École CNRS ICaRE97
Irrégularité : Capa - Rumeur - Exec

Coordonnateurs

- D. Barth
- J. Chassin de Kergommeaux
- J.-L. Roch
- J. Roman

Avant-Propos

La parallélisation d'une application de grande taille nécessite la maîtrise de différentes formes d'irrégularités :

- Irrégularité de l'application dont le traitement nécessite souvent la génération et la manipulation de structures de données complexes et non structurées (files de priorité, matrices creuses ou graphes par exemple).
- Irrégularité de l'algorithme puisque la complexité des dépendances de données rend imprévisible le schéma de calcul (graphe de précédence des tâches et mouvements de données).
- Irrégularité du support d'exécution en raison de la multiplicité des machines parallèles disponibles et de l'émergence de supports hétérogènes performants (réseaux de stations de travail pour le calcul intensif).

Trois des plus importants thèmes de recherche du Groupement de Recherche PRS (Parallélisme, Réseaux et Systèmes) du CNRS (Capa - Rumeur - Exec) spécialisés sur ces différents points se sont regroupés pour proposer une École scientifique qui traite de la parallélisation d'applications irrégulières depuis la construction de l'algorithme parallèle jusqu'à sa mise en oeuvre effective sur une architecture distribuée.

L'École est organisée sur cinq journées. Après une journée d'introduction présentant la problématique générale et les outils de base, chaque journée est consacrée au traitement d'une forme spécifique d'irrégularité, de sa formulation théorique aux outils systèmes nécessaires à l'implémentation. L'École aborde les différentes formes d'irrégularités, en allant du parallélisme de tâches jusqu'à l'optimisation des schémas de communications. En outre, une journée est consacrée à des approches langage ainsi qu'à des présentations de machines parallèles.

Les organisateurs tiennent à remercier :

- Les conférenciers qui par la qualité de leur documents écrits et de leurs présentations ont assuré le succès de l'École.

- Le CNRS dont l'aide financière, au titre de la formation permanente, a été déterminante.

- Le GDR PRS et l'INRIA Rhône-Alpes pour leurs soutiens financiers.

D. Barth, J. Chassin de Kergommeaux, J.-L. Roch et J. Roman

Table des matières

Première partie

De l'algorithmique au support

Chapitre 1

Supports d'exécution : paradigmes

Isabelle Demeure (ENST-Paris), Jean-François Méhaut (LIFL)

1.1 Introduction

Ces dernières années, les évolutions technologiques des machines et architectures parallèles et distribuées ont concerné à la fois les fois les unités de calcul (microprocesseurs) mais aussi les dispositifs de communication comme par exemple les réseaux locaux à haut-débit (Myrinet [5], MPC [28], SCI [18], DEC Memory Channel [16]). Des recherches très actives ont également été menées pour concevoir des environnements de programmation parallèle complets comprenant un ensemble très riche d'outils allant du dévermineur de programmes parallèles jusqu'aux supports d'exécution de bas niveau, en passant par les régulateurs de charge. Nous nous intéresserons dans ce cours principalement aux supports d'exécution en dégageant les principaux paradigmes qu'ils mettent en œuvre. Nous analyserons l'adéquation de ces supports par rapport aux besoins issus d'applications au comportement irrégulier. Nous illustrerons le cours par de nombreuses recherches sur les supports d'exécution, en particulier ceux menés par la communauté Française très dynamique dans ce domaine.

Le premier constat à faire autour des architectures parallèles et distribuées est que les constructeurs ont retenu comme système d'exploitation de leur plateforme, **UNIX** ou un de ses dérivés. Les qualités de ce système sont indiscutables en termes d'outils de développement, d'interactivité ou de disponibilité de logiciels. Cependant, le modèle de programmation UNIX et son interface de programmation ne correspondent pas exactement aux besoins des développeurs d'applications parallèles. Des fonctionnalités élémentaires comme la création de processus (**fork**) ou la communication par (**socket**) nécessitent des connaissances en système que possèdent rarement les développeurs de code scientifique. Le besoin s'est donc fait sentir de développer des couches logicielles intermédiaires offrant des paradigmes et des opérateurs plus évolués et plus simples à mettre en œuvre.

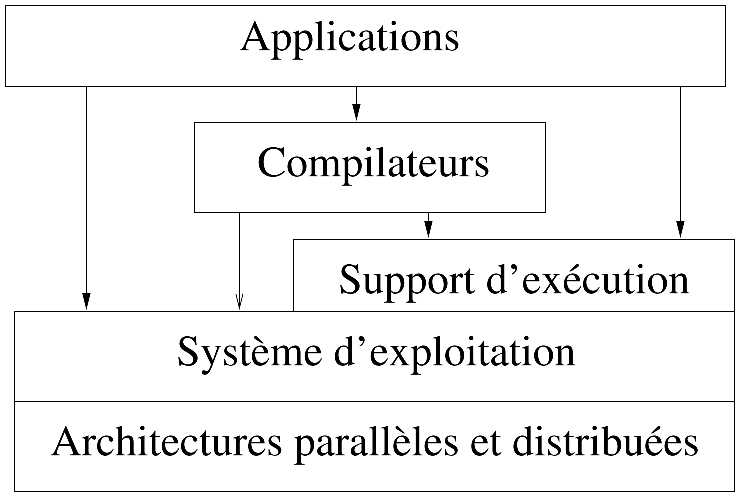


Figure 1.1 : Positionnement des supports d'exécution

Un support d'exécution est donc généralement une couche logicielle développée juste au dessus du système d'exploitation et qui fournit aux programmeurs des paradigmes de programmation parallèles plus évolués que ceux disponibles au niveau des systèmes. Les fonctionnalités du support sont accessibles, soit directement à partir de fonctions appelables d'un langage séquentiel (C, Fortran, C++), soit à partir d'extension de langages (C-Linda), soit encore au travers de langages parallèles généraus développés au dessus des supports d'exécution.

Le second constat concerne les systèmes d'exploitation des architectures parallèles et distribuées. Le programmeur peut être confronté au fait que les différentes implantations et versions d'UNIX peuvent différer d'un constructeur à l'autre et le programmeur doit alors effectuer un travail de portage et de paramètrage qui peut s'avérer fastidieux. D'autres problèmes, comme celui de l'hétérogénéité des architectures, peuvent aussi rendre difficile et ingrat le travail du développeur. Un des rôles des supports d'exécution sera de contribuer à améliorer la portabilité des applications, voire même de faire interopérer des applications s'exécutant dans des contextes hétérogènes.

Les décideurs et utilisateurs d'applications exigent du parallélisme et des programmeurs un niveau d'efficacité proportionnel aux investissements matériels et logiciels. Le contexte de développement (algorithmique, langage, support) doit permetre aux programmeurs de tirer profit des performances intrinsèques des architectures parallèles. Le **support d'exécution** ne doit pas être un frein à l'obtention de bonnes performances. Le surcoût généré par ces environnements devra rester raisonnable en fournissant les paradigmes de programmation les plus évolués avec un niveau d'efficacité suffisant.

Le caractère irrégulier des applications [31] fait qu'il est indispensable de mettre en place des politiques de régulation dynamique de la charge. Les stratégies de régulations pourront être intégrées directement dans le code applicatif ou indirectement par des bibliothèques spécialisées de régulations. Pour mettre en place de la régulation dynamique, il est indispensable de retrouver au niveau du support d'exécution un minimum de fonctionnalités comme par exemple le placement de tâches ou la récolte d'informations sur l'état de charge de la machine [13].

La première partie du cours décrit les paradigmes que l'on retrouve dans les supports parallèles pour exprimer le parallélisme qu'on a identifié dans la phase de conception de l'algorithme. Nous étudierons en particulier les différents niveaux de granularité qu'on peut retrouver dans les environnements ainsi que les outils nécéssaires pour la régulation. La seconde section aborde les paradigmes de communication et de synchronisation. Il est particulièrement important de retenir les bons paradigmes par rapport aux applications ou aux architectures et systèmes qu'on vise.

1.2 Expression du parallélime : les paradigmes

L'algorithmique parallèle [7][8] vise la conception, dans des modèles théoriques tels que le modèle PRAM (Parallel Random Access Machine)[1], d'algorithmes à la fois très parallèles et efficaces. Pour certaines classes d'application, un graphe de précédence peut se construire, statiquement ou dynaniquement, permettant d'identifier les différentes tâches à exécuter et les relations de précédence entre ces tâches. De tels algorithmes et graphes devraient pouvoir facilement être *codés* et pris en charge par les supports d'exécution.

Ces supports d'exécution fourniront donc aux programmeurs la possibilité de créer des tâches sur les différents processeurs de l'architecture. Cette fonctionnalité sera implantée par le support en utilisant les notions d'activités disponibles au niveau des systèmes d'exploitation. Les environnements de programmation classiques, comme PVM[14], MPI [22], LANDA [24] implantent la notion de tâche avec les processus UNIX. Ces environnements sont dits alors comme étant à gros grain car la création d'une tâche entraîne la création d'un processus UNIX et donc la réservation au niveau système de ressources assez importantes (segments de code et de données, table des descripteurs de fichiers...).

Une des évolutions les plus marquantes des systèmes d'exploitation a été d'offrir une nouvelle famille de processus habituellement appelée *processus légers* (thread). Des micro-noyaux comme Mach [1] et Chorus [3] avaient été des pionniers de cette

[1] Ces modèles théoriques sont étudiés dans le cours *Algorithmes irréguliers*.

technologie; des systèmes plus complets et populaires comme Solaris [32] ou même Windows-NT [11] les fournissent en standard à leurs utilisateurs et développeurs. L'objectif prioritaire était principalement d'exploiter efficacement les architectures multiprocesseurs à mémoire partagée (SMP). Les processus légers sont également d'une grande utilité pour réaliser des serveurs de fichiers efficaces, grâce aux possibilités de recouvrement des entrées-sorties. Plus récemment, les concepteurs d'environnements de programmation parallèle ont cherché à définir des modèles de programmation et d'exécution reposant sur les processus légers. Ces environnements [17] [12] [26] [25] seront présentés plus en détails dans le cadre d'un autre cours.

1.2.1 Processus et systèmes d'exploitation

Les processus ont toujours tenu une place importante dans les systèmes d'exploitation, qu'ils soient centralisés ou distribués [33]. D'une manière non formelle, un processus correspond à un programme en cours d'exécution. Le système alloue des ressources mémoire (segments de code, données, pile) mais également un contexte d'exécution (compteur ordinal, registres). L'exécution des processus progresse séquentiellement, c'est à dire qu'à n'importe quel moment une seule instruction au plus est exécutée au sein d'un processus. Les traitements et calculs des utilisateurs sont pris en charge au niveau système par des processus. Nous étudierons dans les sections suivantes les différents types de processus que proposent les systèmes.

1.2.1.a Processus lourds

Nous illustrerons la gestion des processus dans les systèmes par le cas du système **UNIX**. Chaque processus UNIX est identifié par son *identificateur* (*Process_IDentifier*) qui est un entier unique connu et dont la portée reste locale. UNIX ne fournit pas de mécanisme de désignation globale des processus sur un ensemble de sites. Les processus UNIX sont créés par l'appel système **fork**. Le nouveau processus est constitué à partir d'une copie de l'espace d'adressage du processus père. Ce mécanisme permet facilement au processus père de communiquer avec son processus fils. Malheureusement, l'opération **fork** reste locale au site du processus père et ne permet donc pas de créer des processus au travers d'un ensemble de processeurs ou de sites. Les supports d'exécution fourniront des paradigmes permettant de créer et répartir des processus sur différents processeurs ou sites.

L'opération de création de processus UNIX est lourde et gourmande en ressources systèmes. Le nombre de processus supportés par le système est en général limité et lorsque le nombre de processus créés devient trop important, la pagination et le va-et-vient altèrent fortement les performances. C'est pourquoi avec des environnements à gros grain les opérations de création de processus sont souvent limitées à la phase d'initialisation des programmes parallèles. C'est ce que nous constaterons avec les environnements PVM et MPI présentés dans les sections suivantes.

1.2.1.b Processus légers

Si un système d'exploitation supporte l'exécution simultanée de plusieurs programmes (multiprogrammation), le but des processus légers est de multiprogrammer les processus lourds. La multiprogrammation des processus lourds s'appele la multiprogrammation légère et se trouve supportée par des processus légers [21]. Plusieurs processus légers vont donc s'exécuteur au sein d'un même processus lourd et vont se partager les ressources système du processus mais également les données ou variables globales du programme. De même qu'il existe des outils de synchronisation et de communication inter-processus, des outils de synchronisation entre processus légers seront disponibles et permettront aux différents processus légers de se synchroniser pour l'accès aux données et variables partagées.

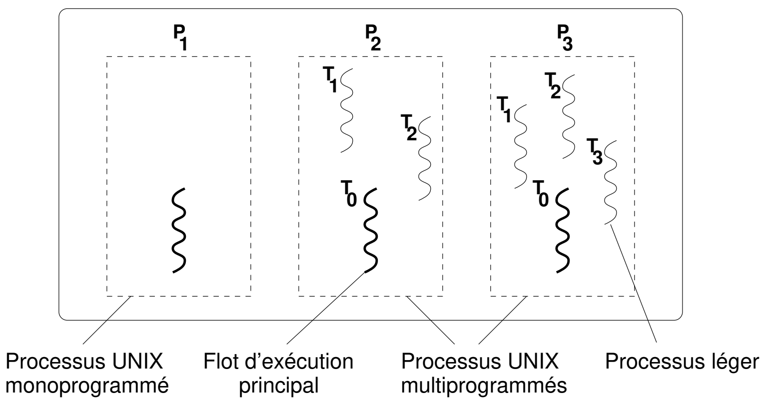


Figure 1.2 : Processus UNIX et processus légers.

Le domaine d'utilisation des processus légers s'est depuis peu étendu au parallélisme grâce à des environnements comme Athapascan, PM^2 ou Nexus. Les qualités des processus légers permettent en particulier dans un contexte parallèle de remédier très simplement au problème du recouvrement des communications par du calcul. Lorsqu'un processus léger effectue une communication, celui-ci se bloque tant que la communication n'est pas terminée, mais d'autres processus légers peuvent pendant ce temps poursuivre leurs calculs. Ces environnements seront présentés plus longuement dans un autre cours.

1.2.2 Processus et environnements de programmation parallèle

Comme nous venons de le constater dans la section précédente, les système d'exploitation ne permettent pas de manipuler facilement des processus répartis sur un ensemble de processeurs ou sites. Les environnements de programmation parallèle vont y remédier en permettant la création distante de processus ou tâches. Un mécanisme de désignation globale permet ensuite d'identifier et de localiser les processus ou tâches lancées par l'environnement. Les identificateurs de tâches peuvent être de simples entiers, comme sur MPI, ou des structures plus complexes renfermant les numéros de processeurs et de tâches (PVM). Les processus sont ensuite **désignés explicitement** dans les opérations de communication et de synchronisation.

Les environnements offrent aussi aux utilisateurs des mécanismes de désignation indirecte permettant à des processus d'interagir sans avoir à se connaître explicitement. C'est pas exemple le cas des groupes de tâches de l'environnement PVM. Un groupe peut se définir comme une collection de processus ou tâches qui ont adhéré ou rejoint le groupe. Ces mécanismes de désignation indirecte permettent la mise en œuvre d'opérations globales de communication et synchronisation.

1.2.2.a Introduction à PVM

PVM (Parallel Virtual Machine) [14] est un environnement de programmation parallèle conçu à l'Oak Ridge National Laboratory. L'originalité de cet environnement est d'avoir introduit le concept de **machine virtuelle** qui a beaucoup influencé l'approche du calcul parallèle en fournissant un environnement simple à installer et disponible sur une grande variété de machines. Une application développée avec l'environnement PVM est constituée d'un ensemble de tâches qui s'exécuteront sur une collection hétérogène de machines qu'on appelle des ¡¡ hosts ¿¿ dans la terminologie PVM.

Nous présenterons dans la section suivante les tâches qui constituent l'unité élémentaire d'exécution dans PVM. Nous insisterons plus précisément sur l'opération de création et sur les différentes possibilités qu'offre PVM pour répartir le calcul.

Création de tâches PVM La primitive *pvm_spawn* permet la création d'une ou plusieurs tâches PVM. Le programmeur spécifie en entrée le type de tâche à exécuter (nom d'un fichier exécutable) et le nombre de tâches à créer. *pvm_spawn* retourne les identificateurs (*tid* : task identfier) des tâches créées.

Le programmeur peut choisir le site ou l'architecture du site de création des tâches à créer. Si aucune information sur le site de création n'est fournie par le programmeur, PVM choisit lui même ce site par un algorithme de répartition cyclique qui ne tient pas compte de la charge réelle (CPU, mémoire, IO) des sites.

Un des modèles de programmation les plus simples à mettre en œuvre sous PVM est celui du Maître-Travailleur. L'initialisation de la tâche Maître consiste généralement à créer des processus travailleurs sur chacun des sites de la configuration.

```
/* Creation des taches travailleuses par le maitre */

nb_tasks = pvm_spawn("travailleur", 0, PvmTaskDefault,
                     "", nb_hosts, tids);
if ( nb_tasks < nb_hosts )
  {
   pvm_exit();
   exit();
  }
```

Le mécanisme de placement offert par PVM est assez restrictif avec comme principal inconvénient que le coût de la création est assez important. Les opérations de création sont donc en général limitées aux phases d'initialisation. Avec un environnement comme PVM les mécanismes de régulation reposent généralement sur un placement des tâches à l'initialisation, suivi d'une distribution des données pendant l'exécution. La répartition des données n'est pas prise en charge par PVM et c'est donc au programmeur de les répartir convenablement sur ses différentes tâches.

1.2.2.b Introduction à MPI

MPI [22] est un standard définissant une interface de communication destinée à l'écriture de programmes parallèles portables. Un des objectifs de cette standardisation est de permettre des communications très efficaces sur les principales plates-formes parallèles. Des fonctionnalités très riches sont fournies par MPI comme :

- communications point à point,
- communications et opérations collectives,
- topologies de processus,
- interface de prise de trace,
- accès à l'environnements d'exécution.

Les implantations de MPI arrivent maintenant à maturité et sont disponibles sur de nombreuses plates-formes. Les implantations du domaine public MPICH [2] et LAM [3] sont parmi celles qui sont le plus utilisées à ce jour. Les constructeurs de machines parallèles (CRAY, IBM) proposent à leurs clients des versions MPI propriétaires très efficaces et optimisées pour le réseau de communication de leurs

[2]URL : `http://www.mcs.anl.gov/mpi/mpich`
[3]URL : `http://www.osc.edu/lam`

machines. Des réalisations de MPI sur les réseaux à haut-débit sont également disposnibles, comme MPI-BIP [29], MPI-FM [20].

Les fonctionnalités de communication seront plus longuement décrites dans une des prochaines sections. Nous présentons maintenant les opérations de manipulation de tâches disponibles dans le standard MPI.

Gestion de tâches MPI La première version du standard MPI ne prévoyait pas de primitive de création dynamique de tâches. L'implantation MPI se devait de fournir un dispositif pour lancer les processus de l'application MPI. C'est souvent par l'intermédiaire d'une commande du type *mpirun* que l'utilisateur peut lancer l'exécution d'un programme en mode SPMD.

Les processus MPI sont identifiés par un numéro de rang unique allant de *0* à *N-1*. Un processus du système devient processus MPI en appelant la primitive *MPI_Init*.

L'initialisation d'une application MPI se décrit très classiquement de la manière suivante:

```
#include <mpi.h>

main (int argc, char **argv)
{
 int rank ;

 MPI_Init (&argc, &argv) ;
 MPI_Comm_rank (MPI_COMM_WORLD, &rank) ;

 if (rank == 0)
  {
   /* tache principale */
   principal () ;
  }
 else
  {
   /* taches travailleuses */
   travailleuse (rank) ;
  }
}
```

Le modèle de processus défini par MPI-2 [23] autorise la création dynamique de processus après que l'application ait été lancée. MPI-2 fournit un mécanisme permettant d'établir des communications entre les nouveaux processus créés et ceux qui ont été lancés au démarrage de l'application. MPI-2 permet aussi à deux applications MPI d'interopérer, même si elles ont été démarrées séparemment.

1.3 Paradigmes pour l'expression de la communication et de la synchronisation

Dans la section précédente, nous avons passé en revue les paradigmes pour l'expression du parallélisme. Dans cette section nous nous intéressons aux paradigmes qui permettent aux unités parallèles de communiquer et de se synchroniser.

Avant de les présenter, nous discutons de caractéristiques architecturales qui ont un impact direct sur la communication et la synchronisation.

1.3.1 Considérations architecturales

Les classifications des architectures parallèles établissent une distinction entre les architectures où les processeurs ont accès à un espace d'adressage partagé, et les architectures où chaque processeur possède son propre espace d'adressage. On utilise souvent les termes de machine à *mémoire partagée* et de machine à *mémoire distribuée*.

Comme l'indique la table 1.1, ces termes peuvent recouvrir différentes réalités matérielles ; Raina [30] a proposé une classification des machines permettant le partage de données qui distingue la notion d'espace d'adressage partagé de la notion d'accès à des modules de mémoire partagée. Il identifie ainsi les architectures :

- SASM : Shared Address space Shared Memory ;
- SADM : Shared Address space Distributed Memory.
- DADM : Distributed Address space Distributed Memory;

On distingue aussi parfois les architectures UMA (Uniform Memory Access) qui sont telles que le temps d'accès aux modules mémoire est le même, quel que soit le module auquel on accède, et quel que soit le processeur qui fait l'accès, des architectures NUMA (Non Uniform Memory Access) qui n'ont pas cette propriété.

1.3.2 Communications par messages

Dans les architectures distribuées, l'échange de messages est le mode de communication le plus "naturel".

1.3.2.a Caractéristiques des communications par messages

Les communications par messages peuvent, tout d'abord, impliquer deux participants, ou plus. Dans le cas de deux participants, la communication est dite *"point-à-point"* ; le participant qui envoie le message est appelé émetteur ; celui qui le reçoit est le destinataire ou récepteur.

Partage physique de la mémoire	Espace d'adressage physique commun	Partage logique	Architecture	Exemples
Non	Non	Non	Répartie	Echange de messages ex : Réseaux de stations, Paragon
Non	Non	Oui	Répartie DADM	partage de données ex : LINDA
Non	Oui	Oui	Multiprocesseur SADM	MVP ex : Munin, DASH, KOAN architectures NUMA
Oui	Oui	Oui	Multiprocesseur SASM	Mémoire partagée ex : Sequent Symmetry

Tableau 1.1 : Partage de "mémoire"

Dans le cas de plusieurs destinataires il s'agit de *diffusion* partielle ou *multicast* (diffusion à un groupe de destinataires) ou de *diffusion généralisée* (à tous). L'identification des destinataires peut se faire soit en les désignant explicitement, soit en désignant un "groupe" auquel ils appartiennent.

Les communications par messages se caractérisent également par le mode de synchronisation entre l'émetteur et le récepteur. Un échange est *synchrone* s'il ne peut avoir lieu que si l'émetteur et le récepteur sont prêts pour la communication (ie. il y a une forme de "rendez-vous" entre eux). L'échange est *asynchrone* si l'émetteur émet lorsqu'il est prêt et le récepteur reçoit lorsqu'il est prêt. Il faut un système de tampons pour mémoriser les messages en attente (émis mais pas encore reçus.

Un autre aspect important est le caractère *bloquant* ou non bloquant d'une primitive d'émission ou de réception. L'émission est bloquante lorsque le système attend au moins que le message ait été envoyé sur le réseau. La réception est bloquante lorsque le destinataire reste bloqué en attente du message jusquà ce que celui-ci soit reçu (ou qu'un temporisateur arrive à échéance). La réception est non bloquante si le destinataire est débloqué immédiatement, même en l'absence de message.

1.3.2.b Les messages sous PVM

Sous PVM, les communications se font en trois étapes ; il faut, tout d'abord, initialiser le tampon qui contiendra le message à émettre ; ceci se fait au moyen des appels *pvm_initsend()* ou *pvm_mkbuf()*. Ensuite, le message doit être composé grâce aux primitives d'empaquettage *pvm_pk** qui permettent de placer des données dans le tampon. PVM prend en compte l'hétérogénéité des nœuds de la machine virtuelle, s'il y a lieu. Enfin, le message est émis vers son (ou ses) destinataire(s). Pour cela, le programmeur a le choix entre les primitives suivantes :

- *pvm_send(int tid, int msgtag)* : associe l'étiquette *msgtag* au message, et l'envoie à la tâche PVM identifiée par *tid*.
- *pvm_mcast(int *tids, int ntask, int msgtag)* : associe l'étiquette *msgtag* au message, et le diffuse aux *ntask* tâches PVM dont les identificateurs sont donnés dans le tableau *tids*.
- *pvm_bcast(char *group, int msgtag)* : elle associe l'étiquette *msgtag* au message, et le diffuse aux tâches qui appartiennent au groupe désigné par *group*.

La primitive *pvm_psend(int tid, int msgtag, void *vp, int cnt, int type)* empaquette et envoie le tableau pointé par *vp*, de type *type* et de taille *cnt*.

La réception se fait en utilisant l'une des primitives suivantes:

- *pvm_recv(int tid, int msgtag)* : cet appel est bloquant. La primitive attend l'arrivée d'un message de type *msgtag*, en provenance de la tâche désignée par *tid*. Pour supprimer le filtrage sur le numéro de tâche ou le numéro de message il suffit d'affecter la valeur -1 au paramètre correspondant.
- *pvm_nrecv(int tid, int msgtag)* : cette primitive est non bloquante. Si un message attendu est présent lors de l'appel, il est reçu. Sinon la primitive retourne la valeur 0.

Le message reçu est placé dans un tampon de réception. Les données sont extraites de ce tampon au moyen de primitives *pvm_unpack** (de façon symétrique à l'opération d'empaquetage).

La primitive *pvm_precv* sert à recevoir et à désempaqueter un tableau (de façon similaire à *pvm_psend*).

Nous donnons, ci-dessous, un exemple d'échange sous PVM.

1.3.2.c Les messages sous MPI

MPI multiplie les primitives de communication en prévoyant des modes de synchronisation plus nombreux que PVM. Les primitives *MPI_send* et *MPI_recv* sont bloquantes. Il existe également des primitives non-bloquantes qui rendent immédiatement la main : ce sont *MPI_Isend* et *MPI_Irecv*. De plus, il existe quatre modes de communication : standard, synchrone, "tamponné" et prêt. Dans le mode tamponné, si le destinataire n'a pas demandé à recevoir un message lorsque l'émission est faite, le message doit être conservé dans un tampon local. Au contraire, le mode

```
/**     operation d'emission **/
float tabReel[10] ;
int tabEntier[10], taille = 10 ;

/* initialise le tampon d'emission */
pvm_initsend(PvmDataDefault) ;

/* insertion des donnees dans le tampon */
pvm_pkfloat(tabReel, taille, 1);
pvm_pkint(tabEntier, taille, 1);

/* emission du message de type 99 */
/* vers la tache identifie par remote_tid */
pvm_send(remote_tid, 99);

/** operation de reception **/
float tabReel[10] ;
int tabEntier[10], taille = 10 ;

/* reception du message de type 99 */
pvm_recv(-1, 99);
/* extraction des donnees */
pvm_upkfloat(tabReel, taille, 1);
pvm_upkint(tabEntier, taille, 1);
```

Figure 1.3 : Exemple de communication sous PVM

prêt suppose que la réception ait été demandée. Ceci permet des implémentations plus performantes.

Enfin, MPI et PVM continuent à évoluer et à s'enrichir de nouvelles primitives. A l'image de ce qui se fait dans des systèmes comme les "Active Messages" [34], les nouvelles versions permettront aux programmeurs de définir des gestionnaires de messages (message handler) à exécuter à la réception des messages pour permettre la programmation événementielle.

1.3.2.d Mise en œuvre du passage de messages

Les systèmes de passage de messages tels que PVM et MPI, s'appuient sur les protocoles de communication disponibles sur la plate-forme sur laquelle ils opèrent.

Ils sont très souvent portés sur la couche socket des systèmes Unix utilisés en support (donc au-dessus des couches TCP/IP ou UDP/IP). Ceci a un impact non négligeable sur les performances résultantes, car le passage d'un message donne lieu à la traversée de plusieurs couches de logiciel et à de multiples recopies du message.

C'est pourquoi, l'on voit se développer des mises en œuvre s'appuyant sur des couches de communication plus basses de façon à tirer un meilleur parti des performances offertes par le matériel. C'est le cas, par exemple, du portage de MPI qui a été fait par une équipe du LIP, sur une plate-forme Myrinet [5].

1.3.3 Paradigmes de plus haut niveau

Le passage de message est un paradigme d'assez bas niveau : il nécessite la manipulation explicite des données à envoyer (ex : empaquetage et dépaquetage, gestion des tampons d'émission et de réception).

Le besoin de paradigmes de plus haut niveau, tels que les appels de procédures distantes, se fait donc sentir. Ce sont des mécanismes de ce type que nous examinons dans cette section.

1.3.3.a Appels de procédures

Le paradigme d'appel de procédures distantes (en anglais Remote Procedure Call ou RPC) a été introduit par Birell et Nelson [BAD84]. La sémantique des RPC, est identique à celle des appels de procédures locaux, à ceci près que dans le cas des RPC la procédure appelée peut se trouver sur un site distinct de celui de l'appelant, site où elle sera exécutée.

En général, un appel de procédure distante est synchrone (l'appelant reste bloqué jusqu'à la fin de l'exécution de la procédure).

Il existe cependant des variantes asynchrones où l'appelant peut soit :

- ne pas attendre la fin de la procédure et se faire rappeler quand celle-ci se termine ;

- poursuivre son exécution sans resynchronisation en fin d'exécution (ce qui suppose que la procédure ne renvoie pas de résultats).

Dans le système Athapascan [6], un calcul est découpé en sous-calculs. Le noyau exécutif parallèle ATHAPASCAN-0, définit une machine abstraite qui permet ce découpage, la distribution des données initiales aux sous-calculs et la recombinaison des résultats des sous-calculs en un résultat global.

Athapascan met en œuvre des communications par appels de procédures distantes. Il propose également, un concept de multiprocédure inspiré du sytème GOTHIC [4] de l'IRISA qui permet d'utiliser de manière transparente des opérations de communication de groupe (diffusion, concentration ou échange total).

1.3.3.b Opérateurs de réduction et communications collectives

Les applications parallèles procèdent souvent en découpant un calcul en souscaculs qui peuvent être effectués en parallèle par plusieurs processus. Il faut alors distribuer les données sur lesquelles portent ces sous-calculs aux processus participants, les laisser calculer, pluis collecter les résultats et les combiner en un résultat final.

MPI comme PVM offrent des primitives pour faciliter ces opérations. Nous présentons ici celles de MPI qui sont plus riches que celles offertes par PVM :

- *MPI_Reduce(void* operand, void* result, int count, MPI_Datatype datatype, MPI_Op operator, int root, MPI_Comm comm)*, les valeurs d'*operande* de chaque processus participant (définis par le communicateur *comm*) sont combinées au moyen de l'opération *operator* ; le résultat est placé dans la variable *result* du processus de rang *root*.

  La primitive *MPI_Allreduce* fait la même chose mais retourne le résultat à tous les processus participant.

- *MPI_Gather(void* send_data, int send_count, MPI_Datatype send_type, void* recv_data, int_recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm)*, collecte les valeurs sauvegardées dans les variables *send_data* de chaque processus participant et les place dans la variable *recv_data* du processus de rang *root*.

  La primitive *MPI_Allgather* fait la même chose mais retourne le résultat à tous les processus participant.

- *MPI_Reduce(void* send_data, int send_count, MPI_Datatype send_type, void *recv_data, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm)* distribue les valeurs sauvegardées *send_data* à tous les processus participant.

1.3.4 partage de données

Le passage de messages présente l'inconvénient d'obliger les processus participant à un traitement parallèle à expliciter les données qu'ils veulent envoyer, à qui ils veulent les envoyer et quand la communication doit avoir lieu. Dans les applications irrégulières les schémas d'échanges peuvent être complexes, et le partage de données par échange de messages peut s'avérer très fastidieux.

Un autre paradigme de communication d'utilisation répandue dans les applications parallèles est le *partage de données*. Il est perçu comme étant plus facile d'utilisation que le passage de messages, car il permet de transposer les habitudes de programmation séquentielle à la programmation parallèle.

Certains environnements de programmation parallèle offrent donc un support pour le partage de données sur une plate-forme distribuée ; c'est le cas, par exemple, de Linda [9], [10], et de TreadMarks [2].

L'objectif de ces environnements est de fournir une abstraction pour le partage de données sur des architectures où il n'y pas pas d'espace d'adressage commun.

Dans les architectures de type "SASM" ou "SADM" évoquées plus haut, des processus s'exécutant sur des processeurs distincts ont accès à un espace d'adressage commun : ils peuvent donc naturellement partager des données.

Dans les architectures de type "'DADM" il n'y a pas d'espace d'adressage commun : c'est donc aux couches logicielles d'émuler le partage de données.

1.3.4.a Modèles de cohérence

Dans une architecture monoprocesseur, une opération d'écriture change la valeur de l'emplacement mémoire écrit et une opération de lecture retourne la dernière valeur écrite. Il n'y a pas, alors, d'ambiguïté possible sur la notion de "dernière valeur écrite" ou sur l'ordre des accès à la mémoire.

Il n'en est pas de même lorsque plusieurs processeurs accèdent concurremment à un espace mémoire. Plusieurs modèles de *cohérence* mémoire sont possibles : ils décrivent les règles régissant les accès concurrents à un ensemble d'emplacements mémoire, en particulier en termes d'ordre relatif. On précise ainsi quels accès doivent être traités avant quels autres, et quel sera le résultat d'un accès en fonction des autres accès en cours et à venir.

Par exemple, suivant le modèle de *cohérence séquentielle*, l'exécution d'un programme parallèle apparaît comme un entrelacement de l'exécution de processus parallèles sur une machine monoprocesseur. Intuitivement, une lecture renvoie la dernière valeur écrite [19]. C'est un modèle très simple mais qui limite les possibilités d'optimisation et ne donne donc pas les meilleures performances.

Notons que dans la communauté matérielle travaillant sur la mémoire partagée on distingue les notions de cohérence et de *consistance*. Nitzberg [27] propose d'utiliser le terme de “cohérence” comme un terme général pour désigner la sémantique des opérations sur la mémoire, et le terme de “consistance” pour désigner une stratégie de cohérence mémoire spécifique.

Dans la communauté logicielle la distinction est rarement faite. Dans la suite de ce chapitre nous n'utiliserons que le terme de “cohérence”.

Des exposés traitant de façon plus détaillée des modèles de cohérence et de la mise en œuvre du partage en environnement distribué seront fait plus tard dans cet ouvrage.

1.3.4.b Partage de données sous Linda

Linda construit un espace de T-uples partagé qui fonctionne comme une mémoire à accès associatif. Un tuple est une suite de champs dont le type est l'un de ceux acceptés par le langage hôte (ex : scalaires, tableaux et structures en C-Linda). Les processus peuvent introduire un tuple dans l'espace à l'aide de la primitive *out* ; ils peuvent sortir un tuple de l'espace (pour le modifier) à l'aide de la primitive *in* ; ils peuvent lire un tuple à l'aide de la primitive *read*.

Il faut noter qu'une caractéristique importante de LINDA est qu'il n'y a jamais d'écritures concurrentes d'un même tuple. Pour modifier un tuple, il faut l'avoir sorti de l'espace des tuples, puis le réintroduire. Les écritures se font ainsi en exclusion mutuelle.

```
/** Programme maitre **/
/* ecrire le tuple contenant la donnee data */
/* a elever au carre */
out("operande", data);
/* creer le processus qui va faire le calcul */
eval("carre", carre());
/* extraire le tuple contenant le resultat */
in("result", ?carre-data)

/** Programme esclave **/
void carre()
{
/* Lire la valeur a elever au carre */
in("operande", ?data);
/* Ecrire le resultat */
out("result", data*data);
}
```

Figure 1.4 : Exemple de communication sous LINDA

1.3.4.c Partage de données sous TreadMarks

TreadMarks est une bibliothèque utilisateur qui a été développée par l'équipe de Willy Zwaenepoel à l'université de Rice. Elle est mise en œuvre au-dessus du système Unix et utilise l'interface socket (UDP/IP) ; les langages hôte sont C, C++ et FORTRAN. Elle a deux objectifs principaux :

- Limiter le nombre de messages nécessaires à la gestion des données partagées (en utilisant un modèle de cohérence dit cohérence à la libération) ;
- Eviter les faux-partages (en utilisant un protocole à écrivains multiples qui permet de modifier différente zones d'une structure de données puis de propager et de fusionner ces modifications).

L'interface utilisateur se compose des primitives suivantes :

- *void Tmk_startup(int argc, char **argv)*, qui intialise Treadmarks et démarre les processus distants ;
- *void Tmk_exit(int status)*, qui termine le processus appelant ;
- *char *Tmk_malloc(unsigned size)*, qui alloue le nombre d'octets spécifié dans la mémoire partagée ;
- *void Tmk_free(char *ptr)*, qui libère la mémoire allouée par Tmk_malloc ;
- *void Tmk_lock_acquire(unsigned id)*, qui bloque l'appelant juqu'à ce qu'il ait acquis le verrou ;
- *void Tmk_lock_release(unsigned id)*, qui relâche le verrou ;
- *void Tmk_barrier(unsigned id)*, qui bloque le processus appelant jusqu'à ce que tous les autres processus arrivent à la barrière;

L'exemple ci-dessous illustre l'utilisation de variables partagées sous TreadMarks.

TreadMarks met en œuvre un modèle de cohérence à la libération.

La cohérence à la libération distingue deux types d'accès : les accès dits ordinaires (lecture - écriture) et les accès de synchronisation : *acquisition* (*acquire*) et *libération* (*release*).

La libération est un accès qui met les modifications apportées par le processeur local depuis la dernière libération effectuée à la disposition de tous les autres processeurs. Dans la mesure où la libération donne l'accès aux modifications effectuées précédemment, il n'est pas utile de retarder des opérations ordinaires si une libération est en cours. En revanche, une libération ne peut être effectuée tant que des opérations ordinaires sont en cours.

```
/* Jacobi sous TreadMarks */
#define  M  1024
#define  N  1024
float    **grille ;    /* tableau partage */
float    brouillon[M][N] ;  /* tableau local */
int  i, j, debut, fin, taille, nb_ite;

init(&nb_ite);

Tmk_startup();
if (Tmk_proc_id =0){
    grille = Tmk_malloc(M * N * sizeof(float));
    initialize_grille();
}
Tmk_barrier(0);
taille = M / Tmk_nprocs ;
debut = taille * Tmk_proc_id ;
fin = taille * (Tmk_proc_id + 1) ;

while (nb_ite){
  for (i = debut ; i < fin ; i++)
    for (j = 0 ; j < N ; j++)
      brouillon[i][j] = (grille[i-1][j] + grille[i+1][j] +
        grille[i][j-1] + grille[i][j+1]) / 4 ;
  Tmk_barrier(1);
  for (i = debut ; i < fin ; i++)
    for (j = 0 ; j < N ; j++)
      grille[i][j] = brouillon[i][j] ;
  Tmk_barrier(2);
  nb_ite--;
}
```

Figure 1.5 : Exemple de programmation sous TreadMarks

L'acquisition assure que la dernière version des données est présente localement. La dernière version correspond à la valeur des données suite à la dernière libération effectuée dans le système. Dans la mesure où cet accès valide les données dernièrement "libérées", il n'a pas besoin d'être retardé pour attendre la fin d'opérations ordinaires locales. Remarquons qu'une opération ordinaire ne peut être effectuée tant qu'une acquisition est en cours. Pour plus de détails sur ces modèles le lecteur pourra se reporter à [15].

1.3.5 Synchronisation

L'échange de messages permet l'échange de données entre processus, mais également leur synchronisation (en utilisant le caratère synchrone/asynchrone des échanges, le caractère bloquant/ non bloquant des primitives).

Par ailleurs, les environnements de programmation parallèle comme PVM et MPI mettent en œuvre la synchronisation par barrières. Une barrière est mise en œuvre au moyen d'une fonction bloquante : une fois qu'un processus l'appelle, il ne sera débloqué que lorsque tous les autres processus l'auront eux-même appelée. Ceci correspond à un rendez-vous multiple.

Sous MPI la primitive de barrière est :
int MPI_Barrier(MPI_Comm com) ; la primitive correspondante sous PVM est :
*int info = pvm_barrier(char *group, int count)* où *count* désigne le nombre de processus du groupe *group* qui doivent s'attendre à la barrière.

Le système TreadMarks possède un appel équivalent : *Tmk_barrier* ; il profite de cette synchronisation pour rendre globales les modifications faites localement aux variables partagées.

Enfin, lorsque plusieurs processus accèdent concurremment à des données partagées, il est nécessaire de permettre l'accès en exclusion mutuelle.

A cet effet, TreadMarks fournit deux primitives de "verrouillage" et "déverrouillage" : *void Tmk_lock_acquire(unsigned id)* et *void Tmk_lock_release(unsigned id)* où id désigne un numéro de verrou.

1.4 Conclusion

La difficulté pour un environnement de programmation réside dans les délicats compromis qu'il faut adopter entre les paradigmes qui sont offerts aux programmeurs, le niveau de portabilité et l'efficacité globale de l'environnement.

Le compromis **paradigmes/portabilité** concerne principalement la distance entre les paradigmes de programmation et les modèles d'exécution architecturaux et système. D'une manière générale, plus les paradigmes seront évolués et puissants (migration), plus ils seront difficiles à mettre en œuvre sur de nouvelles architectures et cela affaiblira le niveau de portabilité de l'environnement.

Le compromis **portabilité/efficacité** concerne principalement l'implantation des environnements. Si l'implantation s'appuie exclusivement des bibliothèques spécialisées fournies par les constructeurs, les performances de l'environnements seront certainement excellentes. Cependant, la portabilité d'un tel environnement sera alors le point délicat car le code est alors difficilement réutilisable sur les plate-formes des autres constructeurs qui ne fourniront pas ces mêmes bibliothèques.

Le compromis **paradigmes/efficacité** est la conséquence du fait que l'ajout de paradigmes a souvent des répercussions sur le fonctionnement général de l'environnement. Un exemple simple à comprendre est celui de la gestion des tampons PVM qui sont très simples d'utilisation, mais qui génèrent des surcoûts importants par rapport à des communications où les données pourraient être directement écrites sur les canaux de communication (*sockets* UNIX) ou même directement sur les cartes d'interface de communication.

Bibliographie

[1] Acetta (M.), Baron (R.), Golub (D.), Rachid (R.), Tevanian (A.) et Young (M.). – *Mach: A kernel foundation for UNIX development.* – Rapport technique, Carnegie Mellon, Department of Computer Science, 1993.

[2] Amza (C.), Cox (A.), Dwarkadas (S.), Keleher (P.), Lu (H.), Rajamony (R.), Yu (W.) et Zwaenepoel (W.). – Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, vol. 29, n° 2, February 1996.

[3] Armand (F.), Herrmann (F.), Lipkis (J.) et Rozier (M.). – Multi-threaded processus in chorus/mix. *In : EUUG Spring'90 Conference.* – Munich, germany, 1990.

[4] Banâtre (J.-P.) et Banâtre (M.). – Les systèmes distribués : Expérience du projet gothic. *InterEditions*, 1991.

[5] Boden (N.), Cohen (D.), Feldermann (R.), Seitz (C.) et Su (W.). – Myrinet: A gigabit per second local area network. *IEEE-Micro*, vol. 15, feb 1995, pp. 29–36.

[6] Briat (J.), Christaller (M.) et Roch (J.-L.). – Une maquette pour athapascan-0. *In Luc Bougé, editor, Actes des 6-ièmes Rencontres francophones du Parallélisme, RenPar'6*, Juin 1994, pp. 231–235.

[7] Briat (J.), Gautier (T.) et Roch (J.). – Application irrégulière et ordonnacement en ligne. *In : placement dynamique et répartition de charge.* pp. 81–105. – Presqu'île de Giens France, Juil. 1996.

[8] Briat (J.), Gautier (T.) et Roch (J.). – On-line scheduling. *In : European School of Computer Science : Parallel Programming Environnement for High Performance Computing*, pp. 95–108. – Alpe d'Huez France, Avr. 1996.

[9] Carriero (N.) et Gelernter (D.). – How to write parallel programs: A first course. *MIT Press*, 1990.

[10] Corbel (A.) et Fleter (F.). – LINDA : un modèle de programmation parallèle. *Calculateurs Parallèles*, vol. 7, n° 2, 1995, pp. 159–172.

[11] Custer (H.). – *Inside Windows NT.* – Microsoft Press, 1993.

[12] Foster (I.), Kesselman (C.) et Tuecke (S.). – The Nexus task-parallel runtime system. *In : Proc. 1st Intl Workshop on Parallel Processing.* – Tata Mc Graw Hill.

[13] Garcia (J.) et Monteil (T.). – Un modèle d'exécution et de prédiction de charge pour le placement de tâches du système network-analyser. *In : placement dynamique et répartition de charge*. pp. 263–281. – Presqu'île de Giens France, Juil. 1996.

[14] Geist (A.), Beguelin (A.), Dongarra (J.), Jiang (W.), Mancheck (R.) et Sunderam (V.). – *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, 1994.

[15] Gharachorloo (K.), Lenoski (D.), Laudon (J.), Gibbons (P.), GUPTA (A.) et Henessy (J.). – Memory consistency and event ordering in scalable shared-memory multiprocessors. *In proceedings of the ISCA conference*, 1990, pp. 15–26.

[16] Gillett (R.). – Memory Channel fot PCI. *IEEE Micro*, vol. 16, n° 2, February 1996, pp. 12–18.

[17] Ginzburg (I.). – *Athapascan-0b: Intégration efficace et portable de multiprogrammation légère et de communications.* – LMC, Thèse de PhD, Institut National Polytechnique de Grenoble, Sep 1997.

[18] IEEE. – *Standard for Scalable Coherent Interface (SCI)*, Aug. 1993. Standard no. 1596.

[19] Lamport (L.). – How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, vol. 28, n° 9, September 1979, pp. 690–691.

[20] Lauria (M.) et Chien (A.). – MPI-FM: High performance MPI on workstation clusters. *Jounal on Parallel and Distributed Computing*, no40 (01), 1997, pp. 4–18.

[21] Lewis (B.) et Berg (D.). – *Threads Primer. A Guide to Multithreaded Programming.* – Prentice Hall, 1996.

[22] Message Passing Interface Forum. – *MPI: A Message-Passing Interface Standard*, March 1994. available from http://www.mpi-forum.org.

[23] Message Passing Interface Forum. – *MPI-2: Extensions to the Message-Passing Interface*, July 1997. available from http://www.mpi-forum.org.

[24] Monteil (T.), Garcia (J.) et Guyaux (P.). – LANDA : Une machine virtuelle parallèle. *Calculateurs Parallèles*, no7(2), 1995, pp. 119–137.

[25] Namyst (R.). – *PM² : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* – Thèse de PhD, Université de Lille 1, Jan 1997. Available at *http://www.lifl.fr/~namyst/these.html.*

[26] Namyst (R.) et Mehaut (J.). – PM²: Parallel Multithreaded Machine. a computing environment for distributed architectures. *In : ParCo'95 (PARallel COmputing).* pp. 279–285. – Elsevier Science Publishers.

[27] Nitzberg (R.) et Lo (V.). – Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, vol. 24, n° 8, August 1991, pp. 52–60.

[28] Potter (F.). – *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour machines multiprocesseurs.* – Thèse de PhD, Université Pierre et Marie Curie – Paris VI, Avril 1996.

[29] Prylli (L.) et Tourancheau (B.). – *Protocol design for high performance networking: A Myrinet experience.* – Rapport technique n° RR97-22, Ecole Nationale Supérieure de Lyon, 69364 Lyon Cedex 07, Laboratoire d'Informatique du Parallélisme, july 1997.

[30] Raina (S.). – Virtual shared memory: A survey of techniques and systems. *Technical report, Dept. of Computer Science, University of Bristol*, 1991.

[31] Roucairol (C.). – *Stratagème : Une méthodologie de programmation parallèle pour les problèmes non structurés.* – Rapport de Recherche, Versailles, PRiSM, Dec 1995.

[32] SunSoft. – *SunOs5.2 Guide to Multi-Thread Programming*, 1993.

[33] Tanenbaum (A.). – *Modern Operating Systems.* – Prentice Hall International Editions, 1992.

[34] von Eicken (T.), Culler (D.), Goldstein (S.) et Schauser (K.). – Aactive messages : a mechanism for integrated communication and computation. *In proceedings of the 19th International Symposium on Computer Architecture*, 1992.

Chapitre 2

De l'algorithme au support

V.-D. Cung (PR*i*SM),
P. Fraigniaud (LIP),
T. Gautier (INRIA/LMC),
D. Trystram (LMC)

2.1 Introduction

L'utilisation d'une architecture parallèle pour la résolution d'un *problème* passe par l'écriture d'un programme parallèle : l'*algorithmique parallèle* permet alors de construire un programme *performant* et d'estimer ses performances relativement à un modèle de machine.

En première approximation une architecture parallèle peut être vue comme un ensemble de ressources permettant de calculer et de communiquer. Un programme parallèle est composé d'un ensemble d'activités concurrentes utilisant ces ressources. Une exécution performante d'un programme nécessite donc de bien utiliser ces ressources. Le critère de performance est généralement le temps d'exécution du programme. Il s'agit donc :

- d'écrire un programme performant à partir d'un algorithme parallèle performant, nous renvoyons aux chapitres sur l'algorithmique PRAM [33, 34] pour une description des techniques relatives à la création d'algorithmes parallèles efficaces,
- d'exécuter efficacement le programme en se basant sur des mécanismes de *régulation de charge* afin d'utiliser au mieux les ressources disponibles.

Dans ce chapitre, nous considérons uniquement le second point. Nous nous attacherons, d'une part, à présenter les principales techniques utilisées pour obtenir exécution efficace et, d'autre part, à dégager une typologie des méthodes de régulation de charge.

2.1.1 Contexte et définitions

Le passage d'un programme source à son exécution par une machine parallèle peut être décomposé en trois étapes.

- La détermination du parallélisme du programme : suivant le langage de programmation utilisé, cette étape nécessite l'utilisation des techniques des compilateurs paralléliseurs (typiquement HPF, *High Performance Fortran* [47]), ou, plus simplement, elle est intégrée explicitement dans l'écriture du programme (typiquement ADA). À la fin de cette étape une représentation du programme original sous forme de *graphe*[1] est disponible pour l'étape suivante.
- Le groupement en *tâches* séquentielles : cette étape, a pour objectif d'augmenter la *granularité* du programme parallèle afin de réduire le coût dû aux communications. Le graphe obtenu à la fin de cette étape est appelé le *graphe de tâches* du programme.
- Le calcul d'un ordonnancement et d'un placement pour l'exécution des tâches : les tâches sont alors affectées (*algorithme de placement*) aux ressources de calcul pour y être exécutées, l'ordre de leur exécution peut aussi être déterminé (*algorithme d'ordonnancement*).

Suivant la modélisation faite du programme, plusieurs types de graphes peuvent être considérés. De même suivant la modélisation des coûts choisis, différents modèles d'exécution ont été proposés. La terminologie couramment employée dans la littérature pour décrire ces modélisations est la suivante.

Modèle de programme :

Le *graphe de tâches* : les nœuds du graphe qui représente le programme parallèle sont des *tâches* (une activité séquentielle, par exemple un bloc d'instructions d'un programme). Une tâche peut correspondre à une instruction comme à un ensemble d'instructions résultant d'un groupement. Le graphe peut être orienté et l'on parle alors d'un *graphe de précédence*. Dans le cas où le graphe n'est pas orienté on parle d'un *graphe de dépendance*.

[1] Nous préciserons plus en avant le type de représentation.

Le *graphe de flots de données* : la précédence entre tâches est induite par les circulations de données. Typiquement, les tâches correspondent à l'évaluation d'une instruction et les précédences aux accès en lecture ou écriture des opérandes.

Modèle d'exécution :

Les *processeurs* : suivant le modèle considéré les processeurs sont dits *identiques* s'ils sont capables d'exécuter chaque tâche et que la durée d'exécution d'une tâche T est la même quel que soit le processeur. Ils sont dits *uniformes* si leurs vitesses diffèrent. Chaque tâche peut être exécutée par chacun des processeurs. De plus si un processeur est k fois plus rapide qu'un autre pour exécuter une tâche, il sera aussi k fois plus rapide pour toutes les autres tâches. Enfin, les processeurs sont dits *sans relation* lorsqu'ils ne peuvent pas exécuter indifféremment chacune des tâches ou que la durée d'exécution dépend des tâches considérées.

Les *communications* : un programme parallèle est amené à échanger des informations. Ces communications peuvent s'effectuer soit par une mémoire partagée par l'ensemble des processeurs (PRAM [28]) soit par un réseau d'interconnexion (XPRAM).

Si le modèle PRAM considère la durée d'accès à un élément mémoire comme étant constant (accès *uniforme*), d'autres modèles tiennent compte d'accès *non uniforme*[2] (par exemple, LPRAM [1]).

D'autres modèles se concentrent sur la *localité* des communications. Par exemple le modèle LXPRAM [5] n'autorise que les communications entre «voisins» et permet de prendre en compte la topologie du réseau d'interconnexion.

Quelques modèles autorisent le *recouvrement* des communications par des calculs et modélisant en cela certaines architectures de machine (par exemple l'Intel PARAGON) qui permettent de décharger le processeur de calcul lors des communications grâce à l'utilisation d'un processeur de communication dédié.

Certains modèles ont un fonctionnement *asynchrone* (par exemple le modèle APRAM [18]) permettant de tenir compte de la variation de la vitesse des processeurs due aux interruptions d'un système d'exploitation. Le modèle BSP [57] est fondé sur un fonctionnement *synchrone* : un programme BSP est une séquence de super-pas (*super step*) de calcul de durée fixée qui se terminent par une étape de synchronisation garantissant la terminaison des communications initiées au cours d'un super-pas.

[2]Par exemple à cause de l'existence de différent types de mémoire.

La donnée d'un modèle d'exécution permet d'étudier l'impact de ses caractéristiques dans l'efficacité d'un programme. Les coûts (temps de calcul, temps communication) d'une exécution y sont déterminés. Le graphe de tâches peut alors être pondéré et valué par ces informations de coût. Celles-ci se divisent en deux catégories : d'une part les informations permettant de calculer un ordonnancement (ou un placement), d'autre part les informations permettant de mesurer la qualité d'une exécution d'un programme. Ces dernières sont très importantes puisqu'elles permettent de mesurer la qualité de la sortie des algorithmes d'ordonnancement ou de placement utilisés.

Les informations de coût les plus utilisées sont les suivantes.

- Le *temps d'exécution* : lorsqu'on parle d'un programme c'est le temps qui s'écoule entre le début et la fin d'exécution de *toutes* les tâches qui le compose ; sur le graphe de tâches cette notion se retrouve dans la longueur du plus long chemin du graphe pondérée par les différents coûts associés aux tâches (temps de calcul) en tenant compte des relations de précédences (temps de communication).

- Le *travail d'un programme parallèle* est la somme des temps d'exécution de l'ensemble des tâches du graphe. Dans un modèle où les temps des instructions élémentaires sont unitaires (par exemple le modèle PRAM [28, 33]), cela correspond au nombre d'instructions exécutées par l'ensemble des tâches. Un programme parallèle de *travail efficace* est un programme qui possède un travail équivalent au travail d'un programme séquentiel semblable[3].

- Intuitivement, plus l'exécution d'un programme parallèle est *efficace* plus les ressources de calcul sont utilisées. L'*efficacité* est associée à la notion de rendement des ressources d'une architecture parallèle.

- Le *grain* d'un programme parallèle est le rapport entre son coût en calcul et son coût en communication. Un programme qui possède une granularité importante calcule plus qu'il ne communique.

- Le *grain* d'un modèle d'exécution caractérise le rapport entre sa capacité à calculer et à communiquer.

- Un programme est dit *extensible* si son temps d'exécution est inversement proportionnel aux nombres de ressources de calcul utilisées. Plus généralement, un programme est extensible s'il peut utiliser un nombre quelconque de ressources (dans un intervalle spécifié) tout en ayant une bonne efficacité.

[3]Certaines définitions considèrent le «meilleur» programme séquentiel équivalent, d'autres le programme parallèle exécuté sur un processeur.

Ces notions sont définies précisément par la donnée d'un modèle d'exécution donné, tels qu'un de ceux cités précédemment.

2.1.2 Objectifs de la régulation de charge

Les algorithmes de régulation de la charge ont pour but une meilleure utilisation des ressources (partagées) d'une architecture parallèle afin d'améliorer les performances globales des programmes.

Les techniques mises en œuvre sont proches d'autres préoccupation lors du développement d'un programme parallèle :

- le rendre tolérant aux pannes,
- assurer sa portabilité d'une architecture à une autre tout en maintenant une bonne performance à l'exécution.

Dans ce chapitre, nous nous intéresserons essentiellement à l'amélioration des performances. Plus précisément, la suite de ce chapitre est organisée de la manière suivante :

- la section 2.2 présente brièvement les différentes techniques permettant de passer d'un programme à une modélisation sous forme de graphe,
- dans les sections 2.4.1 et 2.5 nous présenterons quelques algorithmes de d'ordonnancement et de placement dans un cadre statique ou dynamique,
- enfin nous présenterons dans la section 2.6 la problématique et certaines solutions au problème du routage pour réduire le coût des communications.

2.2 Du programme à son exécution

Dans cette section, nous présentons brièvement les différentes techniques de construction des graphes modélisant à partir de l'expression d'un programme dans un langage. Ces graphes seront ensuite traités par des algorithmes de placement, d'ordonnancement qui utilisent des algorithmes de routage.

2.2.1 Construction d'un graphe de tâches

Toute exécution d'un programme peut être représentée par un graphe de dépendance de données, d'opérations ou de tâches. Suivant les programmes, ces graphes peuvent être déterminés totalement avant exécution ou pendant l'exécution, ou bien ils ne peuvent être déterminés que partiellement voire uniquement après l'exécution (*cf.* section 2.2.2). Notons que la génération proprement dite est un problème difficile, il n'existe que peu de réalisations pratiques.

Ce problème de la construction du graphe de dépendance peut être résolu par des techniques de compilation (analyses de dépendance) telles qu'utilisées par les compilateurs vectoriseurs ou paralléliseurs (HPF [47]) ou plus simplement elles peuvent être intégrées au mécanisme d'expression/d'exécution[4] des programmes (Cilk [10], Ath1 [48, 21]). À l'heure actuelle, les approches suivies par Cilk/Ath1 semblent plus indiquées pour le traitement des applications irrégulières.

2.2.2 Ordonnancement, Placement et exécution de programme

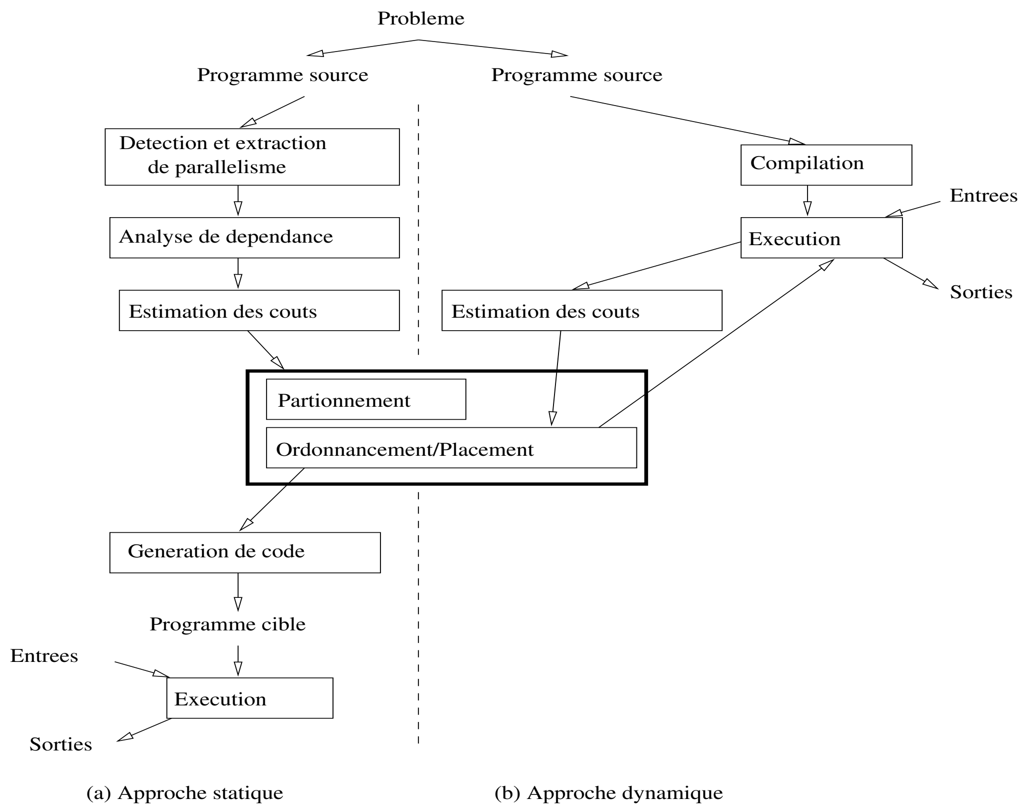


Figure 2.1 : Contexte d'utilisation des outils pour la régulation de charge.

Comme le présente la figure 2.1, suivant que la phase de construction du graphe se situe au moment de la compilation ou durant l'exécution on parle d'ordonnancement statique ou dynamique. Cette distinction est insuffisante pour prendre en compte le champ d'application des algorithmes d'ordonnancement.

La détermination du graphe de tâches permet de mieux cerner les techniques à utiliser.

[4]Le graphe est construit puis ordonnancé lors de l'exécution.

Graphe prévisible : les techniques classiques de placement et d'ordonnancement [9, 16, 53] peuvent être utilisées, que ce soit avant ou pendant exécution.

Graphe semi-prévisible : l'exécution d'un programme consiste en des phases de construction de graphe suivi par des phases d'ordonnancement et d'exécution pouvant utiliser les techniques pour les graphes prévisibles. Du fait du calcul d'un placement ou d'un ordonnancement en cours d'exécution, l'algorithme utilisé ne doit pas être trop coûteux, par exemple en utilisant un algorithme parallèle.

Graphe imprévisible : il s'agit d'utiliser des techniques de placement ou d'ordonnancement dynamiques (cas des problèmes combinatoires d'optimisation) qui ne nécessitent pas d'informations de coût provenant de l'application.

Une étape d'ordonnancement ou de placement avant exécution peut être couplée à l'utilisation d'un mécanisme de régulation dynamique de charge.

Que ce soit dans le cadre statique ou dynamique, les algorithmes d'ordonnancement ou de placement sont chargés du contrôle de l'exécution du programme. Ces algorithmes d'ordonnancement ou de placement peuvent nécessiter certaines opérations, comme la *préemption* de l'exécution d'une tâche, sa *migration* sur un autre site d'exécution (continuation de son exécution) ou sa *reprise* (re-démarrage au début de son exécution).

L'architecture générale d'un mécanisme de régulation de charge est présentée dans la section 2.3.

2.2.3 Exemples de langages ou d'environnements de programmation

Plusieurs langages ou environnements de programmation parallèle intègrent des outils d'ordonnancement ou de placement. Nous en présentons quelques uns en ne retenant que les points clés suivants.

- L'expression du parallélisme : celle-ci peut être *explicite* ou *implicite*.
- Le mécanisme de régulation dynamique de la charge peut, lui aussi, être explicite (à la charge du programmeur d'utiliser certains primitives) ou implicite. Certains outils permettent de définir un régulateur de charge spécifique à un problème donné : nous dirons qu'il est *paramétrable*.
- Le champ d'application de ces outils (*statique* ou *dynamique*) et certaines applications traitées.

Cilk, Ath1. Cilk [10] est une extension du langage de programmation basée sur le langage C. Un programme en Cilk est un ensemble de procédures, chacune étant constituée d'une séquence de *thread*[5]. Un thread est une fonction non-bloquante, *i.e.* une fois lancé un thread ne peut jamais bloquer sur une synchronisation. La communication entre un thread et son créateur (par exemple une valeur de retour) nécessite l'expression explicite de la *continuation* de l'exécution du créateur par le lancement d'un thread.

Le passage des arguments ou le retour des résultats introduisent des précédences entre les différents threads. Le graphe de tâches est construit et ordonnancé en cours d'exécution par un algorithme de *work-stealing* (*cf.* section 2.5.2).

Ath1 [21] est un langage similaire à Cilk qui permet, à la différence de Cilk, de manipuler des objets en mémoire partagée[6] en spécifiant leurs modes d'accès (lecture/écriture). En particulier, Ath1 défini les modes d'accès en lecture concurrente et écriture concurrente par accumulation. Des informations de coût peuvent être spécifiées lors du lancement d'un thread et qui sont utilisées par certains algorithmes d'ordonnancement.

Actuellement, Ath1 s'utilise à travers une interface applicative écrite en C++. Un ensemble d'algorithmes d'ordonnancement sont offerts (*work-stealing*, *cyclique centralisé*, *aléatoire*) et des outils de développement[7] permettent à l'utilisateur d'en écrire de nouveaux.

Cilk et Ath1 sont dédiés aux applications irrégulières ayant un comportement dynamique (création dynamique d'activités). Leur principal intérêt est la prédictibilité des performances des exécutions et la portabilité des programmes.

HPF. Le langage HPF [47] se base sur Fortran (séquentiel) augmenté de quelques instructions «parallèles» (boucle parallèle) et de directives de compilation qui précisent une distribution des données. Le compilateur est alors chargé de générer le code pour l'ensemble des processeurs, y compris la gestion des communications.

La régulation de charge consiste essentiellement en le choix d'une bonne distribution des données permettant d'équilibrer les calculs sur les différents processeurs. Cette distribution peut être éventuellement remise en question en cours d'exécution.

Le langage HPF est essentiellement destiné aux applications qui manipulent des structures de données régulières. Des extensions sont proposées en vue

[5] Processus léger : le terme «léger» vient de ses coûts de création, et de gestion plus faibles que pour les processus au sens «Unix» appelés *processus lourd*.

[6] Éventuellement «virtuellement» partagée sur une architecture à mémoire distribuée.

[7] Ensemble de classes C++.

d'intégrer un parallélisme à base de tâches [47] ainsi que des outils [3] pour la manipulation de structures de données irrégulières (par exemple des matrices creuses).

Pyrros. Pyrros [60] est un outil d'ordonnancement statique de graphe de tâches[8] orienté sans cycle, et de génération de code pour des architectures de type MIMD (processeurs identiques). La prise en compte du recouvrement des communications par des calculs est possible.

L'expression du parallélisme est considérée comme explicite : l'utilisateur doit fournir le graphe de tâches. La taille du graphe de tâches pouvant être très important, certaines études tentent d'utiliser une paramètrisation. L'ordonnancement de ce graphe est constitué de deux étapes. La première consiste en le regroupement des tâches en *clusters* en utilisant un algorithme de type DSC (*Dominant Sequence Clustering*). Ces clusters sont ensuite regroupés en P *super-clusters* (P est le nombre de processeurs), puis placés sur les processeurs.

Les applications visées par Pyrros sont les applications dont on connaît le graphe de tâches avant exécution (graphe prévisible).

2.3 Architecture d'un processus de régulation de charge

Il est maintenant bien acquis [33, 27, 55] qu'un processus de régulation peut être schématisé comme indique la figure 2.2.

Le **placement statique spatial** (*mapping*) et **temporel**, i.e. l'**ordonnancement** (*scheduling*) des objets se font à la compilation et à l'édition des liens. Ces placements sont adaptés aux graphes prévisibles. Nous verrons dans la section 2.4.1 quelques modèles mathématiques de placement/ordonnancement et les algorithmes de résolutions associés. Cette étape de répartition statique de charge présente aussi un intérêt certain pour les applications à graphe semi-prévisible. Elle assure dans ce cas un bon placement/ordonnancement initial.

Pour les graphes (semi-/im)prévisibles, seule la régulation dynamique peut résoudre le problème du déséquilibre des charge encours de calcul afin d'éviter des situations de famine sur certains processeurs et de surcharge sur d'autres. À chaque instant de l'exécution, il est quasi impossible d'estimer avec exactitude la quantité de travail restant à effectuer dans ce type d'applications.

Cependant, il convient de distinguer deux types de régulation dynamique de charge, une non préemptive appelée placement dynamique, l'autre préemptive comme l'équilibrage de charge ou le partage de charge. Le second type nécessite la mise en

[8]Un langage de description est fourni. Il permet de définir les coûts en calcul et communication.

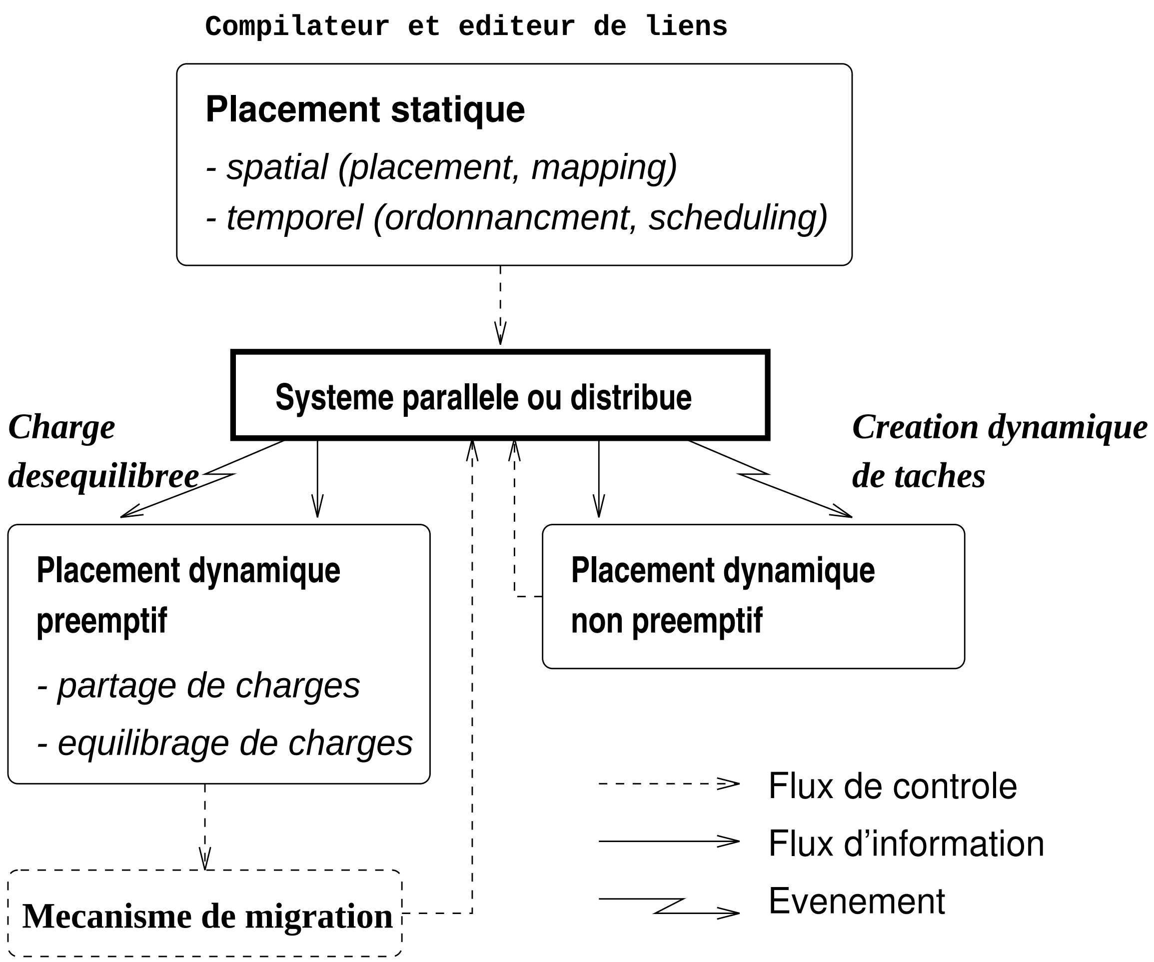


Figure 2.2 : Architecture d'un processus de régulation de charge.

place d'un mécanisme de migration afin de transférer une tâche d'un processeur à un autre.

2.4 Placement et ordonnancement statique

Dans cette partie, nous présentons les problèmes posés par le placement et l'ordonnancement dans un cadre statique et quelques unes de leurs solutions.

2.4.1 Le placement statique

2.4.1.a Une formulation des problèmes

Les hypothèses usuellement adoptées pour le placement statique sont les suivantes.

- Une application est généralement représentée par un graphe prévisible de tâches. Ce graphe est indépendant des données en entrée de l'application.

- Les temps de calcul et de communication sont connus *a priori* et obtenus du compilateur ou de l'éditeur de liens.

Le placement est la plupart du temps considéré comme un problème d'optimisation combinatoire. La fonction objective à optimiser est généralement le temps total de calcul de l'application, i.e. le temps de la dernière tâche. Ce temps dépend du modèle d'architecture de machine utilisée : avec ou sans entrelacement des communications et des calculs, avec ou sans communications silmutanées, etc.

Dans ce cas, le placement peut être alors défini comme une fonction $placer()$ qui associe une tâche t à un processeur p, $\forall\, t \in T, \exists\, p \in P, placer(t) = p$, où T est l'ensemble des tâches à placer et P l'ensemble des processeurs.

Si nous dénotons par PL l'ensemble de tous les placements réalisables, $|P|$ le nombre de processeurs dans P et $|T|$ le nombre de tâches dans T, alors $|PL| = |P|^{|T|}$ et en général $|T| \geq |P|$. Soient $t_{comp}(t_i)$ le temps de calcul de la tâche t_i et $t_{comm}(t_i, t_j)$ le temps de communication de la tâche t_i à la tâche t_j. Alors une fonction objective, sans tenir compte des communications par les calculs ni communications simultanées, s'écrirait :

$$t_{placer}(p_k) = \sum_{t_i / placer(t_i) = p_k} [t_{comp}(t_i) + \sum_{t_j / placer(t_i) \neq placer(t_j)} t_{comm}(t_i, t_j)],$$

$t_{placer}(p_k)$ traduit le temps cumulé sur le processeur p_k alors que le membre droit représente le temps d'exécution de la tâche t_i pour un placement donné. Par conséquent, le temps total d'un placement est $T_{placer} = \max\limits_{p_k \in P}(t_{placer}(p_k))$, et la solution optimale est $T^* = \min\limits_{placer \in PL}(T_{placer})$.

Le problème du placement est connu comme étant combinatoirement difficile (dure pour la classe NP [30]).

Des considérations autres que celles déjà évoquées sur les communications, comme l'hétérogénéité des processeurs [50, 41], peuvent être intégrées au problème de base. Il est également possible d'utiliser d'autres fonctions objectives qui tiennent compte en plus un critère d'équilibrage de charge entre les processeurs en minimisant l'écart des temps de chacun des processeurs par rapport au temps moyen. On écrirait alors la fonction objective comme suit :

$$t_{placer} = \sum_{p_k \in P} |t_{placer}(p_k) - \frac{\sum\limits_{p_l \in P} t_{placer}(p_l)}{|P|}|.$$

2.4.1.b Méthodes de résolution

Le problème du placement de tâches se formule comme un problème d'optimisation combinatoire difficile. En général, les méthodes de résolution sont des heuristiques, inspirées des domaines de la Recherche Opérationnelle ou de l'Intelligence Artificielle. Les méthodes exactes (type «Branch-and-Bound» ou des méthodes de la théorie des graphes comme des couplages) sont proscrites pour des problèmes de taille raisonnable dû au phénomène d'explosion combinatoire.

Certaines méthodes exactes de résolution sont fondées sur des hypothèses peu réalistes comme $|T| < \frac{|P|}{2}$.

En ce qui concerne les méthodes approchées, nous pouvons citer essentiellement :

- les algorithmes constructifs [2, 7, 46],
- les algorithmes de recherche locale : 2-échanges [12], recuit simulé [8, 13, 35, 45], tabou [50, 49],
- les algorithmes évolutifs comme les algorithmes génétiques [56, 42].

Ces algorithmes ont l'avantage d'obtenir des solutions quasi optimales dans des temps raisonnables. En revanche, les inconvénients majeurs sont d'une part une mise au point difficile des paramètres, pour ne pas dire «au pif», et d'autre part, l'obtention des solutions peu sembables (problème de stabilité) au cours des processus de recherche.

Le domaine du placement ne concerne pas que les tâches (processus). Beaucoup d'utilisateurs utilisent comme parallélisme une répartition de leurs données (matrices, domaines physiques, etc). Ici aussi, des heuristiques simples (glouton) peuvent être mises en œuvre comme une répartition cyclique de blocs ou plus simplement modulo. Elles peuvent se revéler inefficaces pour des données irrégulières. Des techniques de partitionnement de graphes peuvent alors être utilisées. Nous détaillons ci-dessous une des plus populaires : les «coupes récursives».

Le principe des coupes récursives consiste en séparant un espace contenant des données (par exemple, l'espace contenant des particules pour la simulation de leur dynamique, ou encore une matrice creuse) suivant l'une de ses dimensions. Cette première «coupe» est choisie afin d'obtenir des coûts équivalents pour les deux parties. Chacune des parties est alors découpée en fonction d'une autre dimension. Finalement les parties obtenues après une phase de découpes suivant toutes les dimensions considérées peuvent être re-découpées suivant la même méthode.

2.4.2 L'ordonnancement statique

Le problème ici est d'associer à chaque sommet du graphe de précédence (représentant un calcul donné, dont on connaît le coût) une date d'exécution. Ce domaine a sucité un nombre considérable de recherche, nous tentons d'en dresser les principales caractéristiques.

Du point de vue des paramètres importants venant de la représentation du progrmme, on distingue la structure du graphe (organisation des opérations, arbre, grilles,...), le temps des tâches (*UET*) et la valuation des arcs (taille des messages qui circulent d'une tâche à une autre). Coté architecture de machine, on distingue le nombre de processeurs (fixé ou non borné) et le type de communications (asynchrones ou non, qui peuvent être recouvertes ou non), etc.. De plus, une politique est caractérisée par des stratégies comme par exemple la possibilité de dupliquer une tâche ou non.

2.5 Le placement et l'ordonnancement dynamique

Les applications ici sont représentées par des graphes semi-prévisibles ou totalement imprévisibles.

Dans le cas où les graphes sont semi-prévisibles (machine dictionnaire, calcul formel), les temps de communications et de calculs peuvent être obtenus précisément à l'exécution. Un placement dynamique avec création dynamique d'objets sans préemption est alors suffisant. Les outils comme Athapascan [15, 17, 48], PM^2 [43], Parlist (pour le placement de données) [26], Landa [19] et PVM/MPI [31, 29] sont adaptés.

Lorsque les graphes sont imprévisibles (Branch-and-Bound, simulation de particules), les temps de communications et de calculs ne peuvent être obtenus à l'exécution avec précision. Les temps sont complétement dépendants des données en entrée, voire l'ordre dans lequel les données sont traitées. Un placement dynamqique avec préemption est alors nécessaire. Athapascan, PM^2, Landa et PVM/MPI sont des supports exécutifs nécessaires mais ils doivent être couplés à des algorithmes d'ordonnancement qui soient globaux (par exemple comme Cilk[10],Atha1[21]).

Pour d'éviter à l'exécution que certains processeurs soient dans l'état oisif alors qu'il reste du travail en attente, au moins deux strategies sont possibles. La première, le **partage de charge** (*load sharing*), consiste à répartir le travail de telle façon à ce que l'ensemble des processeurs soit actif à tout instant. La seconde, l'**équilibrage de charge** (*load balancing*), consiste non seulement à garder les processeurs actifs, mais aussi à répartir *équitablement* le travail entre les processeurs. L'idée sousjacente et intuitive, est de maximiser le taux d'utilisation des processeurs.

Néanmoins cette seconde stratégie n'est pas adéquate dans les systèmes d'exploitation. Elle réduit le temps de calcul global des applications, mais le temps de réponse des «petites» processus est fortement pénalisés. Or, des statistiques ont montré qu'environ 75% des processus nécessitent un temps de calcul inférieur à une seconde.

2.5.1 Composantes d'un algorithme de placement dynamique

Dans le processus de placement dynamique, nous pouvons généralement distinguer deux composantes principales.

1. Celle de l'information (estimation) est chargée d'une part de la collecte des informations locales à chaque processeur, d'autre part, du calcul de la charge globale ou partielle du système. La collecte est effectuée suivant un protocol d'échanges d'information.

2. Celle du contrôle (décision) est quant à elle responsable des stratégies de transfert et de localisation moyennant un protocol de placement. Ces stratégies sont prises en fonction des informations issues de la composante d'information.

2.5.2 La composante de contrôle

Il est à remarquer que certains algorithmes de placement dynamique ne possédent pas de composante d'information. Ils ne prennent donc pas en compte l'état courant du système. Ces algorithmes sont qualifiés d'« aveugles » (*blind*).

Nous pouvons citer deux exemples.

Algorithmes aléatoires. [23, 4, 24, 59] La composante de contrôle place un objet de manière aléatoire sur un processeur destinataire, choisi parmi tous les processeurs (méthode globale) ou un ensemble de processeurs « voisins » [9] (méthode locale).

Un autre critère de sélection des processeurs destinataires consisterait à avoir une probabilité proportionnelle à la puissance des processeurs.

Algorithmes cycliques. (PVM et Hélios [40]) Le choix des processeurs destinataires est fait suivant un ordre cyclique déterminé [58].

Algorithmes à base de listes. Ces algorithmes d'ordonnancement sont bien étudiés et permettent de calculer des ordonnancements de graphe de tâches « assez bon », *i.e.* pas trop éloigné de l'optimal : les solutions calculées sont jamais à plus de 2 de l'optimal [32, 54] dans le cas de processeurs identiques en négligeant le coût des communications).

- Algorithmes centralisés Une liste de tâches prêtes est partagée par l'ensemble des processeurs. Tant qu'il en reste, ceux-ci en choisissent une et l'exécute.

[9] Ce voisinage peut être logique ou physique.

Cet algorithme engendre une contention lors de l'accès à la liste : dans le cas d'un programme ayant un grain faible créant un grand nombre de tâches le surcoût d'accès devient important. Il est en revanche bien adapté aux programmes à gros grain [14].

La variante LPT *Largest Processing Time*, basée sur le classement de la liste en fonction du coût de chacune des tâches[10], permet d'obtenir des solutions qui sont au plus à 4/3 de l'optimal [32, 9] (tâches indépendantes, machine avec des processeurs identiques).

- Algorithme de *work-stealing*. L'algorithme de *work-stealing* (voleur de travail) est un algorithme distribué. Chaque processeur maintient une liste de travaux près à être exécutés. Tant qu'un processeur possède du travail, il en consomme. Dans le cas où celui-ci n'en possède plus, il choisi une victime et lui vole du travail.

  Une version de cet algorithme qui utilise le tirage aléatoire uniforme de la victime est montré efficace tant du point de vue théorique [11] qu'expérimental [10]. Des variations existent dans le choix du travail dans la liste.

L'avantage essentiel de ces algorithmes est leur simplicité de mise en œuvre et un surcoût de calcul faible. Ils ont montré leur efficacité essentiellement dans les systèmes massivement parallèles [24]. Cependant, ces algorithmes aveugles peuvent s'avérer inadéquats pour les applications irrégulières. En effet, l'absence de composante d'information occasionne, dans certains cas, de mauvais placements et peut ainsi provoquer des transferts coûteux par la suite [36].

2.5.3 La composante d'information

La première difficulté dans la réalisation de cette composante est l'indicateur de charge. Il dépend fortement des applications et des outils d'instrumentation offerts par les systèmes.

Il existe communément deux façons d'indiquer la charge d'un processeur. L'un consiste à donner une valeur précise, mais cette méthode pose un problème dans un système hétérogène où il est difficile de trouver une métrique unique. L'autre serait beaucoup plus simple et plus facile en indiquant seulement si un processeur est *faiblement*, *normalement* ou *fortement* chargé. Cette seconde solution, issue de la gestion des stocks et appelée stratégie à seuil, est fondée sur deux valeurs limites. L'une est le seuil de surcharge au delà duquel, un processeur est considéré comme fortement chargé et refuse du travail supplémentaire. L'autre est le seuil de souscharge en dessous duquel un processeur est considéré comme faiblement chargé

[10] Qu'il s'agit de connaître avant ordonnancement !

et peut accepter un supplément de travail. Entre les deux seuils, un processeur est dit dans un état de charge normal (figure 2.3).

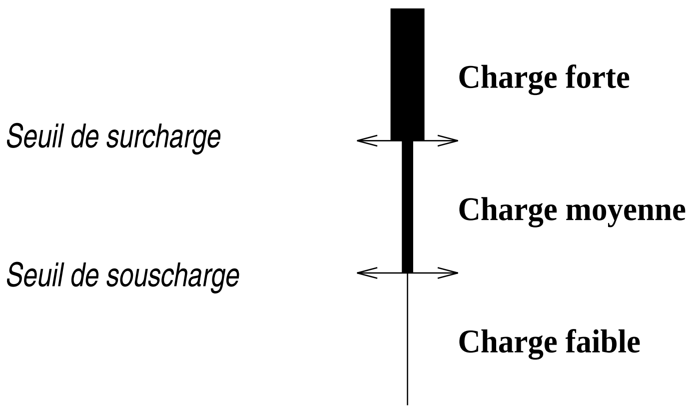


Figure 2.3 : Stratégie à seuil.

Un bon indicateur de charge doit satisfaire au moins les propriétés suivantes :

- refléter précisément la charge du système,
- prédire justement la charge dans un futur proche,
- rester stable et peu sensible aux fluctuations à court terme.

2.5.3.a Exemples d'indicateurs de charge du système

Ces indicateurs sont définis essentiellement pour mesurer la charge des systèmes. Mais on peut s'en inspirer au niveau applicatif.

La longueur des files d'attente CPU et d'entrée-sortie. Ferrari et Zhou [25] ont montré qu'il existe une forte corrélation entre cet indicateur et le temps de réponse des processus.

Seule, la longueur de la file CPU ou des files d'entrée-sortie, n'est pas suffisante puisqu'un processus peut effectuer beaucoup d'entrée-sortie et perdre énormément de temps CPU, et vice-versa.

Ils proposent de modéliser la chage d'un processeur par $T_p = \sum_q \alpha_{pq} Rq$, où α_{pq} est un coefficient dépendant du comportement du processus p utilisant la ressource q, et R_q est le nombre de processus attendant dans la file de ressource q.

Le nombre de messages en attente dans un processeur peut être utilisé pour mesurer la charge d'un futur proche [4].

L'âge des processus. En moyenne, un processus qui est actif depuis longtemps influence plus fortement la charge dans un futur proche qu'un processus récent.

Le taux d'occupation du CPU [44].

Le facteur d'«étirement» (*stretch factor*), proposé par [6], est le ratio entre le temps d'exécution d'un processus sur un processeur fortement chargé et le temps d'exécution du même processus sur un processeur faiblement chargé.

L'occupation mémoire a en général une faible influence jusqu'à ce qu'elle soit saturée. Cet indicateur est utile pour éviter des transferts ou migrations de processus vers de processeurs dont la mémoire est pleine. La charge est alors exprimée par $C = N + \frac{\beta}{1-M}$, où N est le nombre de processus en attente, M est le taux d'utilisation de la mémoire locale et β est une constante fixée à 0,01 par Lin et Keller [39].

D'autres caractéristiques internes aux processus dépendantes de l'application peuvent également être utilisées si elles sont disponibles [22, 51].

2.5.3.b Phase de collecte d'information

Cette phase a pour l'objectif de répondre à trois questions :

1. **Où** l'information doit être envoyée ?

2. **Qui** déclenche l'envoi de l'information ?

3. À **quelle fréquence** l'information doit être envoyée ?

Deux stratégies sont possibles pour répondre à la première question : **centralisée** ou **distribuée**.

La stratégie centralisée consiste à dédier un processeur du système à la collecte d'information (*all to one*). Cette stratégie a l'avantage d'être facile à implanter et nécessite un coût de gestion faible. Néanmoins, il existe un goulot d'étranglement au processeur central et donc peu extensible (*no scalable*), ni tolérant aux pannes même s'il est possible de doubler le processeur central afin de tolérer quelques pannes limitées.

La stratégie distribuée quant à elle se subdivise en deux. Dans la première, tous les processeurs peuvent envoyer leurs informations locales à tout autre (*all to all*). Alors que dans la seconde, chaque processeur envoie ses informations à ses voisins (*all to neighbours*). Les avantages sont la tolérance aux pannes et l'extensibilité. En revanche, ces stratégies sont très difficiles à implanter et gérer. L'extensibilité n'est pas non plus assurée si on considère que l'importance des coûts de communication (premier cas) et l'obsolescence des informations.

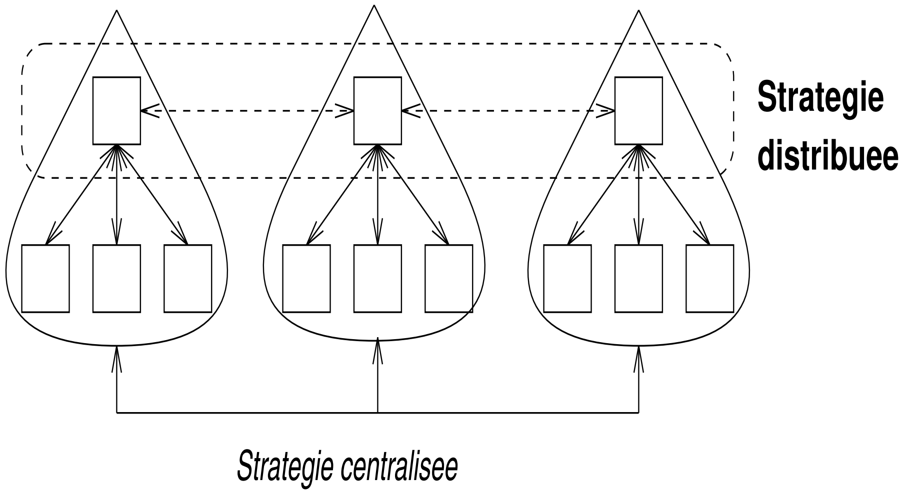


Figure 2.4 : Stratégie mixte hiérarchique.

Des stratégies mixtes (figure 2.4) et hiérarchique (*clustering*), peuvent être élaborées sur la base des deux précédentes stratégies.

En ce qui concerne la prise de décision d'envoi (question 2.), il existe également deux stratégies possibles : **volontariste** ou **sur demande**.

La stratégie volontariste consiste à expédier les informations locales suivant une décision locale. Alors que la stratégie sur demande n'envoie les informations que sur réception d'un message. L'avantage de cette seconde stratégie est qu'il n'y a pas d'information inutile. Par contre, le nombre de messages est doublé et l'état des informations peut être obsolète.

Une stratégie mixte peut être élaborée. Pour les processeurs chargés, il est préférable d'adopter la stratégie volontariste. Ainsi, ces processeurs déjà surchargés se mettent hors du processus de régulation de charge. Une stratégie sur demande dans ce cas aurait augmenté encore plus la quantité de travail de ces processeur. En revanche, les processeurs faiblement chargés peuvent adopter la stratégie sur demande pour augmenter efficacement la régulation de charge puisqu'il n'y a plus d'information circulant inutilement.

Par exemple, la stratégie de jeton circulant est justement une stratégie mixte. Elle est d'une part involontaire (sur demande d'autres processeurs qui envoient le jeton) puisqu'un processus doit attendre le jeton, et d'autre part volontaire, car chaque processeur peut garder aussi longtemps qu'il souhaite le jeton.

Enfin, la fréquence des envois d'information peut être **périodique** ou **apériodique**.

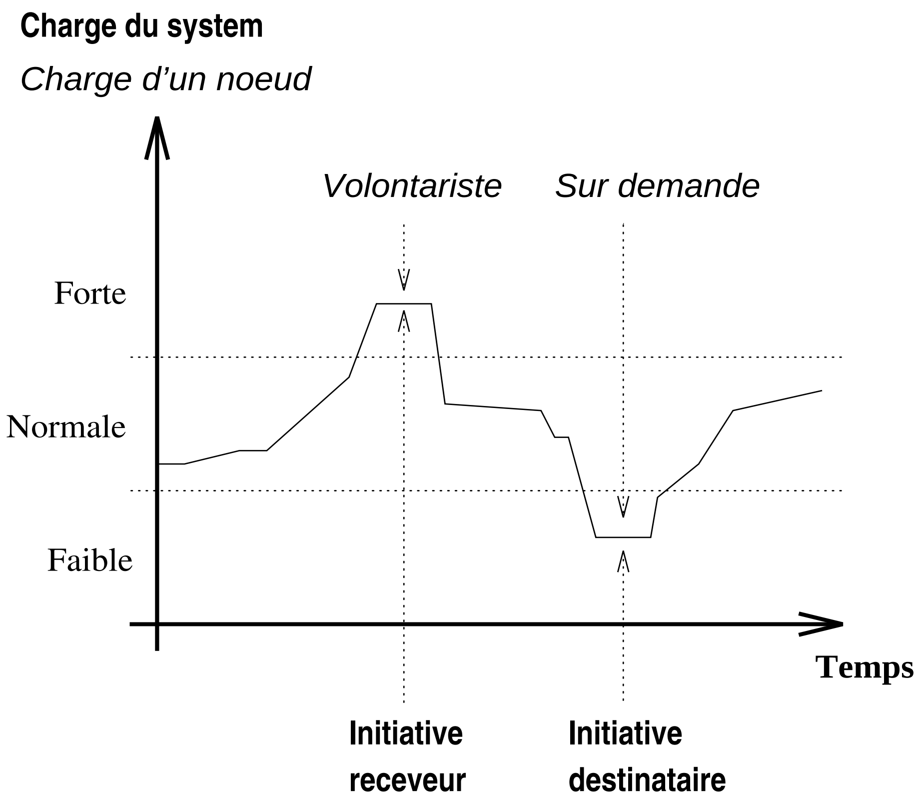


Figure 2.5 : Stratégie volontariste versus sur demande.

Dans le cas apériodique, un processeur effectue l'envoi chaque fois qu'il change d'état. Ou encore, uniquement lorsqu'il passe d'un état faiblement ou normalement chargé à un état fortement chargé (échange relatif [55]). Kuchen et Wagener [37] utilisent une graduation exponentielle pour les seuils de charge. Un processeur n'envoie son état courant uniquement si sa charge augmente de 2^{i-1} à 2^i, ou décroît de 2^i à 2^{i-1}. L'idée est que plus la charge est importante, moins le besoin est de connaître la charge précisément.

Zargham et Purcell [61] suggèrent d'intégrer les informations dans les messages des applications envoyés par les processeurs (échange continu [55]). L'avantage de cette méthode est qu'il ne fait pas intervenir de messages supplémentaires et c'est facile à intégrer dans le matériel. Par contre, cela complique les fonctions de communication du noyau système. De plus, les informations peuvent être obsolètes si les processus communiquent peu.

Dans les méthodes périodiques, la fréquences est un paramètre crucial puisque le nombre de messages engendré en dépend. Ce paramètre pour être efficace doit être réglé dynamiquement suivant la variation globale du système. Par exemple, il serait inutile d'envoyer des informations si l'état des processeurs restent inchangé. Il faut noter lorsque la fréquence est basse, les informations risquent d'être obsolètes et le système peu tolérant aux pannes.

2.5.3.c La composante de contrôle

La composante d'information, après avoir analysé l'état du système transmets ces informations à la composante de contrôle. L'analyse de cette composante consiste également à apporter des réponses à trois questions :

1. Faut-il effectuer un **transfert** ?
2. **Qui** prend la décision de transfert ?
3. **Où** transférer le travail ?

Pour la question de transfert, les stratégies employées sont identiques à celles de la composante d'informations, **centralisée** ou **distribuée**. La décision de transfert est fondée généralement sur des seuils de déclenchement déjà évoqués dans la composante d'information.

La décision de transfert peut être prise soit **par le destinataire** (*receiver initiative*) ou **par l'expéditeur** (*sender initiative*).

Lorsque l'initiative est prise par l'expéditeur, les processeurs fortement chargés cherchent ceux faiblement chargés pour y placer leurs charges. Cette stratégie est intéressante, car un travail peut être placé immédiatement après sa création, ce qui ne nécessite pas, *a priori*, un mécanisme coûteux de migration. L'inconvénient est que lorsque le système est fortement chargé, il devient difficile pour les expéditeurs de trouver d'éventuels receveurs.

En revanche, lorsque le receveur a l'initiative, il doit effectuer une requête avec la mise en place d'un mécanisme de migration généralement coûteux. De plus, lorsque le système est peu chargé, cette stratégie engendrerait trop de messages. Puisqu'il devient alors difficile pour les receveurs de trouver des expéditeurs qui ont encore du travail.

D'où, de nouveau l'idée de stratégie mixte, à l'intiative de l'expéditeur lorsque la charge du système est faible et à celle du receveur dans le cas contraire.

Néanmoins les seuils de déclenchement doivent être réglés dynamiquement si l'on veut éviter les phénomènes de *domino* (transitivité) ou de *boumerang* (réflexivité). Le transfert de travail sur un processeur faiblement chargé a lieu uniquement si le processeur passe ou reste dans un état normal après le transfert.

Les stratégies de localisation sont du type **premier sélectionné** (*first fit*) ou **meilleur sélectionné** (*best fit*) avec choix aléatoire en cas d'égalité. Une exécution locale est déclenchée si jamais la décision de transfert n'a pas eu lieu.

2.5.4 Outils pour la régulation de charge

Dans les sections précédentes, nous avons présenté les problèmes et certaines méthodes de régulation dynamique de charge. En pratique, les principaux problèmes posés par une mise en œuvre sont :

1. le *repliage* de l'exécution du programme sur un nombre fixé de processeurs : généralement les programmes parallèles sont conçus pour fonctionner avec un nombre non borné de processeurs (cas des programmes PRAM) ou utilisant un nombre de processeurs différents de la machine sur lequel il s'exécute ;
2. *le recouvrement* des communications par des calculs ;
3. *la création* d'un grand nombre de tâches : les programmes dits irréguliers génèrent dynamique un grand nombre de tâche pour s'adapter au parallélisme réel du problème ;
4. *la migration* des tâches : il peut s'avérer important de remettre en cause un placement afin de transfert une tâche sur un autre processeur ;
5. *la distribution le placement* des données sont fondamentaux afin d'obtenir de bonnes performances (choix du grain, (ré-)utilisation de la localité des données).

Différentes fonctionnalités sont disponibles pour répondre à ces problèmes. Un support exécutif pour le développement de mécanisme de régulation de charge devrait offrir les suivantes.

1. La multiprogrammation légère (utilisation de processus léger ou *thread*) est une technique qui permet de simuler des processeurs virtuels (repliage), de réduire les coûts de création d'activités concurrentes et qui offre un bon cadre de développement pour des techniques de recouvrement des communications par des calculs.
2. La continuation[11] d'une tâche sur un autre processeur un problème difficile qui, dans le cadre de processeurs homogènes, peut être résolu automatiquement mais partiellement par l'utilisation de technique de compilation (par exemple PM^2). Dans un cadre plus général, le programmeur doit fournir des fonctions de fermeture de l'environnement d'exécution d'une tâche et spécifier les points de reprise du calcul.
3. L'utilisation d'une mémoire (virtuellement) partagée facilite l'adressage des données entre différents processeurs. Si certains protocoles pour assurer la cohérence des données peuvent avoir un coût de gestion relativement important,

[11]Transfert du contrôle à partir d'un point (arbitraire) de l'exécution.

certains langages (Jade[12], Ath1) de programmation utilisent les informations d'accès aux données (écriture, lecture, accumulation...) et une sémantique particulière pour offrir une implantation plus efficace.

Enfin, les techniques de routage des données permettent de minimiser le coût dû aux communications, c'est le sujet de la présentation des sections suivantes.

2.6 Le routage

Comme nous l'avons vu dans les sections précédentes, la régulation et l'équilibrage de charge reposent souvent fortement sur une modification dynamique de la localisation des données, et sur un déplacement, parfois massif, de ces données d'un ensemble de processeurs à un autre. La forme de ce déplacement peut être spécifique (une permutation, une diffusion, etc.) ou arbitraire. L'objectif de cette section est d'introduire la terminologie relative à ce type de problèmes et de donner quelques indications sur la manière de les résoudre.

Dans un premier temps, nous considérerons le routage de communications arbitraires. Nous nous focaliserons ensuite sur quelques cas très particuliers, mais relativement fréquents, de communications. Pour plus de détails sur ce qui suit, nous renvoyons les lecteurs à [38], [20] et [52].

2.6.1 La problématique du routage

Les machines parallèles disposent maintenant, tout comme les réseaux locaux LAN ou mondiaux WAN, de composants matériels dédiés au routage. On appelle ces composants les *routeurs*. En général, chaque nœud du réseau (processeur, station de travail, etc.) dispose d'un routeur spécifique. L'objet de cette section est de décrire le fonctionnement des routeurs.

2.6.1.a Interface réseau et mode de commutation

L'*interface réseau* réalise le lien entre l'application résidente sur les nœuds d'un multi-processeur (par exemple) et le réseau. Elle peut se caractériser par la présence d'une file d'attente en entrée, de type FIFO, qui filtre l'envoie des messages dans le réseau (ceci est plus typique d'un réseau de stations de travail que d'une machine parallèle). L'interface calibre les messages devant être transmis. On distingue le *message* du *paquet*. Le message est l'entité que l'application souhaite transmettre. Il est souvent de taille arbitraire. L'interface réseau peut alors éventuellement découper le message en entités de taille bornée que l'on appelle paquets. L'avantage de traiter des paquets est que les ressources disponibles sur la route des paquets n'ont pas

[12] Voir le chapitre 9 page199 de ce livre pour une présentation.

à être arbitrairement grande. Le problème est que l'interface réseau doit alors être capable, en reception, de reconstruire un message à partir de plusieurs paquets.

Le *mode de commutation* indique la façon dont sont acheminés les messages ou les paquets de nœud en nœud, de la source vers la destination. On distingue plusieurs types principaux de commutation :

Commutation de paquets. Chaque routeur dispose d'un certain nombre de tampon mémoire, chaque tampon pouvant stocker un paquet. Lorsqu'un routeur reçoit un paquet, il le stocke dans un de ses tampons avant de le retransmettre à destination du routeur suivant sur la route du paquet. Si le routeur suivant ne dispose pas d'un tampon libre pour accueillir ce paquet, celui-ci reste stocké jusqu'à qu'une place se libère. Chaque paquet possède un entête dans lequel est stockée de l'information permettant au routeurs intermédiaires de calculer la route à suivre.

Commutation de circuits. Lors de l'émission d'un message, la source procède de la même façon que lorsque vous établissez une liaison téléphonique. Une requête est envoyée par la source jusqu'à la destination. Tout au long de la route entre la source et la destination, des liaisons internes aux routeurs sont établies afin de construire un circuit physique de bout-en-bout. Lorsque la destination reçoit la requête, elle renvoie un accusé de reception et le message est alors transmis directement de la source à la destination. Une fois la connexion établie, le coût d'une communication distante n'est guère différent de celui d'une communication voisin-à-voisin. Par contre, la connexion peut être difficile à établir en cas de surcharge du réseau (un nœud peut se retrouver en situation de famine) et un grand nombre de ressources sont requises pour une faible quantité d'information transmise.

Wormhole. Le terme *wormhole* provient de l'anglosaxon *worm* et *hole*, c'est-à-dire respectivement *vers* et *trou*. Ce mode de commutation consiste à découper le message ou le paquet en petite entitées appelées *flits* (pour *flow control digits*). Comme ce nom l'indique, les flits sont les plus petites entités sur lesquelles le réseau établi un contrôle de flot. A chaque lien physique monodirectionel de communication correspond une mémoire tampon susceptible de stocker un ou plusieurs flits. (Si plusieurs tampons sont associés à un même canal, on parle alors de canaux *virtuels*. Les canaux virtuels jouent un rôle important dans la prévention des inter-blocages, mais ce n'est pas l'objet de cette section de traiter de ces problèmes.) Les flits avancent à la queue-leu-leu dans le réseau. Le premier flit est un flit d'entête contenant l'information permettant au routeurs intermédiaires de calculer la route à suivre. Les autres flits ne contiennent que des données ; ils doivent donc impérativement se suivre. Le dernier flit libèrent les connexions établies par l'entête. Le nombre de ressources est donc proportionnel à la taille du message transmis (contrairement à la commutation de circuits).

L'apparition des techniques de transport basées sur l'optique met en évidence d'autres type de commutation. (Par "optique", nous entendons routeurs optiques, car l'utilisation de fibres optiques ne modifie guère les techniques de commutation si les routeurs restent électroniques.) A titre d'exemple, citons les *étoiles optiques passives* permettant de faire des connexions un-vers-plusieurs dans les architectures multi-connexion, ou la technique WDM (pour *Wavelength Division Multiplexing*) consistant à utiliser différentes longueurs d'ondes partageant un même canal. La technologie optique a donné lieu au *routage par déflexion*. C'est un routage par paquet sauf qu'il n'y a pas de mémoire tampon locale, outre celle nécessaire au stockage du message le temps de décoder son entête et de le router (ce peut être un simple boucle de fibre optique). Dès que le lien de sortie a été calculé à partir de l'entête, le message est renvoyé sur ce canal. Si le canal de sortie est déjà réservé pour une autre communication, le message est défléchi sur un autre canal, même si cela doit l'éloigner de sa destination finale. L'avantage de ce mode de routage est de ne nécessiter que peu de ressource locale. Si les composants sont optiques, il ne nécessite que la traduction de l'entête en électronique (les données sont retardées dans des boucles de fibres le temps de calculer la route). Par contre ce type de commutation ne permet souvent pas de borner *a priori* le temps d'acheminement d'un message, ce dernier pouvant être défléchi un nombre arbitrairement grand de fois.

2.6.1.b Fonction de routage

La manière dont est calculée la route des messages est indépendante du mode de commutation. Ce calcul est attribué à la *fonction de routage*. Nous allons introduire un cas très simple : une fonction de routage dans un réseau $G = (V, E)$ où V représente l'ensemble des routeurs, et E l'ensemble des liens inter-routeurs, est défini par

$$R : V \times V \mapsto \mathcal{P}(E).$$

Cette fonction doit s'interpréter de la façon suivante : lorsqu'un message à destination d'un nœud y arrive sur le routeur x, il doit être routé sur un des canaux de sortie $R(x, y)$. Si $|R(x, y)| = 1$ pour tout x et y, le routage est dit statique, sinon il est dit dynamique ou adaptatif. Un routage adaptatif permet de choisir localement la route la moins chargée. Son contrôle est cependant plus délicat. La fonction de routage est codée soit sous forme d'un algorithme (routage XY dans une grille par exemple), soit au moyen de tables de routage ($R(x, y) = T[x, y]$). Réduire la taille des fonctions de routage a conduit au développement de plusieurs techniques dont le routage par intervalle : pour tout routeur x et tout lien e incident à x, l'ensemble des destinations y telles que $T[x, y] = e$ est codé sous forme d'union d'intervalle.

En choisissant habilement la numérotation des sommets, on peut limiter le nombre d'intervalle et passer d'une complexité mémoire $O(n \log \Delta)$ pour les tables en $O(\Delta \log n)$ pour un réseau de n nœuds dont les nœuds sont connectés à au plus Δ voisins.

2.6.1.c Performances

Il existe plusieurs façon d'estimer les performances d'une fonction de routage. Certains de ces paramètres sont statiques. Par exemple :

- La *congestion* d'un lien (ou d'un nœud) est le nombre de routes passant au travers de ce lien (resp. de ce nœud). L'arête-congestion (resp. sommet-congestion) d'un routage est le maximum de la congestion de chaque lien (resp. nœud). Plus la congestion est faible, moins les chances sont grandes que des messages se retrouvent en conflit pour l'utilisation d'une ressource.

- La *dilatation* d'un routage est la longueur de la plus longue route suivie par un message de sa source à sa destination. (La longueur est exprimée en nombre de liens de communication traversés.) Plus la dilatation est faible, plus les distances sont courtes. Par contre, une dilatation faible peut entrainer une congestion élevée, et réciproquement.

Le *débit* du réseau en fonction de la charge offerte est un des paramètres dynamiques les plus utiles pour caractériser l'efficacité d'un réseau. La figure 2.6 représente la forme typique d'un tel débit : lorsque la charge offerte croît, le réseau débite en proportion ; puis, lorsque la chrage offerte atteint un certain seuil, le nombre de conflits augmente à l'intérieur du réseau et celui-ci ne passe plus la charge en proportion directe avec la demande ; si la charge offerte augmente encore, le réseau sature et ne peut plus débiter que son maximum. (Lorsqu'un réseau sature, les queues de l'interface réseau débordent.) On cherche alors à augmenter le seuil de saturation et à augmenter le débit maximum. Notons que certains modes de communication peuvent entraîner un effondrement du débit lorsque la charge offerte dépasse le seuil de saturation du fait de nombreuses collisions entre paquets à l'intérieur du réseau.

2.6.2 Problèmes spécifiques

Nous n'avons pour l'instant parlé que du routage un-vers-un, c'est-à-dire d'une source vers une destination. Les messages circulent alors dans le réseau sans qu'il soit possible d'organiser les communications autrement qu'au moyen d'une fonction de routage efficace. Il existe cependant des cas où le routage peut être considéré globalement. C'est en particulier le cas lorsque les processeurs procèdent à un rééquilibrage

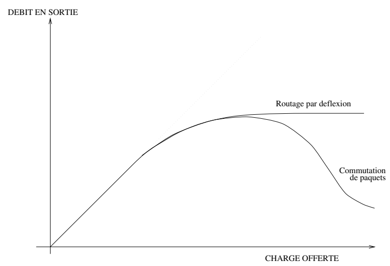


Figure 2.6 : Courbe typique du débit du réseau en fonction de la charge offerte en chaque nœud. Le débit croît proportionnellement à la charge jusqu'à saturation. Au delà du seuil de saturation, le débit peut rester constant ou s'effondrer.

de leur charge, ou lorsqu'ils simulent un réseau virtuel. En ce cas, on peut imaginer d'organiser les communications de façon à faciliter le travail de la fonction de routage.

On distingue deux façons de procéder. On parle de méthodes *précalculées* lorsque certains paramètres de routage nécessitent un calcul global, et on parle de méthodes *directes* lorsque tout peut être calculé au vol en cours de simulation. Ainsi, si, lors d'une simulation, l'exécution des mouvements de données repose sur le remplacement d'instructions du type `envoyer` $M_{x,y}$ `de` x `à` y par `envoyer` $M_{p(x),q(y)}$ `de` $p(x)$ `à` $q(y)$ où p et q sont des fonctions localement calculables par x (et y), pour tout x (et pour tout y), alors la simulation est directe. A l'inverse, si p et q nécessitent la prise en compte globale des informations distribuées au sein du réseau, alors la simulation doit être précalculée pour être efficace (car le calcul distribué de p et q peut prendre un temps considérable). Nous allons voir des exemples dans les paragraphes suivants.

Sans rentrer trop dans les détails (des chapitres de ce livre sont entièrement consacrés à certains de ces problèmes), nous allons considérer brièvement trois cas particuliers.

2.6.2.a L'émulation

Emuler un réseau consiste à simuler le comportement de ce réseau sur un autre. En général, on émule un algorithme ou une classe d'algorithme mise en œuvre sur un réseau particulier plutôt que le réseau lui-même. Par exemple, on peut simuler un algorithme *consécutif* d'un hypercube sur un réseau *mélange-parfait*.

Informellement, un hypercube se définit récursivement de la façon suivante : un 0-cube est un réseau réduit à un unique nœud ; un $(d+1)$-cube s'obtient à partir de deux copies d'un d-cube en reliant les sommets jumaux. Un 2-cube est donc le carré usuel, et le 3-cube est le cube usuel. Un hypercube de dimension d (ie. un d-cube) possède d dimensions, une par copie successive. Un algorithme *normal* sur un hypercube est un algorithme qui procède par étape, chaque étape impliquant des échanges dans une dimension unique. Un algorithme *consécutif* est un algorithme normal tel que les deux dimensions de deux étapes consécutives sont consécutives.

On peut vérifier qu'un hypercube peut être défini comme un ensemble de 2^d sommets numérotés par des chaînes binaires de longueur d telles que deux sommets sont adjacents si et seulement si leur deux chaînes diffèrent en exactement une position. Un mélange-parfait de dimension d possède également 2^d sommets numérotés par des chaines binaires de longueur d. Le sommet $x_1x_2\dots x_{d-1}x_d$ est connecté aux sommets $x_dx_1x_2\dots x_{d-1}$, $x_2\dots x_{d-1}x_dx_1$ et $x_1x_2\dots x_{d-1}\overline{x_d}$ où $\overline{x}$ désigne le complément à 1 de x. Le mélange-parfait n'a donc qu'au plus 3 connexions par sommet alors que l'hypercube en a un nombre proportionel à sa dimension.

En deux mots, voici comment on simule un algorithme consécutif sur mélange-parfait. Regardons comment simuler un échange dans la dimension i, c'est-à-dire un échange entre $x_1x_2\dots x_i\dots x_{d-1}x_d$ et $x_1x_2\dots\overline{x_i}\dots x_{d-1}x_d$. Pour cela, il suffit que le message de $x_1x_2\dots x_i\dots x_{d-1}x_d$ parvienne successivement par rotations droites à $x_dx_1x_2\dots x_i\dots x_{d-1}$, $x_{d-1}x_dx_1x_2\dots x_i\dots x_{d-2}$ jusqu'en $x_{i+1}\dots x_dx_1x_2\dots x_i$ et, de là, en $x_{i+1}\dots x_dx_1x_2\dots\overline{x_i}$ pour ensuite être acheminé en $x_1x_2\dots\overline{x_i}\dots x_{d-1}x_d$ par rotations gauches. Une telle émulation prendrait un temps $O(d)$, mais en utilisant le fait que l'algorithme est consécutif, on peut ne pas bouger les données de $x_{i+1}\dots x_dx_1x_2\dots x_i$ car une seule rotation sera necessaire pour simuler une communication dans la dimension $i-1$ ou $i+1$. La simulation prendra ainsi un temps $O(1)$. Il s'agit ici d'une simulation directe.

2.6.2.b Le routage de permutations

Le routage de permutation est un cas typique de réordonnancement des données au sein d'un multi-processeur, les données situées sur le processeur i devant aller au processeur $\sigma(i)$ où $\sigma \in \Gamma_n$ est une permutation quelconque. Plutôt que de lister les méthodes connues pour router des permutations dans des réseaux classiques, il est préférable de profiter de ce thème pour faire deux remarques.

D'une part, lorsque l'on route un schéma spécifique telle qu'une permutation, les notions de congestion et dilatation du réseau ne sont pas forcément une bonne mesure de la façon dont s'exécutera la communication. Il est parfois ainsi préferable de parler de congestion *relative*, c'est-à-dire pour laquelle on ne prend pas toutes les routes en compte, mais seulement celles induites par le schéma considéré.

D'autre part, pour un ensemble de routes données, on peut *ordonnancer* les communications afin de limiter encore la congestion. Prenons l'exemple de n routes, par exemple les n routes liées au routage d'une permutation dans un réseau à n sommets, et supposons un mode de commutation par paquets. Soit d la longueur maximum des routes, et soit c la congestion de ces routes, i.e. le nombre maximum de routes passant par un même lien de communication. Si l'on exécute en direct le routage, des conflits pour l'accès à certains liens vont se produire. Il y aura au plus c conflits par lien, et la longueur des routes est au plus d. Le routage s'effectuera donc en au plus $c.d$ étapes de façon directe. Leighton, Maggs et Rao ont montré qu'on pouvait router en $O(c+d)$ étapes. Cependant ce routage est précalculé. L'ordonnancement des messages repose sur l'insertion de délais au départ et tout au long du chemin suivi par un message. Ces délais sont ajustés globalement et dépendent de l'ensemble des chemins.

2.6.2.c Le *Multicast*

Le problème du multicast consiste à résoudre le routage un-vers-plusieurs plutôt qu'un-vers-un. Ce type de routage pose des problèmes spécifiques (y compris des problèmes d'inter-blocage). Le multicast a connu un succès important ces dernières années en particulier grâce à la possibilité matériel de copie-transmission : il est maintenant possible pour un message de transiter à travers un nœud tout en étant recopié dans la mémoire du processeur et ce même dans un mode de commutation type wormhole. Les degré de liberté à optimiser sont nombreux : découpage de l'ensemble des destinations en sous-ensembles auxquels on fera parvenir le message de façon indépendante, ordonnancement des destinations au sein d'un même sous-ensemble pour spécifier l'ordre dans lequel seront visitées les destinations successives, etc.

2.7 Conclusions

Dans un soucis de clarifier une démarche globale nous avons présenté un schéma générique pour une gestion efficace des ressources lors du processus de parallélisation, jusqu'à l'exécution proprement dite du programme. Les solutions existantes (souvent partielles) que nous avons présentées et discutées sont prometteuses, mais il reste encore de très nombreux problèmes scientifiques à résoudre.

Les détails des différentes parties de cette présentation se retrouvent dans plusieurs autres chapitres de ce livre, que ce soit en aval à propos des supports d'exécution, en amont pour les aspects programmation parallèle, ou bien encore pour la description certains algorithmes en-lignes d'ordonnancement dynamique.

Bibliographie

[1] A. Aggarwal, K. Chandra, and M. Snir. Communication complexity of pram's. *Theoretical Computer Science*, 71:2–28, 1990.

[2] S. Antonelli, F. Baiardi, S. Pelagatti, and M. Vanneschi. Communication cost and process mapping in massively parallel systems: A static approach. Technical report, Università degli studi de Pisa, Dipartimento de Informatica, 1989.

[3] R. Asenjo, G. Trabado, M. Ujaldon, and E. Zapata. Compilation issues for irregular problems. In *Proc. of ESPPE'96, Parallel Programming Environments for High Performance Computing*, pages 187–199, April 1996.

[4] W. C. Athas. Fine grain concurrent computations. Master's thesis, California Institute of California, Pasadena, California, May 1987. 5242:TR:87.

[5] E. Bampis, J.-C. König, and D. Trystram. Optimal parallel execution of complete binary trees and grids into most popular interconnection networks. In *PARLE'94, Athens Greece*, volume 817 of *LNCS*, 1994.

[6] S. A. Banawan. An evaluation of load sharing in local distributed system. Master's thesis, University of Washington, Seattle, USA, Aug. 1987.

[7] D. Benhamamouch and G. Plateau. Task allocation in distributed computing systems. Technical report, LIPN, Univerité Paris-Nord Villetaneuse, 1989.

[8] F. Berman. Experience with an automatic solution to the mapping problem. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, 1989.

[9] J. Blazewicz, K. Ecker, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1993.

[10] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95), Santa Barbara, California*, pages 207–216, Jul 1995.

[11] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[12] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 1981.

[13] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *International Conference on Parallel Processing*, 1988.

[14] J. Briat, T. Gautier, and J.-L. Roch. Application irrégulière et ordonnancement en ligne. In G. Bernard, J. C. de Kergommeaux, B. Folliot, and C. Roucairol, editors, *Placement dynamique et répartition de charge: application aux systèmes répartis et parallèles*, pages 81–106. 1996.

[15] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Europar'97 Conference*, pages 590–5999, Passau, Germany, 1997. Springer Verlag.

[16] P. Chrétienne, J. Edward G. Coffman, J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley & Sons Ltd., 1995.

[17] M. Christaller. Athapascan-0b for pvm : Adding threads to pvm. In *Proceedings of EuroPVM User's Group*, Oct. 1994.

[18] R. Cole and O. Zajicek. The apram: Incorporating asynchrony into the pram model. In *Proc. of STOC*, 1989.

[19] DELTA PARTENERS S.A., Licence LAAS-CNRS, `landa@delta.fr`. *LANDA : Local Area Network for Distributed Algorithms*, 1997. Manuel d'Utilisation Version 2.5.0.

[20] F. Desprez and P. Fraigniaud. Les bibliothèques de communications. *Ordinateurs et Calculs Parallèles*, (19):293–312, 1997.

[21] M. Doreille, G. Cavalheiro, and J.-L. Roch. Régulation dynamique en athapascan: Exemple d'un tri parallèle probabiliste optimal. In *8ème Septièmes Rencontres Francophones du Parallélisme (RenPar'8), Bordeaux, France*, pages 181–184, May 1996.

[22] S. Dowaji and C. Roucairol. Load balancing strategy and priority of tasks in distributed environments. In *Fourteenth Annual IEEE Conference - International Phoenix Conference on Computers and Communications*, pages 15–22, Scottsdale, Arizona (USA), Mar. 1995.

[23] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptative load sharing in homogeneous distributed systems. *IEEE Transactions on SoftWare Engineering*, 12(5):662–675, May 1986.

[24] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, Department of Mathematics and Computer Science, Aug. 1993.

[25] D. Ferrari and S. Zhou. A load for index dynamic load balancing. In *Fall Joint Computer Conference*, pages 684–690, Dallas, Texas, Nov. 1986.

[26] F. Feschet, S. Miguet, and L. Perroton. *Parallélisme et Applications Irrégulières*, chapter ParList : Une Structure de Données Parallèle pour l'Équilibrage des Charges, pages 177–201. Hermès, 1995.

[27] B. Folliot, J. Chassin de Kergommeaux, and C. Roucairol, editors. *Journées de Recherche sur le Placement Dynamique et la Répartition de Charge : Application aux Systèmes Répartis et Parallèles*. Université Pierre et Marie Curie - Paris, May 1995.

[28] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. *ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[29] M.-P. I. Forum. *MPI-2 : Extensions to the Message-Passing Interface*. University of Tennessee, Knoxville, July 1997. `http://www.mcs.anl.gov/Projects/mpi/index.html`.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[31] A. Geist, A. Beguelin, and J. Dongarra et al. *PVM : Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[32] R. Graham. Bounds for certain multiprocessor anomalies. *Bell System Tech J.*, 45:1563–1581, 1966.

[33] Groupe PRC-GDR PRS CAPA. *Algorithmes Parallèles : Analyse et Conception*. Hermès, 1994.

[34] Groupe PRC-GDR PRS CAPA. *Parallélisme et Applications Irrégulières*. Hermès, 1995.

[35] P. Haden and F. Berman. A comparative study of mapping algorithms for an automated parallel programming environment. Technical report, UC San Diego, 1988.

[36] F. Hémérv. *Étude de la Répartition Dynamique d'Activités sur Architectures Décentralisées*. Thèse de doctorat d'université, Université de Lille 1, July 1994.

[37] H. Kuchen and A. Wagener. Comparison of dynamic load balancing strategies. In K. Boyanov, editor, *Parallel and Distributed Processing*, pages 303–314, Amsterdam, The Netherlands, Mar. 1991. North Holland.

[38] F. Leighton. *Introduction to Parallel Algorithms: Arrays, Trees, Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1991.

[39] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, Jan. 1987.

[40] P. S. LTD. *The Helios Parallel Operating System*. Prentice Hall, 1991.

[41] D. A. Menascé, D. Saha, S. C. D. S. Porto, V. A. F. Almeidia, and S. K. Tripathi. Static and dynamic processsor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, (28):1–18, 1995.

[42] H. Muehlenbein, M. Gorges-Schleuter, and O. Kraemer. New solutions to the mapping problem of parallel systems : the evolution approach. *Parallel Computing*, 1987.

[43] R. Namyst and J.-F. Méhaut. Pm^2 : Parallel multithreaded machine. a computing environment for distributed architectures. In *Proceedings of ParCo'95*, Sept. 1995.

[44] L. M. Ni, C. W. Xu, and T. B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, 11(10):1153–1161, Oct. 1985.

[45] J.-L. Pazat. *Outils pour la Programmation d'un multiprocesseur à mémoire distribuée*. Thèse de doctorat d'université, Université de Bordeaux I, 1989.

[46] F. Pellegrini. *Apllications de Méthodes de Partition à la Résolution de Problèmes de Graphes Issus du Parallèlisme*. Thèse de doctorat d'université, numéro 1244, Université de Bordeaux I, 1995.

[47] G.-R. Perrin and A. Darte, editors. *The Data parallel programming model: foundations, HPF realization, and scientific applications*, volume 1132 of *LNCS*. Springer, Feb. 1996.

[48] B. Plateau. Apache : Algorithmique parallèle et partage de charge. Technical report, LMC-IMAG, 1994.

[49] S. C. S. Porto and C. C. Ribeiro. Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints. *Journal of Heuristics*, (1):207–223, 1995.

[50] S. C. S. Porto and C. C. Ribeiro. A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. *International Journal of High Speed Computing*, 7(1):45–71, 1995.

[51] P.-G. Raverdy. *Gestion de Ressources et Répartition de Charge dans les Systèmes Hétérogènes à Grande Échelle : Application aux Environnements Mobiles et Parallèles*. Thèse de doctorat d'université, Université Paris VI, June 1996.

[52] J. D. Rumeur. *Communication dans les réseaux de processeurs*. Collection Etudes et Recherches en Informatique. Masson, 1994.

[53] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, 1989.

[54] D. Shmoys, J. Wein, and P. Williamson. Scheduling parallel machines on-line. *SIAM, J. Comput.*, 24(6):1313–1331, 1995.

[55] E.-G. Talbi. Allocation dynamique de processus dans les systèmes distribués et parallèles. Technical report, LIFL-Université de Lille 1, 1995.

[56] E.-G. Talbi and P. Bessière. Un algorithme génétique massivement parallèle pour le problème de partitionnement de graphes. Technical report, IMAG, 1991.

[57] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, 1990.

[58] Y. Wang and R. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, 34(3), Mar. 1985.

[59] J.-C. Weill. *Programmes d'Échecs de Championnat : Architecture Logicielle. Synthèse de Fonctions d'Évaluation, Parallélisme de Recherche*. Thèse de doctorat d'université, Université Paris 8, Jan. 1995.

[60] T. Yang and A. Gerasoulis. A parallel programming tool for scheduling on distributed memory multiprocessors. In *Proc. of SHPCC'92*, pages 350–357, 1992.

[61] M. R. Zargham and R. D. Purcell. A protocol for load balancing on csma networks. In *5th International Conference on Parallel Processing*, pages 163–165, Aug. 1985.

Chapitre 3

Langages, Outils et Systèmes pour la construction des applications réparties

Michel Riveill (INRIA–SIRAC)

Chapitre 4

Fonctionnalités des bibliothèques de communication

P. Guyaux, T. Monteil, J.M. Garcia (LAAS-CNRS)

4.1 Introduction

Nous assistons ces dernières années à quelques évolutions marquantes du parallélisme. D'un côté, les machines parallèles s'orientent vers une architecture MIMD. Leurs noeuds sont similaires à des stations de travail, c'est à dire qu'on y retrouve les mêmes processeurs standards, avec de la mémoire locale, un processeur de communication et parfois un disque (Cray T3E, IBM SP2, ...). Ces noeuds sont indépendants et interconnectés par un ou plusieurs réseaux de communication rapides (switch, crossbar, bus, ...). De l'autre côté, les réseaux de stations, jusque là dédiés au travail personnel avec partage de fichiers, sont de plus en plus performants et intéressent fortement la communauté du parallélisme. Cet engouement est principalement dû à la puissance des processeurs et à la mise en place de liens de communication rapides comme ATM, fast-Ethernet ou Myrinet [2]. Le réseau de stations ayant une architecture de type MIMD, on peut, par le biais d'un environnement logiciel, l'assimiler à une machine parallèle virtuelle. L'intérêt d'une telle machine virtuelle est double. D'une part, cela permet d'utiliser des ressources qui sont le plus souvent inoccupées [13] et d'accéder à une machine parallèle à moindre frais. D'autre part, vu que ces environnements d'exécution sont portés sur plusieurs architectures parallèles, les programmes mis au point sur un réseau de stations peuvent être aisement utilisés sur une machine parallèle. Ces environnements d'exécution s'appuient sur le concept de communication par échange de messages (*Message Passing*) qui est l'objet de ce cours. Nous étudierons tout d'abord quelques éléments de programmation pour la communication par échange de messages. Puis, nous verrons quelques environnements d'exécution comme PVM [8], MPI [12] et LANDA [11].

4.2 Quelques éléments de programmation pour le message passing

4.2.1 Un modèle de programmation

Les 2 modèles de programmation les plus connus sont le *partage de données* (LINDA [5], Phosphorus [6]) et le *passage de messages*. Le premier, beaucoup plus naturel, est typiquement associé à une architecture multiprocesseurs à mémoire commune. Le deuxième, moins évident à programmer, est généralement utilisé sur des machines à mémoire physiquement distribuée. Sur ce type d'architecture les tâches d'une application parallèle sont des processus lourds et leurs données sont privées; c'est une tendance générale de longue date bien que l'orientation actuelle favorise l'utilisation de processus légers. La coopération passe donc nécessairement par des échanges de messages (explicites ou non). L'application peut être vue comme des séquences de code s'exécutant indépendamment les unes des autres en parallèle et s'échangeant des données via des messages.

Le schéma général d'une application parallèle sur une machine à passage de messages est le plus souvent décomposé en 2 parties : une tâche de contrôle et des tâches de calcul. Les tâches de calcul sont chargées d'explorer un type de solution ou de travailler sur une partie des données. L'activité d'une telle tâche peut se résumer à une boucle comportant 3 phases : réception de données, calcul sur les données, renvoi du résultat. La tâche principale, qui peut aussi être une tâche de calcul, contrôle l'évolution de l'algorithme et détermine son aboutissement; lorsque le but recherché est atteint, elle stoppe tous les processus ayant participé à son élaboration. Chaque tâche participant à l'application parallèle commence par se faire reconnaître du système et termine en se déconnectant de celui-ci. Ce schéma de programmation (figure (4.1)) n'est pas unique, mais il est simple et souvent utilisé.

4.2.2 Les éléments d'un environnement d'exécution

Un environnement d'exécution parallèle est constitué d'un ensemble de logiciels pour l'exécution d'applications parallèles. Le premier élément est le système de communication; il est composé de la bibliothèque de communication par échange de messages et de son noyau de communication. La bibliothèque propose à l'utilisateur un ensemble de fonctions pour émettre et recevoir des messages. Le noyau, généralement invisible à l'utilisateur, réalise la machine virtuelle; il est généralement constitué de

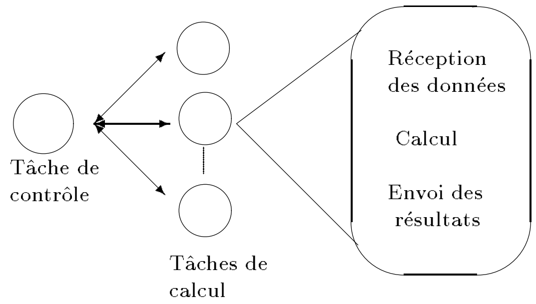


Figure 4.1 : Schéma de programmation

démons s'exécutant sur les stations. Il fournit un système de nommage permettant d'identifier les tâches d'une application de manière unique, et rend transparent à l'utilisateur la répartition géographique des sites sur lesquels s'exécutent ses tâches. Le deuxième élément assure un contrôle de l'application, de son lancement à sa terminaison, c'est le noyau d'exécution. Les derniers éléments sont des outils de traces, de mesures de performances, de deboggage ou bien de suivi (*monitoring*) de l'application.

4.2.3 La communication par échanges de messages

4.2.3.a Le message

Un message est un ensemble de données associé à une entête. Les données peuvent être structurées ou non, c'est à dire être composées d'éléments, ou paquets, de types différents. L'entête est l'enveloppe ajoutée par le système de communication pour acheminer le message de l'émetteur vers le destinataire. Elle contient au minimum l'identificateur de l'émetteur et du destinataire ainsi qu'une étiquette utilisateur. Le message est typé afin que le système puisse coder correctement les données lors des transferts sur un réseau hétérogène. L'étiquette permet à l'utilisateur de sélectionner un message particulier dans sa boîte aux lettres plutôt que de les lire dans l'ordre d'arrivée.

4.2.3.b Principes de base du système de communication

Les bibliothèques de communication utilisées pour le passage de message s'appuient en majorité sur le concept de boîtes aux lettres. C'est à dire qu'il y a par principe un stockage temporaire du message dans une zone tampon. La communication est donc par nature asynchrone. Elle se fait par le biais de 2 primitives du type *send* et *receive*. Les variantes de ces primitives font référence à la synchronisation avec les notions de communication bloquante ou non-bloquante.

Le contrôle de flux est assuré de façon transparente par le système de communication, cela veux dire qu'il gère autant de tampons que le nécessite l'application jusqu'à la limite imposée par le système opératoire. Les messages sont stockés par ordre d'arrivée sous le mode FIFO (First In First Out). Il n'y a donc pas de déséquencement pour les messages issus d'une même tâche. Par contre, le système ne garantie pas l'ordre des messages émis par des tâches différentes. En d'autres termes, un destinataire recevra les messages d'une même tâche dans l'ordre où ils ont été émis, et les messages de tâches différentes dans un ordre chronologiquement non garanti.

Dans les systèmes de communication par échange de messages s'appuyant sur le concept de boîte aux lettres, on entend par "communication" l'opération acheminant le message de l'émetteur à la boîte aux lettres du destinataire (envoi) ou de la zone de stockage au destinataire (réception). Il y a 2 copies, de l'espace mémoire de l'émetteur à celui de la boîte aux lettres du destinataire, et de la boîte aux lettres à l'espace mémoire du destinataire. La communication ne demande pas à priori que les protagonistes soient simultanément dans un état d'émission pour l'un et de réception pour l'autre.

4.2.3.c La communication point à point

La communication non-bloquante fait appel à la bufferisation du message. Dans le cas du *send*, le système de communication mémorise les données jusqu'à ce que le destinataire effectue une lecture. La tâche émettrice reprend la main sans savoir si le message a été ou sera consommé. Pour le *receive*, le système récupère le message précédemment reçu ou retourne une erreur s'il n'est pas présent. Il est à noter que lorsque le système de communication n'assure pas un stockage des données (bufferisation) ou ne dispose que de zones trop petites, l'envoi échoue si le destinataire n'est pas à l'écoute.

La communication bloquante met en relation directe (via le système de boîtes aux lettres) 2 tâches qui exécutent simultanément l'une la primitive *send* et l'autre la primitive *receive*. On ne sort de la fonction bloquante que lorsque la communication est réalisée, on est alors sûr que le message a été émis et consommé. L'envoi bloquant synchronise 2 processus (synchronisation de l'émetteur sur le récepteur), l'émetteur étant libéré lorsque le message est consommé par le destinataire (lecture bloquante ou non-bloquante). La lecture bloquante ne synchronise pas 2 processus car le message peut être dèjà présent dans la boîte aux lettres (figure 4.2).

Une troisième variante insiste sur le recouvrement calcul/communication, il s'agit

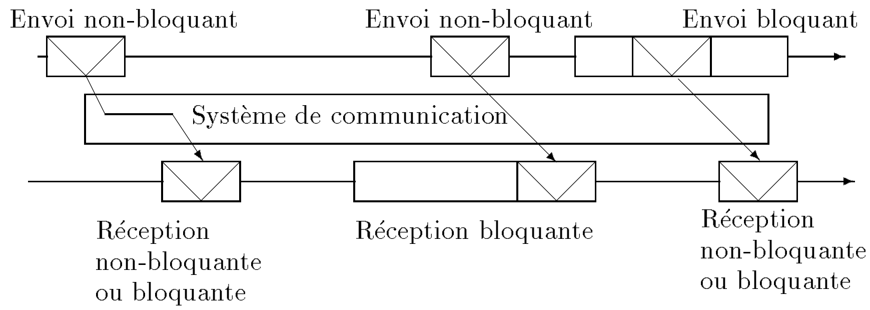


Figure 4.2 : Quelques types de communication point à point

de la communication en arrière-plan. La tâche demande alors au système de communication d'effectuer l'opération (*send* ou *receive*) en tâche de fond pendant qu'elle fait autre chose. Cette variante est accompagnée de 2 primitives supplémentaires; l'une teste l'état de la requête (en cours ou terminée) et l'autre bloque la tâche en attente jusqu'à la terminaison de la requête.

Les decriptions des communications bloquantes, non-bloquantes et en arrière-plan données précédemment peuvent ne pas correspondre exactement à certaines définitions trouvées dans la littérature. En effet, certains considèrent la communication comme l'opération acheminant le message de l'espace de travail d'un émetteur à celui d'un destinataire; on sous-entend ici que le message est consommé, la communication est synchrone.

4.2.3.d Communication globale et notion de groupe

La communication globale met en relation un ensemble de tâches (ou groupe). Les principales opérations sont la diffusion (totale ou partielle / *broadcast-Multicast*) et le regroupement (*gathering*). La diffusion transmet le même message aux destinataires du groupe. Certaines bibliothèques ajoutent aussi la possibilité d'envoyer à chaque élément du groupe une partie du message (diffusion personnalisée / *scatter*). Le regroupement consiste à reconstituer une donnée à partir des différents messages reçus des membres du groupe. En règle générale il s'accompagne d'une opération simple effectuée sur les messages: la concaténation, la somme, le minimum, ou autre.

A la communication globale est associée la notion de groupe. Le groupe permet de rassembler sous un même identificateur un ensemble de tâches ayant les mêmes caractéristiques aux yeux de l'utilisateur. Il est pratique pour effectuer des communications implicites , mais aussi pour réaliser des barrières de synchronisation. La définition du groupe et son implémentation diffèrent d'une bibliothèque à l'autre en fonction de sa constitution, de son rôle ou du niveau où il est géré. Il peut être local à la tâche ou global à l'application; il peut être statique (défini une fois pour toute) ou dynamique (on intègre le groupe ou en sort).

4.3 Un aperçu de différents environnements d'exécution

4.3.1 PVM (Parallel Virtual Machine)

PVM est un logiciel développé au Oak Ridge National Laboratory. Il est distribué gratuitement (sauf quelques versions propriétaires) et donc largement utilisé de par le monde. Sa bibliothèque de communication est interfacée pour des applications écrites en C ou en Fortran, et le logiciel est disponible sur un grand nombre de plateformes (SUN, RS6000, HP_UX, SP2, Cray, etc). Un certain nombre d'outils, développés par ailleur, gravitent autour de PVM; on peut citer entre autres HENCE pour la spécification et le contrôle d'applications, XAB pour la génération de traces et XPVM pour l'analyse des traces.

4.3.1.a Machine virtuelle, Application PVM et Modèle d'exécution

La machine virtuelle est réalisée par la présence d'un démon PVM sur chaque processeur (station) utilisé par l'application. La mise en place d'un démon est effectuée soit par le biais de la console PVM (programme interactif de contrôle de la machine virtuelle), soit directement par programmation (*pvm_addhosts*). Un démon PVM, au moins, doit exister sur une machine pour exécuter la tâche initiale, les autres processus pouvant être lancés dynamiquement en cours d'exécution de l'application (*pvm_spawn*). PVM tient compte de l'hétérogénéité du réseau. Les exécutables des tâches doivent être placés sous un répertoire utilisateur spécifique à PVM. Le chemin de l'exécutable utilisé est calculé par PVM à partir de ce répertoire et de l'architecture.

Une application PVM est constituée d'un ensemble de tâches (exécutables compilés avec la bibliothèque PVM) et de démons chargés de réaliser la connexion entre chaque station utilisée par l'application parallèle. Chaque tâche de l'application est un processus Unix (processus lourd) s'exécutant sur un des sites de la machine virtuelle. PVM ne contrôle pas la terminaison des processus, c'est à l'utilisateur d'implémenter un protocole de terminaison en fonction de son application. Chaque tâche est identifiée de manière unique par PVM qui lui associe un entier. Pour communiquer, les tâches doivent récupérer l'identificateur PVM de leurs correspondants.

La figure 4.3 représente un modèle d'exécution possible.

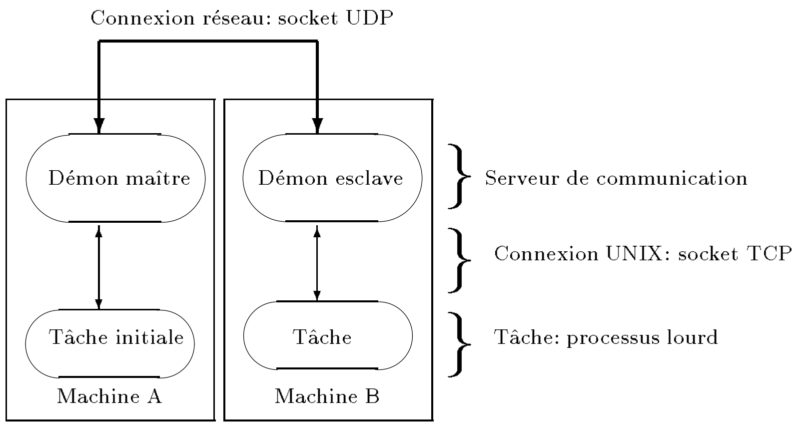


Figure 4.3 : modèle de fonctionnement

4.3.1.b La communication

Pour la communication PVM utilise des tampons de stockage (buffer). Un buffer est fourni par défaut (*pvm_initsend*), mais on peut en créer d'autres (*pvm_mkbuf* - *pvm_setsbuf* - *pvm_setrbuf*). L'échange des données se passe toujours en deux phases. Il faut dans un premier temps constituer le message. Pour cela, PVM fournit un ensemble de fonctions (*pvm_pk* ...) pour remplir la zone tampon d'émission (tampon PVM) avec les données à émettre; c'est *l'empaquetage*. Les types de données manipulés par les tampons sont des types simples (entiers, réels, caractères, ...). Ce principe permet d'émettre des messages contenant des données de différents types (message structuré). Le deuxième temps consiste à l'émission proprement dite du message soit à une tâche (*pvm_send*), soit à un ensemble de tâches (*pvm_mcast*). Une estampille utilisateur (tag) est alors associée au message en même temps que l'entête (identificateur de l'émetteur et du destinataire). Le message est transmis au démon local de la tâche émettrice, puis du démon local au démon distant, et enfin du démon distant à la tâche destinatrice (figure 4.3).

Une tâche peut vérifier si un message particulier est présent dans sa boîte aux lettres (*pvm_probe*). La réception du message (*pvm_recv, pvm_nrecv, ...*) exécute les mêmes opérations que l'émission mais en ordre inverse. La sélection est faite en fonction de l'émetteur et du tag en parcourant la boîte aux lettres dans le sens FIFO; ces critères (émetteur, tag) peuvent être non significatifs. L'opération de lecture récupère en fait les données (le contenu du message) dans le tampon positionné en réception. Il ne reste plus au lecteur qu'à dépaqueter le message (*pvm_unpk* ...), si possible de la même manière qu'à l'empaquetage. PVM offre les modes de lecture bloquant (*pvm_recv*) et non-bloquant (*pvm_nrecv*) avec en plus la possibilité de spécifier un temps d'attente maximum (*pvm_trecv*).

Il faut noter que PVM ne fait pas de contrôle (ni de sélection) sur les types des données ou sur leurs tailles. On peut très bien envoyer des entiers et lire des réels ou des caractères. L'utilisateur a donc en charge le contrôle sur la validité des opérations effectuées à l'empaquetage et au dépaquetage des messages. Le codage des données pour la communication entre machines hétérogènes est spécifié à la création du tampon d'émission; c'est donc à l'utilisateur de savoir à l'avance l'architecture des machines qu'il utilise. Un autre point est l'interbloquage entre plusieurs processus excutant une lecture bloquante; PVM ne fait aucune détection, c'est à l'utilisateur de bien vérifier son algorithme et les interbloquages possibles.

4.3.1.c Le groupe

La gestion des groupes dans PVM est dynamique, c'est à dire que le nombre de groupes et le nombre de tâches rattachées à un groupe peut varier au cours de l'exécution de l'application. Les groupes sont gérés de manière centrale par un démon PVM spécialisé. C'est lui qui contrôle la validité des groupes et la liste des participants aux groupes. Un groupe est identifié par un nom donné par l'utilisateur (une chaîne de caractères). Une tâche rejoint (*pvm_joingroup*) ou quitte (*pvm_lvgroup*) un groupe quand elle le désire en cours d'exécution. Le groupe est utile à deux niveaux: la communication globale (*pvm_bcast, pvm_reduce*) et la synchronisation à travers les barrières (*pvm_barrier*).

Le caractère dynamique du groupe est un avantage mais aussi un problème majeur. En effet, c'est à l'utilisateur de bien connaître la composition du groupe et de vérifier que les opérations qu'il effectue s'appliquent bien aux membres du groupe au moment de l'exécution de la fonction, sinon il risque de placer son application dans un état bloquant.

4.3.1.d La création dynamique de tâches

La création dynamique de tâches est une fonctionnalité indispensable pour configurer dynamiquement son application à l'architecture de la machine ou à la complexité du problème à traiter. PVM offre un fonction intéressante (*pvm_spawn*) car elle permet d'effectuer un placement automatique des tâches sur la machine virtuelle. Le placement actuel utilise un algorithme de type round-robin (circulaire), mais plus tard il tiendra compte de la charge des processeurs. Le choix peut être restreint à un sous ensemble de stations ayant la même architecture.

4.3.1.e Le contrôle de l'application

PVM fournit des fonctions (*pvm_tasks, pvm_mytid, pvm_parent, ...*) pour obtenir des informations sur les tâches de l'application comme l'identificateur de la tâche, l'identificateur du père de la tâche, l'identificateur du démon associé à une tâche.

PVM fournit deux types de fonctions pour la signalisation. La première transmet un signal Unix à un des processus de l'application (*pvm_sendsig*); l'utilisateur a en charge l'installation d'une procedure d'interception sur le signal. La deuxième interrompt la tâche sur un événement particulier (notification) comme l'arrêt d'une machine ou d'un processus (*pvm_notify*). Cette dernière possibilité peut être intéressante par exemple pour prévenir les tâches esclaves que l'application se termine (mort de la tâche maître). L'utilisateur peut agir sur son application (manuellement ou par programmation) en émettant des signaux Unix (*pvm_sendsig*), en créant ou en stoppant des processus (*pvm_exit, pvm_kill, pvm_spawn*).

4.3.2 MPI (Message Passing Interface)

MPI [12] [7] est avant tout une spécification des fonctionnalités d'une bibliothèque à passage de messages, son but est de standardiser l'écriture des programmes utilisant ce principe pour plus de portabilité. Ce standard défini la syntaxe et la sémantique d'une bibliothèque de communication par passage de messages. MPI est développé par le MPI Forum qui regroupe près de 80 personnes provenant d'une quarantaine d'organisations comme des constructeurs d'ordinateurs, des chercheurs universitaires de tous pays, des laboratoires ou encore des industriels. Il existe de nombreuses implementations de MPI au-dessus de bibliothèques de communication diverses; on peut citer entre-autres LAM [3] construit au-dessus du Trollius Operating System (Ohio Supercomputer Center), CHIMP [1] (University of Edinburgh), MPICH [9] construit au-dessus du Chameleon System avec P4 [4] et PVM [8] (Argone National Laboratory) ou encore LANDA_MPI [11] construit sur le système LANDA.

4.3.2.a La notion de communicateur

Toute communication MPI fait référence à un communicateur. Le communicateur est un objet associant un groupe de tâches et un contexte. Le groupe est une collection ordonnée de tâches; il identifie chaque tâche par son rang dans le groupe (de 0 à N). Les groupes sont gérés dynamiquement. Le contexte permet de réaliser une partition de l'espace des communications; c'est une donnée système. Il assure que les messages seront différenciés. Deux tâches ne peuvent communiquer qu'à travers le même communicateur.

Il existe deux types de communicateurs: les intra-communicateurs et les inter-communicateurs. Les premiers rassemblent un groupe et un contexte définissant un espace où la communication peut être point à point ou collective. Les seconds sont constitués de deux groupes et un contexte permettant uniquement des communications point à point. L'intra-communicateur rassemble en fait dans un même espace de communication un ensemble de tâches collaborant (par échanges de messages) à la résolution d'un même travail. L'inter-communicateur rassemble deux groupes dans un même espace de communication pour que l'un fournisse du travail et l'autre des résultats.

Deux communicateurs sont créés automatiquement par MPI à l'initialisation, *MPI_COMM_WORLD* et *MPI_COMM_SELF*, tous les autres sont dérivés de ceux-ci soit à partir d'un sous-groupe des tâches (*MPI_Comm_split, MPI_Group_xxx, MPI_Comm_create*), soit par duplication (*MPI_Comm_dup*).

4.3.2.b La communication

MPI offre une des gammes de fonctions pour la communication parmi les plus importantes pour les bibliothèques de passage de messages. Ceci vient du fait que MPI se présente comme un standard pour la programmation parallèle en insistant sur la portabilité. MPI autorise la communication de messages typés associés à une étiquette (tag). Une des particularités de MPI est la création dynamique de types complexes pour les données des messages (*MPI_Type_struct*) avec la possibilité de préciser des alignements. Le message est composé d'une entête et d'un tableau dont les éléments peuvent être de type complexe. La prise en compte d'architectures hétérogènes est assurée par la bibliothèque de communication quand les messages sont typés.

MPI propose une large gamme de modes de communication qui définissent en fait le comportement du système: standard, synchrone, bufferisé ou prêt. Les fonctions d'émission et de réception bloquante, non-bloquante ou en arrière plan ne fonctionnent pas de la même manière selon le mode de communication choisi par l'utilisateur à la création du communicateur. Chaque message possède une enveloppe qui contient les informations suivantes: source, destination, tag, communicateur. Des fonctions permettent d'avoir des informations sur les caractéristiques des messages reçus (*MPI_Probe*).

La bibliothèque MPI est particulièrement riche en ce qui concerne la communication collective. Elle s'appuie sur le communicateur. On peut citer: la barrière de synchronisation (*MPI_Barrier*), le broadcast d'un membre vers tous les autres (*MPI_Bcast*), le regroupement de tous les membres vers un (*MPI_Gather*), la diffusion personnalisée (*MPI_Scatter*), l'opérateur global sur tous les membres (*MPI_Reduce*), le tous vers tous (*MPI_Alltoall*), etc.

4.3.2.c La notion d'architecture logique

MPI offre la possibilité de définir une topologie virtuelle et de nommer ses processus en conséquence. C'est une facilité offerte à l'utilisateur. MPI n'introduit pas de vérification sur l'utilisation des liens de communication dans cette topologie virtuelle; c'est à l'utilisateur de l'utiliser correctement. Des fonctions de description à l'aide d'un graphe de voisinnage (*MPI_Graph_create*) ou à l'aide d'un pavage régulier (*MPI_Cart_create*) sont fournies.

4.3.2.d Les apports de MPI-2

La norme MPI-2 corrige certains défauts de MPI-1 et ajoute de nouvelles fonctionnalités. Elle permet de mieux contrôler l'application parallèle. La création dynamique de tâches est possible (*MPI_Spawn*). Un contrôle par l'utilisation des signaux a été ajouté. On peut donc émettre un signal (*MPI_Signal*) et capter un signal provenant d'un autre processus MPI. Un processus peut aussi observer un autre processus et connaitre son état (*MPI_Monitor*).

Au niveau des commmunications, MPI-2 apporte la création de communications entre des processus indépendants; c'est à dire n'appartenant pas à la même application. C'est une méthode de communication inspirée du modèle client/serveur. Un nouveau protocole permet la lecture et l'écriture dans des mémoires distantes (*MPI_Put, MPI_Get*). Les communications collectives peuvent être non bloquantes; la notion de "callback" a été introduit pour cela. Les fonctions de manipulation des communicateurs ont été complétées par des fonctions de comparaison et d'identification.

L'aspect temps réel est pris en compte. Il y a une gestion des interblocages par l'ajout de temporisateurs. De même, le temps maximum passé dans une fonction MPI est garanti; ceci se fait en utilisant des temporisateurs. La gestion de priorité par exemple dans les communications est aussi possible.

Enfin, la gestion des entrées/sorties est faite au niveau des fichiers. MPI offre les fonctions nécessaires à la gestion consistante de fichiers dans le cas de leurs manipulations par plusieurs processus simultanément.

4.3.3 LANDA (Local Area for Distributed Applications)

LANDA est développé depuis 1989 au LAAS-CNRS (Laboratoire d'Automatique et d'Architecture des Systèmes) par le groupe OFP (Optimisation, Filtrage et Parallélisme). C'est un environnement d'exécution parallèle complet intégrant une bibliothèque de communication, des outils graphiques et un système de supervision et de mesure de la machine virtuelle. Sa bibliothèque est interfacée pour le C, le fortran et le C++.

4.3.3.a La machine virtuelle

Une des originalités de LANDA est de prendre en compte la machine virtuelle de façon à ce qu'elle soit une entité connue, mieux maîtrisée et mieux utilisée par les applications parallèles. La machine est définie de manière statique; on connait chaque station la composant et ses caratéristiques. Sur chaque station s'exécute deux démons formant l'ossature de la machine virtuelle: le *serveur de tâches* et le démon *Network-Analyser* (DNA). Le serveur de tâches sert de relais aux applications LANDA pour l'exécution de processus à distance et pour la signalisation. Il s'exécute avec des droits *root*. Le DNA [10] mesure et prédit la charge du processeur, il constitue avec ses pairs le système Network-Analyser. Ce système a pour but de surveiller l'état global de la machine et d'utiliser au mieux et de façon cohérente les ressources. Il est organisé hiérarchiquement sur 3 niveaux. Le premier (au plus bas) constitue le système de mesure et de prédiction. Le deuxième regroupe en grappe certaines stations (division par brin réseau, par type ou par secteur ...). Le troisième, représenté par un seul démon (le superviseur), offre des services pour le placement de tâches. L'ensemble est tolérant aux pannes (réélection du superviseur et des responsables de grappe).

4.3.3.b Application LANDA et modèle d'exécution

L'utilisateur lance une application par la commande "*landa -f config_file*", ce qui a pour résultat d'exécuter un serveur d'application LANDA. Ce serveur charge un fichier de configuration écrit par l'utilisateur et décrivant son application. Dans ce fichier on retrouve la déclaration des tâches sous la forme du triplet "*identificateur utilisateur, nom de l'exécutable, machine cible*". Le serveur d'application a en charge le lancement, le contrôle et la terminaison d'une application (figure 4.4). Les tâches sont des processus lourds compilés avec la bibliothèque LANDA. Il est à noter que les tâches sont identifiées par un entier donné par l'utilisateur et non pas par le système. L'application comporte au minimum une tâche initiale, les autres tâches pouvant être créées dynamiquement (*Ltcreate, Ltdup*).

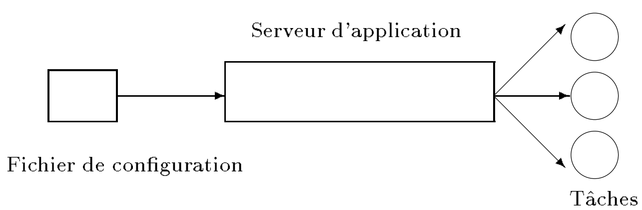


Figure 4.4 : Lancement d'une application

Une application LANDA est constituée de différents modules: le serveur d'application, le système de boîtes aux lettres et les tâches de l'utilisateur. Dans sa version courante, LANDA centralise la gestion des boîtes aux lettres de l'application en un même point (une des stations de la machine virtuelle) ce qui est intéressant pour les communications globales et pour détecter les situations d'interblocage, mais ceci présente un goulôt d'étranglement pour les applications synchrones à grain fin.

Le modèle d'exécution de la version courante peut être représenté comme sur la figure 4.5

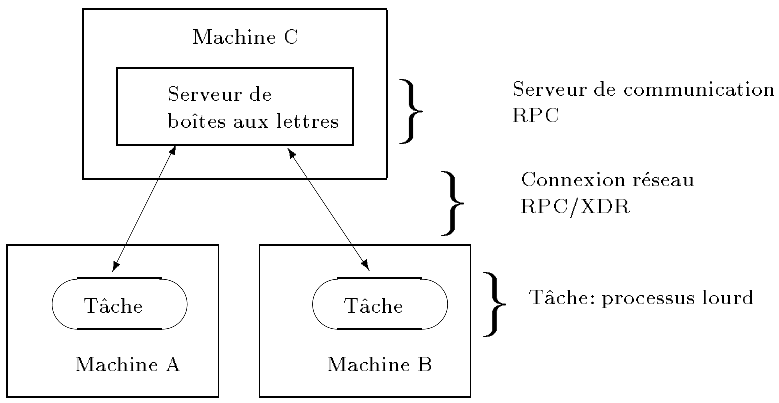


Figure 4.5 : Modèle d'exécution

4.3.3.c La communication

Le système de communication de LANDA véhicule des messages dont les données sont de type simple (un tableau), il n'y a pas de type complexe comme des structures de données bien que le noyau de communication l'autorise (pour les bibliothèques de compatibilité PVM et MPI). L'entête, qui caractérise le message, contient en plus des informations habituelles (émetteur, destinataire et tag) le type des données transmises et la taille du tableau (nombre d'éléments). Lors de la réception, le message lu est sélectionné en fonction de tous ces critères, en particulier le type et la taille; on ne peut donc pas lire un message contenant des entiers en spécifiant des réels (à moins de positionner le type non significatif). L'envoi (*Lsend*) est effectué en une seule opération, il transmet à la boîte aux lettres de la tâche destinatrice le message (données + entête).

LANDA offre plusieurs types de fonctions pour connaître le contenu de sa boîte aux lettres (*Lminfo, Lreadh*) sans lire le contenu des messages (les données), ou être averti de l'arrivée de certains (*Lring*). La réception se fait sur 2 modes: bloquant (*Lreadb*) ou non-bloquant (*Lread*). Une autre particularité de LANDA est de permettre la lecture bloquante sur un ensemble de messages (*Lreadw, Lwait*), ceux-ci étant consommés au fur et à mesure de leur arrivée. Le codage des données, lors des communications sur un réseau de stations hétérogènes, est assuré de façon transparente par LANDA.

4.3.3.d Le groupe

Le groupe est une donnée locale à chaque tâche, il n'est ni connu ni partagé avec les autres tâches de l'application. Le groupe est en fait une manière simple de désigner un ensemble de tâches pour la communication globale (*Lsendg* - broadcast ou multicast). Un groupe existe par défaut, c'est le groupe "ALL" qui désigne toutes les tâches en activités dans l'application.

Le groupe est géré dynamiquement par la tâche. Elle peut les créer, les détruire ou les modifier dynamiquement (*Lgedit*).

4.3.3.e La signalisation

LANDA fournit 2 types de fonctions pour interrompre une tâche en émettant soit des signaux Unix (*Lskill*), soit des signaux LANDA (*Lsendsignal*). Une fonction particulière (*Lcatchsignal*) est donnée pour attacher une procédure à un signal LANDA. La bibliothèque de communication LANDA n'étant pas réentrante, il faut éviter de recevoir ou d'émettre un message dans les interruptions.

4.3.3.f La création dynamique de tâches

Comme PVM, LANDA offre cette possibilité (*Ltcreate, Ltdup*), mais l'utilisateur a en plus les services du système Network-Analyser (de façon transparente) pour placer judicieusement ses tâches sur la station qui offre et offrira la ressource la mieux adaptée afin d'obtenir un meilleur rendement de l'application.

4.3.3.g Le contrôle de l'application

Les tâches peuvent connaître l'état de l'application à tout moment (*Ltinfo*). Cela permet, entre autres, d'adapter le comportement général en fonction du nombre de tâches ou de leur état. L'utilisateur agit sur son application par programme (*Ltcreate, Lskill, Lsendsignal, Lterminate*) ou par le biais de l'interface (*abort*). L'interface graphique peut se connecter ou se déconnecter d'une application à tout moment. Elle visualise l'état instantanné des tâches et affiche les liens de communication en cours ainsi que le contenu des boîtes aux lettres (sans les données). L'utilisateur a la possibilité d'arrêter son application en cliquant sur le bouton "*abort*".

4.3.3.h Les développements en cours

Les évolutions que nous mettons en place dans LANDA sont de plusieurs ordres. Tout d'abord au niveau du système Network-Analyser qui donne des mesures sur le trafic réseau afin de tenir compte d'architectures d'interconnexion complexes. Ensuite au niveau de la bibliothèque centrale qui est l'API du noyau de communication pour les bibliothèques de haut niveau comme LANDA, PVM ou MPI. Et enfin au niveau du système de communication lui-même qui doit utiliser les médias les mieux adaptés en fonction de la localisation des tâches.

Le système Network-Analyser mesure le trafic réseau via un ou plusieurs démons spécialisés placés aux endroits stratégiques. Ces données permettent de connaître le trafic général et celui dû à LANDA (ou à d'autres) ainsi que les liens de communication les plus sollicités. A partir de ces données ainsi que de la charge des stations (instantannée, moyenne et prédite) nous allons pouvoir indiquer une localisation pour les données des messages (la station émettrice, la station réceptrice ou une troisième station) afin d'assurer que les applications parallèles bénéficient du maximum de ressources possibles.

La bibliothèque LANDA est en fait conçue en 2 parties. La première est la bibliothèque utilisateur qui offre des fonctionnalités de haut niveau pour les applications parallèles. La deuxième est la bibliothèque centrale qui est en fait l'API du noyau de communication. Les bibliothèques de compatibilité PVM et MPI sont construites sur cette API. La bibliothèque centrale offre toutes les fonctionnalités de base pour la communication par échanges de messages comme la création d'une boîte aux lettres, l'émission de messages structurés, la réception bloquante, non-bloquante ou en arrière plan, etc. En plus de ces services nous voulons mettre en place des fonctions réalisant une mémoire commune distribuée afin de donner la possibilité d'écrire et de lire directement dans l'espace mémoire qu'une tâche veut partager.

Le système de communication de la version courante implémente une gestion centralisée des boîtes aux lettres. Ce principe est efficace pour des applications asynchrones de forte granularité. Il favorise la communication de groupe (broadcast, multicast) et permet de détecter les interbloquages. Cependant, il est moins performant pour des applications à grain fin car il devient un goulôt d'étranglement. De plus, ce système est inadapté lorsque des tâches s'exécutant sur la même machine (station multiprocesseurs) communiquent en passant par le réseau. Le système que nous mettons en place distribue totalement les boîtes aux lettres directement dans l'espace mémoire des tâches. Cette implémentation favorise la communication directe. Elle limite les accès réseau pour la communication point à point, mais les augmente pour la communication globale. Un des avantages est de pouvoir utiliser le média le mieux adapté en fonction de la localisation des tâches, en particulier lorsqu'elles s'exécutent sur la même station (mémoire commune ou autres).

4.4 conclusion

PVM a été sans nul doute l'environnement de programmation parallèle le plus utilisé et reste de nos jours largement employé. Il est disponible sur de nombreuses plate-formes. Son inconvénient majeur est qu'il est constitué d'outils disparates développés par la communauté du parallélisme, sans qu'il n'existe vraiment de coopération entre ces différents outils. De plus, la bibliothèque offerte par PVM peut être restreinte sur certaines machines favorisant ainsi l'émergence de bibliothèques propriétaires.

L'arrivée de la norme MPI qui est de plus en plus utilisée, a pour but d'offrir une totale portabilité sur une large gamme d'architecture de machines parallèles. Les constructeurs ont fait de réels efforts d'implémentation sur leurs machines. Ceci rend en général la bibliothèque MPI trés efficace. Malheureusement, il manque parfois des outils pour mettre au point ses programmes. Une solution est alors d'utiliser des environnements de programmation parallèle sur des réseaux de stations de travail et de PC comme LANDA-MPI. Ces derniers offrent la norme MPI et des outils d'aide au développement. A terme, les environnements de programmation parallèle offriront certainement tous une bibliothèque normalisée MPI. Toutefois, il est nécessaire de continuer à développer des bibliothèques spécifiques dont certaines fonctionnalités pourront être reprises par MPI.

L'arrivée des réseaux haut débit bon marché dans le monde des réseaux locaux va permettre d'obtenir des machines parallèles efficaces à faible prix. Elles seront constituées de stations ou de PC, d'un ou plusieurs réseaux haut débit et d'un environnement de programmation parallèle efficace et convivial. L'intérêt de ces grappes de processeurs est multiple. D'une part, leur souplesse de mise en place et de configuration en font un support de recherche idéal pour le parallélisme. Cette souplesse doit s'appuyer sur une séparation des couches de communication basse (accès aux médias, routage, etc), moyenne (communication réseau, zone tampon, multicast, etc) et haute (bibliothèques parallèles, scatter, gather, reduce, etc). D'autre part, la rapidité des réseaux va permettre d'implémenter efficacement des fonctionnalités pour la mise en place de mémoire commune distribuée.

Bibliographie

[1] R. Alasdair, A. Bruce, J. Mills, and A. Smith. Chimp/mpi user guide. *Technical report EPCC-KTP-CHIMP-V2-USER 1.2, Edinburgh Parallel Computing centre*, june 1994.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet – a gigabit per second local area network. *IEEE micro*, february 1995.

[3] G. Burns, R. Daoud, and J. Vaigl. Lam: an open cluster environment for mpi. *Proceedings of supercomputing symposium '94*, pages 379–386, 1994.

[4] R. Butler and E. Lusk. Monitors, messages and clusters: P4 parallel programming system. *Parallel Computing*, (20):547–564, 1994.

[5] A. Corbel and F. Fleter. Linda: un modèle de programmation parallèle. *Revue calculateur parallèle, Hermès*, 2(7):159–172, 1995.

[6] I. Demeure. Phosphorus. *http://www.inf.enst.fr/demeure/phosphorus/-phosphorus.html*.

[7] M. P. I. Forum. Mpi-2: Extensions to the message-passing interface. july 1996.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderman. Pvm: Parallel virtual machine, a users's guide and tutorial for networked parallel computing. *The MIT Press Cambridge*, 1994.

[9] W. Gropp and E. Lusk. User's guide for mpich, a portable implementation of mpi. *Technical report ANL-96/6, Argonne National Laboratory*, 1994.

[10] T. Monteil. Etude de nouvelles approches pour les communications, l'observation et le placement de tâches dans l'environnement de programmation parallèle landa. *these*, LAAS(96471), novembre 1996.

[11] T. Monteil, J. Garcia, and P. Guyaux. Landa: une machine parallèle virtuelle. *Revue Calculateur Parallèle, Hermès*, 7(2):119–137, 1995.

[12] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. Mpi the complete reference. *The MIT press*, 1996.

[13] M. Theimer and K. Lantz. Finding idle machines in a workstations-based distributed system. *IEEE Transactions on Software Engineering*, 15(11), november 1989.

Deuxième partie

Ordonnancement et Régulation de charge

Chapitre 5

Machines virtuelles et techniques d'ordonnancement

Jean-Claude Konig (LAMI) – Jean-Louis Roch (LMC-IMAG)

De manière générale, que ce soit dans un cadre dynamique ou statique, l'ordonnancement d'un programme parallèle prend en entrée un graphe de tâches et un modèle de machine et fournit en sortie un ordonnancement à un facteur borné (aussi proche de 1 que possible) de l'optimal. Dans le cadre dynamique (ou à la volée), le graphe est inconnu car construit au cours de l'exécution; la durée d'une tâche ou d'une communication n'est connue qu'après sa terminaison. Dans le cadre statique, le graphe est supposé connu avant l'exécution du programme. Dans cet article, nous introduisons les différents modèles de machines considérés et les techniques d'ordonnancement qui leur sont associées dans les cadres statique et dynamique.

5.1 Introduction

Le placement des tâches sur l'architecture cible dans le but de minimiser la durée d'exécution est un problème complexe qui intègre un grand nombre de paramètres qui seront étudiés dans cette école et qui sont liés à des choix algorithmiques (routage par exemple) ou des contraintes physiques (mode de communication, bande passante par exemple).

Le placement et l'ordonnancement des tâches doivent donc être décomposés en plusieurs étapes pour limiter à la fois la complexité et augmenter la portabilité et la généricité. La démarche classique est donc de placer et d'ordonnancer les tâches sur une machine virtuelle et d'émuler ensuite la machine virtuelle par la machine cible. Or il semble difficile (impossible) de concevoir une machine virtuelle simple et universelle qui soit acceptable (comme l'est la machine de von Neumann pour le calcul séquentiel) c'est-à-dire qui puisse servir de référence pour mesurer la complexité. Une raison clé de cet échec est l'impossibilité de faire abstraction des transferts de données (de coûts non uniformes de part la hiérarchie mémoire) pour obtenir des résultats de complexité ayant un sens expérimental.

Ainsi les modèles (de type PRAM synchrone [30] ou asynchrone [11, 4]) permettent d'analyser le coût d'un algorithme à partir de paramètres caractéristiques des graphes associés à ses exécutions : nombre d'opérations (les tâches sont séquentielles), volume d'accès en mémoire partagée (les arêtes de dépendance de données correspondent potentiellement à des accès distants), ... Mais pour que les résultats ainsi obtenus puissent être exploitables et validés expérimentalement, il est nécessaire de montrer comment l'ordonnancement du graphe (indépendant du nombre de processeurs) peut être réalisé. Il s'agit d'une part de montrer qu'un ordonnancement optimal du programme apporte l'efficacité et d'autre part d'exhiber un algorithme permettant de calculer un ordonnancement optimal ou à un facteur de l'optimal (α-compétitif). Ces deux études sont complémentaires; ce chapitre est surtout centré sur la présentation d'algorithmes α-compétitifs. Dans ce cadre, un ordonnancement optimal pour un programme donné peut éventuellement être séquentiel.

L'ordonnancement trivial sur une machine PRAM théorique (principe de Brent) ne peut fournir une émulation raisonnable sur une machine, même simple, car il repose sur des hypothèses fausses comme l'indépendance entre les temps de latence pour accéder à la mémoire partagée et la possibilité de répartir équitablement sans surcoût de temps l'ensemble des instructions d'une étape sur les processeurs.
Une façon de rendre le modèle plus fiable est de montrer la possibilité d'ordonnancer des classes de graphes spécifiques sur des modèles de machine plus réalistes, comme :

- granularité globale : le nombre d'accès potentiels en mémoire distante est négligeable devant le volume de calcul;
- granularité grossière : toutes les tâches sont de même durée et ont chacune un volume d'accès distants négligeable devant leur nombre d'opérations;
- topologie particulière : le graphe de dépendance est un arbre dont les tâches sont de même coût.

De tels graphes sont suffisament génériques pour être effectivement rencontrés dans des applications irrégulières : les classes précédentes englobent en effet de nombreux cas où par exemple durées de tâches et volumes d'accès à des données distantes sont inconnus ou variables, ou encore le graphe est inconnu et construit au cours de l'exécution par la création de nouvelles tâches.

Nous nous proposons dans ce chapitre de présenter les résultats connus pour l'ordonnancement dynamique et statique de tels graphes sur un modèle simpliste et d'étudier leur évolution lorsqu'on intègre au modèle certains paramètres caractérisant l'architecture cible comme :

- le nombre de processeurs
- la latence liée aux communications
- l'hétérogénéité des processeurs

Il est clair qu'un modèle intégrant seulement ces trois paramètres pour être à peu prêt valide nécessite de nombreuses hypothèses du type : seule l'application considérée s'exécute sur le système, la bande passante est suffisante pour ne pas entrainer une saturation des débits ... Malgré celà un tel modèle devrait permettre de mieux analyser le parallélisme exploitable d'une application et de donner un ordonnancement plus efficace à l'exécution.

Organisation. Après la définition d'un problème d'ordonnancement, la première section précise les mécanismes systèmes nécessaires à sa mise en œuvre ainsi que les critères permettant d'évaluer ses performances. Puis, les algorithmes classiques d'ordonnancement lorsque les processeurs sont supposés identiques et les communications non prises en compte, sont rappelés. L'analyse de leurs performances permet d'introduire les techniques de base utilisées pour l'évaluation théorique d'un algorithme d'ordonnancement. Dans les sections suivantes, les principaux algorithmes permettant de prendre en compte les communications sont présentés : ordonnancements dynamiques (section 3) et ordonnancements statiques (section 4). Enfin, les principaux résultats concernant l'ordonnancement sur des machines hétérogènes sont présentés.

5.2 Le problème d'ordonnancement

Dans cette partie, nous définissons le formalisme de graphe le plus général pris en entrée d'un algorithme d'ordonnancement. L'exécution d'un programme parallèle décrit (de manière dynamique dans le cadre le plus général) un graphe de dépendance. Ce graphe est orienté sans cycle : les nœuds correspondent à des blocs d'instructions (tâches) et les arêtes à des relations de dépendance sur l'accès à une donnée (transitions).

5.2.1 Caractérisation du problème : graphe de dépendance

De manière générale, l'exécution d'un programme parallèle peut-être décrite par un graphe de flots de données (macro data-flow) :DFG. Ce graphe est biparti, orienté et sans cycle avec comme ensemble de nœuds $J = \{j_1, \ldots, j_n\}$ correspondant aux blocs d'instructions séquentielles (j signifiant *job*, nous dirons tâche) et $T =$

$\{t_1, \ldots, t_m\}$ correspondant aux opérations de lecture et écriture d'une donnée (vue comme un bloc de mots) en mémoire globale (t signifiant *transition*).

Tous les nœuds d'entrée (i.e. de degré entrant nul) et de sortie (i.e. de degré sortant 0) sont des transitions; les premiers correspondent aux données en entrée du programme (fichiers en lecture par exemple) alors que les derniers correspondent à ses sorties.

Le graphe définit un ordre partiel sur J, dit de précédence et noté $\prec$. Pour tout $1 \leq i, k \leq n$, $j_i \prec j_k$ si et seulement s'il existe un chemin orienté de j_i vers j_k.

N.B. Dans toute la suite, nous nous limitons à l'ordonnancement de graphes *d'arité bornée* (qui est celui des algorithmes dits EREW). Dans le cas général (algorithme CRCW), nous supposons donc que le graphe (le programme) a été transformé en une description sous forme graphe de tâches à arité bornée. Une telle transformation peut augmenter d'un facteur quadratique au plus le nombre de sommets (tâches et transitions) du graphe.

Contraintes de placement. Généralement, des fonctions de contraintes ϕ_t : $T \mapsto \mathcal{P}(\{M_i, 1 \leq i \leq p\})$ et ϕ_j : $J \mapsto \mathcal{P}(\{P_i, 1 \leq i \leq p\})$ permettent de spécifier des contraintes de localisation pour certaines transitions ou pour certaines tâches. Une transition en entrée peut par exemple n'être disponible que sur un module mémoire, une tâche ne peut être exécutée que sur un sous-ensemble de processeurs. Dans le cadre de cet article, nous nous limitons au cas où aucune contrainte n'est spécifiée, i.e. $\phi_t(x) = (\{M_i, 1 \leq i \leq p\})$ et $\phi_j(y) = (\{P_i, 1 \leq i \leq p\})$ pour tout $x \in T$, $y \in J$.

Déterminisme des exécutions. Le graphe de dépendance est directement associé à une exécution; il ne peut souvent être complètement déterminé qu'après exécution compléte du programme. Cependant, dans tous les cas, nous supposerons que l'exécution reste *déterministe* : le graphe de dépendance peut être construit complétement dynamiquement, mais reste le même pour les mêmes valeurs en entrée quel que soit l'ordonnancement effectué (même séquentiel). Cette contrainte est liée à la difficulté (impossibilité si $P \neq NP$) de trouver un ordonnancement polynômial pour un algorithme de coût polynômial sur une machine non déterministe (i.e. dont toute exécution admet un ordonnancement polynômial).

5.2.2 Ordonnancement du graphe sur une machine

Une architecture distribuée est vue comme un ensemble d'unités de calcul P_i chacune associée à un module de mémoire M_i. L'ordonnancement consiste alors à affecter à chaque transition t (qui correspond à une donnée "en assignation unique") un module mémoire $M(t)$ et à chaque job j le processeur $P(j)$ sur lequel il sera exécuté. L'ordonnanceur spécifie en outre quels évènements conditionnent le démarrage ou la préemption d'une tâche (généralement fin de communication et fin de tâche de calcul).

5.2.2.a Ordonnancement global et ordonnancement système

Deux niveaux d'ordonnancement peuvent être distingués :

- *macroscopique* : ordonnancement global des nœuds du graphe de dépendance (tâches et données) sur les ressources (processeurs et modules mémoire).
- *microscopique* : ordonnancement local des processus concurrents sur le processeur et des instructions atomiques pour minimiser les temps d'attente (par exemple anticipation du chargement d'une ligne dans le cache ou avancement des communications).

Nous ne nous intéressons dans ce chapitre qu'à l'ordonnancement macroscopique, l'ordonnancement microscopique étant supposé réalisé localement sur chaque processeur par le système.

Calculer un ordonnancement consiste donc à allouer à chaque nœud du graphe (donnée ou tâche) une ressource en complétant la relation de précédence $\prec$ initiale. Chaque exécution du programme après ordonnancement définit une relation d'ordre partiel $\prec_i$ sur l'ensemble des différentes instructions atomiques exécutées. Dans tous les cas, $\prec_i$ doit être compatible avec $\prec$: l'ordonnancement calculé est dit "admissible" si, pour toute exécution et pour tout couple d'instructions atomiques (i_1, i_2) appartenant respectivement aux jobs j_1 et j_2 alors :

$$(j_1 \prec j_2) \Longrightarrow (i_1 \prec_i i_2).$$

5.2.2.b Opérations utilisées pour réaliser l'ordonnancement

Pour réaliser et contrôler l'ordonnancement calculé, l'ordonnanceur utilise certaines primitives supposées fournies par le système. Sauf précisé, toutes ces primitives sont supposées de coût constant.

Exécution d'une tâche . Une tâche (un nœud du DFG) peut être exécutée sur un processeur quelconque.

Communication. Un processeur peut lire ou écrire un objet dans n'importe quel module mémoire distant. Le coût de l'accés à n données est supposé borné par $h.n$ où h est une constante de la machine [30, 9]. Les accès mémoire sont asynchrones. Pour contrôler l'ordonnancement (via les transitions, une tâche accède en lecture à des données écrites par ses prédécesseurs), des outils de synchronisation globaux sont supposés disponibles (verrous, sémaphores).

Duplication. La duplication consiste à exécuter plusieurs fois une même tâche sur différents processeurs; ceci peut permettre de construire des ordonnancements minimisant les coûts de communication.

Migration. Certains algorithmes d'ordonnancement utilisent en outre une opération de préemption, qui permet de suspendre une tâche au cours de son exécution pour la relancer, éventuellement plus tard, sur un autre processeur [29, 3]. Deux types de préemption sont distingués :

- *migration par redémarrage :* lorsqu'une tâche est déplacée sur un autre processeur, son exécution reprend depuis son début.
- *migration :* lorsqu'une tâche est déplacée sur un autre processeur, son exécution reprend à partir de sa dernière instruction exécutée.

Un algorithme d'ordonnancement *non-préemptif* n'utilise aucune de ces opérations : il perd tout contrôle sur une tâche dés qu'il l'a démarrée sur un processeur, sauf information sur sa terminaison. Le coût d'une opération de migration est supposé constant.

Remarque. L'utilisation de la migration est trés liée au modèle de programmation (comment sont implémentées les tâches) et au modèle de machine. Par exemple, si les tâches sont sans bloquage (c'est le cas dans le graphe de dépendance), la migration est inutile lorsque la machine est composée de processeurs identiques. Par contre, elle peut s'avérer utile sur des machines hétérogènes [29].

5.2.3 Notation des problèmes d'ordonnancement

Nous allons utiliser pour classifier les problèmes d'ordonnancement la notation introduite dans [14] et étendue dans [34] pour les problèmes avec communications. Nous l'étendons pour préciser le cas où le graphe de dépendance est construit en cours d'exécution, les coûts des tâches et des communications étant inconnus jusqu'à leur terminaison.

Cette notation est constituée de trois champs : α, β, γ. Ces trois paramètres permettent de règler le niveau d'abstraction de la machine virtuelle et les hypothèses sur le type de graphes considérés.

Le paramètre α définit le type des processeurs et leur nombre. Pour nous :

- $\alpha = P$: signifie que les processeurs sont identiques et que leur nombre m est une entrée du problème.
- $\alpha = \overline{P}$: signifie que les processeurs sont identiques, leur nombre étant suffisant ou non borné.
- $\alpha = P_m$: indique un nombre fixé de processeurs identiques.

Le paramètre β définit le type du graphe de précédence, les temps d'exécution des tâches, le coût de communication, l'autorisation ou pas de la duplication, l'autorisation ou pas de la préemption.
Donc $\beta = \beta_1\beta_2\beta_3\beta_4$ où :

- $\beta_1 \in \{prec?, prec, arbre, chaine, ..., \bullet\}$: Type du graphe de précédence des tâches.
  - $\beta_1 = prec$: le graphe de précédence des tâches est quelconque mais de structure connue;
  - $\beta_1 = prec?$: le graphe de précédence des tâches est quelconque mais inconnu (construit en cours d'exécution);
  - $\beta_1 = arbre$: le graphe de précédence des tâches est un arbre;
  - $\beta_1 = \bullet$: les tâches sont indépendantes.

- $\beta_2 \in \{com, com?, c_{jk}, c, \bullet\}$: communication.
  - $\beta_2 = com$: temps de communication donnés par le graphe de précédence des tâches.
  - $\beta_2 = com?$: temps de communication inconnu avant fin de la communication effective mais constant.
  - $\beta_2 = com? < \alpha \sum p_i$: temps de communication inconnu mais volume total de données communiquées borné par une constante fois le volume total de calcul.
  - $\beta_2 = c_{jk}$: le temps de communication entre la tâche j et la tâche k est c_{jk}.
  - $\beta_2 = c$: temps de communication partout égaux.
  - $\beta_2 = \bullet$: temps de communications nuls.
- $\beta_3 = \in \{dup, mig, \bullet\}$: la migration et la duplication sont autorisées ou non
- $\beta_4 \in \{p_i = 1, p_i = ?, \bullet\}$: durée des tâches;
  - $\beta_4 = (p_i = 1)$: toutes les tâches ont une durée d'execution unitaire.
  - $\beta_4 = (p_i = ?)$: les tâches sont de durées quelconques et inconnue avant exécution.
  - $\beta_4 = \bullet$: les durées des tâches sont définies par le graphe de précédence des tâches.

Le paramètre γ définit l'objectif. Dans ce chapitre, le but sera toujours de minimiser la durée d'exécution, ce but est noté $\gamma = C_{max}$.

5.2.4 Évaluation du coût d'un algorithme d'ordonnancement

Dès que l'on considère des machines virtuelles non triviales, le problème de trouver un ordonnancement optimal d'un graphe devient NP-difficile. On s'intéresse donc à rechercher des heuristiques polynômiales (voire linéaire) pouvant fournir un ordonnancement proche de l'optimal.

L'efficacité d'une heuristique peut-être évaluée de diverses façons. Dans ce cours, nous nous contenterons de mesurer la qualité d'un algorithme d'ordonnancement par deux critères :

- **ratio** (ou compétitivité) : le maximum du rapport sur tous les graphes entre la durée obtenue par l'ordonnancement de l'heuristique et la durée obtenue par un ordonnancement optimal.
- **surcoût** d'ordonnancement : le nombre d'opérations requises pour la mise en œuvre effective de l'ordonnancement.

5.2.4.a Ratio de performance ou compétitivité

Définition 5.2.1 *Soit h un algorithme d'ordonnancement. Le ratio de performance de h, noté $R(h)$, est le maximum du rapport sur tous les graphes entre la durée donnée par l'ordonnancement fourni par h et la durée de l'ordonnancement optimal, c'est-à-dire* $R(h) = Max\frac{C_{max}(h)}{C_{max}^{opt}}$.
On dit qu'un algorithme h est ρ-approché (ou encore ρ-compétitif) si $R(h) \leq \rho$.

Un algorithme h est d'autant plus efficace que son ratio $R(h)$ est petit. La démarche dans le cas d'un problème NP-complet est donc de trouver la meilleure heuristique possible. En fait la plupart du temps nous ne sommes capables que de donner une borne supérieure du ratio de la meilleure heuristique (en étudiant le ratio d'une heuristique connue) et une borne inférieure de ce ratio.

Une technique classique pour obtenir une borne inférieure du ratio est basée sur le résultat suivant, facile à démontrer et appelé "théorème de l'impossibilité" [25].

Théorème 5.2.1 *Etant donné un problème d'ordonnancement et un entier c, si la question de savoir si un graphe G peut-être ordonnancé en c unités de temps ou moins est NP-complet alors on ne peut pas espérer avoir une heuristique pour ce problème d'ordonnancement ayant un ratio inférieur à* $\frac{c+1}{c}$.

La valeur du ratio de performance permet de cerner la complexité pratique du problème et de traduire l'augmentation de la complexité du problème en fonction des nouveaux paramètres intégrés au modèle.
D'un point de vue pratique, l'heuristique de meilleur ratio n'est pas forcément celle à intégrer si le surcoût induit par sa mise en œuvre s'avère trop important.

5.2.4.b Surcoût d'ordonnancement

Définition 5.2.2 *Soit h un algorithme d'ordonnancement et G un graphe de dépendance quelconque. Le surcoût d'ordonnancement de h, noté $\sigma(h)$, est le nombre d'opérations effectuées par h pour exécuter G.*

Il est souvent difficile de séparer a priori le surcoût lié à la mise en œuvre de l'ordonnancement de la longueur de l'ordonnancement lui-même (caractérisé par le ratio). Aussi, son étude est basée sur l'analyse du ratio et complétée par la prise en compte des opérations pour l'ordonnancement.

Le surcoût peut être séparé en deux parties : d'une part les opérations requises pour le calcul de l'ordonnancement et d'autre part celles utilisées pour exécuter le graphe de tâches en garantissant le respect de la relation de précédence.

Cadre statique. Dans le cadre statique, le graphe étant connu à la compilation, le calcul de l'ordonnancement n'implique pas de surcoût. Par contre son contrôle nécessite un surcoût lié – au moins – à la taille du graphe. La réalisation d'un ordonnancement pré-calculé étant triviale, nous considérerons uniquement le ratio de performance pour l'analyse des algorithmes d'ordonnancement statique.

Cadre dynamique. Dans le cadre dynamique, au surcoût de contrôle s'ajoute aussi celui du calcul de l'ordonnancement. Dans tous les cas, chaque nœud et chaque transition devant être placés, il est borné inférieurement par la taille du graphe.

Notation. Pour exprimer le surcoût d'ordonnancement, dans la suite nous désignerons par N_a le nombre de tâches et par N_t le nombre de transitions. Comme tous les nœuds sont supposés d'arité bornée, nous désignerons par $n = N_a + N_t$ le nombre de nœuds dans G et par n_d le degré de G. Une borne sur la taille du graphe est donc nn_d.

5.3 Ordonnancements sans communication

Nous présentons d'abord les résultats préliminaires concernant l'ordonnancement (dynamique ou statique) sans communication.

5.3.1 Problème $P_m|prec, p_i =?|C_{max}$: algorithme de liste

Dans le cas de la régulation dynamique de la charge, l'ordonnancement le plus rencontré en pratique consiste basiquement, lorsqu'un processeur devient inactif, à lui allouer une tâche prête s'il en existe. Un tel ordonnancement est appelé "algorithme de liste" : il requiert de gérer une structure évoluant dynamiquement et permettant d'accéder aux tâches prêtes.

Dans cette section, seul le ratio de performance d'un tel algorithme pour un graphe de précédence quelconque est étudié. Le surcoût induit par sa mise en œuvre sur une architecture distribuée (gestion du graphe et contrôle de l'ordonnancement d'une part, introduction des communications d'autre part) est étudié dans la section 5.4.

Théorème 5.3.1 *[12] Si les communications ne sont pas prises en compte, tout algorithme d'ordonnancement de type liste a un ratio de performance borné par* $\left(2 - \frac{1}{m}\right)$ *sur une machine constituée de* m *processeurs identiques.*

Nous rappelons la preuve de ce résultat car son schéma intervient dans de nombreuses démonstrations.
Soit T_m la longueur de l'ordonnancement fourni par l'algorithme de liste sur m machines. Un algorithme de liste est tel que, à chaque instant, au moins un processeur exécute une tâche. Ainsi, si à un instant donné un processeur est inactif alors il existe au moins un processeur qui exécute une tâche. Soit t_{j_1} l'une des tâches qui s'est terminé à la date T_m et soit d_{j_1} la date de début d'exécution de t_{j_1}. Deux cas peuvent être distingués :

cas 1 : soit aucun processeur n'a été inactif avant d_{j_1}.

cas 2 : soit il existe au moins un processeur inactif à un certain instant avant d_{j_1}. Soit θ la plus grande date avant d_{j_1} à laquelle au moins un processeur est inactif. À θ, t_{j_1} n'est pas prête car sinon elle aurait été affectée à un processeur inactif. Donc il existe une tâche t_{j_2} telle que t_{j_2} est en cours d'exécution à θ et $t_{j_2} \prec t_{j_1}$. Soit d_{j_2} la date de début d'exécution de t_{j_2}.

En appliquant récursivement ce schéma jusqu'à ce que le cas 1 arrive, nous construisons une séquence de tâches $t_{j_k} \prec \ldots \prec t_{j_2} \prec t_{j_1}$ telle que, à tout instant, soit tous les processeurs sont actifs soit un processeur exécute une tâche t_{j_i} $(1 \leq i \leq k)$.

Soit T_∞ le temps minimal sur un nombre infini de processeurs et T_1 le nombre total d'opérations exécutées (temps sur un processeur séquentiel, appelé aussi travail). Le temps total d'inactivité $\#I$ est défini par $\#I = mT_m - T_1$. Pour $1 \leq i \leq k$, soit l_i la durée de la tâche t_{j_i}. Nous avons $\#I \leq (m-1)\sum_{i=1}^{k} l_i$, d'où :

$$pT_m \leq T_1 + (p-1)\sum_{i=1}^{k} l_i.$$

En outre, les tâches t_{j_i}, $1 \leq i \leq k$ étant sur un chemin critique, $\sum_{j=1}^{k} l_i \leq T_\infty$. On a finalement :

$$T_m \leq \frac{T_1}{m} + \left(1 - \frac{1}{m}\right) T_\infty \tag{5.1}$$

Par ailleurs, soit $T_m^\star$ la longueur de l'ordonnancement optimal. On a $T_\infty \leq T_m^\star$ et, comme T_1 opérations doivent être exécutées par tout ordonnancement, $mT_m^\star \geq T_1$. Remplaçant dans 5.1, on obtient : $T_m \leq \left(2 - \frac{1}{m}\right) T_m^\star$. □

Le théorème précédent 5.3.1 est donné dans une version restreinte. [3]. De manière plus générale, la borne 5.1 reste inchangée lorsque la relation de précédence $\prec'$ considérée par l'algorithme de liste est plus forte que celle $\prec$ utilisée pour calculer l'ordonnancement optimal. La preuve est directe puisque on aura encore $t_{j_k} \prec \ldots \prec t_{j_2} \prec t_{j_1}$. Clairement, la même remarque s'applique si la durée des tâches est augmentée artificiellement. Celà implique que ni l'ajout de contraintes de précédence telles que des barrières de synchronisation pour obtenir un graphe de dépendance bien structuré (par exemple semi-parallèle), ni l'insertion artificielle d'opérations de découpe pour que toutes les tâches aient la même durée ne peuvent améliorer le ratio d'un algorithme d'ordonnancement en ligne.

Remarque : principe de Brent constructif. En corollaire de l'encadrement 5.1, on obtient une démonstration constructive générale du principe de Brent pour des tâches de longueur arbitraire (et non forcément unitaires).

Corollaire 5.3.1 *Si un programme parallèle effectue T_1 opérations (temps séquentiel) et peut s'exécuter en temps parallèle minimal T_∞ sur un nombre infini de processeurs, alors il peut être exécuté sur m processeurs en temps*

$$Max\left\{\left\lceil\frac{T_1}{m}\right\rceil, T_\infty\right\} \leq T_m \leq \left\lceil\frac{T_1}{m} + T_\infty\right\rceil \tag{5.2}$$

5.3.2 Bornes inférieures sur le ratio

Une question naturelle est alors de déterminer s'il est possible d'obtenir un ratio de performance inférieur à $\left(2 - \frac{1}{m}\right)$, soit sur le même problème soit en utilisant des opérations plus puissantes pour réaliser l'ordonnancement, comme le tirage aléatoire ou la migration. Ce problème est étudié dans [29] où la proposition suivante est prouvée.

Théorème 5.3.2 *[29] Si les temps de communication ne sont pas pris en compte, $\left(2 - \frac{1}{m}\right)$ est une borne inférieure pour le ratio d'un algorithme d'ordonnancement déterministe avec ou sans migration.*
Un algorithme probabiliste sans préemption ne peut avoir un ratio de performance inférieur à $\left(2 - \frac{1}{\sqrt{m}}\right)$, ratio qui est obtenu par un algorithme de liste dans lequel les tâches affectées aux processeurs inactifs sont tirées au hasard parmi l'ensemble des tâches prêtes.

Seule la preuve de la première partie est ici présentée car elle utilise la construction d'un adversaire, technique fréquemment utilisée pour obtenir des bornes inférieures sur un algorithme à la volée. Il s'agit de construire une instance du problème réalisant le pire cas et de montrer que ce pire cas ne peut être évité par aucun algorithme d'ordonnancement.

Le pire cas est obtenu pour l'instance suivante due à Graham [12]. G contient $1 + p(p-1)$ tâches indépendantes; une tâche α est de longueur p tandis que les β_k, $1 \leq k \leq p(p-1)$ autres sont de longueur 1. L'ordonnancement optimal est de longueur p; il est obtenu en exécutant α_1 sur un processeur et les $p(p-1)$ tâches β_k sur les $p-1$ autres processeurs.

Aucun algorithme d'ordonnancement à la volée ne peut faire d'hypothèse sur la longueur d'une tâche qui est inconnue tant que la tâche n'est pas terminée. La technique de l'adversaire consiste alors à faire en sorte que la la tâche α soit démarrée le plus tard possible. Chaque fois que l'ordonnanceur lance l'exécution d'une tâche sur un processeur, nous supposerons donc que cette tâche est une tâche β_k, de longueur 1. Ainsi l'ordonnanceur ne peut demander l'exécution de la tâche α qu'après que toutes les tâches β_k aient été exécutées, donc au plus tôt au top $(p-1)$. La longueur de l'ordonnancement délivré est alors de longueur au moins $(p-1)+p$, ce qui montre la borne inférieure. Il est clair que la préemption ou la migration ne peuvent améliorer ce ratio. $\square$.

En conséquence, ni la migration ni l'introduction d'aléatoire ne peuvent améliorer le ratio de performance en comparaison d'un algorithme de liste.

5.3.3 Problème : $P|indep, p_j|C_{max}$

Si la durée des tâches est supposée connue, trier la liste (par exemple en ordonnançant en priorité les tâches les plus longues) ne permet pas d'améliorer la borne de Graham. Par contre, il est possible d'obtenir une amélioration en restreignant la classe de graphes considérés.

Ainsi si les tâches sont indépendantes le problème reste NP-complet (réduction au problème de la partition) mais l'heuristique LPTF – *largest processing time first* – (algorithme de liste avec la liste des tâches triée en ordre décroissant de durée) permet d'obtenir un ratio de $\dfrac{4}{3} - \dfrac{1}{3m}$ [13].

Une restriction plus intéressante est celle où toutes les tâches ont la même durée, ce sous problème étudié dans la section suivante servira de base de comparaison lors de l'étude de l'impact des communications.

5.3.4 Problème : $P|prec, p_j = 1|C_{max}$

Théorème 5.3.3 *Il n'existe pas d'algorithme d'ordonnancement pour le problème $P|$ prec, $p_j = 1|C_{max}$ ayant un rapport de performance inférieur à $\frac{4}{3}$ si $P \neq NP$.*

Ce résultat est déduit du résultat suivant par le théorème de l'impossibilité :

Lemme 5.3.1 *[22] Le problème de décider pour un graphe de précédence quelconque s'il existe un ordonnancement valide de longueur au plus 3 est NP-complet.*

Nous allons utiliser une reduction du problème CLIQUE qui est un problème connu NP-complet. On rappelle qu'une *clique* un graphe dont tous les sommets sont adjacents. Il est évident que si $G_1 = (V_1, E_1)$ est une clique et si $G_2 = (V_2, E_2)$ a le même nombre de sommets que G_1 ($|V_1| = |V_2|$) mais n'est pas une clique, alors $|E_1| > |E_2|$. Le problème CLIQUE est le suivant :

- **Instance :** Un graphe non-orienté $G = (V, E)$ et un entier k.
- **Question :** Existe-t-il dans G une clique de taille k ?

Nous réduisons une instance de CLIQUE à une instance de $P|prec, p_j = 1|C_{max}$ de la manière suivante (la figure 5.1 illustre la réduction). Soit $l = \frac{(k-1)k}{2}$ le nombre d'arêtes dans une clique de taille k; Soit $k' = |V| - k$ et $l' = |E| - l$.
On considère l'instance suivante de $P|prec, p_j = 1|C_{max}$:

- Le nombre de machines est $m = \max\{k, l + k', l'\} + 1$.
- Pour chaque sommet $v \in V$, on introduit une tâche T_v et pour chaque arête $e \in E$ une tâche L_e. On a $T_v \prec L_e$ si v est une extremité de e.
- On ajoute $(3m-l-k-l'-k')$ tâches $X_x(x = 1, ..., m-k), Y_y(y = 1, ..., m-l-k')$ et $Z_z(z = 1, ..., m - l')$ avec les précédences $X_x \prec Y_y \prec Z_z, \forall x, y, z$.

Le nombre total de tâches est $3m$. Un ordonnancement valide avec $C_{max} = 3$ ne peut donc pas avoir de temps d'inactivité.

- Supposons que G contient une clique de taille k, alors il existe un ordonnancement valide de longueur 3 (voir figure 5.1).
- Supposons qu'on a un ordonnancement valide de longueur 3, alors G contient une clique de taille k.

Il est évident que les tâches qui appartiennent à des chemins de longueur 3 doivent être exécutées au plus tôt. Ainsi X_x, Y_y et Z_z sont exécutés aux instants 1, 2 et 3 respectivement.

A l'instant 1 on peut exécuter au plus k tâches de type T. Supposons que seulement $k_1 < k$ tâches forment une clique. Ceci dit qu'il y aura certains processeurs qui vont rester inactifs pendant la deuxième unité de temps puisque les tâches de type L (c'est-à-dire les arêtes adjacentes aux k sommets exécutés à l'instant 1) qui sont liberées sont moins de l. Par conséquent, l'ordonnancement ne peut pas terminer à $t = 3$. □

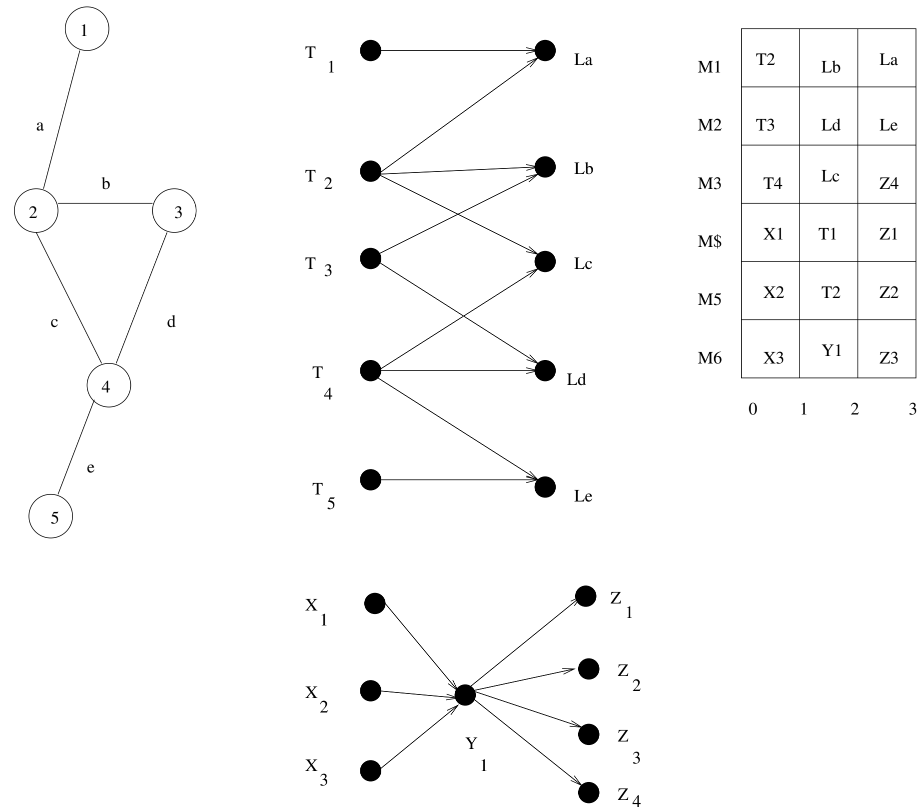


Figure 5.1 : Le graphe de précédence des tâches, l'instance correspondante du problème d'ordonnancement où $m = 6$, et un ordonnancement en trois unités de temps.

5.3.5 Une borne supérieure pour $P|prec, p_j = 1|C_{max}$: l'algorithme de Coffman-Graham

La solution du problème utilise un algorithme d'étiquettage qui associe à chaque tâche du graphe un entier. L'étiquettage se fait de la façon suivante [18]. R désigne l'ensemble des tâches non étiquettées dont tous les successeurs immediats sont déjà étiquettés.

- **Etape 1 :** On distribue de façon arbitraire des étiquettes aux tâches qui n'ont pas de successeurs.
- **Etape 2 :** A chacune des tâches de R on fait correspondre la séquence décroissante des étiquettes de ses successeurs immédiats S_i. La tâche qui a la plus petite (lexicographiquement) séquence S_i prend l'étiquette suivante.

- **Etape 3 :** On itére l'étape 2 jusqu'à ce que toutes les tâches de G soient étiquettées.
- **Etape 4 :** Construction d'une liste de priorité en ordre décroissant des étiquettes.

Théorème 5.3.4 *[21] L'algorithme de CG a un ratio de performance borné par*

$$2 - \frac{2}{m}$$

La preuve assez technique de ce résultat est omise. Elle permet d'obtenir le corollaire suivant.

Corollaire 5.3.2 *L'algorithme de CG est optimal pour* $P2|prec, p_j = 1|C_{max}$.

Etant donné que la complexité en temps de l'algorithme de CG est en $O(n^2)$, il est évident qu'on a intérêt de l'utiliser seulement pour des petites valeurs de m. Pour des grandes valeurs de m les gains par rapport à un algorithme de liste quelconque deviennent minimes.

Remarque. La complexité du problème $P_m|prec, p_j = 1|C_{max}$ n'est pas connue quand $m \geq 3$.

5.4 Ordonnancements dynamiques avec communications

Nous nous intéressons à la mise en œuvre d'un algorithme de liste sur une architecture distribuée qui fournit un ordonnancement de longueur presque optimale sur un nombre arbitraire de processeurs identiques (cf corollaire 5.3.1). L'intérêt d'un tel algorithme est d'être compatible avec une création dynamique du graphe de dépendance : une tâche t peut créer lors de son exécution de nouvelles tâches t_k, en nombre quelconque, en ajoutant de nouvelles dépendances. Il est donc particulièrement adapté à des applications irrégulières où la génération de parallélisme est souvent réalisée en cours d'exécution.

Deux problèmes sont alors à considérer :

- l'implémentation de l'algorithme de liste lui-même : gestion du graphe et contrôle de l'ordonnancement par affectation de tâches prêtes aux processeurs inactifs.
- la prise en compte du coût des communications.

Les résultats présentés sont appliqués à l'ordonnancement de programmes dont le nombre d'opérations est grand devant le nombre de tâches et de dépendances (*granularité globale*).

Remarque : Hypothèse sur la construction à la volée du graphe de tâches. Pour faciliter la gestion du graphe de dépendance (qui induit une relation de précédence entre les tâches), nous supposerons que :

- les tâches t_k créées ont une relation de précédence avec la tâche t qui les crée, c'est à dire : $\forall k$, si t crée t_k alors $t \prec t_k$;
- les nouvelles dépendances de données ajoutées au graphe ne peuvent concerner que les tâches ayant préalablement une relation de précédence avec t, c'est à dire : soit t_1 et t_2 deux tâches et $t_1 \prec t_2$ une dépendance ajoutée par t, alors, avant l'ajout de cette dépendance, on avait $t \prec t_1$ et $t \prec t_2$.

Ces contraintes, bien que non restrictives en ce qui concerne la description de l'exécution d'un algorithme sous forme de graphes de tâches (cf les chapitres sur la construction de graphes de tâches), ne sont pas fondamentales : leur introduction est motivée par l'obtention d'une implémentation concise en ce qui concerne l'algorithme de listes.

5.4.1 Coût des communications : simulation d'une mémoire partagée

Pour prendre en compte le coût des communications (contention et latence), nous simulons une mémoire partagée. La simulation utilise des fonctions de hachage (choisies aléatoirement dans une classe de fonctions de hachage universelles) pour distribuer les objets accessibles par tous les processeurs (i.e. les transitions dans le graphe de dépendance) sur les différents modules mémoire [28, 20].

Le délai d'une simulation est le temps requis pour l'accès à une donnée de taille unité. Il est lié à l'évaluation de la fonction de hachage, à la contention mémoire (quand plusieurs accès sont effectués vers un même module) et à la latence (temps de routage). Dans [28], une simulation de délai $\Theta(\log p)$ sur un réseau “butterfly” est donnée pour des accès exclusifs (de type EREW). Dans [19], une simulation de délai $O(\log\log p \log^\star p)$ est donnée pour des accès aléatoires (de type EREW ou CRCW) sur une architecture disposant d'un réseau d'inteconnexion complet (la latence n'est pas prise en compte); pour des accès concurrents (CRCW), la simulation est à un facteur $\log^\star p$ de l'optimal.

Pour obtenir une simulation optimale, ces délais doivent être masqués par des calculs. La technique classique (virtualisation ou “parallel slackness” [20, 31, 19]) consiste à simuler q ($q > p$) processeurs virtuels sur les p processeurs de l'architecture physique, en utilisant généralement des processus légers. La simulation est alors dite optimale si le délai pour un accès est proportionnel à q/p.

Dans la suite de cette section, nous supposerons disposer d'une simulation optimale de délai h; le nombre de processeurs m requis pour une telle simulation peut alors être supérieur au nombre de processeurs p de l'architecture. L'ordonnancement sera donc effectué sur m processeurs et non p, mais comparé à l'ordonnancement optimal sur p processeurs. Un processeur (parmi les m) peut lire ou écrire un objet dans n'importe quel module mémoire distant. Le coût de l'accés à une donnée de taille n est supposé borné par $h.n$ où h est une constante de la machine.

Synchronisation. Les accès en mémoire distante ne permettent pas directement de synchroniser les processeurs (à la différence des communications). Sur une architecture distribuée asynchrone, des outils spécifiques de synchronisation (verrous, sémaphores) doivent alors être utilisés. Nous supposerons aussi que le coût d'accés à un verrou global, lorsque celui-ci est libre, est lui aussi borné par h (il requiert un aller-retour).

5.4.2 Problème $P_m|prec?, p_i =?|C_{max}$: algorithme de liste

Lorsque le graphe de dépendance est construit et déterminé en cours d'exécution, l'implémentation d'un algorithme de liste nécessite de gérer une structure permettant de déterminer les tâches prêtes (i.e. dont les prédécesseurs sont terminés) et d'affecter des tâches distinctes à chacun des processeurs devenus inactifs.

Directe en séquentiel, cette affectation est cependant complexe en distribué. Dans [7], un algorithme basé sur des graphes d'expansion est utilisé pour ordonnancer n tâches indépendantes effectuant au total $T_1 = O(n)$ opérations en temps $O(\log n)$ sur $\frac{n}{\log n}$ processeurs d'une EREW PRAM; outre le caractère théorique d'un tel ordonnancement (qui suppose une synchronisation globale à chaque instruction), la constante masquée dans le O est loin d'être négligeable (> 237) devant le ratio de performance (proche de 2)[1].

Dans ce qui suit, nous considérons dans un but simplificateur une gestion séquentielle de la structure permettant de calculer les tâches prêtes (i.e. la liste) et de les affecter aux processeurs inactifs. À défaut d'être la plus performante sur un grand nombre de processeurs, elle permet d'estimer précisément le surcoût lié à la gestion de l'ordonnancement. Pour éviter de dédier un processeur à la gestion de cette structure, elle est stockée en mémoire (virtuellement) partagée et accédée par les processeurs en exclusion mutuelle; l'exclusion est réalisée par un verrou global.

Théorème 5.4.1 *Soit un programme parallèle dont l'exécution génère n tâches et n_d dépendances (transitions). Soit T_1 le temps d'une exécution séquentielle (sur un processeur) et T_∞ le temps minimal sur un nombre infini de processeurs. Alors, si les communications ne sont pas prises en compte, G peut être ordonnancé sur m*

[1] Une constante beaucoup plus faible peut être obtenue si l'ordonnancement est réalisé sur moins de processeurs. Par exemple, l'utilisation d'un calcul de préfixe permet de calculer un ordonnancement à un facteur 2 de l'optimal sur $\frac{n}{\log^2 n}$ processeurs [6].

processeurs identiques en temps T_m borné par :

$$T_m \leq \frac{T_1}{m} + T_\infty + O(n + n_d + m).$$

Comme pour le théorème 5.3.1, la preuve est basée sur la construction d'un ensemble de tâches situées sur un chemin critique dans G. Pour contrôler l'ordonnancement, nous utilisons une structure globale, visible par tous les processeurs et accédée en exclusion mutuelle grâce à un verrou global `verrou-graphe`; outre le graphe G, cette structure contient :

- la liste L_t des tâches prêtes (initialisée avec les tâches de G de degré entrant nul);
- la liste L_i des processeurs inactifs; à l'initialisation, seul un processeur (disons P_0) est supposé artficiellement actif – il simule la terminaison d'une tâche fictive –, les autres $m - 1$ processeurs étant mis dans L_i.

L'ordonnancement des tâches est géré de la façon suivante : lorsqu'un processeur p termine une tâche t (par exemple P_0 au top 0), il exécute les instructions de *post-traitement* liées à t suivantes :

1. Prendre le verrou `verrou-graphe`.
2. Mettre à jour G en ajoutant les nouvelles tâches créées et les dépendances associées.
3. Mettre à jour la liste L_t des tâches prêtes en ajoutant les tâches successeurs immédiats de t dont tous les prédécesseurs sont terminés.
4. Tant que ($L_i \neq$ vide) et ($L_t \neq$ vide) faire
   Enlever une tâche t' de L_t et un processeur p' de L_i;
   Lancer l'exécution de t' sur p'.
5. Si ($L_t \neq$ vide) alors
   Enlever une tâche t'
   Relâcher le verrou `verrou-graphe`;
   Exécuter t' (sur le processeur p)
6. sinon
   Ajouter p à la liste des processeurs inactifs
   Relâcher le verrou `verrou-graphe`;

De manière évidente, l'ordonnancement ainsi construit est admissible, i.e. ne viole aucune contrainte de précédence dans G. Soit T_m le temps d'exécution avec cet ordonnancement sur m machines. En reprenant le schéma de la preuve précédente (5.3.1), il s'agit de borner les tops d'inactivité *potentielle*. Les tops $\{1, ..., T_m\}$ sont partitionnés en trois sous-ensembles, A (activité), O (ordonnancement) et I (inactivité) de la façon suivante :

- à tout top dans A, tous les processeurs exécutent une tâche, i.e. une instruction de l'application.
- à tout top dans O, une instruction de post-traitement (i.e. gestion de l'ordonnancement) est en cours.
- à tout top dans I, la liste L_i des processeurs inactifs est non vide mais aucune instruction de post-traitement n'est en cours d'exécution.

En remarquant que, lorsqu'un processeur est en attente du verrou, une instruction de post-traitement est nécessairement en cours sur un autre processeur, il n'y a pas dans $A \bigcup C$ de tops où un processeur est en attente du verrou.

$\#A$ peut être trivialement borné par $\frac{T_1}{m}$. L'exécution en exclusion mutuelle des instructions de post-traitement permet de borner facilement $\#O$. En effet, une fois le verrou pris, le post-traitement ne consiste qu'en la mise à jour des listes (les opérations de base sont en coût constant) et au lancement de nouvelles exécutions éventuelles. Lorsque le verrou est libre, le temps nécessaire à sa prise est borné par h. Lorsqu'il n'est pas libre, il existe nécessairement un autre processeur en train d'exécuter une instruction de post-traitement (éventuellement en train de prendre ou de relâcher le verrou). Par suite, le nombre de tops dans $\#O$ est borné par $O(h(n + n_d + m))$ (i.e. la taille du graphe).

$\#I$ est borné de manière analogue à la preuve de Graham, en remarquant que lorsqu'un processeur est inactif et aucun en cours de post-traitement, il existe nécessairement une tâche sur un chemin critique qui est en cours d'exécution. Par suite, $\#I < T_\infty$.

Pour conclure, $mT_m = m\#O + m\#A + (m-1)\#I$. D'où[2] $T_m \leq \frac{T_1}{m} + T_\infty + O(n + n_d + m)$. □

Conclusion sur le surcoût d'ordonnancement. Borner le surcoût requiert d'analyser l'algorithme qui contrôle l'ordonnancement. Dans tous les cas, ce surcoût fait apparaître un terme lié à la taille du graphe de dépendance. Pour le diminuer, on peut s'intéresser à structurer ce graphe; mais les remarques énoncées à la section précédente montre que celà ne peut qu'entraîner une perte en terme de ratio.

[2] Le ratio $(2 - 1/m)$ de Graham se retrouve directement dans le cas où $\#O = 0$.

Implémentation d'un algorithme de liste. En pratique, la gestion centralisée des listes et l'accès en exclusion mutuelle est évité. D'une part, le graphe est stocké de manière distribuée; d'autre part, lorsqu'un processeur est inactif, il scrute les autres processeurs pour trouver une nouvelle tâche à exécuter (on parle de "vol" de tâche). Une telle stratégie est analysée en moyenne d'un point de vue théorique dans [5, 17] et permet d'obtenir un résultat similaire à 5.4.1.

5.4.3 Problème $P_m|prec?, p_i = ?, com?|C_{max}$: communications

L'algorithme précédent ne prend pas en compte le délai entre l'affectation d'une tâche prête à un processeur et le moment où elle démarre son exécution, les données dont elle dépend étant disponibles localement sur ce processeur. De part la technique de simulation utilisée pour la mémoire partagée (§5.4.1), ce délai est borné par h fois le volume des données accédées. Le résultat s'étend donc directement.

Théorème 5.4.2 *Soit une machine simulant m processeurs identiques accédant avec un délai h à une mémoire virtuellement partagée.*
Soit un programme parallèle dont l'exécution génère n tâches et n_d dépendances (transitions). Soit T_1 le temps de l'exécution séquentielle (sur un processeur), W_c le volume total maximal d'accès distants et T_∞ le temps minimal sur un nombre infini de processeurs (avec communications). Alors, G peut être ordonnancé sur m processeurs identiques en temps T_m borné par :

$$T_m \leq \frac{T_1}{m} + h\frac{W_c}{m} + T_\infty + hO(n + n_d + m).$$

Le résultat s'obtient à partir de la preuve précédente (théorème 5.4.1) en bornant $m\#A$ par $T_1 + W_c$ et $\#I$ par T_∞ (qui prend en compte les communications sur le chemin critique). □

5.4.4 Granularité globale : $P_p|prec?, p_i = ?, com? < \alpha \sum p_i|C_{max}$

Le théorème précédent ne permet pas d'évaluer le ratio de performance par rapport à l'ordonnancement optimal. D'une part, ce dernier peut éventuellement supprimer toutes les communications et d'autre part, la simulation de délai optimal utilisé requiert un nombre de processeurs m supérieur au nombre de processeurs p physiques. Le temps effectif sur p processeurs est alors $\frac{m}{p}T_m$; il ne peut être garanti à un facteur constant de l'optimal que si hW_c négligeable devant T_1.

Lorsque le volume maximal de communications est borné par αT_1 (problème $P_p|prec?, p_i = ?, com? < \alpha \sum p_i|C_{max}$), l'algorithme de liste précédent (théorème 5.4.2) permet d'obtenir un ordonnancement de longueur bornée par :

$$\frac{m}{p}\left((1 + \alpha h)\frac{T_1}{m} + T_\infty + hO(n + n_d + m)\right).$$

Si pour une entrée de taille suffisante, les valeurs de α (borne supérieure du rapport entre volume de communications et volume de calcul), T_∞ (temps parallèle minimal) et $(n+n_d)$ (nombre de tâches et de dépendances) sont asymptotiquement négligeables devant le temps séquentiel d'exécution, le ratio de performance est asymptotiquement optimal (i.e. 1). D'où le théorème suivant.

Théorème 5.4.3 *Soit un réseau de processeurs constitué de p processeurs identiques communiquants via un réseau d'interconnexion.*
Soit π un programme parallèle dont l'exécution construit un graphe de dépendance avec un volume de dépendances de données asymptotiquement négligeable devant le temps séquentiel T_1 et un nombre d'arêtes de dépendances asymptotiquement négligeable devant le temps parallèle sur un nombre infini de processeurs. Alors π peut être exécuté en temps T_p asymptotiquement optimal par un algorithme de liste :

$$T_p \leq (1+\epsilon)\frac{T_1}{p}.$$

On obtient donc un ordonnancement asymptotiquement optimal pour des problèmes très parallèles et requérant peu de communications. Ce cadre est plus général que celui de la granularité grossière (certaines tâches pouvant requérir un temps de communication grand devant le temps de calcul [10]). Il est à noter que la création dynamique de tâches et la gestion du graphe n'implique pas de surcoût pour autant que la taille du graphe des tâches reste négligeable devant le temps parallèle optimal.

5.5 Ordonnancements statiques avec communications

En reprenant le même modèle de communication qu'au paragraphe 5.4.1, nous considérons que le temps de communication entre deux tâches ayant une dépendance directe est nul si les deux tâches sont exécutées sur le même processeur et est de durée constante et connue (proportionnelle à la taille de la donnée communiquée) sinon. Soit d_i la date de début de la tâche t_i sur le processeur P_i; si (i,j) est un arc du graphe de précédence et P_i le processeur qui exécute la tâche i, alors

$$\begin{cases} d_j - d_i \geq p_i, & si \quad P_i = P_j \\ d_j - d_i \geq p_i + c_{ij}, & si \quad P_i \neq P_j \end{cases}$$

où c_{ij} est le coût de la communication interprocesseur entre les tâches i et j.

5.5.1 Granularité grossière : $\bar{P}|prec, c=1, p_j=1|C_{max}$

Théorème 5.5.1 *Le problème de décider si une instance de $\bar{P}|prec, c=1, p_j=1|C_{max}$ a un ordonnancement de longueur au plus 6 est NP-complet.*

La preuve est basée sur une transformation par le problème 3-SAT (voir [16]).

Corollaire 5.5.1 *Il n'existe pas d'heuristique ρ-approchée telle que* $\rho \leq \frac{7}{6}$.

Ainsi nous savons que nous sommes dans l'impossibilité de trouver une heuristique qui permette de garantir que quelque soit le graphe en donnée la solution obtenue par l'heuristique est à moins de 16% de la solution optimale.

Ce problème avec $c = 0$ était un problème facile. Une heuristique triviale pour $c = 1$ est d'intercaler entre 2 étapes de calcul une phase de communication. Cette heuristique a trivialement un ratio égal à deux. Récemment dans [26], les auteurs donnent une heuristique de ratio $\frac{4}{3}$. Cette heuristique est basée sur la résolution d'un problème de programmation linéaire.

En effet, soit G=(V,E) le graphe représentant une instance du problème $\bar{P}|prec, c = 1, p_j = 1|C_{max}$. Le problème peut-être formulé par le problème linéaire en nombres entiers suivant :

$$\begin{array}{c} Min \quad w \\ \forall i \in V, \quad t_i \;\geq\; 0 \\ \forall (i,j) \in E, \quad x_{(i,j)} \;\in\; \{0,1\} \quad (*) \\ \forall (i,j) \in E, \quad t_i + 1 + x_{(i,j)} \;\geq\; t_j \\ \forall i \in V, \quad \sum_{j \in \Gamma^+(i)} x_{(i,j)} \;\geq\; |\Gamma^+(i)| - 1 \\ \forall i \in V, \quad \sum_{j \in \Gamma^-(i)} x_{(j,i)} \;\geq\; |\Gamma^-(i)| - 1 \\ \forall i \in V, \quad t_i \; +1 \;\leq\; w \end{array}$$

Avec cette formulation $x_{(i,j)} = 0$ implique que les tâches i et j sont exécutées sur le même processeur. Lorqu'on relâche les contraintes d'intégrité (c'est-à-dire on remplace $(*)$ par $\forall (i,j) \in E, \;\; x_{(i,j)} \;\in\; [0,1]$) on obtient un problème de programmation linéaire qui peut être résolu en temps polynômial.

Intuitivement, dans la solution réelle obtenue plus $x_{(i,j)}$ est proche de 0 plus on a intérêt à supprimer la communication entre la tâche i et la tâche j. Dans [26], les auteurs montrent qu'effectivement en mettant à 0 tous les $x_{(i,j)}$ strictement inférieurs à $\frac{1}{2}$ et à 1 les autres, on obtient une solution réalisable à moins de 33% de la solution réelle (qui est elle-même une borne inférieure de la solution optimale).

Théorème 5.5.2 *L'heuristique proposée est un algorithme* $\frac{4}{3}$*-approché.*

Dans [8], il est montré que ce problème devient facile si on autorise la duplication des tâches.

5.5.2 Le problème $P|prec, c = 1, p_j = 1|C_{max}$

Théorème 5.5.3 *Le problème de décider si une instance de $P|biparti, c = 1, p_j = 1|C_{max}$ a un ordonnancement de longueur au plus 4 est NP-complet.*

La preuve est omise; elle utilise une réduction du problème de la clique [1].

Ainsi nous savons que nous sommes dans l'impossibilité de trouver une heuristique qui permettent de garantir que quelque soit le graphe en donnée la solution obtenue par l'heuristique est à moins de 25% de la solution optimale pour le problème $P|prec, c = 1, p_j = 1|C_{max}$.

Une solution classique pour obtenir un ordonnancement sur un nombre limité de processeurs est de partir d'un ordonnancement sur un nombre illimité de processeurs et de plier cet ordonnancement sur le nombre de processeurs considéré. Le théorème suivant permet d'évaluer une telle démarche.

Théorème 5.5.4 *A partir de n'importe quelle heuristique de rapport de performance ρ pour le problème $\bar{P}|prec, c = 1, p_j = 1|C_{max}$ on peut obtenir une heuristique pour le problème $P|prec, c = 1, p_j = 1|C_{max}$ de rapport de performance $1 + \rho$.*

Soit l'heuristique h suivante : on exécute les tâches dans le même ordre que pour l'heuristique, notée h^* dans la suite, utilisée pour le problème $\bar{P}|prec, c = 1, p_j = 1|C_{max}$. Plus précisément, si X_i est l'ensemble des tâches exécutées à l'instant i avec l'heuristique h^*, et β_i le cardinal de X_i, alors dans h, si $\beta_i \neq 0$, on exécute les tâches de X_i en $\lceil \frac{\beta_i}{m} \rceil$ instants où m est le nombre de processeurs. Dans le cas où $\beta_i = 0$, on conserve l'instant d'inactivité.

On remarque que le nombre d'instants où il y a au moins un processeur inactif est inférieur à la durée de l'ordonnancement avec un nombre illimité de processeurs. Les autres instants, tous les processeurs sont actifs. Ainsi l'écart entre le temps donné par l'heuristique et le temps optimal est borné par la durée de l'ordonnancement avec un nombre illimité de processeurs. Si l'on note t^* la durée de l'ordonnancement donné par h^* et t^m la durée de l'ordonnancement donné par h alors $t^m \leq t^* + t^m_{opt}$. Comme par hypothèse $t^* \leq \rho t^*_{opt}$ et comme $t^*_{opt} \leq t^m_{opt}$, la conclusion s'impose. □

Le corollaire immédiat de ce théorème est l'existence d'une heuristique de ratio de performance $\frac{7}{3}$ pour le problème $P|prec, c = 1, p_j = 1|C_{max}$. Cette heuristique est la meilleure connue à l'heure actuelle.

Dans les sections suivantes nous allons citer quelques familles de graphes ayant un comportement spécifique vis-à-vis de ce modèle.

5.5.3 Le problème $P2|arbre, p_j = 1, c = 1|C_{max}$

Une famille des graphes de précédence qui a un intérêt important est la famille des arbres. Depuis 1961, on sait que le problème $P|arbre, p_j = 1|C_{max}$ est un problème polynômial. En fait, T.C. Hu a proposé un algorithme simple qui consiste à construire une liste de priorité où la tâche dont la distance de la racine (le niveau) de l'anti-arbre est la plus grande, est la tâche la plus prioritaire.

Par contre, quand on prend en compte les communications Lenstra et al [24, 33] ont montré que le problème $P|arbre, p_j = 1, c = 1|C_{max}$ devient NP-complet. Lorsqu'on fixe le nombre de processeurs, un algorithme optimal de programmation dynamique de complexité $O(n^{2(m-1)})$ a été présenté dans [32].

Restrictions sur la structure du graphe. Parmi les autres classes de graphes pour lesquels un ordonnancement optimal est facilement calculable figure les graphes ordonnés par intervalles. Un graphe de dépendance est dit ordonné par intervalles si à toute tâche t peut être associé un intervalle réel I_t tel que $t \prec t'$ si et seulement si $\forall x \in I_t$, $\forall y \in I_{t'}$ on a $x < y$. Por de tels graphes, le calcul d'un ordonnancement optimal peut être ramené à un tri [27].
Étant donné un graphe quelconque, le problème est alors d'ajouter un nombre minimal de dépendances qui, sans limiter de manière trop importante le parallèlisme, permettent de le ramener à une famille de graphes pour laquelle le calcul de l'ordonnancement optimal est facile.

5.5.4 Granularité fine : $P|prec, c, p_j = 1|C_{max}$

Sur ce modèle, les résultats connus sont très décevants et de nombreux travaux théoriques et pratiques sont en cours.

Les meilleurs résultats connus pour la borne inférieure du ratio de performance est $1 + \dfrac{1}{c+3}$ [2]. Dès que c devient grand et donc que la granularité est relativement fine, cette borne ne représente aucun intérêt. Réciproquement les meilleures heuristiques connues donnent un ratio de performance égal à $c + 2$. Là aussi, la borne ne représente aucun intérêt puisque ce ratio est le même qu'une heuristique triviale consistant à prendre un ordonnancement négligeant les communications et à introduire entre deux étapes de calcul une inactivité des processeurs de durée c pour permettre les communications.

5.5.5 Tableau récapitulatif

Le tableau ci-dessous résume les différents résultats pour l'ordonnancement statique lorsque les tâches sont de même durée ($p_i = 1$) et les communications de durée c.

	m entrée		m fixé	m suffisant	
	B_{inf}	B_{sup}		B_{inf}	B_{sup}
c = 0	$\frac{4}{3}$	2	m = 2 (Classe P) m = 3 (Pb Ouvert)	1	1
c = 1	$\frac{5}{4}$	$\frac{7}{3}$	m = 2 (Pb Ouvert)	$\frac{7}{6}$	$\frac{4}{3}$
c grand	$1 + \frac{1}{c+3}$	c + 2	m = 2 (NP-Complet)	1	1 + c

5.6 Systèmes hétérogènes

Le rôle croissant que joue les réseaux de stations de travail dans le domaine du calcul parallèle, nous oblige à intégrer ce paramètre dans nos modèles. La plupart de ces réseaux sont des réseaux constitués de stations qui n'ont pas tous la même puissance et il est évident que cette hétérogénéité pose de nouveaux problèmes sur le plan de la modélisation et de l'analyse des problèmes d'ordonnancement associés.

Dans le cas général, même si l'on ne considère pas les coûts de communication, le temps d'une tâche ne peut être connu qu'après sa terminaison et dépend non seulement du processeur auquel elle a été affectée mais encore de l'état de ce processeur durant son exécution. Le problème apparaît donc complexe à modéliser. Le but de cette section est d'étudier cette complexité même sous des hypothèses simplificatrices.

5.6.1 Difficulté d'un ordonnancement optimal

Dans cette section nous nous limitons au cas de systèmes comportant une machine puissante, que l'on va noter P_1 et un nombre illimité de machines de faible puissance, que nous appelerons de type P_2. Ce problème est noté $P_1 \cup \bar{P}_2|prec, p_i^1, p_i^2|C_{max}$ où p_i^1 (resp. p_i^2) représente la durée d'exécution sur P_1 (resp. sur un processeur de type P_2).

Un ordonnancement consiste ici en deux types d'information :

- l'affectation des tâches au processeur (choix du type de processeur)
- l'ordre d'exécution sur chaque processeur (ici sur P_1)

En fait, il a été montré que même si on se limite à des graphes bipartis de profondeur 1, le problème est déjà NP-difficile. Même lorsque l'affectation des tâches au processeur est imposée le problème reste NP-complet dans des versions très simples (par exemple lorsque le graphe est une réunion de chemins de longueur 3). Pour plus de détails voir [2].

5.6.2 Machines uniformes et non-uniformes

Lorsque la durée d'une tâche est inconnue mais ne dépend que du processeur sur lequel elle a été affectée, deux modèles de machine sont distingués (m désigne le nombre de processeurs de la machine) :

- machine *uniforme* : les processeurs sont de vitesses proportionnelles, le rapport des vitesses étant éventuellement inconnu).
- machine *non-uniforme* : le temps d'une tâche dépend du processeur sur lequel elle s'exécute.

Dans ces deux cas, $\Omega(\log m)$ est une borne inférieure pour le ratio de perfomance d'un algorithme d'ordonnancement dynamique avec ou sans préemption [29, 15].

Par ailleurs, des algorithmes polynomiaux ont été proposés pour ces deux modèles qui permettent d'atteindre un ratio $O(\log n)$, n étant le nombre de tâches [23]. En regroupant les tâches affectées à un même processeur, [29] donnent un algorithme d'ordonnancement dynamique de ratio $O(\log \min(m, R))$ où R le rapport de vitesse entre le processeur le plus rapide et le plus lent. Il est à noter que tous ces ratio peuvent être obtenus par des algorithmes d'ordonnancement non-préemptifs; la migration n'apporte un gain que d'un facteur constant dans le ratio.

5.6.3 Simulation d'une machine à processeurs identiques

Une approche pragmatique pour construire des algorithmes ayant une justification théorique (par exemple un algorithme de liste) est de simuler une machine constituée de processeurs identiques sur une architecture hétérogène. Une telle simulation est facilitée par la manipulation de processus légers dont le nombre sur chaque processeur peut varier en fonction de l'état du processeur.

5.7 Conclusion

Les différents résultats énoncés dans ce chapitre montrent la difficulté d'obtenir des résultats analytiques non triviaux lorsqu'on tente d'intégrer dans les modèles d'ordonnancement certains paramètres primordiaux d'un point de vue pratique comme les grands délais de communication. Dans le cas général, il n'existe pas d'algorithmes d'ordonnancement ayant un ratio de performance constant (indépendant de l'architecture) sur une architecture hétérogène. Il est donc important de considérer des classes spécifiques de problèmes à ordonnancer.

Une approche harmonieuse semble être de se restreindre à des graphes de dépendance (éventuellement construits dynamiquement en fonction des entrées) dont le volume maximal de communication est négilgeable devant le volume de calcul. Aussi bien dans le cadre dynamique (algorithmes de liste) que statique (granularité grossière, $p_i = c = 1$), des ordonnancements efficaces sont alors connus. Cependant, l'obtention d'un tel graphe par compilation à partir d'un programme de grain fin nécessite des algorithmes de partition efficaces qui risquent de réduire fortement le degré de parallélisme sur des problèmes ayant un comportement irréguliers.

Bibliographie

[1] Anderson (R.) et Miller (G.). – Deterministic parallel list ranking. *In: AWOC88*. pp. 81–90. – Springer-Verlag.

[2] Bampis (E.), Giannakos (A.) et Konig (J.-C.). – *From simple-machine problems to hetegogeneous scheduling.* – Rapport technique n° 25, LaMI, Université d'Evry, France, 1997.

[3] Blazewicz (J.), Exker (K.), Schmidt (G.) et Węglarz (J.). – *Scheduling in Computer and Manufacturing Systems.* – Germany, Springer-Verlag, 1993.

[4] Blelloch (G. E.), Gibbons (P. B.) et Matias (Y.). – Provably efficient scheduling for languages with fine-grained parallelism. *In: Proceedings of the 7th Symposium on Parallel Algorithms and Architectures.* pp. 1–12. – Santa-Barbara, California, 1995.

[5] Blumofe (R. D.). – *Executing Multithreaded Programs Efficiently.* – Boston, Thèse de PhD, Massachussets Institute of Technology, 1995.

[6] Briat (J.), Gautier (T.) et Roch (J.-L.). – Application irrégulière et ordonnancement en ligne. *In: Placement dynamique et répartition de charge: application aux sy stèmes répartis et parallèles*, éd. par Bernard (G.), Chassin de Kergommeaux (J.), B. (F.) et Roucairol (C.), pp. 81–106. – Collection didactique INRIA, 1996.

[7] Cole (R.) et Vishkin (U.). – Approximate Parallel Scheduling. Part I : The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM Journal on Computing*, vol. 17, n° 1, 1988.

[8] Colin (J.) et Chretienne (P.). – C.P.M. Scheduling with small communication delays and task dupli cation. *Operations Research*, vol. 39, n° 3, 1991, pp. 680–684.

[9] Culler (D. E.), Karp (R. M.), Patterson (D.), Sahay (A.), Santos (E. E.), Schausser (K. E.) et Ramesh Subramonian (T. v. E.). – LogP: A Practical Model of Parallel Computation. *Communications ACM*, vol. 39, n° 11, 1996, pp. 78–85.

[10] Gautier (T.), Roch (J.) et Villard (G.). – Regular versus irregular problems and algorithms. *In : Proc. of IRREGULAR'95, Lyon, France.* – Springer-Verlag.

[11] Gibbons (P.). – A more practical PRAM model. *In : Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures.*

[12] Graham (R.). – Bounds for Certain Multiprocessor Anomalies. *Bell System Tech J.*, vol. 45, 1966, pp. 1563–1581.

[13] Graham (R.). – Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, vol. 17, n° 2, 1969, pp. 416–426.

[14] Graham (R.), Lawler (E.), Lenstra (J.) et Rinnooy Kan (A.). – Optimization and approximation in deterministic scheduling: A survey. *Ann. Disc. Math.*, vol. 5, 1979, pp. 287–326.

[15] Hall (L. A.), Schulz (A. S.), Shmoys (D. B.) et Wei n (J.). – *Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms.* – Rapport technique n° 516/1996, Berlin, Technishe Universität, 1996.

[16] Hoogeveen (J. A.), Lenstra (J.) et Veltman (B.). – Three, four, five, six or the complexity of scheduling with communication delays. *Operations Research Letters*, vol. 16, 1994, pp. 129–137.

[17] Joerg (C.). – *The Cilk system for parallel multithreaded computing.* – Thèse de PhD, Massachussets Institute of Technology, january 1996.

[18] Jr. (C. E.) et Graham (R.). – Optimal scheduling for two-processor systems. *Acta Informatica*, no1, 1972, pp. 200–213.

[19] Karp (R. M.), Luby (M.) et auf der Heide (F. M.). – Efficient PRAM Simulation on a Distributed Memory Machine. *Algorithmica*, vol. 16, 1996, pp. 517–542.

[20] Kruskal (C. P.), Rudolph (L.) et Snir (M.). – A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, vol. 71, 1990, pp. 95–132.

[21] Lam (S.) et Sehti (R.). – Worst case analysis of two scheduling algorithms. *SIAM J. Computing*, vol. 6, n° 3, 1977, pp. 518–538.

[22] Lenstra (J.) et Rinnooy Kan (A.). – Complexity of scheduling under precedence constraints. *Operations Research*, vol. 26, 1978, pp. 22–35.

[23] Lenstra (J.), Shmoys (D.) et Tardos (.). – Approximation algorithms for scheduling unrelated parallel machines. *Math. Programming*, vol. 46, 1990, pp. 259–271.

[24] Lenstra (J.), Veldhorst (M.) et Veltman (B.). – The complexity of scheduling trees with communication delays. *Journal of Algorithms*, vol. 20, n° 1, 1996, pp. 157–173.

[25] Lenstra (J. K.) et Shmoys (D. B.). – Computing near-optimal schedules. *In : Scheduling, Theory and its Applications*, éd. par Chrétienne (P.) et al., pp. 1–14. – J. Wiley, 1995.

[26] Munier (A.) et König (J.-C.). – A heuristic for a scheduling problem with communication delays, A paraître dans Operations Research.

[27] Picouleau (C.). – *Etude des Problèmes d'Optimisation dans les Systèmes Distribués.* – Thèse de PhD, Université Paris VI, France, 1992.

[28] Ranade (A. G.). – How to emulate shared memory. *In : Proceedings 28th Annual Symposium on Foundations of Computer Science.* pp. 185–192. – IEEE.

[29] Shmoys (D. B.), Wein (J.) et Williamson (P.). – Scheduling parallel machines on-line. *SIAM Journal on Computing*, vol. 24, n° 6, 1995, pp. 1313–1331.

[30] Valiant (L. G.). – A Bridging Model For Parallel Computation. *Communications of the ACM*, vol. 33, n° 8, 1990, pp. 103–111.

[31] Valiant (L. G.). – General purpose parallel architectures. *In : Algorithms and Complexity*, éd. par van Leuwen (J.), pp. 944–971. – Elsevier, 1990.

[32] Varvarigou (T.), Roychowdhury (V.) et Kailath (T.). – Scheduling in and out forests in the presence of communication delays. *In : Proceedings International Parallel Processing Symposium*, pp. 222–229. – Newport Beach, CA, 1993.

[33] Veltman (B.). – *Multiprocessor Scheduling with Communication Delays.* – Thèse de PhD, CWI-Amsterdam, Holland, 1993.

[34] Veltman (B.), Lageweg (B.) et Lenstra (J.). – Multiprocessor scheduling with communication delays. *Parallel Computing*, vol. 16, 1990, pp. 173–182.

textextex

Chapitre 6

Technologie et objectifs des supports d'exécution pour la régulation

Jacques Briat (IMAG-LMC), Jean-Marc Geib, Jean-François Méhaut (LIFL)

6.1 Introduction

Si les performances des architectures parallèles et distribuées ne cessent de croître, tant sur le plan de la puissance de calcul (microprocesseurs) que sur le plan des débits de communication (réseaux), force est de constater que les applications n'en tirent que trop rarement profit. Cette caractéristique est particulièrement vraie pour les applications *irrégulières* [52] qui demeurent extrêmement difficiles à paralléliser efficacement avec les environnements de programmation disponibles aujourd'hui. Si l'intérêt des processus légers est maintenant largement connu dans un contexte mono-machine (mono ou multiprocesseurs), leur utilisation dans un contexte parallèle et distribué (plusieurs machines) apporte des réponses innovantes dans chacun des registres suivants :

Modèle de programmation La parallélisation d'une application peut amener le programmeur à solliciter la création dynamique d'un grand nombre de tâches dont le graphe de précédence n'est pas connu à l'avance.

Les processus légers constituent une base solide pour réaliser des modèles de programmation favorisant l'expressivité d'un parallélisme très dynamique.

C'est dans cet esprit que les processus légers ont été utilisés pour implanter un compilateur ADA [43] ou des extensions parallèles de C++[9].

Support exécutif La notion de processus ¡¡ lourd ¿¿, qui représente l'unité d'exécution de la plupart des environnements de programmation actuels, fournit une forme de parallélisme en totale inadéquation avec les exigences des applications massivement parallèles. Cette inadéquation, due à la différence existant entre la *granularité* des traitements à effectuer (activités) et les ressources consommées par leur prise en charge, se traduit par l'incapacité des environnements actuels à assurer une prise en charge efficace des applications fortement parallèles.

Les qualités intrinsèques des processus légers permettent de bâtir des environnements de programmation efficaces autorisant la prise en charge d'un parallélisme de granularité plus fine que celle des processus lourds. Les mécanismes offerts aux programmeurs doivent rester simples et souples d'utilisation.

Équilibrage de charge L'exécution efficace sur une architecture parallèle nécessite l'utilisation de techniques d'équilibrage de charge lorsque l'application n'est pas totalement régulière. Un des résultats du projet Stratagème [52] a été mettre en évidence que les applications irrégulières sont classifiables en différentes familles, chacune ayant des besoins différents en terme d'équilibrage de charge. Parfois, cet équilibrage doit être effectué de manière dynamique.

La souplesse d'utilisation des processus légers et l'efficacité des primitives associées permettent de mettre à disposition du programmeur des mécanismes basiques et ¡¡ génériques ¿¿ nécessaires pour concevoir des politiques de régulation de charge. Le placement et la migration de processus légers sont deux opérateurs sur lesquels pourront s'appuyer des régulateurs.

L'objectif de ce papier est de décrire les solutions apportées dans ces trois domaines par les environnements Athapascan-0b et PM^2. Les principaux choix de conception et de réalisation seront très largement justifiés.

La première section de cet article présente d'abord les motivations majeures pour s'appuyer sur la notion de processus léger dans le cadre de la conception et de l'exécution d'applications parallèles. Nous décrirons ensuite les environnements PM^2 (Parallel Multithreaded Machine) et Athapascan-0b et justifierons pour chacun des environnements les choix de conception et de réalisation. Les points originaux en seront soulignés. L'efficacité de la plupart des paradigmes présentés sera illustrée par des mesures de performances.

6.2 Processus légers et calcul parallèle distribué

Si les processus légers ont convaincu dans le contexte des systèmes d'exploitation, c'est très certainement par les capacités qu'ils possèdent à recouvrir les opérations d'entrées-sorties et ainsi à rentabiliser la ressource processeur. Dans le contexte du parallélisme, les processus légers sont exploités avec la même intention de recouvrement, mais appliqué aux communications qui sont un facteur déterminant pour l'obtention de bonnes performances. Nous présentons dans le paragraphe suivant les possibilités de recouvrement qu'offrent les processus légers.

6.2.1 Recouvrement communications/calculs

Dans un environnement distribué, les différents processus d'une application parallèle doivent fréquemment communiquer entre eux pour échanger de l'information. Ces communications, en l'absence de mémoire commune, passent fréquemment par un dispositif d'acheminement de messages (réseau de communication) entre les différentes machines. Étant donné que les différents processus d'une application peuvent difficilement s'exécuter en mode totalement synchronisé, lorsqu'un processus de l'application attend un message en provenance du réseau, le message peut ne pas être disponible immédiatement. De façon symétrique, lorsque un processus souhaite émettre un message, le réseau peut ne pas être disponible. Dans ce cas, le processus est bloqué par le système et le processeur devient inutilisé si aucun autre processus n'est prêt à s'exécuter localement.

Une solution classique à ce problème est la multiprogrammation des processus c'est à dire d'utiliser plusieurs processus légers concurrents au sein de chaque processus lourd. En procédant de la sorte, on diminue les risques d'apparition de périodes d'inactivité des processeurs : la plupart du temps, lorsqu'un processus léger se met en attente d'un message, il existe un autre processus léger prêt à poursuivre ses calculs et à occuper immédiatement le processeur.

Une solution complémentaire est de laisser le processus communicant décider s'il attend ou tente une autre opération. Il s'agit donc le programmeur décider du recouvrement calcul-communication. Le problème alors est de définir précisement les opérateurs appropriés à l'expression de tels recouvrements.

6.2.2 Virtualitation de l'architecture

Aujourd'hui, le ¡¡ multithreading ¿¿ et les noyaux de processus légers sont capables de gérer plusieurs centaines de flots d'exécution concurrents au sein d'un processus lourd. De ce fait, il est maintenant possible de concevoir des applications distribuées manipulant des dizaines de milliers d'activités concurrentes et de les supporter efficacement pendant l'exécution..

Cette caractéristique est très importante, car elle permet la conception d'applications massivement parallèles capables, dans une certaine mesure, d'appréhender un degré de parallélisme variable. Les processus légers peuvent être considérés comme des processeurs virtuels capables de prendre en charge les tâches des applications parallèles. Ainsi, le programmeur dispose d'une machine virtuelle contenant plus de 10 000 processeurs. Il peut donc focaliser son travail de conception sur le découpage logique de ses applications, en générant de nouvelles tâches à chaque fois que l'algorithme le permet.

Cette virtualisation de l'architecture simplifie fortement les deux étapes de développement des applications, à savoir la conception (qui tend à devenir plus générale) et la réalisation (qui doit se rapprocher le plus possible de la spécification de l'algorithme parallèle).

Bien évidemment, cela nécessite un support d'exécution capable de répartir, voire même de réguler un ensemble de processus légers sur une architecture distribuée. Nous verrons, dans les sections consacrées à PM^2 6.3.1 et à ATHAPASCAN-0B 6.4, les propositions faites à ce propos afin d'éliminer le fossé trop souvent constaté entre la conception d'une application parallèle et sa mise en œuvre.

6.2.3 Comment intégrer processus légers et distribution ?

En dépit des qualités que nous venons d'évoquer, les processus légers demeurent encore relativement peu utilisés en contexte distribué. La principale raison réside dans la difficulté de concilier les deux concepts – *Multithreading et Distribution* – tant sur le plan du modèle de calcul que sur le plan de sa mise en œuvre pratique.

C'est pourquoi un certain nombre de recherches ont été — et sont encore — menées sur le thème de la conception et la réalisation d'environnements de programmation parallèle distribués basés sur les processus légers. Les environnements résultant de ces recherches proposent tous un modèle de programmation centré sur la notion de processus léger mais se distinguent par les fonctionnalités d'interaction entre processus qu'ils fournissent. Il existe plusieurs classes d'interaction, que l'on assimile généralement au ¡¡ modèle de programmation ¿¿ proposé par les environnements concernés. Voici les trois approches les plus répandues :

Processus légers distants Cette approche propose d'étendre, de façon plus ou moins transparente, le contexte d'utilisation des primitives ¡¡ classiques ¿¿ de gestion de processus légers (`pthread_create`, `pthread_mutex_unlock`, etc.) à un cadre global à l'architecture. Dans l'idéal, une application distribuée utilisant les processus légers se programme donc de manière analogue à une application *multithreadée* traditionnelle. Toutefois, il convient de nuancer fortement cette caractéristique d'un point de vue pratique, car la réalisation de telles fonctionalités est difficile et les performances obtenues assez catastrophiques. C'est pourquoi seules quelques approches restreignant ce modèle à l'extension de quelques fonctionnalités ont effectivement été proposées. Des environnements tels que RThreads [19] ou, dans une moindre mesure, Chant [34] ont suivi cette voie.

Processus communicants Dans les environnements à gros grain, l'approche ¡¡

processus communicants ¿¿ demeure la plus utilisée. Aussi n'est-il pas surprenant qu'une grande proportion des environnements distribués à grain fin adopte également une telle approche. Dans ce cas, chaque processus léger peut potentiellement communiquer avec n'importe lequel de ses homologues, pourvu qu'il ait connaissance de son interlocuteur. La désignation de celui-ci peut être directe (identificateur du destinataire) ou indirecte (identificateur d'un port, boite à lettre, canal, etc..). Dans la plupart des environnements de ce type, les communications peuvent non bloquantes. PVC [15], TPVM [20], Athapascan-0b [31], DTMS [14] ou encore Chant [34] en sont des exemples représentatifs.

Relations client/serveur Certaines approches proposent un modèle d'exécution fondé sur la prise en charge d'appels de procédures (potentiellement à distance) par des processus légers. Ce type d'interaction introduit des relations clients/serveurs entre les différents processus légers d'une application et s'avère être particulièrement adapté aux applications issues de la parallélisation d'un algorithme séquentiel. Les environnements Nexus [22], Athapascan-0a [11], DTS [5] ou encore PM^2 [44] sont très représentatifs de cette approche.

Les environnements Athapascan-0b et PM^2 ont une vocation affichée pour le support efficace des applications parallèles comportant de nombreux flots d'exécution asynchrones potentiellement parallèles. Ils diffèrent cependant, soit par le modèle de programmation qu'ils défendent, soit par l'étendue des fonctionnalités qu'ils fournissent, soit par leur adéquation à un domaine d'application particulier, soit encore par les choix technologiques de mise en œuvre.

6.3 PM^2

La conception de l'environnement de programmation PM^2 [18, 44] s'inscrit dans le cadre du projet Espace, qui vise à établir un cadre méthodologique et exécutif pour la conception des applications parallèles et leur support sur architectures distribuées. L'objectif principal du projet est ambitieux : il s'agit de fournir aux concepteurs d'applications un environnement permettant 1°) d'exprimer simplement le parallélisme intrinsèque à une application et 2°) de supporter efficacement (et de façon transparente) son exécution sur une architecture distribuée quelconque.

L'environnement PM^2, qui constitue le **noyau commun** du projet Espace, est présenté dans les sections suivantes. Nous commencerons d'abord par décrire le modèle de programmation prôné par PM^2 et nous détaillerons ensuite l'architecture interne de la plateforme.

6.3.1 Le modèle de programmation PM^2

L'environnement PM^2 propose un modèle de programmation basé sur l'utilisation de processus légers pour la prise en charge des tâches parallèles d'une application. Dans son principe, une *configuration* PM^2 consiste donc en un ensemble de processus légers répartis sur les différents nœuds[1] de l'architecture.

De manière à se rapprocher le plus possible de la notion de *virtualisation totale*, le modèle de programmation PM^2 privilégie trois concepts centraux qui sont la *virtualisation* des processeurs, la *concurrence* des activités et la *mobilité* des activités:

Virtualisation La conception d'une application parallèle comportant un parallélisme massif et irrégulier ne doit pas être guidée par des caractéristiques telles que le nombre de processeurs disponibles sur l'architecture cible. Au contraire, elle doit se focaliser sur l'expression du parallélisme inhérent au problème traité, en *virtualisant* la machine sous-jacente. De même, sa réalisation doit être le reflet le plus fidèle possible de cette spécification, ce qui requiert que le support d'exécution soit capable de prendre en charge un nombre important de flots d'exécution ¡¡ virtuellement parallèles ¿¿. Cette caractéristique, en plus de permettre au système de gérer au mieux les ressources disponibles, est fondamentale en ce qui concerne la *portabilité* des applications sur différentes configurations distribuées.

Le modèle de programmation PM^2 propose le découpage parallèle des applications à l'aide des mécanismes d'*appel de procédure à distance léger* et de *clonage léger*, qui assurent une expression naturelle du parallélisme tout en rendant transparente la gestion des processus légers sous-jacents.

Concurrence La virtualisation décrite précédemment ne peut être satisfaisante que si toutes les tâches éligibles à un instant donné *progressent simultanément*. Cette propriété, qui est très naturelle dans un système d'exploitation classique (par exemple dans UNIX), l'est tout autant dans une application susceptible de comporter des tâches interactives, ou même des tâches devant s'acquitter de travaux périodiques, tels que les agents d'information de certains régulateurs de charge. En l'absence d'une telle garantie, c'est-à-dire avec un système favorisant arbitrairement l'exécution d'une tâche plutôt qu'une autre, l'*équité* de l'accès aux ressources n'est pas respectée et des situations de famine peuvent se produire.

Le modèle d'exécution PM^2 repose sur un partitionnement de l'ensemble des processus légers sur les processeurs de l'architecture, avec un ordonnancement

[1] Un *nœud* est un sous-ensemble de l'architecture ne partageant pas de mémoire avec le reste. Il en résulte qu'un nœud peut être monoprocesseur ou multiprocesseurs, mais qu'il ne communique avec les autres nœuds que par un réseau de communication.

préemptif assurant une exécution réellement concurrente au sein de chacune des parties.

Mobilité des activités Le rôle de PM^2 n'étant pas de fournir un support d'exécution régulant automatiquement la charge mais au contraire de permettre la construction aisée de régulateurs de charge, les différentes fonctionnalités permettant la création de processus légers sont toutes à ¡¡ localisation explicite ¿¿. Un régulateur de charge construit au-dessus de PM^2 peut alors, en utilisant des informations fournies par ce dernier, contrôler la distribution des traitements en effectuant un *placement* de tâches *ad-hoc*. Malheureusement, une telle stratégie n'est pas suffisante pour garantir un bon équilibrage de la charge si l'application présente un comportement dynamique irrégulier, car des déséquilibres peuvent apparaître lors de la terminaison de certaines tâches. Seul un mécanisme de *migration* de tâches peut aider à rétablir l'équilibre dans ce cas.

Le modèle d'exécution PM^2 permet la *mobilité*, c'est-à-dire le déplacement, d'un nœud à un autre, d'une partie des tâches s'exécutant à un instant donné. Pour ce faire, l'environnement PM^2 fournit un ensemble de fonctionnalités permettant aux applications de commander la *migration* de processus légers pendant leur exécution.

6.3.1.a Opérateurs de découpage parallèle

Le modèle de programmation PM^2 est basé sur un découpage des applications en un ensemble de tâches et sur la prise en charge de chacune de ces tâches par un processus léger de l'environnement. Ce découpage des applications est réalisé en utilisant soit le mécanisme d'*appel de procédure à distance léger*, soit le mécanisme de *clonage léger*.

6.3.1.a.1 L'appel de procédure à distance léger L'appel de procédure à distance est un concept destiné à rendre *transparente* l'utilisation de mécanismes de type *client/serveur*. L'objectif est de cacher au *client* les aspects relatifs à la réalisation effective du *serveur*. Pour ce faire, le *client* effectue un appel local ¡¡ classique ¿¿ à une procédure spéciale. Cette procédure renferme des mécanismes capables de transformer cet appel local en une requête qui sera émise vers un *serveur* apte à rendre le service en question. Une fois le service rendu, le résultat est émis vers la procédure spéciale qui, après avoir extrait les données reçues, retourne le résultat en utilisant les conventions de passage de paramètres du *client*.

Dans PM^2, le mécanisme utilisé est un mécanisme d'appel de procédure à distance **léger** (encore appelé *LRPC* pour *Lightweight Remote Procedure Call*) et diffère sensiblement du précédent. Il consiste en la possibilité qu'a un processus léger de déclencher l'exécution d'une fonction se trouvant sur un *nœud* distant, celle-ci étant prise en charge par un nouveau processus léger créé pour l'occasion (figure 6.1).

L'appel de procédure à distance ¡¡ léger ¿¿ (LRPC) consiste en la création d'un processus léger (p_2) dans un nœud spécifié pour prendre en charge l'exécution d'un service. Lorsque l'exécution du code du service est terminée, le résultat est envoyé au processus appelant (p_1) (figure 6.1).

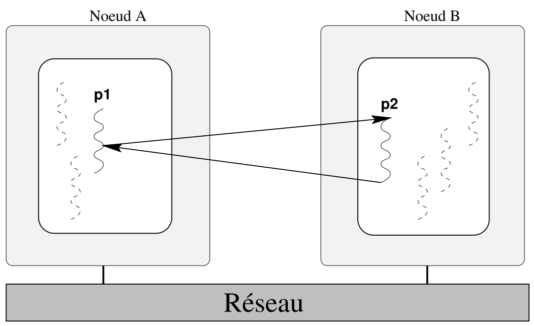


Figure 6.1 : Fonctionnement d'un LRPC

Lorsqu'un processus léger (p_1 sur la figure) effectue un appel de procédure à distance léger, il fournit principalement un nom de fonction à exécuter (on parle de *service*), des arguments et un emplacement mémoire pour le stockage des résultats. De manière interne, ces paramètres sont alors *empaquetés* dans un message, puis celui-ci est envoyé vers le nœud cible. Une fois le message reçu par le nœud, le service à exécuter est identifié et la création d'un nouveau processus léger (p_2 sur la figure) est déclenchée. Ce processus extrait (*désempaquetage*) alors les paramètres du message et commence l'exécution du service. Lorsque l'exécution est terminée, le résultat est *empaqueté* dans un deuxième message, celui-ci est envoyé vers le nœud source et le processus (p_2) disparaît. Une fois le message reçu par le nœud source, le résultat en est extrait et est rangé à l'emplacement mémoire spécifié lors de l'appel. Le processus appelant peut alors — nous verrons dans quelle circonstance par la suite — utiliser le résultat de son appel.

Notons que la distinction *client/serveur* n'a de sens que pour une instance de LRPC particulière, car d'une manière générale les différents processus légers d'une application PM^2 peuvent être à la fois *serveur* d'un LRPC et *clients* de plusieurs LRPC. Trois variantes de l'appel de procédure à distance léger sont disponibles dans l'environnement PM^2 :

Appels synchrones L'appel de procédure à distance léger **synchrone** est le mécanisme dont la sémantique est la plus proche de l'appel de procédure à distance classique. Lorsqu'un tel appel est effectué, le processus appelant est immédiatement *bloqué* par le système (*i.e.* par PM^2) jusqu'à la disponibilité du résultat.

Appels à attente différée L'appel de procédure à distance léger **à attente différée** constitue un opérateur de décomposition parallèle majeur dans l'environnement PM^2. Son fonctionnement est comparable à celui d'un appel *synchrone* pour lequel les opérations d'envoi des paramètres et de réception du résultat seraient séparées. Son utilisation s'effectue donc en deux temps.

Dans un premier temps, le processus appelant effectue l'appel en spécifiant les mêmes paramètres que dans le cas d'un appel synchrone (nom de service, arguments et adresse de stockage du résultat) et en y ajoutant une référence sur une variable d'un type particulier qui lui servira de *clé* par la suite. Cette étape donne lieu à la création d'un nouveau processus léger (pour exécuter le service), mais ne ¡¡ bloque ¿¿ pas le processus appelant qui peut donc librement continuer son exécution.

Dans un second temps, lorsque le processus appelant désire accéder au résultat de l'appel, celui-ci doit effectuer une opération d'attente en spécifiant la clé qu'il a obtenue lors de l'appel correspondant. Cela conduit alors au blocage de ce processus jusqu'à disponibilité du résultat.

Appels asynchrones La dernière forme d'appel de procédure à distance léger proposée par l'environnement PM^2 est dite ¡¡ **asynchrone** ¿¿, car elle offre le moyen de créer un nouveau flot d'exécution indépendant du processus appelant. Ce mécanisme, qui s'apparente plutôt à un ¡¡ déclenchement de traitement à distance ¿¿, diffère des deux précédents par le fait qu'il ne comporte pas de phase de retour de résultat. En ce sens, il est très similaire au mécanisme de RSR (*Remote Service Request*) proposé par l'environnement Nexus [21].

6.3.1.a.2 Le clonage léger Le mécanisme de *clonage léger* est une opération permettant d'engendrer, de manière ¡¡ temporaire ¿¿, un certain nombre de processus *clones* à partir d'un processus léger initial. Deux processus *clones* l'un de l'autre possèdent, à l'instant suivant l'opération dite de *séparation*, un contexte d'exécution strictement identique. Entre autres, leurs compteurs ordinaux référencent la même adresse d'instruction machine et, surtout, leurs piles d'exécution contiennent les mêmes données, qui sont héritées du processus ayant servi de *modèle* à l'opération de clonage.

En réalité, les différents processus issus d'une même opération de clonage possèdent chacun une donnée qui leur est propre : un numéro de rang compris entre zéro et le nombre de clones moins un. En consultant la valeur de ce rang, il est alors possible à chaque clone d'exécuter un traitement différent des autres.

L'opération de *séparation* (qui déclenche le clonage) est comparable à l'opération *fork()* disponible sur les processus UNIX, à ceci près qu'elle s'applique à des processus légers et qu'elle est utilisée de façon beaucoup plus contrôlée. En effet, une deuxième opération, la *fusion*, doit être appelée par tous les clones d'une même génération pour mettre fin à une opération de clonage. Cette opération, effectuée de manière asynchrone par les différents clones, aura pour effet de ne laisser survivre que le dernier clone ayant effectué cette opération (figure 6.2).

Lorsque qu'un processus exécute une opération de séparation, il est ¡¡ cloné ¿¿ en plusieurs exemplaires. Chaque exemplaire peut alors vivre sa propre vie, jusqu'à l'opération de fusion, exécutée par chaque clone de manière asynchrone. Le dernier clone exécutant cette opération sera le seul survivant (figure 6.2).

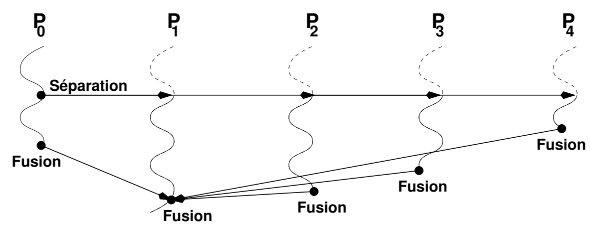


Figure 6.2 : Schéma conceptuel de l'opération de clonage.

L'objectif du mécanisme de clonage étant de permettre le découpage d'un calcul en sous-calculs de manière symétrique, il est essentiel que les différents clones d'une même génération puissent apporter chacun leur contribution au calcul lors de l'opération de *fusion*. Voici donc le détail du déroulement typique d'une opération de clonage :

1. Lorsqu'un processus désire se subdiviser en n exemplaires, il exécute une opération de **séparation** en indiquant le nombre de clones (en plus de lui-même) qu'il faut fabriquer, la liste des nœuds sur lesquels il faut les placer, le type du résultat qu'ils vont retourner lors de l'opération de fusion et l'adresse d'un tableau local pouvant stocker les résultats des différents clones.

2. Le bloc d'instructions suivant une opération de clonage est donc exécuté par tous les clones. Cependant, en fonction de leur *rang*, ceux-ci peuvent manipuler des données différentes (par exemple en utilisant la pseudo-variable *rang* pour indicer des tableaux), ou même exécuter des instructions différentes.

3. Au bout d'un temps fini, chaque clone doit exécuter une opération de **fusion**, en fournissant la valeur du résultat de son calcul. L'appel à cette opération de fusion, pour des raisons de lisibilité, ne doit apparaître qu'une seule fois dans le code exécuté par les clones, de préférence dans le même bloc syntaxique que l'instruction de séparation correspondante.

4. L'instruction suivant immédiatement l'opération de fusion n'est exécutée que par un seul clone, qui constitue le seul survivant de l'opération de fusion. Ce processus peut alors accéder au tableau contenant les différents résultats communiqués par les clones.

L'utilisation du mécanisme de *clonage léger* pour exprimer le parallélisme inhérent à une application exhibe plusieurs avantages :

héritage de contexte Lors d'une opération de séparation, tous les ¡¡ nouveaux ¿¿ processus héritent de l'intégralité du contexte du processus ¡¡ original¿¿. Non seulement cette caractéristique évite au programmeur d'expliciter lui-même le transfert des informations de contexte utiles, mais en plus elle permet à chacun des clones d'une même génération de pouvoir ¡¡ remplacer ¿¿ le processus original lors de l'opération de fusion.

gestion des ressources Lors d'une opération de fusion, étant donné que tous les clones sont ¡¡ égaux ¿¿ devant cette opération, le système peut pleinement exploiter cette propriété en faisant en sorte que le clone effectuant le traitement le plus long constitue le processus ¡¡ survivant ¿¿ à l'opération. Il est donc possible d'éviter le blocage à durée indéterminée[2] de processus dans le système, ce qui constitue un réel progrès en ce qui concerne la gestion des ressources mémoire de la machine.

6.3.1.a.3 Opérateurs de découpage et virtualisation Il est important de remarquer que l'utilisation d'opérateurs de découpage parallèle (plutôt que des envois de messages par exemple) va permettre une prise en charge efficace de l'exécution des processus d'une application même si ces derniers seront susceptibles de voyager dynamiquement entre les processeurs de l'architecture. En effet, dans une application utilisant des LRPC ou des clonages, toutes les références sur un processus sont connues et gérées par le système (et uniquement par lui). De plus certains processus ne sont référencés par aucun autre, ce qui autorisera leur migration de façon transparente. Ces caractéristiques font que l'implantation d'un mécanisme de migration de processus dans ce contexte est une possibilité assez attrayante qui, de plus, peut être réalisée à un coût raisonnable. En fait, les opérateurs de découpage parallèle offrent un modèle d'exécution très peu sensible au changement de localisation dynamique des processus, ce qui constitue une caractéristique majeure dans l'optique d'une virtualisation réaliste de l'architecture.

[2]Par contre, des blocages dus à l'attente d'opérations d'accès mémoire à distance semblent difficilement évitables.

6.3.1.b La migration de processus légers

Le support d'exécution PM^2 fournit quelques fonctionnalités permettant à une application[3] de provoquer la migration d'un processus léger (ou d'un groupe de processus légers) d'un *nœud* à un autre pendant son exécution. Cette opération, déclenchée par le simple appel à une primitive de la bibliothèque PM^2, provoque l'arrêt instantané de l'exécution du processus désigné, son transfert sur le nœud destinataire et la reprise de son exécution au point où elle avait été interrompue (figure 6.3).

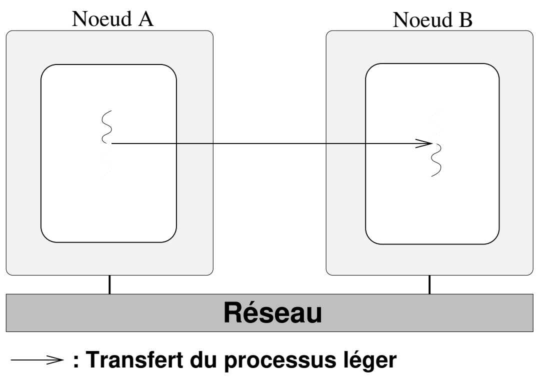


Figure 6.3 : Migration d'un processus léger.

Le déroulement de l'opération est (normalement) transparent pour le processus léger déplacé et survient de façon asynchrone pour ce dernier. Le processus migré n'est donc pas tenu d'exécuter de quelconques traitements périodiques destinés à sauver son état ou à effectuer son ¡¡ auto-migration ¿¿. Ce dernier point est important car, contrairement à ce qui se passe dans d'autres systèmes [40], la migration d'un processus peut réellement être initiée par un autre processus (par exemple un processus chargé d'équilibrer la charge entre les nœuds) et ne nécessite pas une *coopération active* de sa part.

De par sa nature, la migration d'un processus léger provoque uniquement le transfert de son contexte local (pile d'exécution, variables locales). Par conséquent, après sa migration, un processus léger se trouve plongé dans un nouvel environnement global (variables globales du processus, variables ¡¡ système ¿¿ telles que les fichiers ouverts, etc.). Cette caractéristique est toutefois sans conséquence pour la plupart des applications développées avec PM^2, car le modèle de programmation ne préconise ni l'accès ¡¡ direct ¿¿ aux variables globales d'un processus, ni l'accès (sauf sous contrôle) aux primitives de la bibliothèque C standard. Le cas échéant, une

[3]Par exemple un régulateur de charge, qui est une application comme une autre du point de vue du support.

application peut faire usage de primitives permettant de *protéger* temporairement l'exécution d'un processus léger contre son éventuelle migration intempestive.

6.3.1.b.1 Migration et virtualisation Une application PM^2 est composée d'un ensemble de processus légers répartis sur plusieurs nœuds. Cet ensemble, typiquement de cardinalité importante (*i.e.* composé de centaines de processus légers), peut être caractérisé par un comportement extrêmement dynamique dans le temps. En effet, non seulement le nombre global de processus dans le système peut fluctuer à des fréquences élevées (applications irrégulières à parallélisme fin), mais l'état de ces processus peut également osciller entre *migrable* et *non-migrable*.

En n'oubliant pas que, dans PM^2, l'objectif de la migration est de fournir un outil permettant de rééquilibrer la charge à un instant donné, la question que nous nous posons maintenant est : ¡¡ *Comment, dans un tel contexte, fournir une interface aux fonctionnalités de migration qui soit réellement utilisable ?* ¿¿

En fait, le problème principal consiste à trouver un moyen simple de permettre au programmeur de désigner, à un instant donné et sur un nœud donné (supposé surchargé), un ensemble de processus à migrer sur un autre nœud. Ce problème est difficile pour plusieurs raisons :

- Nous avons vu, dans une section précédente, que le mécanisme d'*appel de procédure à distance léger* est intéressant parce qu'il évite au programmeur de manipuler des identifiants de processus. Cette caractéristique, essentielle à une **virtualisation** réaliste de l'architecture, doit être conservée en présence d'un mécanisme de migration de processus. Il faut donc fournir au programmeur le moyen de désigner des processus ¡¡ à migrer ¿¿ tout en lui évitant de gérer lui-même la liste des processus dans le système.

- Si tous les processus légers d'une application ¡¡ étaient égaux devant la migration ¿¿, alors il suffirait de fournir une primitive permettant au programmeur de spécifier une simple *quantité* de processus à migrer et le système s'occuperait du reste. Hélas, la réalité est toute autre, car les processus présentent chacun des caractéristiques différentes (priorité, type de service exécuté, taille mémoire occupée, etc.). C'est pourquoi il faut permettre au programmeur d'**inspecter**, à un moment donné, les processus migrables d'un nœud et lui permettre de choisir, en fonction de leurs caractéristiques, les meilleurs candidats à la migration.

- Dans un environnement où les différents processus légers d'un nœud s'exécutent de façon concurrente (voire de façon réellement parallèle sur machines multiprocesseurs[4]), la désignation d'un processus léger est une opération *hasardeuse* par nature.

[4]Nous verrons dans la section consacrée à la réalisation de PM^2 que cette fonctionnalité n'est toutefois pas supportée dans la version actuelle de PM^2

En effet, pendant le temps s'écoulant entre la récupération du *nom* d'un processus et l'appel effectif à une primitive de manipulation (par exemple une opération de migration) de ce processus, le processus en question peut soit avoir changé d'état (il était migrable mais il ne l'est plus), soit même avoir disparu (son exécution s'est terminée).

6.3.2 Réalisation et performances

La réalisation de l'environnement PM^2 s'appuie, comme c'est le cas pour la plupart des environnements fondés sur une exploitation du parallélisme à grain fin en contexte distribué, sur une bibliothèque assurant la gestion des processus légers [45] et sur une bibliothèque assurant la gestion des communications (figure 6.4).

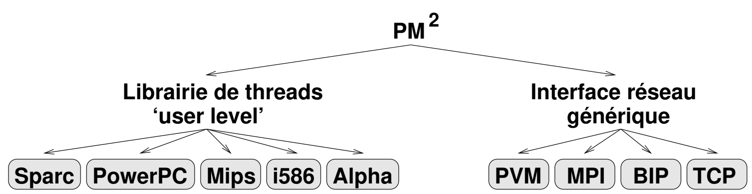


Figure 6.4 : Architecture de l'environnement PM^2

6.3.2.a Gestion des processus légers

Contrairement aux environnements cités en section 6.2.3, les exigences de l'environnement PM^2 vis-à-vis des fonctionnalités des processus légers sont assez importantes (ordonnancement préemptif avec priorités, etc.). Certaines de ces fonctionnalités, comme la ¡¡ migration ¿¿[5] de processus léger, ne sont ni disponibles dans les bibliothèques de processus légers actuelles ni réalisables au-dessus. C'est pourquoi nous avons conçu et développé notre propre bibliothèque de processus légers — nommée Marcel — et avons montré qu'il était possible, en préservant un degré de portabilité élevé, de fournir des fonctionnalités évoluées (piles extensibles, processus migrables) tout en exhibant des performances de tout premier plan.

Marcel est une bibliothèque de niveau utilisateur dont l'interface est constituée d'un large sous-ensemble des fonctionnalités du standard POSIX et de quelques fonctionnalités supplémentaires permettant l'implantation d'opérations telles que la migration ou le clonage de processus légers.

Un accent particulier a été mis sur l'efficacité des différentes primitives ¡¡ critiques ¿¿ (créations de processus, commutations de contexte) dans l'optique d'une utilisation intensive. Par exemple, le nombre de processus légers maximal ne dépend que

[5] À ce niveau, nous devrions plutôt parler de ¡¡ support pour la migration ¿¿.

de la mémoire disponible sur un nœud, l'allocation des processus utilise un mécanisme de *cache* de pile et l'ordonnanceur des processus garantit une durée *constante* des opérations de changement de contexte quel que soit le nombre de processus légers présents dans la file des processus prêts et le nombre de niveaux de priorités autorisés.

Il est important de noter que la bibliothèque MARCEL n'a pas une vocation aussi générale que les bibliothèques POSIX et que son domaine d'utilisation concerne clairement les applications de calcul scientifique. C'est pourquoi il a été possible de s'affranchir de certaines contraintes d'implantation dans ce cadre précis d'utilisation. Par exemple, les primitives de gestion des signaux ¡¡ à la UNIX ¿¿ ne sont pas disponibles dans cette bibliothèque[6].

Cette dernière remarque explique en partie les performances obtenues par rapport aux bibliothèques POSIX existantes, comme il est synthétisé sur le tableau 6.1. Ce comparatif introduit les bibliothèques *Pthreads* de Frank Mueller [42], de Chris Provenzano [50] et du système Solaris[7] [54]. Pour chacune d'elles, nous avons mesuré le temps d'exécution d'une opération de changement de contexte (moyenne sur 1 000 commutations) et d'une création de processus[8] (moyenne sur 100 créations successives).

Tableau 6.1 : Comparaison de la bibliothèque MARCEL avec les principales bibliothèques (POSIX) existantes.

Sparc5/Solaris 85Mhz				
Opération	MARCEL	Mueller	Provenzano	Solaris
Chgt contexte	$8,838\ \mu$s	$220,232\ \mu$s	$34,366\ \mu$s	$48,196\ \mu$s
Création	$37,630\ \mu$s	$45,193\ \mu$s	$26,791\ \mu$s	$86,838\ \mu$s
Création/Terminaison	$85,252\ \mu$s	$532,201\ \mu$s	$109,161\ \mu$s	$305,937\ \mu$s

Si les résultats présentés ici montrent parfois des différences importantes entre les bibliothèques, il convient néanmoins de les relativiser dans un cadre d'utilisation restreint aux applications de calcul intensif. En effet, certains de ces écarts s'estomperont par exemple avec des applications ne sollicitant pas beaucoup d'opérations de création dynamiques (rendant les mécanismes de cache de pile inutiles) ou encore avec des applications n'utilisant tout simplement qu'un petit nombre de processus légers.

[6]Notons qu'il est cependant possible de les définir dans une sur-couche de MARCEL, car le mécanisme de base nécessaire à une éventuelle gestion de signaux (la ¡¡ déviation ¿¿ de processus) est fourni.

[7]Précisons qu'il s'agit des processus légers de *niveau utilisateur* de Solaris.

[8]Deux cas ont été mesurés : la création asynchrone et le cycle création-exécution d'une fonction vide-terminaison (`pthread_create/pthread_join`).

En revanche, une caractéristique telle que l'efficacité de l'opération de commutation de contexte gardera toujours son importance dans le cas d'un ordonnancement préemptif, quelle que soit l'application visée.

La bibliothèque Marcel est actuellement opérationnelle sur six architectures de processeurs: Sparc, ix86, Alpha, PowerPC et Mips. Son portage de PM^2 sur une nouvelle architecture ne nécessite qu'une (légère) adaptation concernant quelques lignes d'assembleur seulement.

6.3.2.b Gestion des communications

L'efficacité des communications conditionne fortement les performances globales d'un environnement distribué, et PM^2 n'échappe évidemment pas à la règle. Toutefois, et contrairement à sa situation vis-à-vis de Marcel, PM^2 n'a besoin que de fonctionnalités très rudimentaires en ce qui concerne l'accès au réseau. Principalement, il s'agit de disposer de mécanismes d'envoi asynchrone et de réception non-bloquante de messages en mode point à point.

De manière à assurer une portabilité maximale de l'environnement, une *interface générique* commune à toutes les implantations a été définie. Cette interface est réduite à quelques primitives élémentaires facilement portables sur la plupart des bibliothèques de communication existantes. Des fonctionnalités évoluées telles que la gestion de tampons de communication sont bâties au-dessus de cette interface générique. L'efficacité de la plateforme restant la priorité principale, l'implantation de PM^2 peut garantir la transmission de données sans recopie intermédiaire si la bibliothèque sous-jacente le permet. C'est par exemple le cas de bibliothèque BIP (Basic Interface for Parallelism [51]), qui est une interface au réseau Myrinet [4] permettant l'accès direct au matériel (en contexte utilisateur) ne générant donc aucune recopie superflue des données transmises.

L'interface générique sur laquelle s'appuie PM^2 est actuellement disponible sur les bibliothèques de communication PVM [28], MPI [41], TCP et BIP [51]. Les performances comparées de ces quatres implantations sur un réseau de PC connectés par un réseau Myrinet sont détaillées sur la figure 6.5. Les mesures sont données pour un appel asynchrone (sans création de thread) en utilisant successivement les couches de communications PVM, TCP, MPI-BIP et BIP.

Ces mesures montrent non seulement que l'implantation de PM^2 est capable de tirer pleinement profit des capacités d'un réseau rapide tel que Myrinet (10 μs pour un appel de procédure distante sans argument), mais aussi qu'il est essentiel de pouvoir exploiter les interfaces de communication les plus basses et les plus rudimentaires, car elles sont aussi les plus efficace. La figure 6.5 illustre bien ce propos. L'écart entre les performances sur PVM et celles sur TCP provient des recopies intermédiaires effectuées par PVM. Le protocole TCP est lui-même pénalisé par rapport aux couches MPI-BIP (implantation de MPI directement au-dessus de BIP) et BIP à cause du coût des appels système et des recopies au sein du noyau qu'il occasionne.

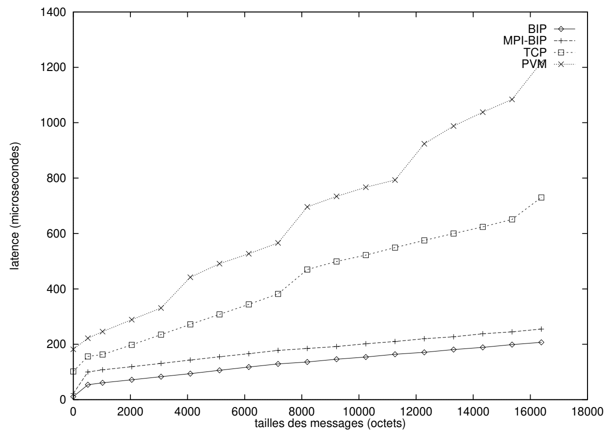


Figure 6.5 : Lattence des appels de procédure à distance légers sur un réseau de PentiumPro 200Mhz connectés par Myrinet.

Enfin, l'implantation MPI-BIP introduit un léger surcoût par rapport à BIP dû à la gestion interne des messages MPI qui exige l'émission de messages de contrôles additionnels.

6.4 Athapascan-0b

Le projet Apache a pour objectif de concevoir Athapascan[9] un environnement de programmation de haut niveau à destination d'applications irrégulières de grande taille à grain fin[10]. De nombreuses tâches de calcul et d'échange de données doivent être mises en œuvre sur un nombre très inférieur de processeurs. Pour atteindre un niveau de grain fin, le mécanisme de découpe doit être peu coûteux. L'utilisation de processus légers (thread) comme support d'exécution d'une tâche de calcul élémentaire fait qu'un algorithme irrégulier s'exprime comme **un réseau dynamique de processus légers communicants**.

Dans cette section, nous présentons Athapascan-0b le **noyau exécutif** de base nécessaire à l'implantation d'un tel environnement. Ciblé vers les applications irrégulières qui nécessitent l'expression d'une évolution dynamique de la découpe des calculs et du partage de données, il doit permettre d'exprimer aisément les décompositions parallèles traditionnelles comme diviser pour régner ou le parallélisme de données. Un tel noyau exécutif est l'association d'un noyau de communication et d'un noyau de gestion de processus légers. Athapascan-0b est donc un applicatif

[9]Athapascan est la famille linguistique à laquelle la langue des Apache appartient.

[10]le terme de grain fin se referre au grain de calcul effectivement exploitable par programmation sur une architecture donnée.

de bas niveau (comparable à Nexus [23] ou PM^2 [44]) dans la mesure où il offre des mécanismes élémentaires de parallélisation mais aucune politique d'utilisation de ceux-ci. Bien qu'utilisable en tant que tel, il est plutot destiné à être la cible de compilateurs (comme Cilk [3], CC++ [10], Cid [47], HPF [35]) ou le support de bibliothèques de haut niveau offrant de telles politiques comme ATHAPASCAN-1 [8] ou Mentat [32].

6.4.1 Objectifs et Principes de conception

Les objectifs d'un noyau exécutif parallèle sont de permettre d'exploiter au mieux les opportunités de parallélisation offertes par les matériels. Nous pouvons distinguer différents types de parallélisme physique exploitable au sein d'une machine parallèle :

- entre nœuds de calcul différents,
- entre les processeurs de calcul ou de communication d'un même nœud,
- entre communications sur le réseau procédant sur des routes disjointes.

Un accroissement d'efficacité peut être obtenue par une exploitation appropriée de chacune de ces formes de parallélisme au sein d'une machine et en choisissant la granularité correcte. L'objectif d'Athapascan étant d'être un noyau universel portable sur toute machine parallèle, il doit proposer une abstraction convenable des machines parallèles actuelles et à venir.

6.4.1.a Un modèle à deux niveaux

Dans cette approche, une machine parallèle générale présente potentiellement toutes les formes de parallélisme : on peut la voir comme un réseau de multiprocesseurs à mémoire partagée ou SMP[11]. Un réseau rapide de stations multiprocesseurs est un exemple d'une telle machine paralléle générique. Toute grappe de stations est aussi une incarnation d'une telle machine car un monoprocesseur peut se voir comme un SMP dégénéré. Les machines parallèles actuelles ne sont que des cas extrèmes de cette machine parallèle générique. Une machine parallèle à mémoire commune comme la SGI Origin2000 ou la Convex Exemplar est une machine réduite à un unique SMP de fort degré de parallélisme alors qu'une machine à mémoire distribuée comme l'IBM SPx ou le Cray T3E est un réseau de monoprocesseurs.

Ces formes de parallélisme ne sont pas du même niveau de grain. Ainsi les interactions au sein d'un nœud sont moins coûteuses que celles entre nœuds. Nous considérons que ces deux niveaux vont perdurer. En effet, l'évolution technologique encourage d'une part l'intégration de plusieurs processeurs dans une même machine (SGI Origin2000, Convex Exemplar, Sun Enterprise, etc.) et, d'autre part,

[11]Symetric Multi-Processors : multiprocesseurs symétriques.

généralise l'interconnexion de telles machines (grappe, réseau local) par des réseaux de débit croissant (ATM, Ether 1Gb/s, Myrinet [4], etc.) ou des composants spécialisés permettant des accès efficaces à la mémoire distante (DEC Memory Channel [30], Bus SCI [36]). Sachant donc que deux niveaux de parallélisme sont exploitables, à l'intérieur d'un SMP via la mémoire partagée et entre SMP via l'échange de messages, nous devons proposer un cadre général dans lequel il est possible d'exploiter ces formes sans introduire de traits trop spécifiques de telle ou telle architecture.

Les programmes parallèles doivent s'organiser selon deux niveaux de découpe: des calculs à gros grain placés sur des nœuds doivent se décomposer en calculs plus élémentaires locaux à un SMP. Ces calculs locaux peuvent coopérer via la mémoire partagée. Les calculs distribués sur plusieurs SMP coopèrent via l'échange de message ou des accès mémoire à distance. La granularité de ces calculs et interactions doit être contrôlée par le programmeur d'applications qui peut exploiter au mieux les propriétés de celles-ci, par des bibliothèques (ATHAPASCAN-1 [8], Cilk [3], Cid [47]) ou par des compilateurs proposant des stratégies de gestion de processus et partage de données tenant compte de ces deux niveaux de grain. C'est le rôle d'un noyau exécutif de proposer les opérateurs de base permettant le contrôle du parallélisme physique.

6.4.1.b Incidence des contraintes d'efficacité

L'efficacité des opérateurs va déterminer la finesse du grain exploitable des différents types de parallélisme. Cependant ces différents types ne sont pas tous directement contrôlables par le programmeur d'une application parallèle. C'est le cas du fonctionnement du réseau d'interconnexion (routage, contrôle de congestion) comme de l'attribution des processeurs aux processus légers. Le programmeur ne peut influer que par l'intermédiaire de la charge de travail en calcul et communication qu'il engendre et, parfois, par la possibilité d'agir sur la stratégie d'ordonnancement. Cependant, les contraintes à respecter pour exploiter au mieux les possibilités logicielles et matérielles sont connues et les opérateurs d'un noyau exécutif doivent s'y conformer.

Interface de communication L'implantation des protocoles standard de communication, organisés en niveaux et conçus pour des réseaux à faible débit procèdent à de nombreuses copies des données à transmettre à travers ces niveaux. Ce phénomène est à l'origine de leur inefficacité sur les réseaux à haut débit (cf. section 6.4.4). Les recherches actuelles sur la conception d'interfaces efficaces comme [48] [51] ont montré la nécessité d'éliminer ces copies. L'interface ATHAPASCAN-0B est conçue de façon à permettre à un noyau de communication d'éviter des copies des données à transmettre et d'utiliser ces même données comme tampon intermédiaire de transfert. Ainsi, les copies pourront être réduites au minimum nécessaire c'est à dire à la lecture en mémoire des données à émettre ou à leur écriture lors d'une réception.

Recouvrement calcul-communication La latence de communication est pour une bonne part liée à la distance physique entre processeurs. Celle-ci restera donc toujours significative dans les réseaux locaux. Il est donc intéressant de recouvrir les délais de communication par du calcul soit par recours à la multiprogrammation des processus légers, soit par recouvrement explicite au sein d'un même processus léger. Dans le premier cas, les opérateurs de communication sont bloquants et un processus léger libère alors le processeur au profit d'un autre. Dans ce cas, le recouvrement est assuré *en aveugle* au gré de la progression relative des calculs et des communications. Quand cela est possible, il est plus efficace de permettre au programmeur ou au compilateur de faire le recouvrement des calculs et des communications au sein d'un processus léger. Ceci nécessite l'utilisation d'opérateurs non bloquants. En ATHAPASCAN-0B le recours à la multiprogrammation ne se fait qu'à partir du moment où le recouvrement au sein d'un processus léger n'est plus assuré.

Un schéma type. Pour faciliter l'expression de tels recouvrements explicites, toutes les opérations impliquant une communication suivront un schéma type. Un opérateur lance l'opération. Cet opérateur est non bloquant pour le processus léger qui l'exécute et termine aussitot que l'action de communication est initiée ou mémorisée par le noyau. Un autre opérateur permet à un processus léger d'attendre la fin de l'opération. Entre ces deux instants, le processus léger peut procéder à toutes autres actions compatibles avec les actions en cours.

De tels opérateurs asynchrones permettent d'exprimer simplement de façon sûre des structures de communication globalement déterministes bien que leurs exécutions soient entrelacées au gré de la disponibilité des ressources [16].

```
Requete_Asynchrone (Action1, Req1);
Requete_Asynchrone (Action2, Req2);
Requete_Asynchrone (Action3, Req3);
<... calcul indépendant des actions...>
Attendre_Fin (Req1);
Attendre_Fin (Req2);
Attendre_Fin (Req3);
 <... calcul dépendant des actions...>
```

Dans l'exemple précédent, un calcul est effectué pendant que des actions de communication à destination d'autres processeurs sont effectuées dans un ordre indéterminé. Par contre, une synchronisation est nécessaire préalablement au calcul suivant qui dépend des échanges de données effectués.

6.4.2 Fonctionnalités

La définition précise des fonctions de ATHAPASCAN-0B doit beaucoup à l'expérience. Une version préliminaire ATHAPASCAN-0a [12] orientée RPC a été utilisée pour une application de dynamique moléculaire [2]. Une extension ATHAPASCAN-0MP[13] incluant la notion de groupe de processus légers a été utilisée pour la bibliothèque de calcul formel parallèle GIVARO [27]. Enfin, une bibliothèque C++ ATHAPASCAN-1 [8] incorporant des stratégies de placement/ordonnancement automatique est en cours de test sur ATHAPASCAN-0B. L'ensemble de ces expériences a contribué à la définition d'un noyau exécutif portable efficace pour machine parallèle.

6.4.2.a Parallélisme de calcul

Une machine parallèle ATHAPASCAN-0B est constitué d'un ensemble de processus lourds ou *nœuds* placés sur les processeurs physiques. Ils sont le support d'exécution des processus légers. Le nombre et le placement de ces nœuds ne fait pas partie du noyau ATHAPASCAN-0B mais plutôt de l'administration de la machine physique.

Les nœuds. Ils sont identifiés par de simples numéros consécutifs. Le même programme parallèle est chargé sur tous les nœuds. Il se compose d'un point d'entrée initial et de procédures exécutables en parallèle appelées **services**. La machine est prête dès que ces nœuds sont activés et qu'un processus léger exécute le point d'entrée initial. Son rôle est d'initialiser les objets globaux au calcul parallèle. Après cette phase, la création de processus légers peut être faite à tout moment **localement** ou **à distance** de façon à satisfaire à la contrainte de dynamicité de construction du réseau de processus légers communicants.

Création et synchronisation locale de processus légers. Un processus léger peut être créé localement par un autre pour exploiter les processeurs d'un nœud SMP ou recouvrir calcul et communication. Ceci est fait en lançant en parallèle une procédure de profil imposé. Ces processus légers peuvent coopérer via leur mémoire commune ce qui nécessite des opérateurs de synchronisation. Les fonctions ATHAPASCAN-0B sont simplement une forme simplifiée du standard POSIX **Pthreads** [37] dont une implantation plus ou moins complète existe sur toutes les machines actuelles.

Définition de service. Un service est une procédure dont l'exécution sur un nœud peut être lançée à distance. C'est aussi une procédure de profil standard sans résultat avec un seul argument. Les services offert par un nœud doivent être enregistrés dans la phase d'initialisation. L'utilisation directe des adresses de procédure devient difficile en milieu hétérogène du fait de l'absence d'un éditeur de lien distribué. Un contrôle explicite de la désignation pallie à ce défaut. Un service est repéré par un simple numéro local à un nœud. Si un service est présent sur plusieurs nœuds, il suffit que les services communs soient enregistrés dans le même ordre pour être numérotés de façon identique selon le principe des *services bien connus* d'INTERNET.

Création à distance de processus légers. Tout processus léger peut initier un calcul à distance. Conformément au schéma type d'action communicantes, ceci se fait en deux temps : initiation et attente de terminaison. A l'initiation sont fournis les numéros des nœuds et services distants et l'argument à transférer.

```
a0RemoteThreadCreate( a0tNode, a0tService,
                      ...Règle d'ordonnacement, priorité,...
                      a0tRequest, a0tFormat, a0tData)
```

Cette opération est non bloquante et un processus léger peut donc initier immédiatement une autre création sur un autre nœud. L'action de création est **localement** terminée dès que les informations nécessaires ont été émises (ou mémorisées par le noyau). Jusqu'à cet instant, les variables contenant numéros de nœud et numéro de service et argument ne sont pas modifiables. Ce moment peut être attendu par :

```
a0WaitRequest( a0tRequest)
```

Cette terminaison ne signifie pas que le processus léger distant ait été créé ou même que la requête de création soit arrivée.

Le choix de cet asynchronisme total permet de masquer les délais de communication et donc de distribuer avec le maximum de rapidité un ensemble de calculs sur plusieurs nœuds. C'est donc un choix privilégiant l'efficacité sans perte de généralité. A l'aide de communications explicites ou de services urgents (cf. plus loin), il est en effet possible de reconstruire facilement des protocoles nécessitant des échanges entre les processus légers créés et créateurs comme les appels de procédure à distance synchrone ou asynchrone, des extensions de ce modèle [13], des schémas de type diviser pour régner [8], des schémas plus exotiques comme les continuations [3] ou le retour direct de résultats depuis les feuilles d'un arbre de découpe récursive de calculs.

Ordonnancement des processus légers. ATHAPASCAN-0B n'offre pas de mécanisme d'ordonnancement global de processus légers. Il permet simplement d'utiliser les ordonnanceurs locaux des nœuds en permettant de fixer le type d'ordonnancement et la priorité d'un processus léger au sens des **Pthreads** [37]. Ce standard distingue deux stratégies. La stratégie *FIFO*[12] définit un ordre d'exécution où le processus léger qui s'exécute est toujours le plus prioritaire. Celui-ci ne perd le processeur que sur terminaison, blocage explicite ou préemption par un processus léger de plus haute priorité. Dans la stratégie *RR*[13], les processus légers de plus haute priorité se partagent le processeur à tour de rôle à une fréquence donnée. Un processus léger de priorité inférieure ne peut tourner que si tous les processus légers de priorité supérieure se sont bloqués ou sont terminés. Une dernière stratégie (*OTHER*) est laissé au libre choix du réalisateur du noyau Pthreads. C'est généralement la stratégie de gestion du processeur effectué par le système hôte qui est reconduite pour les processus légers.

[12] First in, first out ou premier arrivé, premier servi.
[13] Round-robin ou tourniquet.

Service Urgent. ATHAPASCAN-0B permet de lancer à distance l'exécution d'un service urgent similaire au concept d'Active Messages [55] ou de RSR de Nexus [23]. L'exécution d'un service urgent interrompt l'exécution en cours des processus légers comme le ferait une interruption. Un service urgent doit s'exécuter sans se bloquer de façon à éviter tout interblocage. Il se réduit généralement à une mise à jour de données et un réveil de processus légers. Les services urgents sont enregistrés de la même façon que les services normaux.

6.4.2.b Communication

L'échange de données entre processus légers distants est fait par des messages ou datagrammes[14] circulant sur des liaisons logiques unidirectionnelles fiables entre paire de nœuds. Aucune connexion préalable n'est requise. Sur une liaison, l'ordre d'émission est respecté à la réception mais aucune contrainte n'est imposée sur les entrelacements possibles des messages issus sur deux liaisons distinctes.

Adressage. Une liaison est simplement caractérisée par le numéro du site émetteur, le numéro du site récepteur et l'identification d'un port de réception. L'émetteur (respct. récepteur) n'a donc qu'à fournir le numéro du site destinataire (respct. émetteur) et l'identification du port de réception. La gestion des ports est automatique et transparente au programmeur. Par contre, les processus légers doivent s'accorder sur l'attribution des identificateurs de port.

Identification des ports. Un port doit être connu des deux processus légers communicants. ATHAPASCAN-0B offre deux opérateurs pour engendrer des identificateurs de port. Un opérateur globalement déterministe permet de garantir que si tous les nœuds appellent cet opérateur dans le même ordre, ils obtiendront la même suite d'identificateurs. Ceci est destiné à rendre compte de la notion de **port bien connu** (well known port) qui permet de donner une structure initiale de réseau de processus légers. Un second opérateur permet d'engendrer localement un identificateur de port qui sera unique au sein de toute la machine parallèle (numéro de site, numéro d'ordre sur le site). Ce double mécanisme a été choisi pour sa légèreté car il n'implique aucune communication entre nœuds. Un identificateur de port peut être communiqué sans restriction entre processus légers locaux ou non. Un identificateur de port peut être engendré par un nœud et donner lieu à la création de plusieurs ports sur des sites différents si cet identificateur est utilisé dans différentes liaisons bi-points.

Un processus léger communique normalement avec un ensemble fini de *voisins* qu'il est commode de désigner par une suite de numéros. Pour cette raison, une identification de port est composée de deux parties : le **port** proprement dit et un index dit **étiquette** (tag). L'opération de génération d'identificateurs de port est en fait une opération créant une suite d'identificateurs constitués d'un même port et

[14]Un message est obligatoirement une donnée de taille finie.

d'une suite consécutive d'étiquettes.

Emission et réception. Toutes les communications se font selon le schéma type précédemment défini. Une requête de communication est initiée de façon non bloquante ; sa terminaison doit être testée ultérieurement.

```
a0ISend(aotPort, a0tNode, a0tRequest, a0tFormat, Data)
a0IRecv(       .....................                 )
```

Une émission se termine dès que les données à émettre sont émises ou mémorisées par les dispositifs de communication. Ceci n'indique donc ni que le message ait été reçu sur le nœud distant, ni qu'il ait été acquis par le processus léger destinataire. Simplement, les données à émettre sont de nouveau modifiables sans risque par le processus léger émetteur. La situation est symétrique pour une réception où l'initiation définit simplement la liaison et les variables dans lesquelles mettre le message reçu. La réception termine lorsque le message reçu est placé dans ces variables.

Un tel modèle explicite de communication permet de construire aisément des opérateurs de communications plus évolués comme les opérations de communications multipoints. Elles peuvent être construites simplement à partir d'une suite de communications non bloquantes suivies des attentes de terminaison correspondantes. Le programmeur n'a pas à se préoccuper de l'ordre dans lequel le noyau exécute les communications élémentaires. En effet, les communications étant toutes données, celui-ci les effectue au gré de la disponibilité du réseau. Les données à émettre peuvent être partagées par les émissions.

Emballage / Déballage des données. Les opérateurs de base de ATHAPASCAN-0B sont capable d'émettre et recevoir les données typées depuis la mémoire si l'organisation de ces données en mémoire est statique et régulière (tableau de type de base). Le typage des données est nécessaire pour effectuer les conversion de codage nécessaire du fait de l'hétérogénéité des processeurs. Si l'organisation en mémoire des données à émettre ou à recevoir est irrégulière ou dynamique, il est nécessaire d'emballer / déballer explicitement les données dans des tampons. ATHAPASCAN-0B fournit les mécanismes de base permattant de gérer les tampons et d'y emballer/déballer des données typées.

ATHAPASCAN-0B offre une bibliothèque de communication étendue pour rendre transparent ces opérations via le concept de **format**. Un format enregistre une description de l'organisation régulière des données comme les datatype de MPI [41] ou des procédures complexes d'emballage/déballage à la XDR [53]. Ceci permet de séparer l'expression des communications et du calcul de l'expression de la gestion des tampons et de l'emballage des données. Du point de vue d'un opérateur d'émission ou de réception, une donnée à émettre ou à recevoir est toujours considérée comme un vecteur de valeur d'un type donné et codée par un triplet ⟨format, adresse, quantité⟩.

6.4.2.c Accès mémoire à distance

L'utilisation de communications explicites pour transférer arguments et résultats d'un calcul parallèle implique la copie de données. Ceci est inefficace lorque de gros volumes sont propagés à travers de longues chaînes d'initiation de sous calculs, comme dans des appels récursifs. En effet, la propagation de ces copies consomme de façon importante de la mémoire et du temps de stockage-reémission. Ce phénomène est encore accru lorsque l'ignorance a priori de la localité des références aux données exige de transmettre plus de données que nécessaires.

La solution à ce problème est le passage d'arguments ou résultats de sous calculs par référence plutôt que par valeur. La donnée est transférée au plus tard à la demande du calcul l'utilisant. Cette façon de faire a des inconvénients car les accès en lecture peuvent avoir des latences importantes du fait du temps d'aller-retour dans les réseaux sous-jacents. Il est donc intéressant de substituer des écritures à distance aux lectures chaque fois que cela est possible. Des mécanismes de synchronisation sont aussi nécessaires pour contrôler les accès aux données partagées.

Le partage direct de mémoire dans une machine parallèle ATHAPASCAN-0B est seulement possible entre processus légers d'un même nœud. ATHAPASCAN-0B propose donc une abstraction de mémoire accessible à distance par des opérateurs de lecture et écriture explicites. Ces opérateurs sont conçus de façon à être facilement implantables via un protocole de communication du type *message actif* [55] ou via les nouveaux interfaces de réseau rapide directement couplés à la mémoire comme celui du Cray T3D/E ou les produits DEC Memory Channel [30], Bus SCI [36], ...).

DMA-Region. La partie de la mémoire d'un nœud qui peut être accédée à distance doit être déclarée. Appelées DMA-Region (Direct Memory Access Region), elles sont caractérisées par un triplet ⟨format, addresse origine, quantité⟩. Un format est associé à une région de façon à assurer lors des accès les transcodages nécessaires du fait de l'hétérogénéité de la machine parallèle. La déclaration d'une région doit précéder tous les accès.

```
a0NewDMARegion(a0tDMARegion, a0tFormat, a0tAddress)
```

La déclaration engendre un identificateur unique qui doit être présenté lors de tout accès. Cet identificateur peut être communiqué sans restriction entre processus légers locaux ou non. Localement, une région peut être accédée directement. Par contre les accès distants sont fait via des opérateurs spécifiques. Ces opérateurs décident du besoin d'emballer/déballer les données selon l'organisation de celles-ci en mémoire, l'hétérogénéité des processeurs source et cible ou les dispositifs matériels d'accès à distance à la mémoire.

Accès distant. Chaque accès à une région distante se fait par des requêtes de lecture ou écriture. Ces requêtes se conforment au modèle général d'opérateur communicant. Elles sont initiées par un opérateur non bloquant. Leurs terminaisons peuvent être attendues par la suite. Les accès mémoire à distance peuvent être recouverts par du calcul au même titre que les communications ou les créations de processus légers à distance. Une lecture termine dès que la donnée locale s'est vue affectée la valeur lue à distance.

```
a0IRead(a0tDMARegion, a0tRequest, a0tFormat, a0tAddress)
a0IWrite(             .....................            )
```

Ces requêtes précisent l'identité de la région accédée, l'adresse et le format local de la donnée lue ou écrite. Le format distant n'a pas besoin d'être explicité car il l'a été à la déclaration de la région.

Synchronisation. Les accès à une région sont atomiques et les accès issus de différents processus légers sont sérialisés. Si une cohérence plus forte est nécessaire, des **verrous** sont fournis en ATHAPASCAN-0B afin de permettre la synchronisation des accès conflictuels. Ici aussi, les opérations de verrouillage sont asynchrones comme toutes opérations impliquant des communications. La gestion des requêtes de verrouillage sur un verrou sont traitées dans l'ordre de réception.

Si un processus léger doit verrouiller plusieurs régions, il y risque d'interblocage. Pour limiter ce risque, ATHAPASCAN-0B permet à un processus léger de requérir un verrouillage atomique d'un ensemble de verrous. Ceci est simplement réalisé en effectuant les verrouillages selon un ordre standard.

6.4.3 Réalisation

ATHAPASCAN-0B est le noyau exécutif de bas niveau qui s'est dégagé des différentes expériences conduites dans le projet APACHE. Contraignant la programmation parallèle à suivre un modèle procédural, une version préliminaire [12], plus simple qu'ATHAPASCAN-0B avait été construite autour de PVM et de divers noyaux de processus légers disponibles à l'époque[15]. Il n'était pas possible d'exploiter le parallélisme SMP du fait des capacités limités de PVM et des noyaux de processus légers utilisés. Au cours de la conception d'une extension ATHAPASCAN-0MP [13] utilisée pour une bibliothèque de calcul formel parallèle GIVARO [27], le besoin d'un noyau de bas niveau portable et efficace incluant processus légers et communication s'était révélé nécessaire. Dans cette perspective, nous avons collaboré avec IBM[16] au développement d'un prototype de MPI intégrant les processus légers à destination de la machine IBM SPx : MPI/F [24] [25] [26].

La définition d'un environnement de programmation de haut niveau ATHAPASCAN-1[8] conforme aux objectifs du projet APACHE c'est à dire incorporant des stratégies de placement et ordonnancement automatique à destination d'applications irrégulières, a accru le besoin d'un tel noyau pour assurer la portabilité et l'efficacité de cet environnement.

[15]Dont un, élémentaire, réalisé dans le cadre du groupe APACHE.

[16]Via le séjour d'un doctorant au centre de recherche IBM de Yorktown Heights.

Choix de réalisation. ATHAPASCAN-0B aurait du être un simple carrossage de MPI/F. Cependant, l'abandon de celui-ci par IBM et la non disponibilité du source nous a contraint à réaliser une implantation spécifique. Deux options de réalisation étaient possibles. La première consistait à proposer une implantation intégrée des fonctionnalités de communication, d'accès mémoire à distance et de gestion des processus légers. Un tel choix aurait sans doute permis d'atteindre la meilleure efficacité (telle qu'atteinte par MPI/F) au prix de coûts élevés de portage sur toutes les plateformes matérielles envisagées.

Or un des résultats des expériences avec ATHAPASCAN-0a était qu'il était possible avec une perte minime d'efficacité d'assurer le mariage de ces différents noyaux en les traitants comme des boîtes noires. La pérennité de ATHAPASCAN-0B serait alors assurée par le choix de boîtes noires standard et largement répandues. ATHAPASCAN-0B a donc été réalisé comme l'assemblage d'un noyau Pthreads [37] et d'un noyau MPI [41] dont seules les fonctionnalités de base ont été utilisées. Le choix du noyau de processus légers était simple car le standard Pthreads est le seul disponible. MPI a été préféré à PVM du fait de sa plus grande légèreté [29].

Le principe du mariage de noyaux de processus légers et communication est de confiner les communications au sein de procédures en exclusion mutuelle. Fils d'exécution de l'application et processus légers démons coopérent de façon à assurer la cohérence du noyau de communication non prévu pour une utilisation concurrente.

Couplage noyau de processus légers- noyau de communication. Ce couplage est le point critique de l'intégration des communications et des processus légers [23] [33] [38] [39]. En effet, les noyaux de communication actuels partent du principe que l'enchainement des calculs et des communications est complètement gérée par le programme d'application. Le système hôte se charge de la désactivation du processus utilisateur si le réseau n'est pas disponible pour une communication (multiprogrammation). Dans la mesure où l'on souhaite continuer à calculer alors que la communication est en cours, il faut, d'une part, pouvoir initier des communications sans blocage pour les processus légers qui doivent calculer et, d'autre part, assurer l'enchaînement des communications proposées par les processus légers lorsque le réseau est prêt à les faire. D'une manière générale, il faut pouvoir recevoir une donnée aussitôt qu'elle arrive du réseau et, réciproquement, émettre les données (s'il y en a) dès que le réseau est prêt à les acheminer.

Ceci peut être atteint de plusieurs façons selon les modalités de réalisation du noyau de gestion des processus légers [1] [17]. Les processus légers peuvent être gérés par multiprogrammation des entrées/sorties dans les cas d'une implantation dite en *mode système* (kernel space) (ex. AIX 4.2) ou en *mode mixte* [1] (ex. Solaris 2.5). Dans ce cas, les processus légers effectuant les communications seront réveillés directement par le système en fonction de l'état du dispositif d'accès au réseau. Le délai effectif de réveil dépend des modalités d'ordonnancement des processus légers par le système.

Dans le cas où aucune multiprogrammation sur entrée-sortie n'est faite (implantation en *mode utilisateur* (ex. AIX 3.5, MARCEL [46], ..), on peut pallier ce défaut si un signal est engendré lorsque le dispositif d'interface avec le réseau change d'état. En effet, une procédure de traitement du signal permet de réagir à ce changement d'état en faisant progresser les communications en attente et en réveillant les processus légers bloqués si nécessaire. Le problème est ici aussi la réactivité du système c'est à dire le délai que prend le système pour propager cette indication au noyau de gestion des processus légers.

Si le réseau (ou le système) ne remonte pas un tel signal, il est nécessaire de venir tester périodiquement l'état du réseau. Le choix de la période est critique dans la mesure où un test trop fréquent produira une dégradation des performances de calcul alors qu'une période trop longue provoquera un accroissement important du temps de démarrage des communications ainsi qu'une baisse du débit pour les messages nécessitant un transfert en plusieurs paquets.

Les systèmes sur lesquels ATHAPASCAN-0B a été porté sont caractéristiques de ces modes de fonctionnement. Dans tous les cas, un réglage fin de l'interaction avec le réseau est nécessaire pour atteindre un niveau de performance acceptable. L'utilisation d'un signal n'a jamais été possible soit du fait de l'absence d'un tel signal soit du fait d'un surcoût considérable provoquée par la fréquence trop élevée des changements d'état engendrant les signaux (IBM SP1). La scrutation des communication a donc été réalisée de deux façons complémentaires. Une scrutation est effectuée lors de toute exécution d'un opérateur ATHAPASCAN-0B. Dans le cas où la fréquence obtenue est insuffisante, des scrutations complémentaires doivent être effectuées. C'est là le point le plus délicat car il concerne les fonctions de gestion du temps et d'ordonnancement des processus légers dont les implantations sont généralement partielles et dissemblables.

Portage. ATHAPASCAN-0B a été initialement développé sur IBM SPx (AIX 3.5) autour de MPI/F et d'un noyau DCE-threads (mode utilisateur). Il actuellement opérationnel sur la version AIX 4.x, un noyau Pthreads (mode système) et le produit IBM MPI version 2.3. ATHAPASCAN-0B a été porté sur différents réseaux de stations en utilisant un noyau MPI-LAM (du Ohio Supercomputing Center) [6]) et les noyaux Pthreads disponibles sur ces stations : PCx86/Linux (mode utilisateur), PCx86/Solaris (mode mixte), SMP Sun SPARC/Solaris (mode mixte), Station HP/HP-UX 10.x (mode utilisateur). Un portage a été effectué sur un Cray T3E en utilisant une bibliothèque MPI/CHIMP (du Edinburgh Parallel Computing Center) [7] et le noyau MARCEL [46] (mode utilisateur).

Le portage s'est toujours effectué sans problème, aux réglages prêts, dès que les deux noyaux disponibles (Pthreads et MPI) étaient matures.

6.4.4 Performances

Dans cette section, nous nous proposons de montrer que ATHAPASCAN-0B ne dégrade pas les performances des noyaux sur lesquels il est construit. Les mesures faites l'ont été sur un IBM SP1 de 32 processeurs connectés par un réseau commuté rapide (40 Mo/s). Celui-ci est utilisable via un protocole spécifique ou le protocole fédérateur IP. Les tests ont été conduits en utilisant un noyau de processus légers (IBM DCE-threads, draft POSIX 1003.4a) en mode utilisateur et la version prototype de MPI/F [24]. Une série d'expériences a mesuré le surcoût introduit par ATHAPASCAN-0B sur les noyaux POSIX et MPI. Une autre série a eu pour but d'évaluer la réalité du recouvrement calcul-communication dans différents de contrôle parallèle.

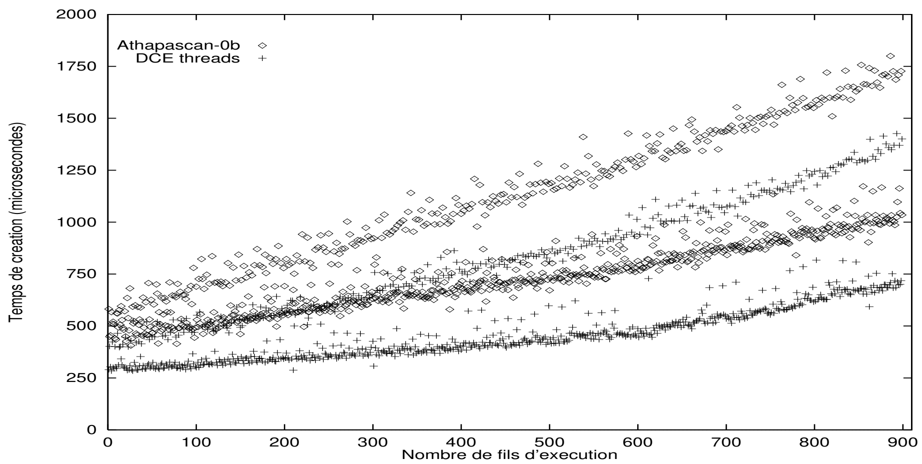


Figure 6.6 : Temps de création de processus légers.

Gestion des processus légers. La figure 6.6 montre le temps de création des processus légers pour le noyau DCE et ATHAPASCAN-0B en fonction de la charge actuelle (nombre de processus légers en cours d'exécution). Il apparait clairement deux modes de création dont l'un est plus rapide que l'autre. Ceci est probablement du à la statégie de gestion de mémoire qui demande une nouvelle allocation tous les 3 processus légers créés. ATHAPASCAN-0B offre le même comportement. Le surcoût est grossièrement de 150 microsecondes en faible charge à 300 microsecondes pour un millier de processus légers. Ce surcoût est due à la gestion des structures nécessaires à l'identification des processus légers, l'allocation de leur mémoire privée et le passage des arguments.

La figure 6.7 donne la durée d'une série de commutation explicite dans un ensemble de processus légers de taille croissante. Les temps de commutation entre processus légers en ATHAPASCAN-0B,sont de l'ordre de 18 microsecondes sauf pour un petit ensemble de processus légers où les mesures sont très instables. Nous n'avons

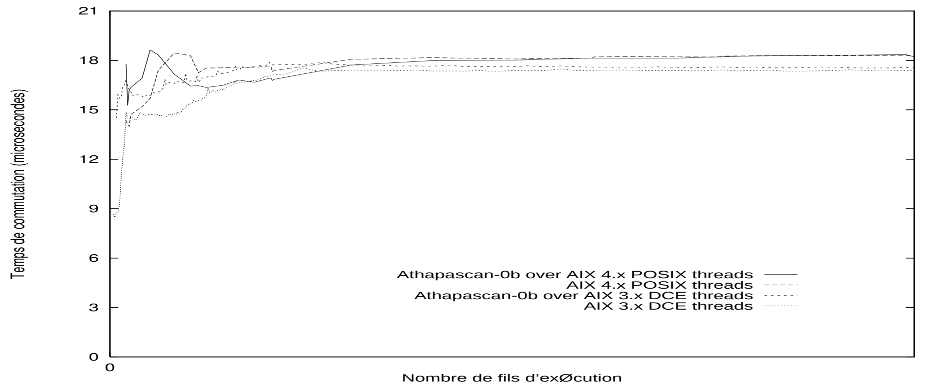


Figure 6.7 : Temps de commutation.

pas d'explication de ce phénomène.

Communication. Les performances pour de petits messages sont données sur la figure 6.8. L'expérience consiste en un programme de ping-pong qui estime les délais de délivrance de messages (1/2 du temps d'aller-retour). Le surcoût Athapascan-0b pour un message vide est de 100% et 30% pour un message de 3 Ko ce qui est important. Ce surcoût trouve son origine dans le retard de réaction à la disponibilité du réseau. Ce délai provient de la multiprogrammation des processus (lourds ou légers[17]) effectués par le système ainsi que du recours au test périodique du réseau. Néanmoins, une autre expérience montre que, lors d'un ping-pong asyn-

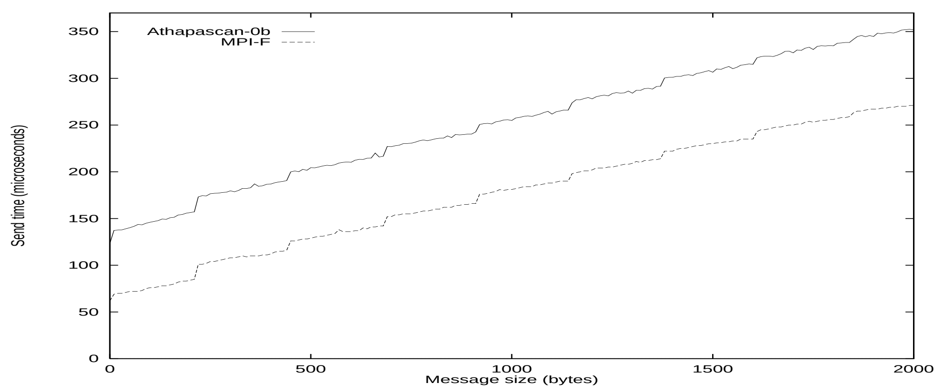


Figure 6.8 : Emission de petits messages.

chrone mettant en œuvre un message de taille nulle, 120 des 230 microsecondes du temps d'aller-retour sont utilisables pour le calcul. Ceci veut dire que le recouvrement calcul-communication est effectivement réalisable même dans des conditions défavorables de couplage du noyau de communication avec le noyau de processus

[17]On retrouve un tel délai en AIX4.x où les processus légerssont en mode système.

légers.

La situation est donc bien plus favorable pour des tailles de messages importantes (figure 6.9). Le surcoût de ATHAPASCAN-0B croit linéairement en fonction de la taille du message émis mais devient relativement petit. Cette croissance linéaire provient du surcoût de gestion des processus légers payé chaque fois qu'un morceau élémentaire du message (paquet) doit être émis par la couche de communication. Cependant, le débit atteint est voisin de celui de la couche de communication MPI. Ainsi le débit maximum de MPI/F sur le réseau rapide du SP1 atteint 28.5 Mo/s alors que celui de ATHAPASCAN-0B est de 27 Mo/s.

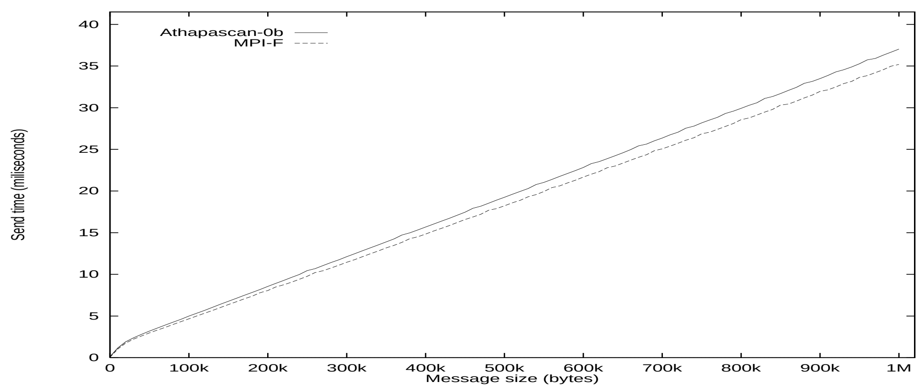


Figure 6.9 : Emission de gros messages.

On peut estimer l'effet de la multiprogrammation des processus légers en regardant les temps nécessaires pour exécuter plusieurs processus légers effectuant des ping-pong depuis un même nœud. La figure 6.10 montre que la multiprogrammation permet d'exploiter au mieux les possibilités de débit du réseau puisque un ping-pong supplémentaire *coûte moins* que le coût d'un unique ping-pong.

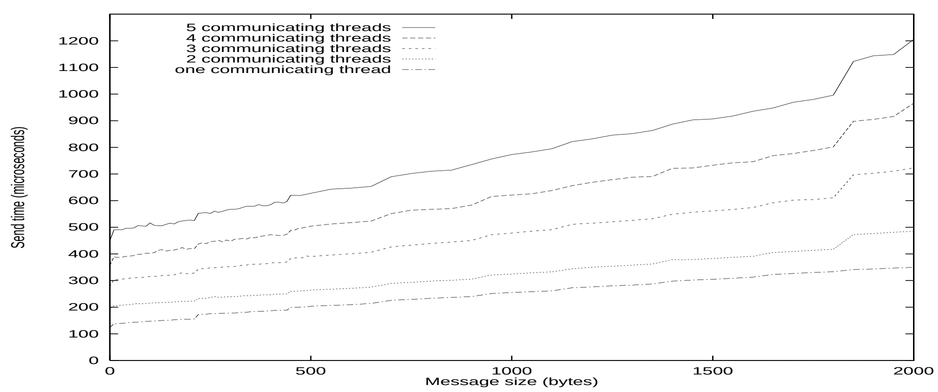


Figure 6.10 : Multiples ping-pongs: petits messages.

Ceci est vérifié tant que le réseau n'est pas saturé comme c'est le cas lors de ping-pongs mettant en oeuvre de gros messages (figure 6.11).

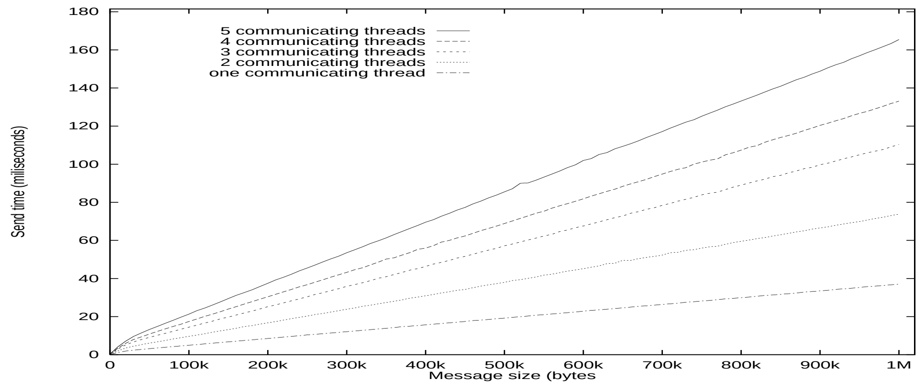


Figure 6.11 : Multiples ping-pongs: gros messages.

Création de processus légers à distance Ces propriétés de recouvrement des communications et des calculs se retrouvent sur des schémas où un processus léger initial *lance* une gerbe de processus légers sur d'autres noeuds (figure 6.12). Les

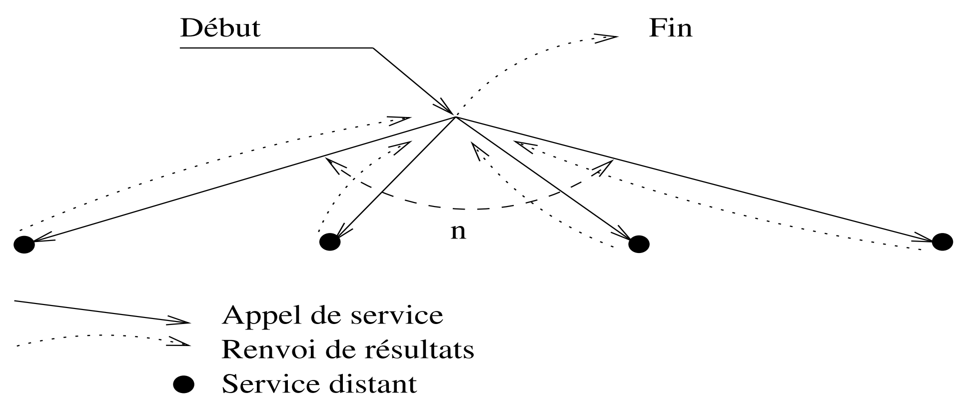


Figure 6.12 : n appels de service *nul* avec collection des résultats (RPC).

résultats montrent un recouvrement des communications pour les petits messages (figure 6.13) jusqu'à saturation du réseau par les gros messages(figure 6.14).

Ces propriétés sont préservées dans des schémas de calcul-communication plus complexes tels que celui où un processus léger initial *lance* une gerbe de processus légers sur d'autres noeuds(figure 6.12) et où ces processus légers enchaînent des étapes de calculs et de communications collectives. L'expérience porte sur le lancement de n services ne faisant aucun calcul mais procédant à un échange total avant de terminer(figure 6.15). Les résultats sont voisins de ceux des expériences précédentes. Il y a recouvrement jusqu'à saturation du réseau (figure 6.16).

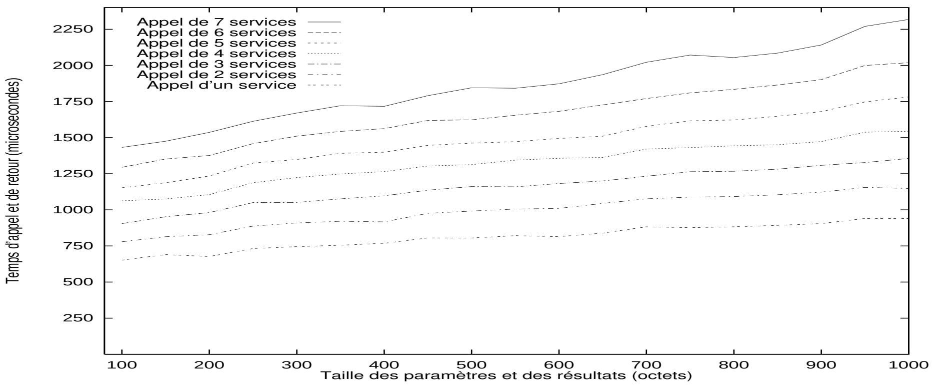


Figure 6.13 : n appels de service(RPC): gros messages.

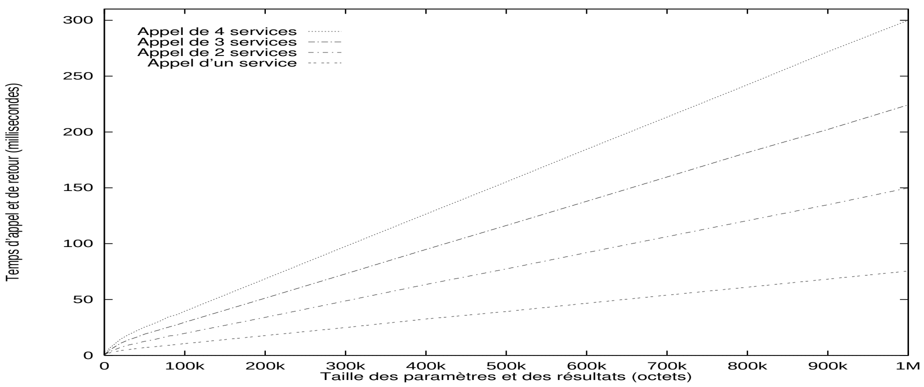


Figure 6.14 : n appels de service(RPC) : petits messages.

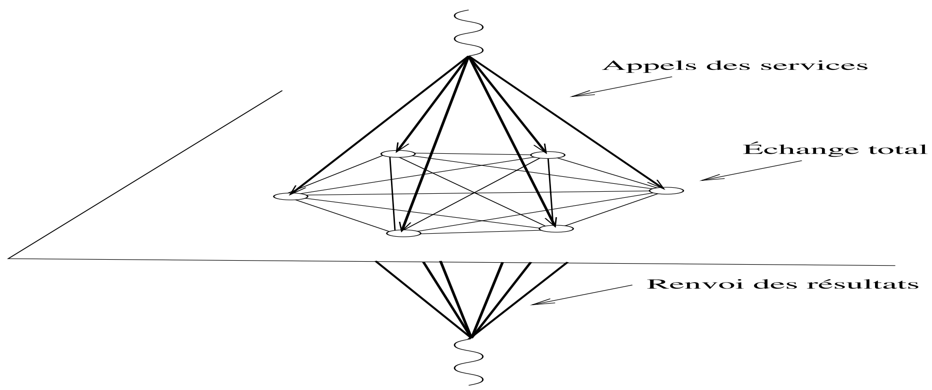


Figure 6.15 : n appels de service avec échange total et collection des résultats.

6.5 Conclusion

Nous avons justifié dans la première partie de ce cours l'utilisation des processus légers pour le calcul parallèle et leurs capacités à recouvrir les communications.

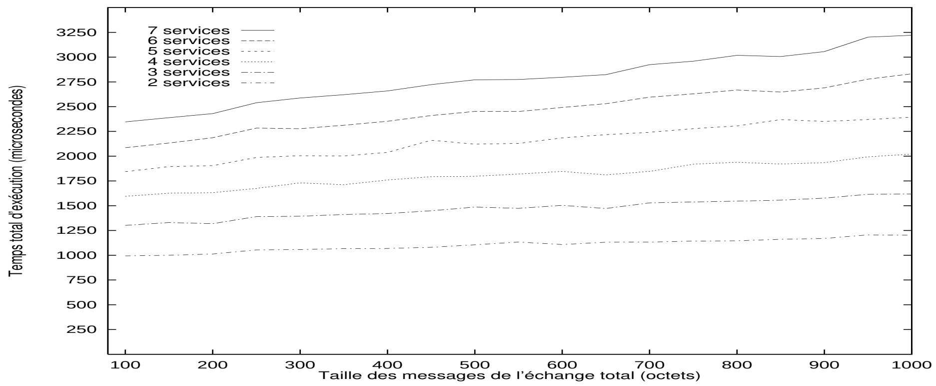


Figure 6.16 : n appels de service avec échange total et collection des résultats : petit messages

Des environnements de programmation parallèle, comme Athapascan-0b et PM^2, proposent des modèles de programmation et d'exécution s'appuyant sur la notion de processus léger. Les grandes lignes des deux plateformes logicielles ont ensuite été décrites et plus précisément le volet ¡¡ communication ¿¿. Les performances de ces environnements sont de tout premier plan et ont été confirmées par des applications significatives.

Nous allons maintenant montrer en quoi les similitudes et différences de ces environnement sont liées aux applications pour lesquelles ils ont été conçus et aux principaux choix techniques de réalisation. Nos évolutions futures sont assez proches et concernent principalement les nouvelles architectures de réseau ou bien encore les bibliothèques de processus légers pour multiprocesseurs à mémoire partagée.

6.5.1 Applications

Les choix de conception de Athapascan-0b et PM^2 ont été guidés par la logique des projets pour lesquels ils ont produits et des applications qui étaient initialement visés. L'équipe Lilloise s'est dans un premier temps intéressée aux applications d'optimisation combinatoire (¡¡ branch-and-bound ¿¿), applications dont le comportement est fortement irrégulier, où le parallélisme potentiel est grand et où il n'est pas a priori possible de prédire et répartir la charge simplement par un simple placement. Les idées originelles de PM^2 sont issues de cette réflexion et prônent le fait qu'un environnement doit être capable de gérer un grand nombre de tâches avec des possibilités de migration pour répartir dynamiquement la charge.

L'équipe grenoble s'est intéressée à deux applications de caractéristiques assez différentes. L'application de dynamique moléculaire[2] correspond à une application à parallélisme de données où un bon placement initial peut être trouvé et où le déséquilibre de charge se produit *lentement*. Il peut être simplement corrigé par des mouvements de données. Dans ce type d'application, le recouvrement calcul communication est primordial. D'où l'effort fait en Athapascan-0b pour permettre ce recouvrement de façon simple. A contrario, l'application de calcul formel [27] développe des calculs de façon imprédictible. Dans ce cas, il est nécessaire de permettre l'expression dynamique de la décomposition de calcul en sous-calculs coopérants à un degré quelconque. Comme en PM^2, l'accent a été mis sur la création à distance de processus. Dans aucun de ces cas, la migration ne s'est révélée utile.

Ces environnements ont été utilisés pour des applications scientifiques et ont confirmé que le niveau des performances obtenues était très intéressant. Certaines de ces applications seront d'ailleurs présentées dans le cadre de cette école. Un autre point à souligner est que les modèles de programmation et l'interface de programmation Athapascan-0b et PM^2 sont simples d'utilisation et il n'est pas besoin d'être un grand expert des environnements parallèles et du système pour les comprendre et les utiliser.

6.5.2 Réalisations

Les plateformes logicielles Athapascan-0b et PM^2 sont principalement bâties autour de deux bibliothèques : une première gère les processus légers et l'autre les communications. L'équipe Athapascan-0b a choisi une réalisation par intégration de deux ¡¡ boîtes noires standard ¿¿ disponibles, soit au niveau du système d'exploitation, soit dans le domaine public. Le premier standard est connu sous le nom **POSIX Threads** et l'autre est le standard **MPI**. La plateforme Athapascan-0b est donc facilement portable sur une architecture dès que les deux implantations de ces standards y sont disponibles.

L'équipe PM^2 a adopté une approche opposée qui consiste, au niveau des processus légers, à construire sa propre gestion de processus légers **MARCEL** et ceci pour répondre aux exigences qui sont données par les applications (gestion efficace, migration). PM^2 s'est dans un premier temps appuyé sur PVM, puis plus récemment une couche de communication générique **MADELEINE** facilement interfaçable avec les principales bibliothèques de communication du marché a été définie et portée à ce jour sur quatres couches de communication.

6.5.3 Travaux futurs

Pour continuer à faire ¡¡ vivre ¿¿ nos environnements, nous pensons qu'il est indispensable de suivre et de s'adapter aux évolutions technologiques, en particulier dans le domaine des réseaux rapides. Des architectures à très hautes performances et faible coût sont maintenant opérationnelles et il est stratégique que nos environnements les exploitent efficacement. Nos premières expérimentations ont été réalisées à partir du réseau Myrinet et les performances obtenues sont très prometteuses. Actuellement, nous nous intéressons aussi à d'autres architectures à haut débit comme SCI [36] ou bien encore la machine MPC [49] développée par l'équipe d'Alain Greiner (LIP6). Nous pensons qu'une interface générique de communication, comme MADELEINE qui a été utilisée pour PM^2, facilitera nos développements futurs et les portages sur les nouvelles architectures de réseaux.

Si ATHAPASCAN-0B a choisi de s'appuyer sur les librairies de processus légers disponibles, soit au niveau du système, soit dans le dommaine public, PM^2 a délibérement opté pour le développement d'une nouvelle librairie de processus légers s'exécutant en contexte utilisateur. Le principal intérêt d'un tel choix concerne le niveau de performances (appel de procédure vs appel système) et les possibilités d'adapter cette librairie aux besoins spécifiques [18] de PM^2. L'inconvénient majeur d'une exécution en contexte utilisateur (MARCEL et les librairies du domaine public) est l'incapacité à exploiter efficacement les architectures multiprocesseurs à mémoire partagée (SMP). La seule solution satisfaisante à ce problème est donc de donner à l'ordonnanceur la possibilité de placer les processus légers sur différents processeurs. Les processus de poids moyen d'un système comme Solaris [54] devraient nous permettre d'implanter efficacement un ordonnanceur de processus légers en contexte utilisateur sur les architectures multiprocesseurs SUN et PC.

6.6 Remerciements

Nous tenons à remercier vivement les étudiants, qui au cours de leur thèse, nous ont fortement aidé à concevoir et à réaliser nos environnements :

- M. Christaller, I. Ginzburg, M. Pasin, M. Rivière (ATHAPASCAN-0B, Grenoble).
- Y. Denneulin, R. Namyst, B. Planquelle (PM^2, Lille).

Leurs efforts et leurs enthousiames ont grandement contribué à la maturité et à l'excellente robustesse de nos plateformes.

[18] Migration, clonage, priorités.

Bibliographie

[1] Anderson (T.), Bershad (B.), Lazowska (E.) et Levy (H.). – Scheduler Activations: Efficient kernel support for the user-level managment of parallelism. *In : Proc. of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 95–105.

[2] Bernard (P.), Plateau (B.) et Trystram (D.). – Using Threads for developing Parallel Applications: Molecular Dynamics as a case study. *In : Parallel Numerics*, pp. 3–16. – Trobec, Gozd Martuljek, Slovenia, 1996.

[3] Blumofe (R.), Joerg (C.), Kuszmaul (B.), Leiserson (C.), Randall (K.) et Zhou (Y.). – Cilk: An efficient multithreaded runtime system. *In : PPOP'95 proceedings ACM SIGPLAN*. – Santa Barbara, 1995.

[4] Boden (N.), Cohen (D.), Feldermann (R.), Seitz (C.) et Su (W.). – Myrinet: A gigabit per second local area network. *IEEE-Micro*, vol. 15, feb 1995, pp. 29–36.

[5] Bubeck (T.). – *Eine Systemumebung zum vereilen funktionalen Rechnen.* – Rapport technique n° WSI-93-8, Germany, Universität Tübingen, Août. 1993.

[6] Burns (G.), Daoud (R.) et Vaigl (J.). – *LAM : an open cluster environnement.* – Rapport technique, Ohio Supercomputing Center, May 1994.

[7] Cameron (K.), Clarke (L.) et Smith (A. G.). – *CRI/EPCC MPI for Cray T3D.* – Rapport technique, Edinburgh Parallel Computing Centre, 1996.

[8] Cavalheiro (G.) et Doreille (M.). – Athapascan: A C++ library for parallel programming. *In : European School of Computer Science : Parallel Programming Environnement for High Performance Computing*, éd. par INRIA, Sophia-Antipolis (F.), pp. 75–85. – Alpe d'Huez France, April 1996.

[9] Chandy (K.) et Kesselman (C.). – CC++: A declarative concurrent object-oriented programming notation. *In : Research Directions in Concurrent Object-Oriented Programming.* – MIT Press.

[10] Chandy (K.) et Kesselman (C.). – Compositional C++: Compositional parallel programming. *In : Proc. Fifth Int'l Workshop on Parallel Languages and Compilers.* – Springer-Verlag.

[11] Christaller (M.). – Athapascan-0a: A control parallelism approach on top of PVM. *In : Proceedings of the 1994 PVM Users' Group Meeting.* – Oak Ridge, Tennessee, 1994.

[12] Christaller (M.). – *Athapascan-0 : vers un support exécutif pour applications parallèles irrégulières efficacement portables.* – Grenoble I, Thèse de PhD, Université Joseph Fourier, Nov 1996.

[13] Christaller (M.), Briat (J.) et Rivière (M.). – Athapascan-0 : Concepts structurants simples pour une programmation parallèle efficace. *Calculateurs Parallèles*, no7(2), 1995, pp. 173–196.

[14] Colin (J.). – *DTMS : Un environnement pour la programmation distribuée à grain indéterminé.* – Thèse de PhD, Université de Mons-Hainaut, 1995.

[15] Courtrai (L.). – *Les Composants Actifs de Communication: Outils pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD.* – Laboratoire d'Informatique Fondamentale de Lille, Thèse de PhD, Université des Sciences et Technologies de Lille, Oct. 1992.

[16] Cypher (R.) et Leu (E.). – *Message Passing Semantics and Portable Parallel Programs.* – Rapport technique, IBM Research, 1994.

[17] Demeure (I.) et Fahrat (J.). – Systèmes de processus légers : concepts et exemples. *Techniques et Sciences Informatiques*, vol. 6, n° 30, 1994, pp. 765–795.

[18] Denneulin (Y.) et Namyst (R.). – PM^2 : Parallel Multithreaded Machine. Un support d'exécution pour applications irrégulières. *In : 7èmes Rencontres du Parallélisme*, pp. 208–212. – Mons Belgique, May 1995.

[19] Dreier (B.) et Zahn (M.). – RThreads — a Uniform Interface for Parallel and Distributed Programming. *In : Proceedings of the Second International Conference on Massively Parallel Computing Systems (MPCS'96)*, pp. 557–561. – Ischia, Italy, May 1996.

[20] Ferrari (A.) et Sunderam (V.). – *TPVM. A Threads-Based Interface and Subsystem for PVM.* – Rapport technique n° CSTR-940802, University of Virginia & Emory University, USA, August 1994.

[21] Foster (I.), Kesselman (C.) et Tuecke (S.). – The Nexus task-parallel runtime system. *In : Proc. 1st Intl Workshop on Parallel Processing.* – Tata Mc Graw Hill.

[22] Foster (I.), Kesselman (C.) et Tuecke (S.). – *The Nexus approach to integrate multithreading and communication.* – Rapport technique, Argonne National Laboratory, USA, 1995.

[23] Foster (I.), Kesselman (C.) et Tuecke (S.). – The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, vol. 37, 1996, pp. 70–82.

[24] Francke (H.), Hochschild (P.), Pattnaik (P.) et Prost (J.). – MPI-F: a MPI prototype implementation on IBM SP1. *In : Environments and tools for parallel scientific computing: 2nd workshop*, éd. par for Industrial (S.) et Mathematics (A.), pp. 43–55. – Townsend, TN, May 1994.

[25] Franke (H.) et Rivière (M.). – *An efficient MPI-threaded implementation for an executive parallel kernel.* – Rapport technique, Yorktown Height, N.Y, IBM T.J. Watson Research Center, 1995.

[26] Franke (H.), Wu (E.), Rivière (M.), Pattnaik (P.) et Snir (M.). – MPI programming environment for IBM SP1/SP2. *In : International Symposium on Computer Architecture.* – Montréal, Canada, 1995.

[27] Gautier (T.). – *Calcul formel et parallélisme : Conception du Système Givaro et Applications au Calcul dans les Extensions Algébriques* . – Thèse de PhD, Institut National Polytechnique de Grenoble, June 1996.

[28] Geist (A.), Beguelin (A.), Dongarra (J.), Jiang (W.), Mancheck (R.) et Sunderam (V.). – *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, 1994.

[29] Geist (G.), Kohl (J.) et Papadopoulos (P.). – PVM and MPI : a comparison of features. *Calculateurs Parallèles*, no8(2), 1996.

[30] Gillett (R.). – Memory Channel fot PCI. *IEEE Micro*, vol. 16, n° 2, February 1996, pp. 12–18.

[31] Ginzburg (I.). – *Athapascan-0b: Intégration efficace et portable de multiprogrammation légère et de communications.* – LMC, Thèse de PhD, Institut National Polytechnique de Grenoble, Sep 1997.

[32] Grimshaw (A.), Strayer (W.) et Narayan (P.). – Dynamic object-oriented parallel. *In : Distributed Technology: Systems and Applications*, pp. 33–47.

[33] Haines (M.), Cronk (D.) et Mehrotra (P.). – On the design of chant: A talking threads package. *In : Proc. of Supercomputing'94*, pp. 350–359. – Washington, November 1994.

[34] Haines (M.), Mehrotra (P.) et Cronk (D.). – *Chant: Lightweight Threads in a Distributed Memory Environment.* – Rapport technique, NASA Langley Research Center, ICASE, May 1995.

[35] High Performance Fortran Forum. – *High Performance Fortran Language Specification.* – Rice University, Houston, Texas, Nov 1994. Version 1.1.

[36] IEEE. – *Standard for Scalable Coherent Interface (SCI)*, Aug. 1993. Standard no. 1596.

[37] IEEE Standard for Multithreaded Programming. – *POSIX.1.c*, August 1993.

[38] Langedoe (K.), Romein (J.), Bhoedjang (R.) et Bal (H.). – Integrating polling, interrupts and thread management. *In : 6th Symposium on the Frontiers of Massively Parallel Computing (Frontier's 96).* – Annapolis, Maryland, 1996.

[39] Maquelin (O.), Gao (G.), Hum (H.), Theobald (K.) et Tian (X.). – Combining polling and interrupts for efficient message handling. *In : Proceedings of 23th Symposium on Computer Architecture.* – Philadelphie, May 1996.

[40] Mascarenhas (E.) et Rego (V.). – *Ariadne: Architecture of a portable threads system supporting mobil processes.* – Rapport technique n° CSD-TR-95-017, Purdue University, March 1995.

[41] Message Passing Interface Forum. – *MPI: A Message-Passing Interface Standard*, March 1995. available from netlib2.cs.utk.edu.

[42] Mueller (F.). – A Library Implementation of POSIX Threads under UNIX. *In : Proceedings of the USENIX Conference*, pp. 29–41.

[43] Mueller (F.), Giering (E.) et Baker (T.). – Implementing ada 9x features using posix threads: Design issues. *TRI-Ada*, September 1993.

[44] Namyst (R.) et Mehaut (J.). – PM^2: Parallel Multithreaded Machine. a computing environment for distributed architectures. *In : ParCo'95 (PARallel COmputing).* pp. 279–285. – Elsevier Science Publishers.

[45] Namyst (R.) et Méhaut (J.-F.). – PM2 : Parallel Multithreaded Machine. A multithreaded environment on top of PVM. *In : 2nd Euro PVM UG Meeting*, pp. 179–184. – Lyon, Sept. 1995.

[46] Namyst (R.) et Méhaut (J.). – *Marcel : Une bibliothèque de processus légers.* – Laboratoire d'Informatique Fondamentale de Lille, Lille, 1995.

[47] Nikhil (R.). – Cid: A parallel, 'shared-memory' c for distributed-memory machines. *In : LNCS 892.* pp. 376–390. – Springer Verlag.

[48] Pakin (S.), Karamcheti (V.) et Chien (A.). – Fast Messages (FM: Efficient, Portable Communication for Workstation Cluster and Massively-Parallel Processors. *IEEE Concurrency*, 1997. – to appear.

[49] Potter (F.). – *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour machines multiprocesseurs.* – Thèse de PhD, Université Pierre et Marie Curie – Paris VI, Avril 1996.

[50] Provenzano (C.). – *Pthreads: an implementation of POSIX 1003.1c.* Available at http://www.mit.edu:8001/people/proven/pthreads.

[51] Prylli (L.) et Tourancheau (B.). – *Protocol design for high performance networking: A Myrinet experience.* – Rapport technique n° RR97-22, Ecole Nationale Supérieure de Lyon, 69364 Lyon Cedex 07, Laboratoire d'Informatique du Parallélisme, july 1997.

[52] Roucairol (C.). – *Stratagème : Une méthodologie de programmation parallèle pour les problèmes non structurés.* – Rapport de Recherche, Versailles, PRiSM, Dec 1995.

[53] Sun Microsystems, Moutanin View, USA. – *External Data Representation*, 1995.

[54] SunSoft. – *SunOs5.2 Guide to Multi-Thread Programming*, 1993.

[55] Von Eicken (T.), Culler (D.), Goldstein (S.) et Schauser (K.). – Active messages: a mechanism for integrated communication and computation. *In : Proc. 19th Int'l Symposium on Computer Architecture.* – Gold Coast, Australia, May 1992.

Chapitre 7

Régulation de charge dans un applicatif CORBA/Objet

Laurent Philippe et Hervé Guyennet (Laboratoire d'informatique de Besançon)

7.1 Introduction

Les systèmes à objets sont actuellement très utilisés dans les systèmes d'exploitation ouverts. L'OMG (Object management Group) mène leur normalisation en définissant CORBA (Commun Object Request Broker Architecture). La modulatité et la légerté du paradigme objet permet d'envisager un meilleur contrôle des applications au niveau du système, grâce à des possibilités d'observation et d'intervention plus fines. Ainsi, les informations collectées permettent l'utilisation de modèles d'observation plus complexes tels que le modèle relationnel, à condition de les adapter aux contraintes propres à notre environnement. Nous présentons, dans une première partie, la norme CORBA et son intérêt dans le cadre des applications parallèles irrégulières. Dans ce cadre, nous posons la problématique du placement dans ce type de systèmes et donnons un apperçu d'un modèle d'observation des systèmes dynamiques, le modèle relationnel. Pour finir, nous étudions l'adaptation de ce modèle à notre environnement et nous présentons une réalisation sur le système COOLv2.

7.2 Les systèmes répartis à objets

Le développement des systèmes répartis à objets a pris de l'ampleur ces dernières années. Nous pouvons en prendre pour preuve le nombre de systèmes commercialisés.

Dans cette partie, nous présenterons la norme CORBA qui définit l'interface de ces sytèmes.

7.2.1 L'OMG et l'OMA

Une des organisations principales dans le domaine des systèmes répartis à objets est l'Object Management Group (OMG). Ce consortium est à l'origine de la norme CORBA (Common Object Request Architecture) qui régit l'interface des systèmes répartis à objet et du modèle OMA (Object Management Architecture) qui définit l'environnement de l'ORB.

L'OMA (Object Management Architecture) est un modèle de référence définissant un système à objets comme un fournisseur de services à des clients. Ces services sont implantés par des objets serveurs et offerts aux clients au travers d'interfaces. Une interface décrit un ensemble d'opérations (méthodes) qu'un client peut demander à l'objet serveur. A la requête d'un client sont associés un objet cible, une opération, éventuellement des paramètres et un contexte optionnel. Les interfaces sont définies grâce au langage de définition d'interface (IDL). Un objet satisfait une interface s'il peut rendre l'ensemble des services définis dans cette interface.

L'OMA définit également des entités spécifiques :

- l'Object Request Broker (ORB) fournit les mécanismes d'échange transparent entre les objets quelque soit leur position dans l'environnement CORBA.
- Les fonctions communes correspondent aux services tels que l'aide en ligne, l'impression, etc.
- Les services objets sont les services optionnels tels que la persistance, le nommage, etc.
- Les objets applicatifs désignent les objets spécifiques des applications (des clients et des serveurs).

7.2.2 L'architecture CORBA

L'architecture CORBA définit le modèle du bus de communication logiciel (ORB). Ce bus permet une communication transparente à la répartition et à l'hétérogénéité. Ses composants sont donnés par la figure 7.1.

On distingue :

- les souches client ou *stub* qui sont générées à partir de la déclaration de l'interface d'un serveur. Le client dispose d'une souche par interface qu'il importe. Cette souche correspond à la mise sous forme de message des appels sur les opérations exportées par l'interface.

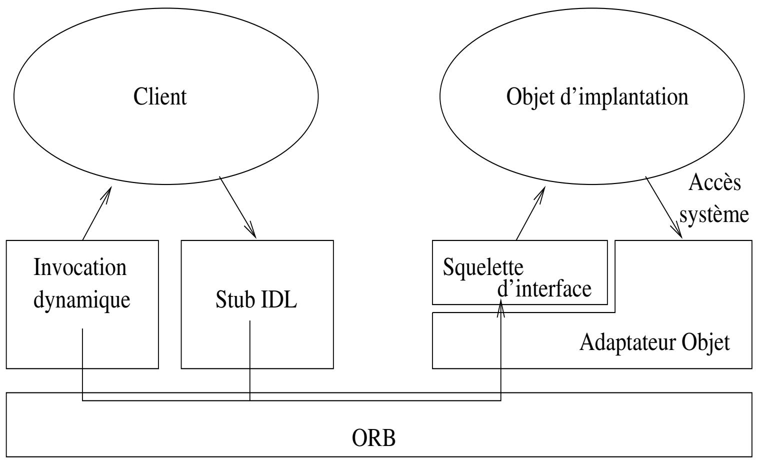


Figure 7.1 : Architecture CORBA

- l'adapteur d'objet qui est un environnement d'exécution (run-time) pour les serveurs. C'est l'adaptateur d'objet qui exécute une méthode lors de son appel par un client.
- le squelette d'implémentation qui est également généré à partir de la déclaration des interfaces. C'est la remise en forme des données d'invocation qui ont été tramisent par le client, de manière à pouvoir réaliser une invocation locale des opérations du serveur.
- le noyau ORB, sur lequel reposent les composants précédents, offre le support et le liant nécessaire à l'intégration de l'environnement. C'est lui qui réalise l'acheminnement des messages, identifie les différentes entités, etc.
- l'invocation dynamique est un composant qui permet à un client de construire une requête sans disposer d'un objet d'interface.

7.2.3 Définition d'applications sur CORBA

La norme CORBA ne définit pas, à priori, de langage spécifique pour les applications qui utilisent un ORB. En pratique, le C++ est le langage le plus couramment proposé mais des langages tels que Java semblent également avoir de l'avenir.

Lorsqu'un serveur veut se rendre accessible de ses clients, il déclare les méthodes qu'il veut rendre accessibles dans un objet d'interface. Les objets d'interface sont écrit en IDL (Interface Definition Langage), syntaxiquement proche du C++. Ces objets sont ensuite compilés et liés au code du serveur sous forme de squelettes.

Le lien entre un client et son serveur est réalisé dynamiquement. Un objet d'interface peut-être exporté vers un serveur de nom qui associe un nom symbolique à chacune des interfaces qui lui est déclarée. Le client importe alors l'objet d'interface en faisant référence au nom symbolique. L'invocation du serveur se fait alors par invocation des méthodes déclarées dans l'objet d'interface.

Ces deux phases supposent un travail d'initialisation des communications, ce qui est courant et pourrait facilement être simplifié par un environnement de programmation. La démarche est résolument orientée Client/Serveur dans le cadre d'une relation mais rien n'empêche d'utiliser cette démarche dans les deux sens pour assurer la symétrie. Pour le reste la programmation des applications est classique.

Divers aspects peuvent présenter des intérêts dans le cadre de la programmation parallèle :

- la modularité du paradigme objet est reconnue comme bonne pour tous les aspect du génie logiciel,
- par rapport à des applications découpées en processus la taille du grain des objets est plus adaptée au parallélisme,
- les possiblités de création dynamique d'objets et de liens dynamiques peuvent intéresser le programmeur d'applications irrégulières.
- la possibilité de gérer des activités, ou *thread*, pour la programmation dynamique.

Notons cependant que nous n'avons présenté ici que les aspects de la norme CORBA qui peuvent intéresser la programmation parallèle. Cette norme est cependant beaucoup plus étendue. En effet, elle donne également des directives sur la sécurité, la définition de domaines, l'hétérogénéité, etc.

7.3 La régulation de charge chez les objets

L'équilibrage de charge est un domaine qui a été largement étudié ces dernières années, en particulier dans le champ du placement de processus dans les environnements de type UNIX. Il commence à apparaître comme une fonction nécessaire pour les systèmes répartis à grand nombre de sites. Les avantages apportés par l'équilibrage de charge sont nombreux : l'utilisation des ressources est rationalisée et optimisée, les applications dynamiques ou irrégulières sont gérées automatiquement, sans souci de placement pour le programmeur.

Comme nous l'avons vu le paradigme objet et les Systèmes Répartis à Objets (SRO) présentent un intérêt dans la programmation d'applications dynamiques, dont les applications parallèles irrégulières. Cependant, les SRO posent de nouvelles questions dans le domaine de l'équilibrage de charge. Comment transférer la technologie développée pour les processus vers le domaine des objets ? Quelles sont les différences entre les systèmes distribués à base de processus et les SRO ? Les objets n'accomplissent une tâche complexe que par coopération, comment gérer cette interdépendance logicielle et le flux de communication qui en découle ?

De plus, il faut noter que, si les objets ont un intérêt incontestable au niveau de la programmation, ils en ont également dans le cadre du placement dynamique. Ainsi, notre travail est motivé par les constatations suivantes :

- la conception d'applications dans les systèmes répartis supportant les objets est en pleine expansion. Le principe de base de ce type de programmation est comparable à l'approche client-serveur, un objet client fait appel à des méthodes d'un objet serveur présent dans le système. Un serveur peut être soit sur le même site que son client (problème de charge du processeur), soit sur un site distant (problème de charge du réseau de communication). Dans les deux cas il faut envisager de gérer le placement pour optimiser le temps de traitement dans le serveur. De plus, cet objet serveur peut lui-même être client d'autres serveurs.

- les objets sont des entités de petites tailles évoluant dans des contextes de petites tailles : ceci doit permettre des interventions fines sur le système et donc mieux adaptées aux besoins des utilisateurs.

- l'apparition de supports d'exécution répartis pour les applications objets qui permet d'envisager deux niveaux d'intervention : au niveau du langage (par analyse du code) ou/et au niveau de l'exécution (par analyse du comportement).

- le standard défini par l'OMG [3] [4] introduit la notion d'IDL. Il s'agit d'un langage de définition d'interfaces standardisant les fonctions d'échanges entre les objets. Ces interfaces standards facilitent l'accès aux informations concernant les communications en implantant des mécanismes standards, les moniteurs, d'observation des échanges. Cet aspect est souvent un problème difficile dans le cadre de l'équilibrage de processus [6]. Il n'est ainsi plus nécessaire d'intervenir dans le système pour obtenir des informations à jour et pertinentes. Dans ce cadre, le placement automatique des objets peut être vu comme un service et donc implanté sous la forme d'un serveur externe. Le gain de modularité est évident. Cette approche plus souple permet également d'offrir un service "à la carte" aux applications.

Bien que le domaine des systèmes répartis à objets soit relativement jeune, plusieurs équipes se sont intéressées au problème de l'équilibrage de charge. Nous présentons ci-dessous trois de ces projets, en décrivant succinctement le système et l'approche choisie pour réaliser l'équilibrage.

- Au sein du projet GUIDE, une étude a été menée pour inclure un système de gestion de charge [8] sur la base de travaux concernant le comportement et la granularité des objets [9].

- *The Bellerophon load-balancer* [5] réalise des groupes d'objets (*clump*) qu'il déplace vers des sites moins chargés. Aux contraintes de charge vient s'ajouter un contrôle du déplacement par des indicateurs nommés *colocator* et *contralocator*. Deux clumps disposant d'un *colocator* l'un pour l'autre sont regroupés sur un même site. Alors que deux clumps disposant d'un *contralocator* ne sont jamais regroupés.

- Dans Muse [10], les informations utilisées sont des forces d'attraction ou de répulsion calculées en fonction du débit de communication et de l'activité des objets. Les objets sont plongés dans un champ, et soumis à des forces. Ils se déplacent pour trouver une position d'équilibre. Des travaux sont menés pour tenter de définir une métrique pour l'espace dans lequel évoluent les objets.

Nous pouvons noter que la plupart des réalisations dans les SRO utilisent les relations entre objets comme information stratégique dans le placement. Pour cette raison, nous avons cherché à définir un modèle de comportement d'observation et de décision pour la prise en compte de ces informations.

7.4 Le modèle d'observation

Le modèle relationnel, baptisé "dérive des connaissances" [2] est issu de réflexions sur l'augmentation des échanges de données électroniques qui pose le problème du rapport entre la notion de connaissance et celle d'information. Le modèle relationnel part de l'hypothèse selon laquelle toute entité individuelle existe surtout à travers le rôle qu'elle joue au sein de collectivités. C'est à partir de cette facette relationnelle représentant les conditions d'évolutions des interactions des entités entre elles, que le modèle relationnel propose des critères de pertinence pour transformer de l'information en connaissance.

L'espace relationnel se décrit suivant trois points de vue qui se complètent et s'influencent mutuellement. Il se compose d'un graphe non nécessairement connexe, où les noeuds représentant les informations en interaction, sont reliés entre eux par des liens fictifs. Les valeurs attachées à ces liens quantifient la nature des interactions entre les noeuds correspondants.

7.4.1 Le point de vue local : les noeuds

Chaque noeud possède sa vision de la base au travers des liens qu'il entretient. Pour chaque noeud ces liens sont de deux types ; il y a ceux qui correspondent aux sollicitations vers ce noeud et ceux qui correspondent aux sollicitations à partir de ce noeud. Plusieurs paramètres décrivent chaque noeud dans l'espace relationnel ; c'est le cas de la masse relationnelle qui est une mesure instantanée de l'ensemble des interactions du noeud. Cette vision locale se retrouve dans les calculs des différents paramètres relationnels, puisque à aucun moment ces calculs n'intègrent pour un noeud donné, le résultat des conséquences dues aux perturbations dont il est indirectement la cause; c'est-à-dire celles qui sortent de son espace local d'interaction. Un noeud détermine sa masse relationnelle en observant les invocations dans lesquelles il est impliqué.

7.4.2 Le point de vue collectif : les formes relationnelles collectives

L'une des propriétés des systèmes observés est que ses éléments (noeuds) interagissent entre eux de façon concurrente. Ceci implique que le modèle travaille à un niveau collectif qui dépasse la vision individuelle de chaque noeud de la base. Pour bien prendre en compte ce phénomène, le modèle introduit la notion de forme relationnelle collective.

Une forme relationnelle collective est un sous-graphe connexe de la base qui est activé sous l'impulsion d'excitations ou sollicitations. Elle correspond donc à un regroupement d'entités fonctionnellement liées entre elles. Pour qu'une forme joue un rôle dans le modèle relationnel, il faut qu'elle soit homogène pendant son existence. L'homogénéité définit une propriété collective de la forme qui autorise des transformations identiques de chaque élément de la forme sans remettre en cause l'identité de cette dernière. Cette propriété garantit la conservation des formes à partir de leur structure invariante. Lorsque des parties de ces formes évoluent différemment des parties restantes, il se produit alors une rupture d'homogénéité. Une première tentative de description catégorielle de la notion de forme a été faite dans [1].

7.4.3 Le point de vue global : la base

La base relationnelle (BCO) regroupe l'ensemble du graphe dans lequel on observe l'évolution dynamique de formes relationnelles. La base possède ses propres paramètres relationnels qui caractérisent son état relationnel statique mais aussi dynamique (évolution). Grâce à ces paramètres relationnels, on peut construire de nouveaux paramètres utiles pour les mécanismes évolutionnistes. Suivant le type d'heuristique qualitative que nous voulons introduire dans le modèle relationnel et les choix algorithmiques d'implémentation (pseudo-répartition ou répartition), les paramètres relationnels de la base peuvent être utilisés dans le calcul du comportement évolutif de chaque noeud de la base.

7.4.4 Les flux excitatoires et entropiques

L'intensité des liens qui relient les noeuds entre eux dépend principalement des activations réellement faites entre ces noeuds. C'est le domaine d'application choisi qui précise la notion d'activation. Plus un noeud active un autre noeud et plus le coefficient (intensité relationnelle [2]) attaché au lien correspondant est grand. Le flux excitatoire précise la façon dont croît ce coefficient. Dans la mesure où l'utilisation du modèle relationnel suit l'heuristique selon laquelle la tendance est de regrouper les entités qui entretiennent entre elles un haut niveau d'interaction, il fallait introduire une composante qui s'oppose à des agglomérations définitives. C'est le rôle que joue le flux entropique en diminuant périodiquement les valeurs des liens en fonction du contexte.

Puisque l'un des apports de ce modèle relationnel est d'expliciter ces formes collectives pour en tenir compte dans le comportement du système, afin d'en améliorer son fonctionnement, l'échelle temporelle dans laquelle se jouent ces émergences est importante. Il faut que les interactions génératrices de formes relationnelles soient significatives, c'est-à-dire qu'elles durent dans le temps.

7.4.5 L'invariance perceptive

C'est grâce au mécanisme d'invariance que l'on va détecter les formes relationnelles et suivre leurs éventuelles évolutions. L'invariance perceptive s'applique à chacun des liens issus de tout noeud N de la base. Il s'agit pour ce noeud N de savoir si l'action des interactions qu'il a vers chacun des noeuds qu'il voit, provoque chez le noeud cible une modification structurelle invariante. En d'autres termes quand N active un lien vers un autre noeud, il regarde si l'évolution de ce noeud est corrélée à cette activation. Une forme sera donc repérée comme stable, lorsqu'il existera des fonctions d'invariance sur chacun de ses liens.

7.5 Application du modèle aux SRO

Le modèle relationnel peut s'appliquer à l'ensemble des environnements dynamiques dans lesquels des entités sont en relation et dont on veut observer lévolution des interactions au cours du temps. Il s'applique donc bien au contexte des SRO. Il nécessite alors une adaptation plus fine, propre à l'environnement, prinicpalement pour les algorithmes de prise décision. Dans le cas des SRO, les entités observées se situent au niveau des objets et les interactions proviennent des invocations distantes. Nous avons donc adapté le modèle relationnel pour l'appliquer au contexte du placement et nous y avons adjoint les algorithmes de prise de décision nécessaires.

Nous considérons donc comme modèle d'application un ensemble d'objets communicant qui traduit le graphe des tâches. La plupart des objets peuvent ainsi être à la fois client et serveur. Le graphe n'est pas nécessairement connexe, ce qui peut traduire l'exécution simultanée de plusieurs applications.

7.5.1 Mise en oeuvre de l'observation

Dans un environnement à grand nombre de sites, il n'est pas réaliste de centraliser les traitements ou de réaliser un état global du système. L'utilisation de ces techniques engendre des points de contention sur le réseau et une faible résistance aux fautes. Dans ce cas, la mise en oeuvre de la base (BCO) d'observation est supposée être distribuée sur l'ensemble des sites concernés par le placement.

La notion de flux excitatoire est relativement aisée à mettre en oeuvre pour le placement. Il faut marquer les liens avec des quantités proportionnelles au débit de communication entre objets. La notion d'entropie est plus difficile à concevoir. En fait, le phénomène entropique traduit l'idée qu'un lien qui n'est pas réactivé périodiquement doit voir son intensité diminuer. Pour ce faire, nous introduisons l'entropie sous deux formes : l'une associée au temps et l'autre associée à la charge des sites impliqués dans la relation.

Pour introduire la notion de temps nous utilisons deux quantités : le débit instantané (d = Volume échangé par unité de temps) et le débit moyen (D = Moyenne pondérée des débits instantanés). D, à l'instant t, est obtenu par le calcul suivant :

$$D_t = \frac{\alpha D_{t-1} + \beta d}{\alpha + \beta} \tag{7.1}$$

Les constantes α et β paramètrent l'inertie du modèle (plus β est grand devant α, plus le modèle est sensible aux variations de débit).

Le modèle relationnel se préocupe peut des contraintes pratiques qui régissent le placement d'objets. Il définit ainsi un plan d'observation du comportement des applications. Son utilisation suppose de projeter ce plan sur le plan physique pour définir le placement. L'introduction de la charge permet de tenir compte des contraintes propres aux sites. Soit I l'intensité relationnelle portée par un lien, I est donnée par le calcul

$$I = \frac{1}{C} D \tag{7.2}$$

Où C est un coefficient calculé à partir de la charge. Si C est proportionnel à la charge, l'oubli de l'objet par disparition du lien est favorisé. Si C est inversement proportionnel à la charge, la migration de l'objet est favorisée.

7.5.2 Le modèle de décision

La détection des formes de notre modèle implique un parcours des chemins d'activation qui peut être coûteux. Dans le cadre du placement nous avons choisi de détecter l'homogénéité sur une longueur de un lien : l'ensemble des objets ayant des liens stables avec un même objet est considéré comme appartenant à sa forme relationnelle. L'apparition d'une rupture d'homogénéité dans une forme déclenche une reconsidération du placement de ces membres.

La procédure de détection d'une rupture d'homogénéité est la suivante. Pour chaque objet, on définit le rapport R obtenu en divisant l'intensité relationnelle I par la somme des intensités relationnelles de tous les liens de l'objet. R traduit l'importance de la relation considérée parmi l'ensemble des relations de l'objet. Soit A un objet entretenant des communications avec n sites notés S_1,..,S_n, et soit I_j l'intensité relationnelle du lien entre A et le site S_j alors R_j, l'importance relative du lien (A, S_j) est donnée par

$$R_j = \frac{I_j}{\sum_{k=1}^{n} I_k} \tag{7.3}$$

Une forme est dite homogène si la part relative de chacun des liens qui la composent est conservée dans le temps. Considérons une forme simplifiée, centrée sur un objet, et composée de l'ensemble des liens issus de cet objet. En régime stationnaire, la part de chaque lien reste stable si les volumes de données échangés restent stables, augmentent ou diminuent en même temps. Examinons le cas d'une excitation, donc d'une augmentation de volume sur un des liens, sans évolution dans les autres relations (une analyse équivalente peut être faite dans le cas d'une diminution). La part relative du lien excité va augmenter et celles des relations stables va diminuer. Le gestionnaire utilise la somme des valeurs absolues des différences entre R_t et R_{t-1} sur chacun des liens pour détecter les pertes d'homogénéité. Soit PH l'indicateur de perte d'homogénéité, et n le nombre de liens attachés à l'objet considéré. PH est obtenu par :

$$PH = \sum_{k=1}^{n} |R_{k,t} - R_{k,t-1}| \tag{7.4}$$

Cette valeur est nulle si l'homogénéité est conservée et positive dans le cas contraire. Une réorganisation du modèle est nécessaire si PH est supérieur à un seuil.

Si une réorganisation s'avère nécessaire, il faut trouver un objet à migrer. L'information sur la perte d'homogénéité n'est pas suffisante car elle est indépendante du volume traité. Aussi le choix de l'objet est fait en fonction des volumes échangés parmi les formes nécessitant une réorganisation, et la migration est réalisée vers le site avec lequel l'objet échange le plus d'information.

Les migrations ont lieu uniquement si le site distant accepte l'excédent de charge induit. En effet, si nous n'utisions que ces critères pour gérer le placement dynamique des objets, il est fort probable que le système évoluerait à moyen terme vers un regroupement des applications sur les mêmes noeuds. Il est donc nécessaire d'introduire dans les critères de décision des barrières qui limitent les échanges d'objet entre les sites dont la différence de charge est trop importante. Ces barrières seront fonction de type d'équilibrage de charge qui veut être obtenu.

7.6 Une réalisation sur COOLv2

L'intérêt du modèle relationnel a été montré au moyen d'une maquette, implantant le placement d'objets dans le système COOLv2. Nous présentons d'abord l'environnement applicatif dans lequel s'est déroulée cette expérience puis nous décrivons la réalisation.

En fonction du type d'environnement géré par le service de gestion de charge on peut avoir des contraintes différentes.

La mise en oeuvre de mécanismes prenant en compte le comportement global du modèle est difficile dans un environnement réparti. En effet, les gestionnaires doivent échanger beaucoup d'informations pour obtenir une connaissance de l'état de la base à un instant donné. Aussi, nous nous restreindrons à l'utilisation de mécanismes locaux (noeuds et sites) et semi-locaux (formes).

7.6.1 Environnement applicatif

L'environnement applicatif visé initialement par le projet se situait plus dans les domaine des systèmes distribués que du parallélisme. Néanmoins les environnements objet peuvent convenir à des applications très variées. Ils ont déjà fait leur preuves pour la plupart des applications classiques en tant qu'outil efficace de développement et de mise au point.

Dans le cadre plus spécifique de la régulation de charge, les applications visées sont à caractère dynamique. Il s'agit donc d'applications dont on ne connaît pas, à priori, le comportement. Une classe de ces applications peut donc être les applications parallèles irrégulières puisque le graphe de tâche qui leur est associé est construit au cours de leur exécution. Le paradigme objet présente quelques avantages pour ce type d'applications, en effet, il permet de construire aussi bien des entités de petite taille que de taille moyenne à grosse. L'environnement objet apporte alors le liant au moyen des communications par invocation de méthode à distance.

7.6.2 COOL v2

Le système COOL (Chorus Object Oriented Layer) [7] fournit une infrastructure pour développer des applications réparties à base d'objets au-dessus du micro-noyau Chorus. COOL v2 est conforme à la spécification de l'OMG pour la réalisation de systèmes répartis à objets. Cette couche étend la norme en introduisant les notions de distribution, de persistance et d'interopérabilité.

L'espace d'adressage du système COOL v2 (figure 7.2) est structuré par des **clusters**, qui sont des groupes d'objets, fortement liés construits au niveau de

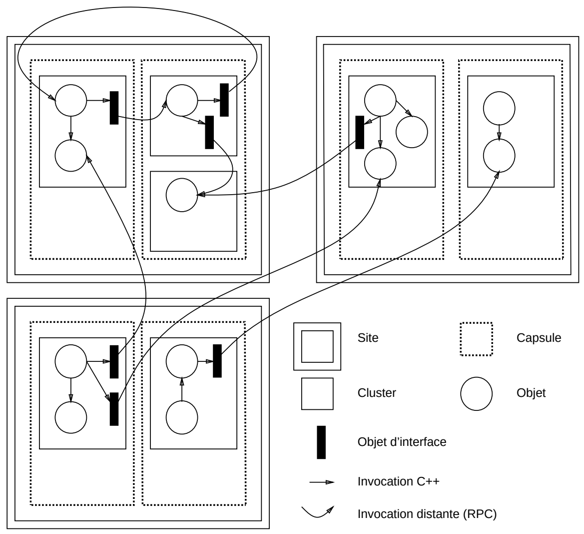


Figure 7.2 : Principales entités de COOL.

l'application. Les clusters sont alloués à l'intérieur de **capsules** qui sont implémentées dans des processus UNIX. Chaque capsule dispose d'un ensemble d'activités prenant en charge les requêtes réalisées sur les objets des clusters contenus dans la capsule.

La migration d'objets est une alternative à l'invocation distante, COOL permet de migrer un objet serveur, distant, dans l'espace d'adressage du client et de procéder ensuite par appel de méthode local (RPC intra-site). La migration ne s'applique pas directement à des objets, mais à des clusters de manière à obtenir une granularité en relation avec le coût de l'opération.

7.6.3 Le serveur de placement

Le placement réalisé se fait sur la base de cluster puisque cette entité est la grain de base de la migration. L'ensemble de la définition du modèle relationnel se fait donc sur cette base, les noeuds étant représentés par des clusters et la valuation des liens est basée sur la somme des communications réalisées par les objets contenus dans un cluster. Nous utilisons la migration pour réaliser un placement dynamique des clusters.

Pour réaliser une véritable gestion répartie du placement, nous équipons chaque site d'un gestionnaire de placement. Ce gestionnaire est chargé de la gestion des clusters locaux, du maintient d'informations sur la charge locale et des sites avec lesquels les cluster locaux entretiennent une relation, et finalement de la détermination, à intervalles de temps réguliers, de la nécessité d'une réorganisation (un déplacement) des objets.

L'observation des relations a, dans un premier temps, été réalisée par surcharge des méthodes d'invocation distante. Il s'agit donc d'une intervention au niveau du langage. Ceci nous a permis de d'implanter notre réalisation sous forme d'une application s'exécutant au dessus de COOLv2. Il est à noter qu'à travers les *common services* définis dans la norme CORBA, il devrait être possible d'intégrer cette réalisation comme un des services standards offerts par l'environnement.

Pour gérer les déséquilibres de charge trop importants, nous introduisons deux traitements exceptionnels qui correspondent à deux situations extrêmes : la sous-charge et la sur-charge. En effet, les sites sous-chargés, ne disposant que de peu de relations, sont peu attracteurs. Dans ces deux cas, les serveurs des différents sites coopèrent pour reéquilibrer la charge en déplaçant les clusters qui modifient le moins la structure relationnelle des applications et en utilisant des algorithmes classiques de gestion de charge.

7.7 Conclusion

Nous avons présenté les systèmes répartis à objet et la problématique de la gestion de charge dans ces environnements. Ce contexte permet une observation fine du comportement des applications et l'utilisation de modèles d'observation complexes tels que le modèle relationnel. Il est ainsi possible d'associer un service système déquilibrage de charge avec des informations obtenues directement au niveau du langage. En généralisant cette approche pour les applications complexes telles que les applications parallèles irrégulières, il est possible des définir des placements adaptés au comportement des applications.

Bibliographie

[1] Albonico (D.). – Dérive des connaissances et théorie des catégories. – Projet de fin études, ENST Bretagne. Brest, mars 1994.

[2] Bourdon (F.). – *Un modèle de dérive des connaissances, application à la bureautique.* – Thèse de PhD, Université du Maine, july 1992.

[3] Consortium (T. O.). – *The Object Management Architecture Guide.* – Rapport technique n° 90.9.1, Farmington MA. USA, Object Management Groupe, 1990.

[4] Consortium (T. O.). – *The Common Object Request Broker: Architecture and Specification.* – Rapport technique n° 93.12.29, Farmington MA. USA, Object Management Groupe, 1993.

[5] Dickman (P.). – Effective load balancing in a distributed object-support operating system. – IEEE, 1991.

[6] Guyennet (H.), Herrmann (B.), Philippe (L.) et Spies (F.). – A performance study of dynamic load balancing algorithms for multicomputers. *In : MPCS'94*, éd. par Ed. (I.). – Ischia, Italie, May 1994.

[7] Jacquemot (C.). – *Chorus/COOL v2 Reference Manual.* – Rapport technique n° CS/TR-94-16, Chorus system, 1994.

[8] Jensen (C. D.). – Elvis: Load distribution in the distributed object-oriented operating system guide. – Rapport préliminaire de master, september 1994.

[9] Lacourte (S.) et Riveill (M.). – Granularity of objects in distributed systems. *In : Workshop on Granularity of Objects in Distributed Systems.* – Kaiserlautern - Germany, july 1993.

[10] Tokoro (M.). – Computational field model: Toward a new computing model/methodology for open distributed environment. June 1990.

Chapitre 8

Ordonnancement de graphes de tâches paramétrés

Michel Cosnard (LORIA, INRIA-loraine), Emmanuel Jeannot (LIP, ENS-Lyon)

8.1 Introduction

Traiter des problèmes de grande taille est un des buts du parallélisme. En ce qui concerne le parallélisme de contrôle, le graphe de tâches est un modèle souvent utiliser pour décrire le programme séquentiel. Il s'agit d'un graphe acyclique où chaque sommet représente des calculs (les tâches) et où chaque arc représente les dépendances entre les calculs (il s'agit souvent de communication). L'ordonnancement du graphe de tâches consiste à attribuer à chaque tâche un processeur et une date de début d'exécution. Les ordonnanceurs statiques ont besoin d'avoir en mémoire tout le graphe de tâches pour l'ordonnancer. Or, quand les calculs sont très importants le graphe de tâches devient très encombrant. Il n'est alors pas possible d'ordonnancer ces graphes là. Nous décrivons ici un algorithme qui permet, pour certains types de programmes, d'ordonnancer le graphe de tâches, sans que celui-ci ne sois jamais entièrement en mémoire. Il s'agit d'un algorithme *dynamique* - l'ordonnancement est construit en cours d'exécution -, qui utilise une représentation symbolique du graphe de tâches: le graphe de tâches paramétré (GTP) [4]. Le GTP est un modèle intermédiaire, grâce auquel une fois la valeur des paramètres du programme instanciée, on peut construire le graphe de tâche. Le fait que cet algorithme soit dynamique permet de construire un programme générique où les valeurs des paramètres sont données en début d'exécution.

8.2 Techniques et modèles concernant l'ordonnancement

Le graphe de tâches est un modèle qui représente les dépendances entre les différentes parties (tâches) d'un programme séquentiel. Ordonnancer un graphe de tâches va consister à déterminer un processeur pour chaque tâche en respectant les dépendances entre elles et en cherchant à minimiser les communications tout en conservant un maximum de parallélisme.

8.2.1 Le graphe de tâches

8.2.1.a Définitions

On appelle:

- **tâche**, une instruction ou un ensemble d'instructions devant être exécutés sans interruption, séquentiellement.
- *graphe de tâches* le quadruplet $G = (V, E, W, C)$ où:
  - V est l'ensemble des sommets. Chaque sommet représente une tâche. On note v le nombre de sommets.
  - $E \subset V \times V$ est l'ensemble des arêtes. Il y a une arête de s_1 à s_2 $(s_1, s_2) \in E$, si et seulement si on ne peut exécuter la tâche s_2 qu'une fois l'exécution de s_1 terminée. On note e le nombre d'arêtes.
  - W est une fonction de V dans $\mathbb{R}$. $W(T)$ représente le coût d'exécution (la durée) de la tâche T.
  - C est une fonction de E dans $\mathbb{R}$. $C(T, T')$ représente le coût de communication (le volume de données transmis) de la tâche T à la tâche T'. Si $\forall T, T', C(T, T') = 0$, on parlera de graphes sans communication.

  Un exemple de graphe de tâches est donné dans la figure 8.1 (la tâche $T4$ a une durée de 2, elle ne peut être exécutée qu'une fois $T3$ terminée. La durée des communications entre $T3$ et $T4$, est de 4 (si elles sont exécutées sur des processeurs différents - sinon il n'y a pas de données à transmettre).

  Le graphe correspond au modèle *macro data flow*. Ce modèle d'exécution des tâches suit pour chaque tâche, le protocole suivant :

  1. Avant de s'exécuter sur son processeur, la tâche reçoit toutes les données nécessaires à ses calculs.
  2. Elle s'exécute ensuite sans interruption. Tous les processeurs sont identiques.

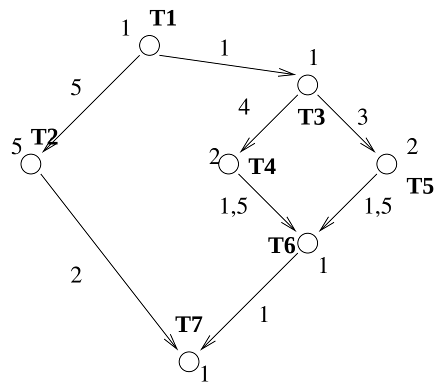


Figure 8.1 : Un exemple de graphe de tâches, avec communication

3. Elle transmet ensuite les données aux tâches "fille" (celles qui ont besoin des données qu'elle a calculées), si nécessaire.

- **dépendance** entre deux tâches T et T', l'existence dans G d'un chemin de T à T', on le note $T \ll T'$.

- **grain** d'un graphe G, le nombre

$$g(G) = \min_{i=1:v} \left(\min \left((\tau_i / \max_{j \in pred(i)} c_{j,i}), (\tau_i / \max_{j \in succ(i)} c_{i,j}) \right) \right)$$

  où $succ(i)$ (resp. $pred(i)$) est l'ensemble des successeurs (resp. prédécesseurs) de i dans G. On dira qu'un graphe de tâches est à **gros grain** si $g(G) > 1$, c'est à dire si tous les coûts de communication sont plus faible que toutes les durées des tâches.

- *ordonnancement*, l'assignation, pour chaque tâche, d'une date de début d'exécution, et d'un processeur. Pour être valide, cette assignation doit respecter les deux contraintes suivantes:

  1. Les *contraintes de ressources* : à un instant donné un processeur ne peut exécuter qu'une seule tâche.
  2. Les *contraintes de dépendances* : Soit deux tâches T et T' telles que, $T \ll T'$ alors on ne peut commencer l'exécution de T' qu'après la fin de celle de T plus d'éventuelles communications si T' n'est pas exécutée sur le même processeur que T.

- **makespan**, le temps d'exécution total de l'ordonnancement. Soient t une tâche, T l'ensemble de toutes les tâches, $debut(t)$ la date de début d'exécution, $W(t)$ la durée de t, M le makespan :

$$M = max_{t \in T}(debut(t) + W(t))$$

8.2.1.b Les heuristiques d'ordonnancement statiques

Nous présentons deux *heuristique* d'*ordonnancement statiques* (l'affectation des tâches aux processeurs est fixée avant l'exécution du programme). Il est donc nécessaire que le graphe de tâches soit entièrement connu. En particulier il faut que la durée de chaque tâche ainsi que le volume de communication soient déterminés à l'analyse du programme source, pendant la phase de compilation.

- *Les heuristiques de liste* : on construit une liste de tâches en fonction d'un critère particulier. Puis on exécute une tâche dès que celle-ci est prête (Earliest Task First [8]). S'il y a plus de tâches prêtes que de processeurs, on ordonnance les tâches par priorité suivant la liste. Plusieurs critères pour établir la liste sont étudiés dans [15].

- *Le regroupement (clustering)*. Dû à Sarkar [10], le regroupement se décompose en deux phases:

  1. Ordonnancer les tâches sur un nombre infini de processeurs. Cette phase fournie des groupes (*clusters*) de tâches, avec la contrainte que toutes les tâches d'un même groupe doivent s'exécuter sur le même processeur.
  2. Si le nombre de groupes est plus grand que le nombre de processeurs de la machine réelle, alors on fusionne ensemble les groupes jusqu'à n'avoir plus qu'un nombre de groupes égal au nombre réel de processeurs.

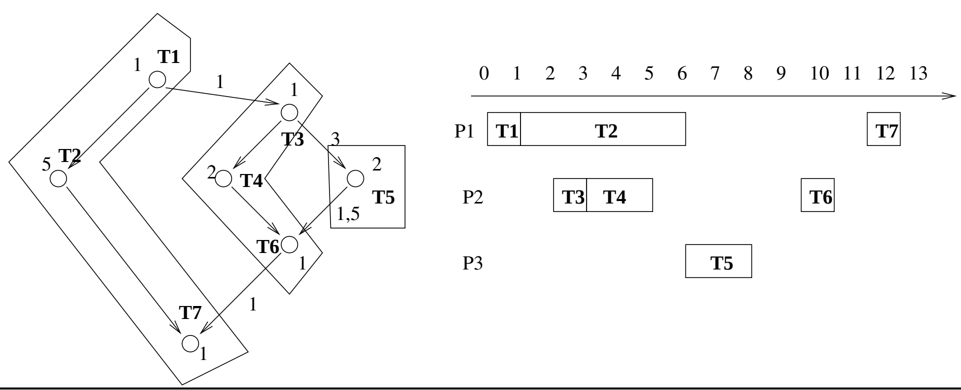


Figure 8.2 : Graphe de tâches groupé, schéma d'exécution

L'exemple de la figure 8.2 montre un regroupement possible de l'exemple 8.1 avec 3 groupes. Avec une exécution sur 3 processeurs, il n'y a plus de communication entre $T1$ et $T2$, puisqu'ils sont sur le même processeur. Il existe de nombreux algorithmes de regroupement [7].DSC de Yang et Gerasoulis [14], est dans le pire cas à 2 de l'optimal pour les graphes à gros grain.

8.2.1.c Ordonnancement statique, placement dynamique

Les méthodes *d'ordonnancement statiques* nécessite d'avoir le graphe de tâches à la compilation. Ceci implique deux problèmes majeurs:

- L'analyse statique du graphes de tâches est coûteux en mémoire. Les méthodes d'ordonnancement statique échouent si le graphe est très gros.

- Un programme parallèle est construit pour chaque graphe de tâches. Ainsi, si on change les paramètres du programme, on doit reconstruire le graphe de tâches. Les méthodes statiques ne permettent pas de construire des programmes génériques.

Pour pallier à ce problème un autre paradigme permet d'exécuter le graphe malgré tout. Les algorithmes de placement dynamique ordonnancent, en cours d'exécution, le graphe de tâches. Ils utilisent les informations partielles qu'ils ont à leur disposition : charge des processeurs, dépendances locales, etc... L'ordonnancement statique permet de faire de l'équilibrage de charge. C'est le cas de *Cilk*[1], qui implante la méthode de *work stealing*.

D'autre part *l'ordonnancement dynamique*, permet de traiter des problèmes irréguliers (le graphe de tâches n'est pas connu à la compilation).

8.3 PlusPyr et le graphe de tâches paramétré

Nous présentons ici des travaux qui ont en grande partie été réalisés au cours de la thèse de Michel Loi [9, 4].

8.3.1 PlusPyr

Dans le cadre du projet EuroTops [1], il a été développé un outil d'aide à la parallélisation : *PlusPyr*. *PlusPyr* est un logiciel qui permet, à partir d'un programme fortran annoté, de construire le graphe de tâches pour l'ordonnancer, et générer le code parallèle. Avec le programme *Pyrros*, *PlusPyr* forme une chaîne complète de parallélisation automatique. Une des contributions les plus originales de ces travaux est le graphe de tâches paramétré. C'est une représentation intermédiaire du graphe de tâches, qui est indépendante de la taille du problème à paralléliser. *PlusPyr* est capable, si le programme fortran respecte certaines contraintes, d'évaluer les coûts de communications entre tâches, et les coûts d'exécution de chacune d'elles.

[1] EuroTops est un projet Eurêka dont le maître d'œuvre est Matra Cap Systèmes et dont l'objectif est la conception d'un environnement de programmation parallèle.

8.3.2 Motivations

Le développement de *PlusPyr*, a été guidé par deux constatations:

- Les performances obtenues par l'utilisation d'outils d'allocation de graphes de tâches sont bonnes et, de plus, des prototypes efficaces de ces outils sont disponibles. Les algorithmes d'ordonnancement modernes permettent d'obtenir de bon résultat malgré la NP-complétude de l'ordonnancement dans le cas général ou, dans de nombreux cas restreints [6, 11]. Le regroupement [7], et en particulier l'algorithme *DSC* de Yang et Gerasoulis [14], permet de réaliser des ordonnancements efficaces, garantissant une performance à un facteur 2 de l'optimal. Des outils comme *Parallax* [5] permettent de comparer les différentes heuristiques d'ordonnancement, et *Pyrros* [13, 12, 5], permet d'ordonnancer et de générer le code à partir du graphe de tâches.

- Il n'est pas réaliste de demander à l'utilisateur de construire explicitement le graphe de tâches correspondant au programme à paralléliser. En effet, calculer les dépendances entre les tâches, les coûts de communication, et les coûts de calcul des tâches, est fastidieux et source d'erreur.

Pour fournir une chaîne complète de parallélisation automatique, il a été décidé de mettre au point un outil qui construit le graphe à partir du programme source.

8.3.3 Les contraintes sur le langage, les tâches

8.3.3.a Les contraintes

Pour pouvoir être analysé, le programme source doit respecter un certain nombre de contraintes. Le langage est du pseudo *Fortran*, où les expressions des bornes de boucles et de tableau doivent être des fonctions affines des paramètres du programme, des constantes et des indices de boucles englobantes. Le langage est augmenté de 2 délimiteurs syntaxiques `task` et `endtask`, qui servent à l'utilisateur pour regrouper des instructions en bloc d'exécution atomique (séquentielle), appelé *tâche*. Les tâches ne peuvent pas être imbriquées et toute instruction d'affectation doit être lexicalement inclue dans une tâche.

La figure 8.3 montre le programme de l'élimination de *Gauss* qui est paramétré par `n`, la taille du problème. Il respecte les contraintes énoncées plus haut.

8.3.3.b Les tâches, définition

Une tâche est définie par son nom, son vecteur d'itération et les délimiteurs de tâches (`task endtask`). Dans l'exemple 8.3, Il y a deux tâches $T1(k)$ et $T2(k,j)$. Le *vecteur d'itération* est le vecteur des indices de boucles englobantes pris dans l'ordre d'imbrication (dans l'exemple il s'agit respectivement de (k) et de (k,j)). On appellera *instance* d'une tâche, une affectation légale de son vecteur d'itération (par

```
param n
assert n >= 3
real a(n, n+1)
real s
for k = 1 to n-1 do
      task                                        /*T1(k)*/
         s= 1 / a(k,k)                            /*S1*/
         for l = k + 1 to n do
            a(l,k) = a(l,k) * s                   /*S2*/
         endfor
      endtask
      for j = k + 1 to n+1 do
        task                                         /*T2(k,j)*/
           for i= k + 1 to n do
                a(i,j) = a(i,j) - a(k,j) * a(i,k) /*S3*/
           endfor
        endtask
      endfor
endfor
```

Figure 8.3 : Exemple de programme: élimination de Gauss

exemple $T1(3)$). Si le vecteur d'itération est vide, alors la seule instance de la tâche T sera noté T.

8.3.4 Le graphe de tâches paramétré et sa construction

Nous abordons ici un des aspects fondamentaux des travaux traités dans ce rapport. Le *graphe de tâches paramétré* est la base des résultats présentés dans ce qui suit.

8.3.4.a Le graphe de tâches paramétré

C'est une représentation indépendante de la taille du problème, le graphe paramétré est composé de :

- Un ensemble de définitions d'instructions (vecteur d'itération, tâche englobante, bornes des indices de boucles englobantes).
- Un ensemble de définitions de tâches (vecteur d'itération, première et dernière instructions). Deux tâches, en plus de celles définies par l'utilisateur ont été

rajoutées. Il s'agit de *T-ENTREE* et de *T-SORTIE*. La tâches d'entrée (*T-ENTREE*) est exécutée avant toutes les autres et elle sert à la saisie des données nécessaires à l'exécution du programme (dans l'exemple 8.3, il est nécessaire de saisir le système que l'on veut résoudre). La tâche de sortie, est exécutée après toutes les autres et sert au retour des résultats.

- D'un ensemble de *règles de communication*, spécifiant les réceptions, les émissions, et les synchronisations entre les instances des tâches.

8.3.4.b Les règles de communication

Les *règles de communication* sont les principales composantes du graphe de tâches paramétré. Elles définissent de manière symbolique les transferts de données entre les instances des tâches. Toutes ces règles sont indépendantes de la taille du problème. Elles représentent une version compactée du graphe de tâches que peut ordonnancer un outil comme *Pyrros*. Les règles de communication permettent de spécifier plusieurs types de communications (diffusion, communication point à point, concentration, etc...). Les règles de communication s'expriment de manière duale en règles d'émission et en règles de réception. La figure 8.4 présente les règles d'émission du programme de l'exemple 8.3.

Considérons la règle de réception 9 de la figure 8.4:

$$\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T1(k)\} : \{A(i,k)|k+1 \leq i \leq n\}$$

Cette règle spécifie qu'à chaque étape k, la tâche $T2(k,j)$ reçoit de la tâche $T1(k)$ la colonne pivot courante $A(i,k)|k+1 \leq i \leq n$. D'une manière générale une règle de réception spécifie :

- la tâche réceptrice et les valeurs du vecteur d'itération valides,
- la tâche émettrice et les valeurs de son vecteur d'itération,
- les données transmises et le vecteur d'itération, quand il s'agit d'un tableau.

A la règle 9 de réception correspond la règle d'émission duale, qui se lit elle aussi de gauche à droite:

$$\{T1(k)|1 \leq k \leq n-1\} \rightarrow \{T2(k,j)|+1 \leq j \leq n+1\} : \{A(i,k)|k+1 \leq i \leq n\}$$

8.3.4.c La construction automatique du graphe de tâches paramétré

Pour construire le *graphe de tâches paramétré*, *PlusPyr* procède en plusieurs étapes:

1. $\{T1(k)|k=1\} \leftarrow \{T-ENTREE\} : \{A(k,k)\}$
2. $\{T1(k)|k=1\} \leftarrow \{T-ENTREE\} : \{A(l,k)|k+1 \leq l \leq n\}$
3. $\{T1(k)|2 \leq k \leq n-1\} \leftarrow \{T2(k-1,k)\} : \{A(k,k)\}$
4. $\{T1(k)|2 \leq k \leq n-1\} \leftarrow \{T2(k-1,k)\} : \{A(l,k)|k+1 \leq l \leq n\}$
5. $\{T2(k,j)|k=1, k+1 \leq j \leq n+1\} \leftarrow \{T-ENTREE\} : \{A(k,j)\}$
6. $\{T2(k,j)|k=1, k+1 \leq j \leq n+1\} \leftarrow \{T-ENTREE\} : \{A(i,j)|k+1 \leq i \leq n\}$
7. $\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T2(k-1,j)\} : \{A(k,j)\}$
8. $\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T2(k-1,j)\} : \{A(i,j)|k+1 \leq i \leq n\}$
9. $\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T1(k)\} : \{A(i,k)|k+1 \leq i \leq n\}$
10. $\{T-SORTIE\} \leftarrow \{T-ENTREE\} : \{A(i,j)|j+1 \leq i \leq n\}$
11. $\{T-SORTIE\} \leftarrow \{T1(j)|1 \leq j \leq n-1\} : \{A(i,j)|j+1 \leq i \leq n\}$
12. $\{T-SORTIE\} \leftarrow \{T2(i-1,j)|2 \leq i \leq n, i \leq j \leq n+1\} : \{A(i,j)\}$

Figure 8.4 : Règles de réception pour l'élimination de Gauss

1. Calcul des dépendances et du flot de données. Il s'agit d'analyser le programme source pour déterminer un ordre partiel entre les opérations du programme. Cet ordre partiel est déterminé par les dépendances entre les opérations qu'il faut respecter lors de l'exécution parallèle du programme. Si le programme respecte les contraintes énoncées sur les bornes d'indices de boucles, alors *PlusPyr* est capable de faire l'analyse exacte des dépendances mettant à jour la totalité du parallélisme du programme séquentiel. Si l'on considère deux instructions $I1$ et $I2$ et une variable r, $I1$ étant exécutée après $I2$, il existe 3 types de dépendances (comme le montre la figure 8.5):
   - La *dépendance de flot* si $I1$ écrit r et $I2$ lit r.
   - L'*anti-dépendance* si $I1$ lit r et $I2$ l'écrit.
   - La *dépendance de sortie* si $I1$ écrit r et $I2$ aussi.

```
I1:  r:=3;  a:=r;  r:=3;
I2:  a:=r;  r:=3;  r:=4;
```

Figure 8.5 : la dépendance de flot ; l'anti-dépendance; la dépendance de sortie

L'analyse du flot de données détermine, pour chaque référence faite par une instruction I à un scalaire ou à une entrée de tableau, l'opération source S qui a écrit la valeur lue par R. Le résultat de cette analyse est une fonction "source" qui donne S en fonction de R. Cette analyse est indispensable pour la génération automatique du graphe de tâches.

2. Prédiction du coût d'exécution des tâches : pour pouvoir allouer correctement en temps et en espace le graphe de tâches, il est nécessaire de connaître le coût de calcul de chaque tâche. Aussi, il a été développé un système qui associe de manière symbolique le coût exact d'une tâche en fonction de ses coordonnées dans l'espace d'itération (en fonction des paramètres du programme). Cela revient à calculer au sein d'une tâche, le nombre de fois que sera exécutée une instruction, et à multiplier le tout par le coût unitaire de l'instruction. Dans l'exemple 8.3 le coût de la tâche $T1(k)$ est $2 + 2 \times (n - k)$: (le coût de $S1$) + (le coût de $S2$)$\times$ (le nombre d'exécution de $S2$).

3. Génération des communications. Il s'agit d'expliciter à la fois les réceptions et les émissions effectuées entre les tâches. Cela ce fait grâce à l'analyse des dépendances et du flot de données et permet de construire les règles de communication comme celles de l'exemple 8.4. Il s'agit de regrouper les arcs du graphe *data-flow*, qui vont d'instruction à instruction, par un seul arc qui va de tâches à tâches, en éliminant les arcs intra tâche.

8.4 Ordonnancement dynamique du graphe de tâches paramétré

Comme nous l'avons vu, il n'est pas possible d'ordonnancer statiquement un graphe de tâches si celui-ci est trop gros. De plus l'ordonnancement statique ne permet pas de créer un programme générique. Le *graphe de tâches paramétré* est une représentation compacte, peu coûteuse en mémoire du graphe de tâches, et *l'ordonnancement dynamique*, avec instanciation des paramètres à l'exécution permet de réaliser un seul programme parallèle pour un programme séquentiel. Nous présentons ici *PTGDS* (Parameterized Task Graph Dynamic Scheduling) , un algorithme d'ordonnancement du graphe de tâches paramétré, qui demande les paramètres du programme à l'exécution. PTGDS, permet donc de traiter des problèmes de grandes tailles, et de construire des programmes génériques.

8.4.1 PTGDS l'algorithme d'ordonnancement

```
1 ordonnance(instance T){
2   pour chaque tâche T' dans père(T) faire
3     si pas(alloué(T')) alors ordonnance(T');
4   finpour
5   allouer T au processeur qui minimise sa date de début d'exécution;
6   pour chaque tâche T' dans père(T) faire
7     si T est le dernier fils de T' qu'il restait à ordonnancer alors
8       effacer de la mémoire toutes les informations sur T';
9     finsi
10  finpour
11  alloué(T)=vrai;
12 }
```

Figure 8.6 : L'algorithme PTGDS

PTGDS (cf Figure 8.6), ordonnance les tâches. Il détermine sur quel processeur et à quelle date chaque tâche doit être exécutée. Pour chaque tâche T, quand la valeur des paramètres est connue :

- nous utilisons l'analyse du code effectuée par PlusPyr pour déterminer la durée de T;
- nous utilisons le graphe de tâches paramétré pour déterminer l'ensemble des fils de T, l'ensemble des pères, et le volume de communication entre les pères ou les fils.

PTGDS part de la tâche de sortie qui est topologiquement après toutes les autres tâches. Il explore récursivement le DAG et pour chaque tâche il ordonnance tous ces pères avant de lui allouer un processeur. Les lignes 7 et 8 se justifient comme suit : chaque fois qu'une tâche T est ordonnancée on pose *alloué(T)=vrai*. Quand tous les fils de cette tâche auront été ordonnancés le test de la ligne 3 ne sera plus effectué pour T, nous pouvons donc retirer de la mémoire les informations concernant T, à savoir la variable *alloué* ainsi que ce qui concerne son processeur et sa date d'exécution.

8.4.2 Exécution dynamique de l'ordonnancement

Nous construisons un programme SPMD qui exécute dynamiquement l'ordonnancement trouvé par PTGDS. Nous utilisons un protocole superviseur/exécuteur. Il y a un superviseur (sur l'hôte de la machine parallèle) et un exécuteur sur chaque nœud de la machine. Le superviseur exécute PTGDS. Chaque fois qu'une tâche se voit attribuée un processeur, le superviseur envoie aux processeurs l'ordre d'exécuter la tâche et aux processeurs qui ont calculé les données nécessaires pour cette tâche l'ordre de les transmettre.

```
tantque vrai faire {
 case type_message of
 execute :
    si toutes les données nécessaires sont en mémoire
       alors exécute la tâche;
             envoie les nouvelles données si nécessaire (vérifie send_list );
             Si possible, exécute une tâche avec les nouvelles données calculées
             (vérifie exec_list );
       sinon stocke l'ordre dans exec_list ;
 envoie :
    si la donnée nécessaire est en mémoire
       alors envoie la donné;
       sinon stocke l'ordre dans send_list ;
 reçois :
    stocke la donnée en mémoire;
    exécute une tâche si possible (vérifie exec_list );
}
```

Figure 8.7 : Code d'un exécuteur

La figure 8.7, présente le code du protocole d'un exécuteur. Chaque exécuteur gère deux listes. Dans *exec_list* sont stockés les ordres d'exécution qui ne peuvent pas être accomplis car les données ne sont pas encore présentes dans la mémoire locale du processeur. Cette liste est mise à jour quand une nouvelle donnée arrive ou est calculée. Dans *send_list* sont stockés les ordres de transmission de données qui ne peuvent pas être exécutés car la donnée n'a pas encore été calculée par le processeur. Cette liste est mise à jour lorsqu'une nouvelle donnée est calculée.

8.4.3 L'efficacité de l'ordonnancement dynamique

Nous avons très largement étudié *PTGDS*, aussi bien en ce qui concerne sa complexité spatiale ou temporelle que son efficacité sur un nombre infini ou borné de processeurs [2, 3]. Nous présentons ici quelques résultats concernant l'efficacité de notre approche.

8.4.3.a Résultats théoriques

Notre méthode est centralisée. Il y a donc un surcoût dû à l'ordonnancement et à l'envoie des ordres par le superviseur. Nous montrons ici que le surcoût de l'ordonnancement dynamique devient négligeable si la durée moyenne des tâches est importante.

Définition 8.4.1 *Nous appelons :*

- $PT_P(G,p)$ *Le temps pris par les p exécuteurs pour exécuter l'ordonnancement.*
- $PTGDE(G,p)$ *le temps total d'exécution du programme parallèle. Exécutant le graphe G ordonnancé par PTGDS sur p processeurs.*
- $T_{sup}(G,p)$*, le temps total d'exécution du superviseur.*
- M_{op} *le nombre moyen d'opérations d'une tâche. Si* τ_i *est la durée de la tâche* i*, et* ω *le temps d'une opération, alors* $M_{op} = \frac{\sum_{i\in V}\tau_i}{\omega v}$

Lemme 8.4.1 $PT_P(G,p) \leq PTGDE(G,p) \leq T_{sup}(G,p) + PT_P(G,p)$

8.4.3.a.1 Preuve du lemme 8.4.1: A cause du surcoût dû à l'ordonnancement, le temps pris par les exécuteurs pour terminer est plus petit que le temps du programme en entier. De plus, puisque simultanénemt PTGDS est exécuté par le superviseur, et que le code parallèle est exécuté sur p processeur, le temps total de l'exécution du programme dynamique est plus petit que le temps de la séquentialisation du code du superviseur et du code des exécuteurs. ∎

Théorème 8.4.1 *Si* $e = O(v)$ *et* $p \max(p, \log v) = o(M_{op})$ *alors*

$$PTGDE(G,p) = PT_P(G,p) + \epsilon$$

où ϵ *est un temps négligeable par rapport à* $PT_P(G,p)$*.*

8.4.3.a.2 Preuve du théorème 8.4.1

$T_{sup}(G,p) \;=\;$ temps d'ordonnancement + temps de gestion des communications

La quantité maximum de communication gérée par le superviseur est $O(e)$ (il doit communiquer aux exécuteurs quand deux processeurs doivent s'envoyer des données). On a montré dans [3] que la complexité temporelle de PTGDS est $O((e+v)(p+\log v)$. Donc :

$T_{sup}(G,p) \;=\; O((e+v)(p+\log v)+O(e)$

Soit $m=\max(p,\log v)$, et puisque $e=O(v)$:

$T_{sup}(G,p) \;=\; O(vm)+O(v)$

Alors, il existe une constante k telle que :

$T_{sup}(G,p) \;=\; kvm\omega \qquad (1)$

Il n'est pas possible d'avoir une accélération superlinéaire :

$$PT_P(G,p) \geq \frac{T_{seq}}{p} \qquad (2)$$

M_{op} est le nombre moyen d'opération de chaque tâche :

$$M_{op} = \frac{T_{seq}}{v\omega}$$

Par hypothèse, $pm=o(M_{op})$

$$\begin{aligned} pm=o(M_{op}) &\implies pm=o(\frac{T_{seq}}{v\omega}) \\ &\implies kpm=o(\frac{T_{seq}}{v\omega}) \\ &\implies kvm\omega=o(\frac{T_{seq}}{p}) \end{aligned}$$

En remplaçant les équations (1) et (2) on obtient $T_{sup}(G,p)=o(PT_p(G,p))$. Puisque d'après le lemme 8.4.1 $PT_P(G,p) \leq PTGDE(G,p) \leq T_{sup}(G,p)+PT_P(G,p)$, alors

$$PTGDE(G,p) = PT_P(G,p)+\epsilon$$

∎

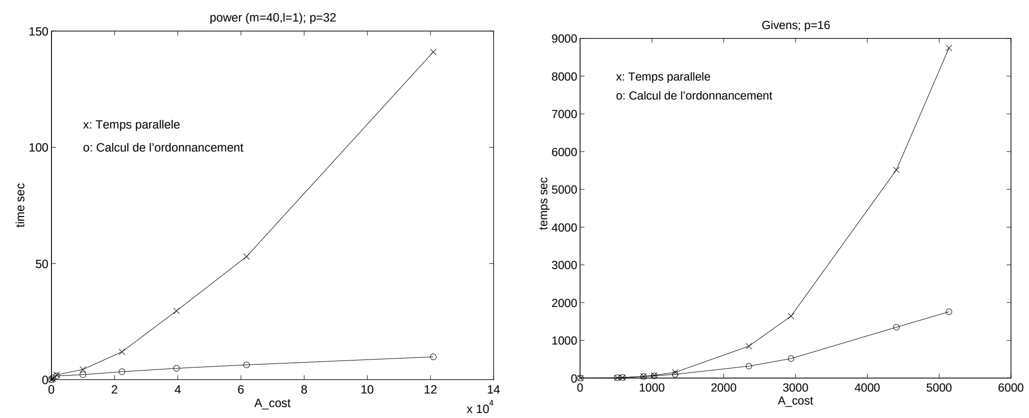


Figure 8.8 : Temps de calcul de l'ordonnancement et simulation du programme parallèle en fonction du nombre moyen d'opérations par tâches

8.4.4 Résultats expérimentaux

Les expériences que nous avons menées concernant le surcoût de l'ordonnancement dynamique montrent que, comme l'énonce le théorème 8.4.1, ce surcoût devient négligeable si le nombre moyen d'opération par tâches est très grand. Dans la Figure 8.8, nous avons simulé l'exécution de deux programmes parallèles *Power* et *Givens*. Pour *Power* nous voyons que lorsque $p^2 \approx 50.A_{op}$, $PT(G) \approx 10.PTGDS(G)$. Pour l'algorithme de Givens, lorsque $p^2 \approx 20.A_{op}$, $PT(G) \approx 5.PTGDS(G)$.

8.5 Conclusion

Nous avons présenté PlusPyr, un outil d'aide à la parallélisation automatique. Nous avons montré qu'il permet de générer une représentation intermédiaire, le graphe de tâches paramétré. Le graphe de tâches paramétré est une représentation indépendante de la taille des données des graphes acycliques de précédence. Nous avons proposé une méthode pour construire un programme générique qui ordonnance dynamiquement le graphe de tâches. Cette approche à l'avantage d'être peu coûteuse en mémoire car à aucun moment le graphe de tâches n'est entièrement en mémoire. Nous avons montré que cette approche est d'autant plus valable que la durée des tâches est grande.

Bibliographie

[1] Blumofe (R. D.), Joerg (C. F.), Kuszmaul (B. C.), Leiserson (C. E.), Randall (K. H.) et Zhou (Y.). – Cilk: An efficient multithreaded runtime system. *In : Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95.* – Santa Barbara, California, Juil. 1995.

[2] Cosnard (M.) et Jeannot (E.). – Automatic Coarse-Grained Parallelization Techniques. *In : NATO workshop : Advances in High Performance Computing*, éd. par Grandinetti and Kowalik. – Kluwer academic Publishers.

[3] Cosnard (M.) et Jeannot (E.). – *Building and Scheduling Coarse Grain Task Graphs.* – Rapport technique n° RR97-03, Ecole Normale Supérieure de Lyon, France, Laboratoire de l'Informatique du Parallélisme, Fév. 1997.

[4] Cosnard (M.) et Loi (M.). – Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, vol. 5, n° 4, 1995, pp. 527–538.

[5] El-Rewini (H.), Lewis (T.) et Ali (H.). – *Task Scheduling in Parallel and Distributed Systems.* – Prentice Hall, 1994.

[6] Garey (M.) et Johnson (D.). – *A Guide to the Theory of NP-Completeness.* – New-York, W.H. Freeman and company, 1979.

[7] Gerasoulis (A.) et Yang (T.). – A Comparaison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. *Journal of Distributed and Parallel Computing*, vol. 16, n° 4, December 1992, pp. 276–291. – special issue on scheduling and load balancing.

[8] Hwang (J.-J.), Chow (Y.-C.), Anger (F. D.) et Lee (C. Y.). – Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, vol. 18, n° 2, Avr. 1989, pp. 244–257.

[9] Loi (M.). – *Construction et exécution de graphe de tâches acycliques à gros grain.* – Thèse de PhD, Ecole Normale Supérieure de Lyon, France, 1996.

[10] Sarkar (V.). – *Partitionning and Scheduling Parallel Program for Execution on Multiprocessors.* – Cambridge MA, MIT Press, 1989.

[11] Ullman (J.). – NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, vol. 10, 1975, pp. 384–393.

[12] Yang (T.). – *Scheduling and Code Generation for Parallel Architectures.* – Thèse de PhD, Rutgers State University of New Jersey, 1993.

[13] Yang (T.) et Gerasoulis (A.). – Pyrros: Static Task Scheduling and Code Generation for Message Passing Multiprocessor. *In : Supercomputing'92*. ACM, pp. 428–437. – Washington D.C., July 1992.

[14] Yang (T.) et Gerasoulis (A.). – DSC Scheduling Parallel Tasks on an Unbouned Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n° 9, September 1994, pp. 951–967.

[15] Yang (T.) et Gerasoulis (A.). – List scheduling with and without communication delay. *Parallel Computing*, vol. 19, 1995, pp. 1321–1344.

Chapitre 9

Langages pour l'expression dynamique de parallélisme et graphes de tâches

François Galilée, Jean-Louis Roch (LMC-IMAG)

La mise en œuvre d'un langage parallèle de haut niveau permettant de faire abstraction de l'architecture de la machine (dans un souci de portabilité) nécessite celle d'un mécanisme automatique d'ordonnancement pour placer les calculs, router les données et synchroniser les tâches dans le but de garantir l'efficacité et la sémantique de l'exécution de tout programme. Cette séparation entre l'application parallèle et son ordonnancement peut être rendue possible par la construction du graphe de flot de données qui définit à la fois les dépendances de données et les précédences des calculs. Nous nous proposons dans ce chapitre de montrer à travers trois environnements de programmation parallèle de haut niveau (Cilk [4], Jade [5, 6] et Athapascan-1 [2]) comment de simples extensions d'un langage séquentiel pour exprimer du parallélisme permettent la construction implicite de graphes de tâches (graphe de précédence ou de flot de données). Dans ces trois environnements cette construction est dynamique car la classe d'applications visée est celle des applications (irrégulières) dans lesquelles le parallélisme est généré en cours d'exécution en fonction des valeurs manipulées.

9.1 Introduction

Face à la grande diversité et à la banalisation des machines parallèles (allant des multi-processeurs aux réseaux de stations) différents environnements de programmation parallèles (dont Cilk, ATHAPASCAN-1 et Jade qui sont au centre de ce chapitre) tentent d'offrir un compromis entre portabilité, efficacité et simplicité.

La **portabilité** est obtenue en faisant une totale abstraction des éléments spécifiques à chaque machine et en offrant une sémantique d'exécution des programmes. Ainsi n'y a-t-il plus aucune référence de faite au matériel (processeurs, mémoire, réseau, ...) : le programmeur explicite le parallélisme de son algorithme sans hypothèse de ressource matérielle. Le système (compilation et contrôle dynamique) garantit le respect de la sémantique du langage pour toute exécution sur une architecture matérielle à ressources nécessairement bornées.

L'**efficacité** est obtenue en gérant de manière fine le parallélisme de l'algorithme exprimé par le programmeur. Cette gestion est assurée par un ordonnanceur, découplé du programme, qui va être à même de tirer pleinement profit des capacités de la machine, bien plus que toute stratégie d'ordonnancement directement codée dans l'algorithme qui présupposent forcément un certain support d'exécution (ce qui bride de surcroît la portabilité). L'ordonnanceur va permettre de faire le lien entre le programme et la machine : il est donc intéressant de pouvoir choisir celui qui sera le mieux approprié au couple (programme, support d'exécution).

Le problème de l'efficacité ne peut pas être totalement séparé du problème de portabilité, l'efficacité d'un algorithme dépendant fortement de l'architecture sur laquelle il est appliqué. La portabilité de l'efficacité peut nécessiter l'adaptation de l'algorithme d'ordonnancement à la machine ciblée pour l'exécution : certaines applications ne peuvent pas être efficacement exécutés sur certaines architectures. Un langage de haut niveau offrant une portabilité de l'efficacité doit donc offrir une représentation abstraite (mais la plus complète possible) du programme parallèle, représentation sur laquelle travaillera l'algorithme d'ordonnancement. Cette représentation peut, par exemple, être un graphe de flot de donnée, objet central en théorie de l'ordonnancement.

La **simplicité** de programmation est obtenue en limitant l'apprentissage nécessaire à l'utilisation de ces langages parallèles en proposant des “extensions” de langages séquentiels déjà existants (C, C++, ...) plus que de nouveaux langages totalement reconstruit et en adoptant une sémantique d'exécution intuitive (proche d'une sémantique séquentielle).

Nous utiliserons comme exemple de base le problème du calcul d'une intégrale par la méthode de Newton-Cotes qui se résume à calculer par morceaux l'intégrale d'une fonction f sur un intervalle $[a, b]$, connaissant la fonction g telle que pour tout $|x_b - x_a| < h$, $g(x_a, x_b) = \int_{x_a}^{x_b} f dx$. Nous utiliserons un algorithme effectuant une découpe récursive jusqu'à atteindre le pas h désiré puis accumulant les intégrales calculées sur chaque morceau : figure 9.1.

<pre>double compute(double a, double b)
{
    if(b-a < h) {
        return g(a,b);
    } else {
        double res1, res2;
 	
        res1 = compute(a, (a+b)/2);
        res2 = compute((a+b)/2, b);
 	
        return (res1 + res2);
    }
}</pre>
	$\int_a^b f dx = \int_a^{\frac{(a+b)}{2}} f dx + \int_{\frac{(a+b)}{2}}^b f dx$
	
	 $\forall \lvert b-a \rvert < h, g(a,b) = \int_a^b f dx$

Figure 9.1 : Calcul par découpe récursive de l'intégrale d'une fonction f sur l'intervale $[a, b]$ par la méthode de Newton-Cotes. L'implémentation de la fonction g est supposée sans effet de bord.

9.2 Représenter l'exécution par un graphe de tâches

Dans le paradigme de programmation parallèle explicite par tâches concurrentes, l'algorithme est exprimé comme la description d'un ensemble de tâches travaillant sur un ensemble de données partagées. Cette description, fournie implicitement par le programmeur qui explicite le parallélisme par l'intermédiaire de mot clés spécifiques, peut être interprétée comme un graphe, soit de précédence, soit de flot de données. Cette interprétation peut avoir lieu soit statiquement lors de la compilation, soit dynamiquement lors de l'exécution.

9.2.1 Deux modèles de graphe : précédence et flot de données

Le graphe de précédence définit un ordre partiel pour les tâches. Cet ordre partiel doit impérativement être contenu dans l'ordre d'exécution de l'ensemble des tâches afin de respecter la sémantique du langage. Le graphe de flot de données définit quand à lui la succession des états d'une donnée en mémoire partagée, chaque état étant le résultat de l'intervention (synthèse ou modification) d'une tâche.

Dans la figure 9.2 sont représentés les graphes de dépendances et de flot de données de l'exécution du calcul d'intégrale (présenté figure 9.1) avec pour paramètres $a = 0$, $b = 1$ et $h = \frac{1}{4}$. Dans toutes les représentations de graphe nous représenterons les tâches par des ellipses, les états (valeur valide à une étape de l'exécution) d'une donnée partagée par des rectangles ; les flèches représentent une dépendance : soit entre deux tâches (précédence), soit entre une tâche et une donnée partagée (correspondant soit à une lecture, soit à une écriture, selon le sens de la dépendance).

La distinction entre ces deux types de graphe se situe au niveau des synchroni-

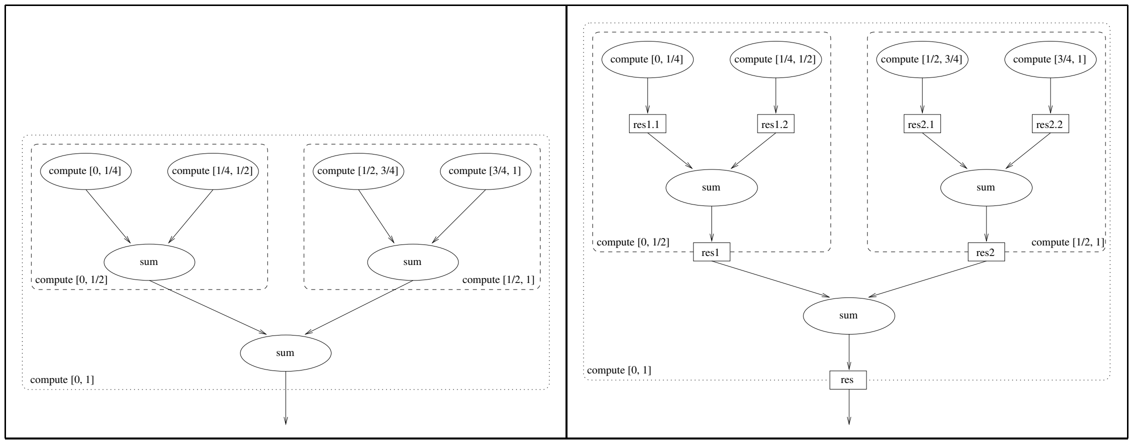


Figure 9.2 : Graphe de précédence et graphe de flot de données. La différence fondamentale se situe au niveau des synchronisations. Dans le cas d'un graphe de précédence une tâche est prête si toutes les tâches qui la précèdent sont terminées, tandis que dans le cas d'un graphe de flot de données une tâche est prête si toutes les données qu'elle accède en lecture sont disponibles.

sations : dans le cas d'un graphe de précédence une tâche est dite prête, c'est-à-dire que son exécution peut avoir lieu, lorsque toutes les tâches qui la précèdent sont terminées, tandis que dans le cas d'un graphe de flot de données une tâche est dite prête si toutes les données qu'elle accède en lecture sont disponibles.

9.2.2 Graphe de flot de données

Ces synchronisations sur l'état prêt des tâches sont nécessaires dans toute exécution concurrente (de la programmation multi-thread à la programmation parallèle) pour garantir une sémantique d'exécution. Cette sémantique est directement liée à l'accès aux données : la dépendance entre les tâches n'a en effet pour origine que le partage des données (dire qu'une tâche t_1 précède une autre tâche t_2 signifie simplement que t_2 utilise une donnée produite (pour partie au moins) par t_1). Chaque donnée partagée peut donc être vue comme un point de synchronisation ayant pour effet de bloquer les lectures tant que toutes les écritures n'ont pas eu lieu.

Ces graphes sont utiles au système car ils contiennent des informations nécessaires à une bonne exécution du programme parallèle : la précédence des calculs, quand et où doivent intervenir les synchronisations entre tâches et de ce fait met en évidence les tâches qui peuvent être exécutées en concurrence. Dans le cas d'un graphe de flot de donnée, l'information supplémentaire porte sur les accès qui sont effectués par les tâches sur les données. Cette information peut être par exemple utilisée pour tirer profit de la localité des données lors du placement des tâches ou pour le routage des données accédées vers le site d'exécution de la tâche.

9.3 Trois langages basés sur une représentation par graphe de tâches

Nous nous proposons d'examiner dans cette section différents langages de programmation parallèle de haut niveau : Cilk [4], Jade [5, 6] et ATHAPASCAN-1 [2]. Nous tâcherons de mettre à jour leurs différences fondamentales et leurs points communs.

9.3.1 Cilk

L'environnement de programmation Cilk [4] est une extension du langage C qui permet le lancement asynchrone de l'exécution d'une fonction (un nouveau thread lui sera alloué) par l'instruction `spawn`. Le résultat retourné ne pourra être consulté qu'après la fin d'exécution de la dite fonction ; l'instruction `sync` attend la terminaison de toutes les fonctions précédemment lancées avant de continuer, ce qui permet à son issue de consulter l'ensemble des résultats des fonctions créées antérieurement par `spawn`.

Un mécanisme de mémoire partagée permet de partager des données entre les threads. La sémantique associée à cette mémoire est telle qu'un thread peut lire dans une variable toute valeur qui est consistante avec une exécution séquentielle quelconque du graphe des threads (i.e. qui respecte l'ordre partiel induit par les précédences); il peut donc y avoir indéterminisme, si deux exécutions séquentielles (par exemple largeur d'abord et profondeur d'abord) ne mènent pas à la même valeur (si deux threads concurrents font des effets de bords sur une même variable). Nous présentons, figure 9.3, un codage du calcul de l'intégrale dans ce langage.

Dans Cilk le grain de calcul est explicite, de même que les synchronisations garantissant un accès cohérent aux données, accès qui est de type CREW. Une procédure se synchronise en créant un nouveau thread (bloqué) qui sera la continuation de la procédure. Ce thread sera débloqué par les retours des procédures lancées en parallèle. Le système maintient une liste de threads exécutables qui est ordonnancée par une politique de "work stealing" de type receiver-initiated. Un graphe de précédence des threads est dynamiquement créé (figure 9.4) afin de garantir la sémantique de l'instruction `sync` (attente de tous les threads créés).

9.3.2 Athapascan-1

ATHAPASCAN-1 [2] est l'interface de programmation de l'environnement de programmation ATHAPASCAN. Cette interface implémente un modèle de programmation parallèle orienté pour l'expression dynamique de la décomposition d'un calcul en sous-calculs grâce à l'instruction `fork`. Ce modèle est basé sur le partage d'objets entre activités paralléles et propose un modèle original de résolution des conflits d'accès aux objets partagés.

```
cilk double compute(double a, double b)
{
    if(b-a < h) {
        return g(a,b);
    } else {
        double res1, res2;

        res1 = spawn compute(a, (a+b)/2);
        res2 = spawn compute((a+b)/2, b);

        sync;

        return (res1 + res2);
    }
}
```

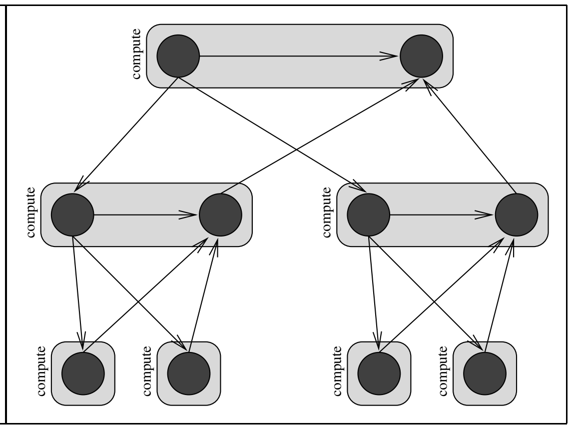


Figure 9.3 : Code Cilk et graphe d'exécution ($a = 0$, $b = 1$ et $h = \frac{1}{4}$) du calcul de l'intégrale par Newton-Cotes. Les threads, représentés par des cercles, sont regroupés en procédure, représentés par des rectangles grisés. Les flèches vers le bas indiquent une création de procédure (conséquence d'un `spawn`), les flèches horizontales une continuation entre deux threads (conséquence d'un `sync`) et les flèches vers le haut un retour de résultat vers la procédure créatrice. Ces trois types de flèches contraignent l'ordre d'exécution des threads.

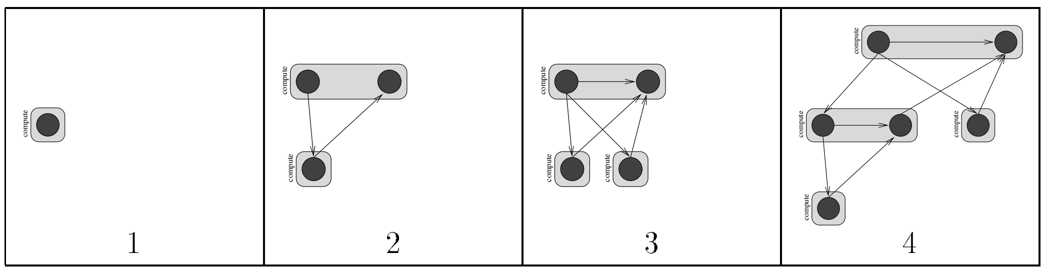


Figure 9.4 : La génération, dans Cilk, du graphe de précédence des threads est dynamique et distribuée. À l'étape 1 la première tâche est créée. Cette tâche crée aux étapes 2 et 3 deux nouveaux threads de calcul (correspondant respectivement aux données `res1` et `res2`) et se synchronise sur l'attente de ces deux paramètres. À l'étape 4, le premier thread (par exemple...) continue la découpe récursive.

Athapascan-1 est implémenté par une bibliothèque pour le langage C++ et s'exécute sur le noyau exécutif Athapascan-0[1] (voir le chapitre sur les supports d'exécutions) ; il permet une description explicite et dynamique du parallélisme par création asynchrone de tâches (appel de procédure sans effet de bord). Un prépro-

[1]Le noyau exécutif Athapascan-0 ; il permet la portabilité logicielle d'un programme parallèle écrit pour un nombre fixé de processeurs virtuels sur une architecture matérielle permettant d'émuler ce nombre de processeurs.

cesseur Ath permet l'interfaçage avec C. C'est la syntaxe de ce préprocesseur qui est ici utilisée.

Quatre modes d'accès sont définis pour les données situées en mémoire partagée : *lecture* (`a1_shared(r)`), *écriture* (`a1_shared(w)`), *lecture/écriture* (`a1_shared(r_w)`) et *accumulation* (`a1_shared(cwf)`). Les trois premiers modes sont classiques, le dernier permet une accumulation[2]. La sémantique des accès[3] en mémoire partagée est telle que chaque lecture voit la dernière écriture effectuée selon l'ordre séquentiel de création des tâches (profondeur d'abord). L'approche est une approche de type flot de donnée. Nous présentons, figure 9.5, un codage du calcul de l'intégrale en ATHAPASCAN-1.

```
task sum(shared(r)<double> res1,
         shared(r)<double> res2,
         shared(w)<double> res)
{
  res.write( res1.read() + res2.read());
}

task compute(double a, double b,
             shared(w)<double> res))
{
    if(b-a < h) {
        res.write( g(a,b) );
    } else {
        shared(w)<double> res1, res2;

        fork compute( a, (a+b)/2, res1 );
        fork compute( (a+b)/2, b, res2 );

        fork sum(res1, res2, res);
    }
}
```

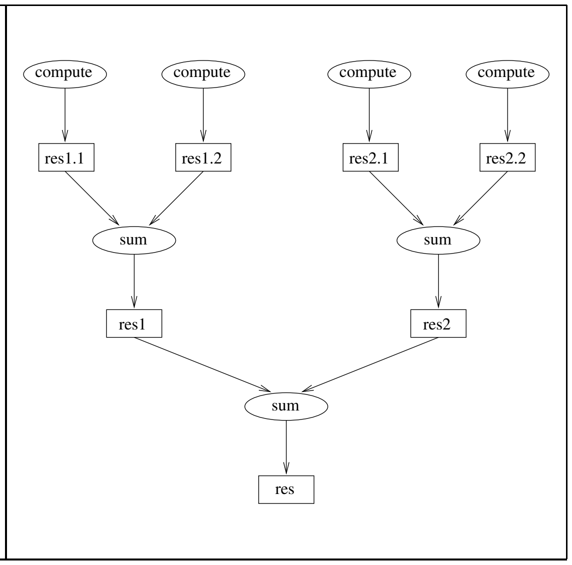


Figure 9.5 : Code ATHAPASCAN-1 et graphe d'exécution ($a = 0$, $b = 1$ et $h = \frac{1}{4}$) du calcul de l'intégrale par Newton-Cotes. Ce sont les accès aux données situées en mémoire partagée qui définissent implicitement les synchronisations entre les tâches.

Le grain de calcul est explicite mais le système peut décider la dégénérescence séquentielle d'une tâche (c'est-à-dire que toutes les créations de tâches sont remplacées par des appels synchrones de procédure) au début de son exécution : il y

[2]La fonction d'accumulation, supposée associative et commutative, doit être fournie par l'utilisateur. Par défaut c'est l'opérateur `+=` de C++ qui est utilisé.

[3]ATHAPASCAN-1 offre d'autres types d'accès en mémoire partagée : des accès différés permettant une expression plus fine du parallélisme (les objets ne pouvant pas être directement accédés, la tâche sert à générer d'autres tâches et plus à calculer) et définition de collections d'objets partagés. Leur implémentation n'est pas décrite ici mais repose sur le principe exposé sur les modes d'accès élémentaires.

a ainsi adaptation du grain à l'état de la machine[4]. L'ordonnanceur par défaut est un ordonnanceur proche de celui utilisé dans Cilk mais tenant compte en plus de la localité des données. Dans la mesure où la politique d'ordonnancement la mieux adaptée un type d'algorithme peut ne pas l'être pour un autre, il est possible de définir sa propre politique d'ordonnancement (ciblée sur une rapidité d'exécution ou une économie des resources mémoires par exemple) en respectant une interface prédéfinie. Un graphe de flot de données est dynamiquement créé (figure 9.6) afin de garantir la sémantique des accès en mémoire ; ce graphe est accessible à toute politique d'ordonnancement qui peut ainsi analyser les relations entre calculs et données.

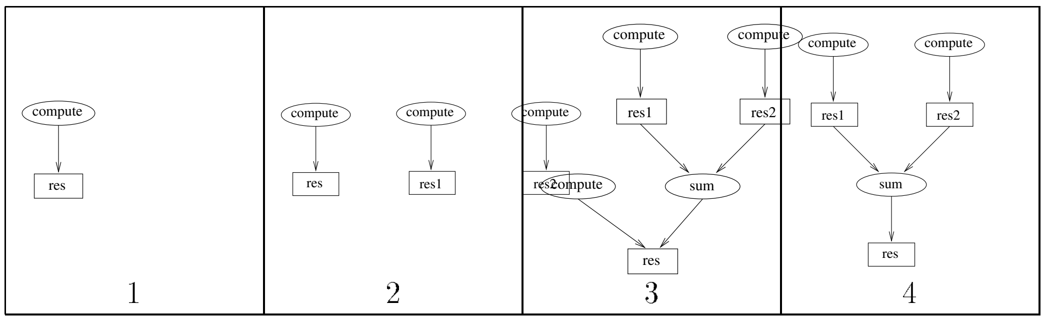


Figure 9.6 : La génération, dans ATHAPASCAN-1, du graphe de flot de données est dynamique et distribuée. À l'étape 1 la première tâche est créée, pointant en écriture sur une donnée en mémoire partagée où sera stoquée la donnée. Cette tâche crée à l'étape 2 deux nouvelles tâches de calcul qui stoqueront leur résulats dans deux données partagées `res1` et `res2`. À l'étape 3 il y a création de la tâche `sum` qui accède en lecture `res1` et `res2` et en écriture à `res`. L'étape 4 représente la fin de la première tâche de calcul : elle se retire du graphe en ôtant tous ses accès.

9.3.3 Jade

L'environnement de programmation Jade [5, 6] est une extension du langage C dans lequel l'utilisateur doit décomposer son programme séquentiel en tâches en spécifiant quels accès seront effectués par chaque tâche sur les données. Une tâche est un bloc d'instructions défini par l'instruction `with_only ... do {...}`. Les instructions sont définies dans la seconde partie de la construction, les accès effectués par ces instructions étant définis dans la première partie. Ces accès sont codés `rd`, `wr`, `rd_wr` pour respectivement un accès en *lecture*, *écriture* et *lecture/écriture*. L'implémentation de Jade parallélise le calcul en identifiant les parties du programme séquentiel qui peuvent être exécutées en concurrence sans changer le résultat de

[4]Par exemple, si l'architecture cible est un monoprocesseur, aucune tâche ne sera créée.

ce programme. Nous présentons, figure 9.7, un codage du calcul de l'intégrale de l'intégrale dans ce langage.

```
double compute(double a, double b)
{
    if(b-a < h) {
        return g(a,b);
    } else {
        double shared res1;
        double shared res2;

        withonly { wr(res1) }
        do (a, b, res1) {
            res1 = compute(a, (a+b)/2 );
        }
        withonly { wr(res2) }
        do (a, b, res2) {
            res2 = compute(a, (a+b)/2 );
        }

        with { rd(res1), rd(res2) } cont;

        return (res1 + res2);
    }
}
```

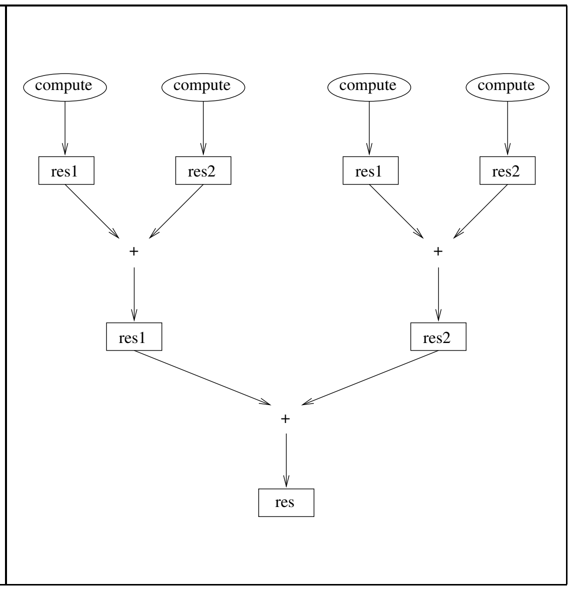


Figure 9.7 : Code Jade et graphe d'exécution ($a = 0$, $b = 1$ et $h = \frac{1}{4}$) du calcul de l'intégrale par Newton-Cotes. Ce sont les accès aux données situées en mémoire partagée qui définissent les synchronisations entre les tâches.

Jade maintient un graphe de type flot de donnée pour garantir la sémantique du programme (le résultat doit être le même que lors d'une exécution séquentielle), les tâches exécutables étant confiées à un ordonnanceur de type liste. Le grain est fixé par l'utilisateur, mais le système ne transforme pas forcément toute spécification de tâche en une tâche s'exécutant en parallèle. Ce mécanisme permet de limiter une génération de parallélisme qui peut être abusive.

9.4 Utilisation du graphe

Nous traitons dans cette section des deux principales utilisations d'un graphe de précédence ou de flot de données ; d'une part garantir la sémantique d'exécution et d'autre part permettre l'implémentation d'algorithmes de régulation fondés théoriquement.

9.4.1 Garantir une sémantique d'exécution

Tout langage possède une sémantique d'exécution, sémantique qui doit être garantie par l'implémentation. La connaissance du graphe d'exécution des tâches apporte une aide appréciable dans la résolution de ce problème.

9.4.1..3 Formalisation La représentation du programme par un graphe de précédence ou de flot de données permet de définir de manière simple et efficace la sémantique du langage en montrant de manière intuitive les points de synchronisation entre les tâches par l'intermédiaire des arêtes du graphe. Par exemple dans le cas de Cilk le graphe de précédence est directement utilisé pour définir [1] la sémantique des accès aux variables partagées : un thread peut lire dans une variable toute valeur qui est consistante avec une exécution séquentielle du graphe des threads. Cette définition de la sémantique à l'aide du graphe de précédence des tâches permet à l'utilisateur de vérifier facilement la correction de son programme.

Cette même approche se retrouve dans Jade ou ATHAPASCAN-1, en considérant cette fois le synchronisation sur les données et non sur les tâches. La sémantique d'accès en mémoire partagée est cependant plus forte : il ne peut y avoir d'indéterminisme (contrairement à Cilk) puisque la valeur lue dans une variable est la valeur qui aurait été lue si l'exécution avait été séquentielle (suppression du parallélisme en remplaçant toutes les créations de tâches asynchrones par des appels de procédures). Le graphe de flot de données permet dans ces langages de formaliser cette sémantique en introduisant la notion d'état[5] sur les variables partagées.

9.4.1..4 Implémentation Considérons par exemple le cas de l'instruction `sync` dans Cilk. La sémantique associée à cette instruction est d'attendre la fin de toutes les procédures appelées en concurrence au cours de l'actuelle procédure. Le graphe de précédence permet d'identifier immédiatemment quelles sont les tâches dont il faut attendre la terminaison.

Dans le cas d'ATHAPASCAN-1, le graphe de flot de données permet d'assurer la sémantique du langage en gérant les accès à la mémoire partagée, et de ce fait en gérant les synchronisations entre tâches concurrentes.

9.4.2 Intérêt du graphe de tâches pour la régulation

Les graphes de tâches sont centraux en théorie de l'ordonnancement, et nombre d'algorithmes utilisent ce graphe comme seul et unique paramètre. Il y a possibilité d'utiliser directement les résultats de la théorie d'ordonnancement, et plus particulièrement les garanties d'efficacité (compétitivité) de certains algorithmes d'ordonnancement en ligne.

Par exemple, pour Cilk et ATHAPASCAN-1, des algorithmes d'ordonnancement dynamique de type liste permettent de borner théoriquement le temps d'exécution d'un programme écrit pour un nombre infini de processeurs en fonction du nombre d'opérations effectuées, du temps parallèle minimal sur un nombre non borné

[5] L'état d'une variable partagée (ou transition) représente une valeur valide (qui sera éventuellement lue) à une étape de l'exécution.

de ressources et du volume maximal de communications (voir chapitre Machines virtuelles et techniques d'ordonnancement).

9.4.3 Séparation entre programme et ordonnancement

L'exécution se déroule comme suit :

1. *Construction distribuée du graphe :* Les tâches actives s'exécutent en concurrence sur les différents processeurs (virtuels ou physiques) et génèrent de nouvelles tâches liées par de nouvelles dépendances. Ces tâches sont insérées dans la partie du graphe possédée localement. Lorsqu'une tâche retire ses accès en mémoire partagée (lors de sa terminaison) le graphe est également modifié localement ; le système recalcule les dépendances des tâches du graphe et détermine l'ensemble des tâches qui peuvent être exécutées (tâches devenues prêtes).

2. *Calcul de l'ordonnancement :* L'ordonnanceur peut parcourir le graphe distribué qui est construit dynamiquement : il a ainsi accès à toutes les informations disponibles concernant l'exécution à cet instant.

3. *Exécution avec l'ordonnancement calculé :* Cette dernière phase ne requiert aucune autre communication que celles des données entre processeurs distincts et la migration des tâches vers leur site d'exécution.

Par défaut, dans Cilk et ATHAPASCAN-1, un algorithme d'ordonnacement dynamique de type liste est utilisé. Cet algorithme prend comme paramètre les informations sur l'état des tâches (prête ou non-prête), informations déterminées par le système à partir du graphe. L'algorithme d'ATHAPASCAN-1 tient compte des localités des données pour faire son placement et examine donc les données associées à chaque tâche dans le graphe de flot de données.

Dans le langage ATHAPASCAN-1, le graphe de flot de donnée peut être utilisé de manière dynamique par un algorithme d'ordonnancement tel DSC utilisé généralement dans un cadre statique [7, 3]. Lors de la création d'une tâche, des attributs de coût peuvent être fournis à `fork` qui seront évalués lors de l'appel [2]. Ces informations sont ajoutées comme décors sur les nœuds (coût de calcul) et arêtes (taille de donnée) du graphe. Il est possible de transformer le graphe de flot de données en un graphe de précédence, utilisé par DSC, par un simple parcours de graphe (figure 9.8).

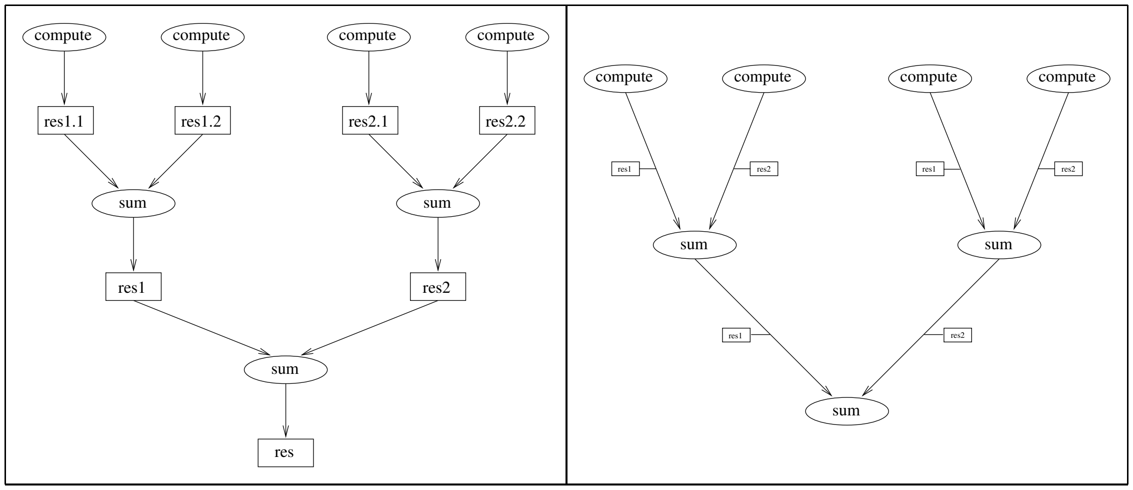


Figure 9.8 : Génération du graphe de précédence adapté à l'ordonnancement à partir du graphe de flot de données en ATHAPASCAN-1.

9.5 Une implémentation concrète : le cas d'Athapascan-1

Nous nous proposons dans cette section d'étudier la création, gestion et utilisation du graphe de flot de données dans le cadre d'ATHAPASCAN-1.

9.5.1 Création dynamique du graphe de flot de données

La création, ou plus exactement l'évolution, du graphe de flot de données se fait de manière dynamique à chaque nouvel accès à une variable située en mémoire partagée. Ce nouvel accès peut avoir pour origine :

- La création d'une référence locale à une **nouvelle** variable en mémoire partagée par l'utilisateur (suite à une déclaration d'une référence statique ou la création dynamique par `new`) ;
- La création (utilisation de `fork`) d'une nouvelle tâche prenant en paramètre une variable située en mémoire partagée (adjonction d'un lecteur et/ou écrivain) ;
- La destruction d'une référence locale à une variable en mémoire partagée (suite à la terminaison d'une tâche pour les références statiques ou à la destruction explicite, par `delete`, dans le cas d'une création dynamique de la référence).

9.5.2 Gestion distribuée

La gestion du graphe est menée simultanément à la construction du graphe. C'est-à-dire que chaque modification apportée à ce graphe peut faire évoluer l'état de certaines de ses composantes (tâche ou transition[6]). Une transition peut prendre trois états : **non-prête** (il reste encore des écrivains), **prête** (toutes les écritures sont terminées) et **terminée** (il n'y a plus ni lecteur, ni écrivain). Une tâche peut prendre deux états : **non-prête** (toutes les données accédées en lecture ne sont pas prêtes, ou, ce qui est synonyme, si cette tâche accède en lecture une transition qui est non prête) et **prête** lorsque toutes les données accédées en écriture sont prêtes.

La détermination de l'état prêt (resp. terminé) d'une transition se ramène au comptage global du nombre restant d'écrivain (resp. de lecteurs). Ce comptage est effectué de la manière centralisée suivante[7] : chaque transition possède un site de référence[8] qui gère le compteur. Lorsque toutes les écritures sont terminées sur un site, il y a émission vers le site de référence du nombre d'écrivains qui se sont retirés afin que le compteur puisse être décrémenté. Lorsque le compteur est à 0, la transition est déclarée prête.

La détermination de l'état d'une tâche est effectuée à l'aide d'un compteur local décrémenté à chaque fois qu'une transition accédée en lecture passe à l'état prêt. Lorsque ce compteur passe à 0, la tâche est déclarée prête et l'ordonnanceur est averti (il peut désormais exécuter cette tâche).

9.5.3 Utilisation

Le graphe est utilisé de manière interne par le système pour garantir la sémantique d'exécution du programme, c'est-à-dire pour garantir la sémantique des accès à la mémoire partagée. Cette sémantique est que toute lecture d'un objet partagé voit la dernière écriture dans l'ordre séquentiel d'exécution (profondeur d'abord). Possédant le graphe de flot de données du programme, le système doit garantir qu'aucune tâche accédant en lecture à une donnée en mémoire partagée n'est commencée avant que toutes les tâches écrivant tout ou partie de cette donnée soient terminées : c'est la détermination de l'état de cette tâche.

Ce graphe est intégralement mis à disposition (sous sa forme distribuée) des ordonnanceurs qui veulent en tirer parti. Une étude est actuellement menée pour intégrer un algorithme d'ordonnancement tel DSC utilisé généralement dans un cadre statique et qui prend comme paramètre de travail un graphe de précédence

[6]Une transition représente l'implémentation de l'état d'une variable partagée (valeur valide,qui sera éventuellement lue) à une étape de l'exécution.

[7]Seule l'idée de l'algorithme est donnée ici, un certain nombre de précautions étant prise pour assurer un comptage global tenant compte des migrations tâches

[8]Ce site est personnel à la transition : deux transitions différentes peuvent donc avoir des sites de référence différents ce qui mène à une gestion distribuée de la cohérence du graphe.

étiqueté par les coûts des tâches et des communications, deux informations qui sont contenues dans le graphe de flot de données ATHAPASCAN-1 (par annotation du programme par l'utilisateur).

9.6 Conclusion

Nous avons montré qu'il était possible, à partir de simples extensions permettant l'expression du parallélisme, de construire implicitement des graphes de tâches (graphes de précédences ou graphe de flot de données), graphes qui peuvent ensuite être utilisés pour mener à bien l'exécution et l'ordonnancement. De tels graphes sont notamment utilisés par les trois environnements de programmation parallèle Cilk, Athapascan-1 et Jade.

En outre, l'utilisation d'un graphe de flot de données (éventuellement annoté par des informations de coût fournies par l'utilisateur) permet de résoudre les problèmes de sémantique (précédence entre tâches sur les accès aux données) et d'ordonnancement (en utilisant des algorithmes travaillant sur le graphe), de localité des données (pour le routage lors de l'exécution ou l'ordonnancement).

Bibliographie

[1] Blumofe (R. D.), Frigo (M.), Joerg (C. F.), Leiserson (C. E.) et Randall (K. H.). – An analysis of dag-consistent distributed shared-memory algorithms. *In : Eighth Annual ACM Symposium on Parallel Algorithm and Architectures (SPAA).* – Padua, Italy, Juin 1996.

[2] Doreille (M.), Galilée (F.) et Roch (J.-L.). – *Athapascan-1b: Présentation.* – Rapport technique n° http://navajo.imag.fr/ath1/, Grenoble, France, Projet APACHE, 1996.

[3] Gerasoulis (A.) et Yang (T.). – PYRROS : Static Scheduling and Code Generation for Message-Passing Architectures. *In : Proc. of the 6th ACM International Conference on Supercomputing*, pp. 428–437.

[4] Joerg (C.). – *The Cilk system for parallel multithreaded computing.* – Thèse de PhD, Massachussets Institute of Technology, january 1996.

[5] Rinard (M.). – *The design, implementation and evaluation of Jade: a portable, implicitly parallel programming language.* – Thèse de PhD, Stanford University, september 1994.

[6] Rinard (M. C.), Scales (D. J.) et Lam (M. S.). – Jade: A high level, machine-independent langage for parallel programming. *IEEE*, Juin 1993, pp. 28–38.

[7] Yang (T.) et Gerasoulis (A.). – DSC scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n° 9, Sept. 1994, pp. 951–967.

Chapitre 10

Techniques de régulation de charge - Applications en Optimisation Combinatoire

Yves Denneulin (LIFL, Université de Lille 1)
Thierry Mautor (PRiSM, Université de Versailles - St Quentin)

Résumé :

L'un des problèmes majeurs posés lors de la résolution exacte de problèmes d'optimisation combinatoire dans un contexte distribué réside dans la forte irrégularité de ce type d'applications. Celle-ci implique l'utilisation d'un mécanisme de régulation de charge afin d'améliorer les performances lors de l'exécution. Dans ce cadre, nous rappelons les spécificités de ce type d'applications en les illustrant sur le problème connu d'affectation quadratique. Nous présentons ensuite différents algorithmes de régulation de charge pour les applications visées, en insistant sur leurs aspects génériques. Enfin nous étendons les concepts présentés dans les parties précédentes aux environnements d'exécution distribués afin de mettre en évidence les points communs entre les besoins d'un support d'exécution distribué et les besoins des applications précédemment étudiées.

10.1 Introduction

L'objectif majeur du parallélisme est de permettre la résolution de problèmes qui de par leur ampleur, en quantité de calcul et en espace mémoire, ne peuvent pas être raisonnablement traités dans un environnement séquentiel. On cite classiquement les besoins des météorologues qui, sans les ordinateurs parallèles, fourniraient les prévisions pour le lendemain avec 2 jours de retard, limitant ainsi leur intérêt. Cette parallélisation des applications se fait en divisant l'application en de multiples éléments soit au niveau des données, on parle alors de parallélisme de données (data parallelism), ou au niveau des activités, on parle alors de parallélisme de tâches. Dans les deux cas, le problème qui se pose alors est celui de la répartition des travaux ainsi générés sur les éléments de calcul, les processeurs. On utilise pour cela des techniques de régulation de charge dont le but est d'assurer une bonne efficacité de l'exécution parallèle de telles applications. La difficulté de réaliser un bon équilibrage dépend en grande partie de l'irrégularité de l'application considérée, une application irrégulière étant bien évidemment bien plus difficile à équilibrer qu'une application régulière telle qu'on en trouve dans le domaine du parallélisme de données.

Cet article a pour objectif de présenter des techniques de régulation mises en place dans le cadre d'une application de résolution exacte de problèmes d'optimisation combinatoire. Cette application figure parmi les plus irrégulières existantes car son comportement est largement imprévisible.

Nous commençons par présenter le schéma général de parallélisation de l'algorithme de Branch and Bound. Dans la section suivante nous présentons en détail l'application qui nous a servi de support pour le test de nos algorithmes de répartition, à savoir le problème d'affectation quadratique. Nous décrivons ensuite deux schémas généraux de régulateurs de charge dont nous illustrons le comportement avec des résultats expérimentaux. Les directions que nous souhaitons maintenant donner à ces travaux sont classiquement présentées en conclusion.

10.2 Stratégies de parallélisation d'un Branch and Bound

Dans cette section, nous décrivons brièvement les approches généralement retenues lors de la parallélisation d'algorithmes Branch and Bound. Pour l'essentiel, les taxonomie et terminologie adoptées dans cette présentation sont celles dues à B. LeCun [7].

La première distinction généralement retenue lors de la description d'un algorithme Branch and Bound est son niveau de parallélisation. Et on peut constater qu'aux parallélisations de bas niveau où ce sont les sous-tâches de l'algorithme, comme le calcul de l'évaluation ou la procédure de branchement, qui font l'objet de la parallélisation sont généralement préférées des parallélisations de plus haut niveau dans lesquelles c'est directemment l'exploration de l'arborescence de recherche qui est parallélisée. Dans ce cas, chaque processeur effectue la procédure générale de recherche sur des parties distinctes de l'arborescence de recherche. Ainsi, les tâches allouées aux différents processeurs sont indépendantes. Ceci définit de plus une parallélisation à gros grain mieux adaptée aux machines parallèles actuelles.

Algorithmiquement parlant, la procédure de recherche locale effectuée par chaque processeur est similaire à la procédure séquentielle. Les opérations parallèles sont effectuées sur la structure de données, appelée file de priorité, où sont regroupées les tâches à allouer aux différents processeurs et qui correspondent à l'exploration d'une sous-arborescence. En conséquence, la forme même de cette file de priorité est essentielle pour définir le schéma de parallélisation. Or, on distingue alors les *files de priorité parallèles à exécution séquentielle* des *files de priorité parallèles à capacité parallèle*.

10.2.1 File de priorité à exécution séquentielle

Dans ce premier modèle, on ne dispose que d'une file de priorité unique et globale sur laquelle les opérations sont effectuées séquentiellement, les différents processeurs y accèdant suivant un protocole d'exclusion mutuelle.

Dans un environnement distribué, ce modèle s'apparente au classique modèle Maître/Esclave, sur lequel de très nombreux travaux ont été réalisés. La file de priorité globale est gérée par le maître qui reçoit les requêtes de travail des esclaves, leur fournit les sous-problèmes à développer localement et reçoit les nouveaux sous-problèmes à insérer dans la file de priorité. Nous y revenons dans la partie implantation.

Quoique généralement simple à implémenter, ce modèle présente deux défauts principaux. Tout d'abord, la capacité du processeur maître qui doit stocker tous les sous-problèmes en attente, représente une première limitation. En outre, la centralisation de la file de priorité peut facilement représenter un goulet d'étranglement, spécialement lorsque les requêtes des esclaves sont fréquentes.

10.2.2 File de priorité à capacité parallèle

Pour pallier à ces éventuels defauts, le parallélisme peut être introduit au niveau de la file de priorité.

Pour ceci, la première possibilité consiste à éclater la file de priorité en plusieurs sous représentations. Chaque processeur détient et gère alors sa propre file de priorité locale. En contrepartie, il devient indispensable de mettre en œuvre une politique d'équilibrage de charge, afin de maintenir une relative équité entre les processeurs en termes de quantité et qualité (charge potentielle) des sous-problèmes. B. Le Cun appelle ce modèle *file de priorité à données parallèles*.

Dans la seconde possibilité, *file de priorité à opérations parallèles*, la file de priorité demeure unique mais des opérations concurrentes y sont effectuées par les différents processeurs. Un processeur accéde à un moment donné à une partie de la file de priorité, des protocoles de verrouillage partiel étant alors utilisés.

Ces deux possibilités peuvent être combinées.

10.3 Le cas particulier du problème d'affectation quadratique

10.3.1 Le problème d'affectation quadratique

Le Problème d'Affectation Quadratique (PAQ) est un problème très classique d'optimisation combinatoire dans lequel n unités doivent être affectées à n sites, de façon à minimiser le coût total de l'affectation. Or ce coût est quadratique car proportionnel à la somme des produits des distances inter-sites par les valeurs des flots inter-unités. Ce problème peut être formulé de la façon suivante :

étant données deux $(n \times n)$ matrices,

$$\begin{array}{rcll} F & = & (f_{ij}) & \text{avec } f_{ij} \text{ : flot entre les unites } i \text{ et } j, \\ D & = & (d_{kl}) & \text{avec } d_{kl} \text{ : distance entre les sites } k \text{ et } l, \end{array}$$

trouver une permutation p de $N = \{1, 2, ..., n\}$ qui minimise la fonction de coût global

$$Cout(p) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij}\ d_{p(i)p(j)}.$$

L'une des principales particularités, et par là même l'un des principaux intérêts, de ce problème NP-complet est l'extrême difficulté de sa résolution exacte. Même exécuté en parallèle sur de puissantes machines, les meilleurs algorithmes exacts récents (Crouse et Pardalos 1989 [10], Mautor et Roucairol 1994 [8], Clausen et Perregaard 1994 [3], Brungger, Marzetta, Clausen et Perregaard 1996 [1]) demeurent fortement limités quant à la taille des applications qu'ils peuvent aborder. Certes, le domaine des algorithmes exacts pour le PAQ demeure tout à fait actif depuis plus de 30 ans et, par suite, la limite est repoussée régulièrement. Mais à l'heure actuelle, celle-ci demeure toutefois située aux alentours de vingt.

Repousser cette limite peut donc être considéré comme un intéressant, bien que très difficile, challenge. De plus, ce problème a une valeur d'exemplarité et d'autres problèmes d'optimisation combinatoire, tel le problème d'allocation de tâches, peuvent bénéficier des avancées réalisées dans sa résolution.

Toutefois, il paraît dorénavant indispensable pour cela d'accroître la puissance globale de calcul du système en faisant appel à une architecture parallèle ou distribuée et en utilisant un grand nombre de processeurs. Les problèmes de parallélisation d'un algorithme Branch and Bound évoqués à la section précédente (équilibrage de la charge, détermination d'une bonne granularité des tâches, ...) deviennent alors cruciaux.

10.3.2 Les particularités de l'algorithme Branch and Bound

Nous ne détaillons pas ici tous les aspects de l'algorithme séquentiel et renvoyons le lecteur intéressé à l'article [8]. En revanche, nous tenons à souligner les particularités de cet algorithme ayant une influence pour sa parallélisation.

Tout d'abord, de par la nature du problème - minimisation du produit scalaire de deux matrices -, les valeurs des solutions réalisables sont généralement assez homogènes. En particulier, il existe pour la plupart des problèmes un grand nombre de solutions dont la valeur est proche de l'optimum. Si ce facteur explique en partie la difficulté rencontrée par les méthodes exactes, il a aussi pour conséquence que les métaheuristiques les plus performantes (Recherche Tabou, Recuit Simulé, Algorithmes Génétiques, Scatter Search) obtiennent d'excellents résultats. Ainsi, la solution de départ utilisée par les méthodes exactes est-elle excellente et même souvent optimale et le travail de la méthode exacte devient-il de plus en plus de prouver son optimalité.

Cet aspect est évidemment fort important lors de la parallélisation d'une méthode exacte. Tout d'abord, il évite l'occurrence d'importantes anomalies d'accélération. Par ailleurs, une autre conséquence est que l'ordre dans lequel les noeuds sont explorés n'est pas vraiment critique. Il est ainsi préférable pour un algorithme séquentiel d'utiliser une stratégie “profondeur d'abord”.

Une autre particularité du PAQ est le fait que, malgré de très nombreuses recherches sur le sujet, il soit resté impossible de trouver une borne inférieure d'excellente qualité. La borne encore utilisée par la plupart des méthodes exactes est celle de Gilmore-Lawler [5, 6]. Or, cette borne ancienne est certes très rapide à calculer mais de piètre qualité et produit des arborescences de recherche rapidement énormes. La tâche correspondant à l'étude et à l'évaluation d'un seul noeud est donc de grain extrêmement fin.

10.4 Modèles implémentés

Dans cette partie, nous allons présenter deux implantations possibles reprenant les concepts énoncés présentés précédemment. Ces implantations sont les représentants typiques des deux classes principales : la classe centralisée et la classe distribuée.

10.4.1 Schéma Maître/esclave

Cette première implantation, appartenant à la famille centralisée, utilise une file de priorité à exécution séquentielle (voir terminologie de la section 2). Deux types d'activités sont définies : l'activité maître et l'activité esclave, comme représenté sur la figure 10.1.

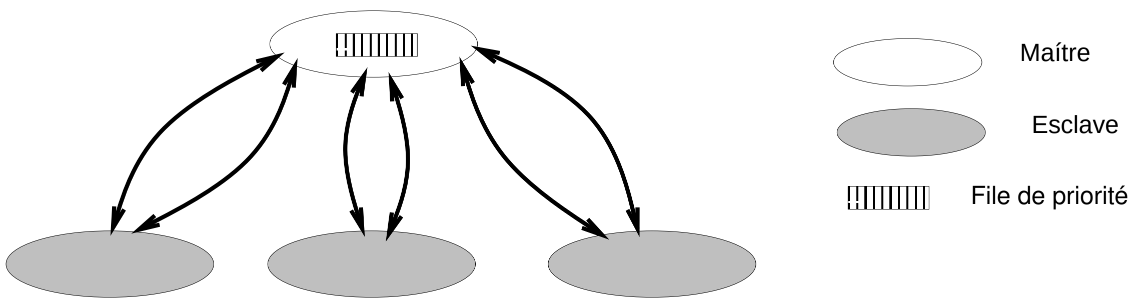


Figure 10.1 : Architecture du modèle maître/esclave

Le rôle du maître est de :

- initialiser le début de l'exécution ;
- stocker les problèmes en attente de traitement et donc gérer la structure de données globale ;
- recevoir des esclaves les requêtes pour du travail et leur envoyer en retour des sous-problèmes à résoudre ;
- recevoir et stocker les nouveaux sous-problèmes générés par les esclaves ;
- stocker et mettre à jour la meilleure solution trouvée jusqu'à présent ;
- détecter la terminaison de l'application.

D'un autre côté, le rôle de l'esclave est de :

- demander du travail au maître ;
- développer, éventuellement partiellement, le sous-arbre correspondant au sous-problème reçu du maître ;
- envoyer au maître des sous-problèmes qui seront explorés plus tard et la meilleure solution trouvée durant une recherche locale quand elle a été améliorée.

L'intérêt d'une telle méthode est qu'elle ne nécessite pas d'algorithmes sophistiqués d'équilibrage de charge. La répartition se fait tout simplement par des requêtes de travail des esclaves quand ils sont inactifs ce qui est une des méthodes de régulation les plus simples que l'on puisse imaginer.

Cependant, dans ce cas, le grain de calcul a un impact significatif sur les performances. D'un côté, un grain fin implique un grand nombre de communications entre les esclaves et le maître puisque l'intervalle de temps entre deux demandes de travail est petit. La file de prorité unique devient alors un goulot d'étranglement qui pénalise les performances. D'un autre côté, un grain trop gros n'exploitera pas totalement l'architecture parallèle car il amènera des périodes d'inactivité des processeurs, en particulier au début et à la fin de l'exécution.

Quand on parallélise une exploration d'arbre de recherche, le grain est proportionnel au nombre de nœuds explorés durant une activité (on se place dans le cadre d'une tâche esclave). Dans l'implantation dont nous présentons les résultats dans la suite, la quantité de nœuds qui peuvent être explorés durant une activité est bornée supérieurement par une valeur qui est un paramètre de l'algorithme. Donc,

- si la taille (en nombre de nœuds) du sous-arbre correspondant au sous-problème reçu par l'esclave est inférieure au paramètre donné, le sous-arbre sera complètement exploré,

- sinon l'esclave développe, suivant une stratégie profondeur d'abord, un nombre de nœuds égal au paramètre transmis et envoie au maître les nœuds correspondants aux sous-arbres restant à explorer.

Durant nos expériences ce paramètre a été positionné à une valeur importante pour :

- minimiser les communications entre le maître et les esclaves,

- minimiser le nombre de nœuds stockés dans la file de priorité gérée par le maître,

- assurer un plus grand usage de la stratégie "profondeur d'abord" qui est plus adaptée au PAQ, comme cela a été mentionné précédemment.

La famine de certains proceseurs durant la phase d'initialisation peut être largement minimisée en réduisant temporairement la valeur de granularité (en positionnant le paramètre décrit ci-dessus à une faible valeur) au début de l'exécution. De cette façon les premières tâches sont achevées plus vite que les suivantes ce qui permet "d'amorcer la pompe" du parallélisme.

L'une des caractéristiques principales de ce modèle réside dans son mélange des stratégies d'exploration. Si les explorations locales se font en "profondeur d'abord", les nœuds sont stockés et sélectionnés suivant une stratégie "meilleur d'abord". Ainsi, le nœud envoyé à un esclave est toujours celui de plus haute priorité. Pour cette raison, la structure de données utilisée par le maître pour gérer les nœuds est une funnel-table [7].

Cette stratégie “meilleure d'abord” participe à un bon équilibrage de la charge, en terme qualitatif. En effet, les nœuds ayant des évaluations similaires, et donc correspondant a priori à des potentialités de travail équivalentes, sont distribués durant la même période aux différents esclaves. De plus, durant la phase de terminaison de l'application, les sous-arbres explorés seront les moins intéressants, car ayant une évaluation proche de la meilleure solution trouvée. Pour cette raison, leur exploration sera effectuée rapidement ce qui limite les risques d'inactivité des processeurs durant cette phase *a priori* génératrice de déséquilibre.

On peut remarquer que cette stratégie d'exploration mixte est originale et que, de ce point de vue, notre modèle ne peut être considéré comme un “pur” modèle Maître/Esclave.

Les résultats expérimentaux obtenus avec cette implantation se trouvent dans la section 10.5 page 224.

10.4.2 Schéma complètement distribué

Selon la terminologie introduite dans la section 2, ce second schéma d'implantation utilise une file de priorité à données parallèles. Ainsi, les données dont le stockage était auparavant centralisé sur un seul processeur sont maintenant distribuées sur l'ensemble des différents processeurs, une file de priorité locale se trouvant sur chaque processeur. L'ensemble de ces files de priorité définit un schéma d'exécution entièrement distribué.

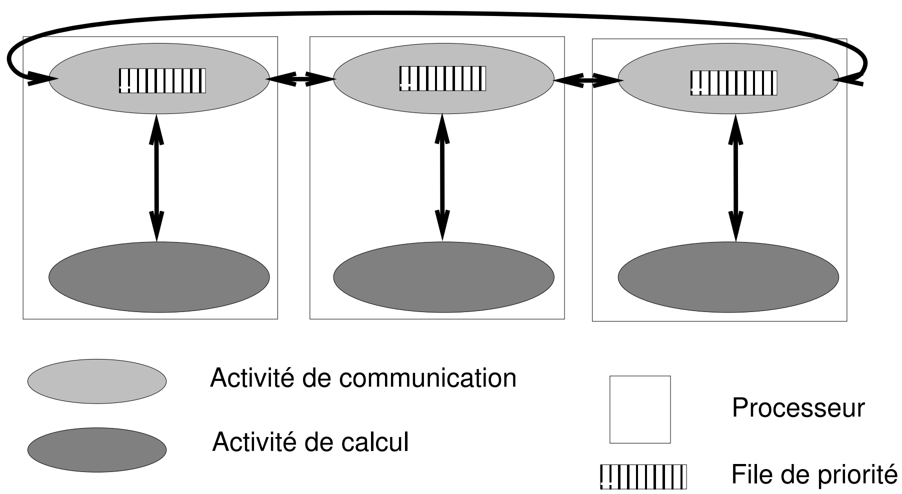


Figure 10.2 : Schéma de communication de l'implantation complètement distribuée

Une des caractéristiques principales de cette implantation réside dans l'utilisation de deux types d'activités autonomes et concurrentes sur chaque processeur comme représenté sur la figure 10.2 :

- L'activité de calcul est similaire à l'activité esclave de l'implantation précédente : elle demande du travail (un sous-problème) à l'activité de régulation se trouvant sur le même processeur qu'elle, explore le sous-arbre correspondant et retourne, s'il y a lieu, les sous-problèmes à stocker dans la file de priorité locale ainsi qu'une nouvelle solution si elle en a trouvé une meilleure. Ces trois opérations sont exécutées séquentiellement.

- L'activité de régulation est chargée de quatre tâches : la gestion de la file de priorité locale, la communication avec l'activité de calcul qui se trouve sur le même processeur, la participation à l'équilibrage global de la charge et enfin la participation à la détection de la terminaison.

La distribution de tous les sous-problèmes entre les processeurs est la solution aux faiblesses exhibées par l'implantation précedente : la saturation mémoire du processeur sur lequel se trouve le maître et l'absence de parallélisme dans la gestion de la file. Cependant, des techniques d'équilibrage plus sophistiquées sont alors nécessaires pour répartir équitablement la charge entre les files locales comme nous allons le voir maintenant.

Les éléments de base d'un protocole d'équilibrage de charge sont la politique d'information et la politique de transfert entre les files locales. Ces deux éléments caractérisent le mécanisme d'équilibrage. Notre politique d'information est basée sur des messages circulant sur un anneau de communication. L'un des buts de tout équilibrage est de ne pas être plus coûteux qu'une simple exécution sans équilibrage, c'est pourquoi nous avons choisi un algorithme utilisant des seuils ce qui évite de continuer à réguler quand la charge est déjà importante. Nous utilisons deux seuils qui caractérisent trois états possibles pour les files de priorité locales : sous-chargé, normal et chargé.

État du receveur	Message reçu	
	Sous chargé	Sur chargé
Sous chargé	Propage le message	Propage le message
Normal	Transmet des nœuds à l'envoyeur	Propage le message
Sur chargé	Transmet des nœuds à l'envoyeur	Transmet des nœuds à l'envoyeur

Tableau 10.1 : Action entreprise à la réception d'un message de changement d'état

Un message est envoyé par une activité de régulation quand son état de charge change (c'est à dire quand un seuil est franchi). Un processeur qui reçoit un message réagit à celui-ci suivant son propre état de charge. Le tableau 10.1 synthétise l'ensemble des actions entreprises suivant l'état local et l'information reçue. Cette réaction peut mener à un transfert de sous-problèmes d'une file chargée vers une file sous-chargée.

L'un des avantages principaux de cette stratégie d'équilibrage est que, quand le système est stable, aucune communication ne s'effectue. En particulier quand toute la machine est chargée, il n'y a aucun surcoût dû à l'équilibrage puisque des messages ne sont envoyés que quand l'une des charges locales franchit un seuil. Nos tests ont ainsi montré que durant la majeure partie de la résolution d'un problème d'affectation quadratique on ne trouve aucune communication entre les processeurs.

10.5 Résultats expérimentaux et commentaires

Nos expérimentations ont été réalisées sur deux machines parallèles : un IBM SP2 avec 32 nœuds de calcul et une ferme d'ALPHA de DEC avec 16 nœuds. De plus, quelques expériences ont été faites sur un Cray T3D comportant 128 processeurs.

Le premier algorithme (maître/esclave) a en fait obtenu, un tout petit peu à notre surprise d'ailleurs, de très bonnes performances parallèles. Nous avons ainsi obtenu un speed-up quasiment linéaire et égal au nombre de processeurs jusqu'à 16 processeurs. Il nous faut cependant préciser que ces performances se dégradent quand le nombre de processeurs augmentent, cette observation découlant des résultats préliminaires que nous avons obtenus sur le Cray T3D.

Dans un environnement parallèle de taille moyenne (16 ou 32 processeurs) faiblement chargé par d'autres applications, les performances du second modèle (complètement distribué) sont réellement semblables à celles présentées ci-dessus pour le premier modèle. Un bon indice de performance concerne l'équilibrage de la charge. La figure 10.3 représente le nombre de nœuds explorés sur chaque processeur durant la résolution d'un problème de Eschermann et Wunderlich de taille 16. Les 32 processeurs utilisés sont homogènes, ils sont de puissance équivalente, et la machine était faiblement chargée. La distribution de la charge est alors très bonne, la différence entre les deux extrêmes étant inférieure à 10%, ce qui montre une très bonne utilisation moyenne des processeurs.

Nos deux algorithmes ont été utilisés pour résoudre des problèmes classiques de la littérature, qui peuvent être trouvés dans la "librairie QAPLIB" [2] :

- des problèmes de taille 16 proposés par Eschermann et Wunderlich,
- le 'célèbre' problème de Nugent de taille 20,

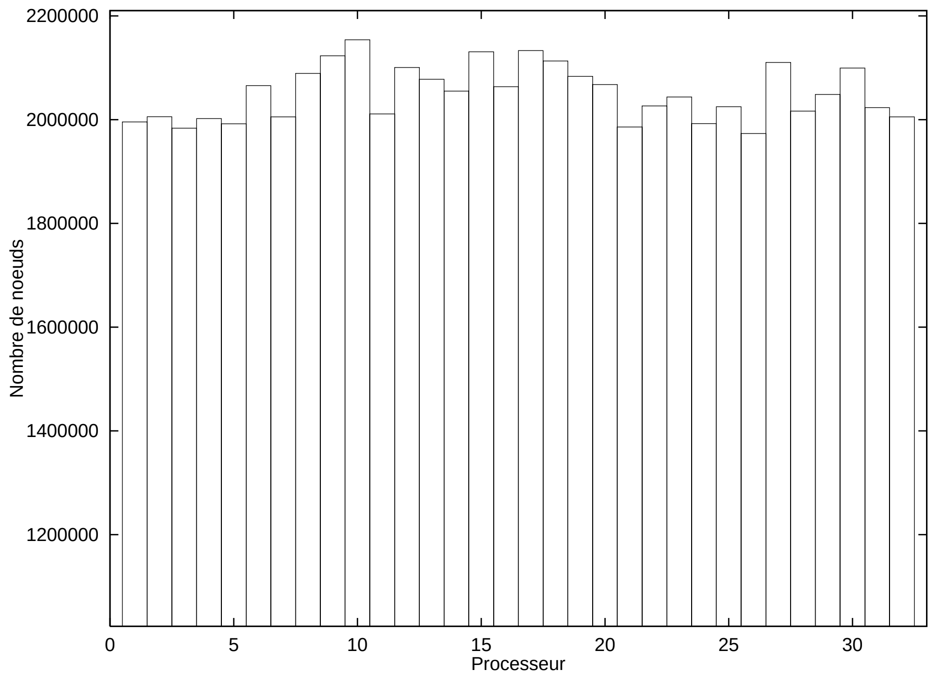


Figure 10.3 : Répartition de l'exploration des nœuds sur les processeurs pour la résolution du problème esc16c

- un autre problème de taille 20 proposé par C. Roucairol.

Les résultats sont donnés dans le tableau suivant indiquant la solution optimale trouvée, ainsi que le temps pris pour la résolution et le nombre de nœuds explorés pour chaque implantation. Tous les résultats de ce tableau ont été obtenus sur un IBM SP2 avec 32 processeurs.

Prob.	Sol.	M/E (sec)	# Nœuds	Distrib. (sec)	# Nœuds
esc16a	68	59	525.285	39	451.631
esc16b	292	333	10.221.885	292	10.197.329
esc16c	160	2087	63.649.543	1792	62.571.257
esc16d	16	1793	43.884.805	1147	39.463.020
esc16e	28	38	452.266	48	434.733
esc16g	26	1752	55.195.289	1605	51.531.127
esc16h	996	9619	332.385.306	9576	331.908.440
esc16i	14	3569	130.559.757	2687	121.855.947
esc16j	8	988	30.596.415	948	30.624.955
nug20	2570	13423	121.169.074	14949	120.278.528
rou20	725522	68342	445.097.368	67591	444.847.900

Globalement, on peut donc dire que les performances des deux approches sont similaires. En moyenne les résultats de l'implantation distribuée apparaîssent très légèrement meilleurs que ceux du maître/esclave. Le modèle distribué apparaît cependant de toute façon plus intéressant pour deux raisons :

- Sur une machine plus chargée, cas classique des environnements partagés entre plusieurs utilisateurs comme les réseaux de stations de travail, le second modèle est plus performant que le premier. Des tests réalisés sur une machine chargée à 50% de sa capacité de calcul ont montré un meilleur comportement du second modèle, dû à un meilleur équilibrage. Ce point est important car l'utilisation d'un réseau de stations de travail est courant pour résoudre des problèmes aussi gros car il est rare de pouvoir utiliser de manière exclusive une machine parallèle dédiée pendant des durées pouvant excéder la journée, voire la semaine.

- Quand le nombre de processeurs devient important, un schéma distribué amène logiquement à une dégradation des performances dûes au goulet d'étranglement que représente la séquentialité des accès à la structure globale. Les tests préliminaires que nous avons menés sur le Cray T3D ont mis clairement ce phénomène en évidence.

10.6 Conclusion et perspectives

Dans cet article nous avons présenté les grandes lignes régissant la parallélisation d'une application d'exploration d'arbres de recherche. Nous nous sommes particulièrement focalisés sur les aspects régulation de charge qui sont au cœur de cette parallélisation à cause du haut degré d'irrégularité des applications visées. Nous avons illustré notre discours par la description d'algorithmes de régulation appartenant aux deux familles principales, centralisée et distribuée que nous avons comparées à l'aide de résultats expérimentaux obtenus sur le problème, appartenant au domaine de l'optimisation combinatoire, de l'affectation quadratique.

La direction principale que nous comptons maintenant donner à ce travail concerne la conception de supports d'exécution pour les applications décrites dans cet article. L'idée de base est que, plutôt que d'avoir des sous-arbres en cours d'exploration et d'autres en attente, il est préférable de les explorer tous simultanément. Cela nécessite un grain de parallélisme plus fin que celui proposé par les environnements traditionnels comme que PVM ou MPI. C'est pourquoi le support d'exécution que nous proposons utilise les processus légers, ou threads, comme activité de base. La forte irrégularité de la classe d'applications visée induit alors un déséquilibre dans la charge, en terme d'activités, de chaque processeur. Pour compenser ce déséquilibre notre support d'exécution propose l'utilisation du mécanisme de migration d'activités entre les processeurs. On peut alors faire un parallèle entre la régulation de charge étudiée dans cet article et l'équilibrage des activités dans le contexte décrit ci-dessus. L'utilisation des processus légers en tant que support d'exécution et l'équilibrage de ces activités via le mécanisme de migration ont d'ores et déjà fait l'objet de plusieurs travaux au LIFL (voir en particulier [4, 9]). Notre perspective consiste donc à adapter ces techniques à la résolution exacte de problèmes d'optimisation combinatoire, tel le problème d'affectation quadratique.

Bibliographie

[1] Brungger (A.), Marzetta (A.), Clausen (J.) et Perregaard (M.). – *Joining forces in solving large-scale QAP in parallel.* – RR n° 96-23, DIKU, 1996.

[2] Burkard (R.), Karisch (S.) et Rendl (F.). – Qaplib - a quadratic assignment problem library. *European Journal of Operational Research*, vol. 55, 1991, pp. 115–119.

[3] Clausen (J.) et Perregaard (M.). – *Solving Large Quadratic Assignment Problems in Parallel.* – Rapport technique n° 1994/22, DIKU, Denmark, 1994. accepted for publication in Computational Optimization and Applications.

[4] Denneulin (Y.). – Programmation parallèle avec priorités et régulation de charge. *In : RenPar'9*, éd. par Schiper (A.) et Trystram (D.). – Lausanne, Mai 1997.

[5] Gilmore (P.). – Optimal and suboptimal algorithms for the quadratic assignment problem. *SIAM Journal on Applied Math.*, vol. 10, 1962, pp. 305–313.

[6] Lawler (E.). – The quadratic assignment problem. *Management Science*, vol. 9, 1963, pp. 586–599.

[7] Le Cun (B.). – *Des structures de données parallèles.* – 4, Place Jussieu, 75252 Paris Cedex 05, FRANCE, Thèse de PhD, Université Pierre et Marie Curie – Paris VI, Jan. 1996. In French.

[8] Mautor (T.) et Roucairol (C.). – A new exact algorithm for the solution of quadratic assignment problems. *Discrete Applied Mathematics*, vol. 55, 1994, pp. 281–293.

[9] Namyst (R.). – *PM^2 : un environnement pour une conception portable et efficace des applications parallèles irrégulières.* – Thèse de PhD, Université des Sciences et Technologies de Lille, Janvier 1997.

[10] Pardalos (P. M.) et Crouse (J.). – A parallel algorithm for the quadratic assignment problem. *In : Proceedings of Supercomputing 89.* ACM, pp. 351–360.

Troisième partie

Langages de Programmation et machines

Chapitre 11

Gestion de l'irrégularité dans HPF (High Performance Fortran)

Philippe Marquet (LIFL, Université de Lille)

11.1 Résumé

HPF (High Performance Fortran) est une extension de la nouvelle norme Fortran visant une programmation efficace des machines parallèles. HPF est un langage data-parallèle. Le parallélisme est exprimé par des affectations de tableaux ; la distribution des tableaux est réalisée par le compilateur à partir des indications données par le programmeur sous la forme de directives de placement.

Les objets manipulés par HPF sont par essence même réguliers (tableaux multi-dimentionels). L'irrégularité est reporté au niveau de directives de placement ad hoc. L'état de l'art des compilateurs HPF aborde maintenant la prise en compte de ces nouvelles directives.

11.2 HPF : un langage data-parallèle extension de Fortran 95

11.2.1 Fortran pour les années 2000 ?

Depuis son origine dans les années 1950, Fortran est le langage de prédilection des scientifiques. L'évolution des architectures des machines dont disposent ces scientifiques et l'intégration de constructions plus récentes introduites dans d'autres langages de programmation ont constamment fait évoluer Fortran jusqu'à la dernière norme Fortran 95. Fortran 95 est une évolution relativement mineure de la norme précédente Fortran 90 alors que ce dernier marquait une étape importante sur son prédécesseur Fortran 77.

Fortran 90 (et donc Fortran 95) promeut les tableaux au rang d'objets primaires du langage : les opérations calculatoires habituelles peuvent avoir des opérandes qui soient des tableaux. L'expression du parallélisme est ainsi réalisée au niveau des données élémentaires formant le tableau : Fortran 90 est un langage data-parallèle. D'autres langages data-parallèles introduisent des structures spécifiques différentes des tableaux pour supporter les ensembles d'éléments sur lesquels porteront les opérations parallèles [10]

Considérant les constructions fournies par Fortran 90 insuffisantes pour tirer parti des performances des machines parallèles, le Forum HPF (HPFF) a été fondé en 1992 par des industriels et des chercheurs du monde académique pour proposer une extension de Fortran visant particulièrement ces machines parallèles. Les travaux de ce groupe sont initiés dans la perspective suivante : la pleine exploitation des machines parallèles nécessite plus d'informations de la part du programmeur qu'il ne peut en exprimer en Fortran 90. Des extensions de Fortran avaient déjà été proposées en ce sens par des constructeurs ou des équipes universitaires : CM Fortran de Thinking Machines [14], MP Fortran de DEC/MasPar [11], MPP Fortran de Cray [13], Fortran D (Rice University) [7], Vienna Fortran (Vienna University) [16].

Les travaux du HPFF associés à de larges consultations sur ses propositions ont abouti à une première spécification du langage nommée HPF 1.0 en 1993 [8]. Certaines propositions de ce langage ont été intégrées dans la nouvelle norme Fortran 95.

Une seconde phase de travaux du HPFF a débouché sur la version actuelle de HPF : HPF 2.0 [9].

11.2.2 Parallélisme de données en Fortran 90

11.2.2.a Les tableaux, une structure de données parallèle

Les structures parallèles de Fortran 90 sont les tableaux. Les opérateurs habituels de Fortran sont étendus aux tableaux. La boucle d'affectations Fortran 77

```
    REAL T, U
    DIMENSION T(10), U(10)
    DO 10, I = 1, 10
      T(I) = U(I) + 5
10  CONTINUE
```

peut donc être réécrite en Fortran 90 :

```
    REAL, DIMENSION (10) :: T, U
    T = U + 5
```

Les variables `T` et `U` sont des variables data-parallèles.

Sémantiquement, l'opération est réalisée sur les éléments correspondants des tableaux opérandes. Les valeurs scalaires utilisées dans une opération avec une structure data-parallèle sont fonctionnellement équivalentes à une structure data-parallèle de taille convenable dont tous les éléments ont la valeur du scalaire.

Notons qu'une instruction d'affectation data-parallèle n'est pas équivalente à la boucle scalaire correspondante. La sémantique de l'affectation data-parallèle est telle que toute la partie droite est évaluée avant le rangement des valeurs dans la partie gauche. Une dépendance entre les parties droite et gauche nécessite donc le passage par une zone temporaire.

Les opérations data-parallèles doivent être appliquées à des opérandes *conformes*, c'est-à-dire à des opérandes de même rang (même nombre de dimension) et de même taille.

11.2.2.b Allocation des tableaux en Fortran 90

L'allocation des tableaux Fortran 77 est statique : la taille des tableaux doit pouvoir être évaluée dès la compilation et cette taille ne peut varier au cours de l'exécution du programme. Trois allocations possibles sont identifiées en Fortran 90 : les allocations statique, automatique et dynamique.

Fortran 90 introduit des tableaux automatiques : ils sont alloués dynamiquement à l'entrée d'une fonction et automatiquement désalloués à la fin de la fonction ; leur taille peut différer d'un appel à l'autre de la fonction ; cette taille est par exemple fonction de la valeur d'un paramètre. La fonction suivante échange les valeurs de deux tableaux de taille quelconque :

```
SUBROUTINE SWAP (A,B)
  REAL, DIMENSION (:) :: A, B
  REAL, DIMENSION (SIZE(A)) :: TEMPOAUTO
  TEMPOAUTO = A
  A = B
  B = TEMPOAUTO
END SUBROUTINE
```

La dynamicité des objets parallèles de Fortran 90 repose sur la dynamicité des tableaux. Un tableau est spécifié dynamique par l'attribut de déclaration `ALLOCATABLE` ; la taille du tableau n'est pas précisée lors de cette déclaration, seul le nombre de dimensions doit être spécifié. Les instructions `ALLOCATE` et `DESALLOCATE` sont utilisées par le programmeur pour explicitement allouer et désallouer ces tableaux. Par exemple

```
INTEGER N
REAL, DIMENSION (:,:), ALLOCATABLE :: A
```

```
...
READ *, N
ALLOCATE (A(N,N*N))
...
DESALLOCATE (A)
```

Notons que les anciennes valeurs d'un tableau sont systématiquement perdues lors d'une opération d'allocation de ce tableau ; il n'existe pas d'opération de "réallocation" des tableaux en Fortran 90.

11.2.2.c Opérations de description

Lors de la manipulation de structures data-parallèles, il est parfois nécessaire d'isoler parmi les éléments d'une structure un sous-ensemble de ceux-ci. En Fortran 90, cette possibilité est offerte par les opérations de description (*gather/scatter*). Une opération de description est une opération réalisant un accès (en lecture ou en écriture) à un sous-tableau. Dans les langages scalaires, ces opérations résultent en une référence sur un élément d'un tableau ; la description est alors réalisée par une valeur scalaire entière : l'indice de l'élément dans le tableau.

Deux types de descriptions sont possibles en Fortran 90 : la description par triplet et la description par un vecteur d'entiers.

La description par triplet sélectionne des éléments contigus ou régulièrement espacés dans une dimension d'un tableau. Ces éléments sont désignés par trois valeurs entières : la borne inférieure, la borne supérieure et le pas. Par exemple, l'expression

```
A (I, 1:N)
```

sélectionne les éléments 1 à `N` de la ligne `T` alors que l'expression

```
A (1:M:2, J)
```

référence les éléments impairs de 1 à `M` de la colonne `J`.

La description par un vecteur d'entiers est une de la description habituelle par une valeur entière scalaire. L'expression (dans laquelle l'expression `(/1, 7, 3, 2/)` dénote une valeur littérale tableau formée des quatre entiers)

```
T ((/1, 7, 3, 2 /))
```

est composée, dans l'ordre, des éléments

```
T(1)  T(7)  T(3)  T(2)
```

Pour les deux tableaux `U` et `V` :

```
U = (/1, 3, 2/)
V = (/2, 1, 1, 3/)
```

les expressions `Z(3,V)` et `Z(U,V)` dénotent respectivement les valeurs

```
Z(3,2)  Z(3,1)  Z(3,1)  Z(3,3)
```

et

```
Z(1,2)  Z(1,1)  Z(1,1)  Z(1,3)
Z(3,2)  Z(3,1)  Z(3,1)  Z(3,3)
Z(2,2)  Z(2,1)  Z(2,1)  Z(2,3)
```

Il est à remarquer qu'un même élément d'un tableau peut être référencé plusieurs fois dans une description par un vecteur d'entiers. Ce fait ne pose pas de problème en lecture. Par contre, en écriture le résultat dépendrait de l'ordre de traitement des éléments (suivant les indices croissants ou décroissants par exemple). De ce fait Fortran 90 interdit les opérations de description par un vecteur d'entiers comportant des doublets en partie gauche d'une affectation.

11.3 Distribution des données en HPF

L'allocation des structures parallèles que forment les tableaux sur la mémoire distribuée d'une machine parallèle nécessite un partitionnement de chacun de ces tableaux entre les différents processeurs. Le parallélisme exprimé sur les tableaux est alors exploitable entre les processeurs : en parallèle, chacun d'entre eux réalise l'opération sur la portion de tableau qui lui est allouée. Le postulat du Forum HPF établit le manque d'informations exprimées par le programmeur permettant au compilateur de définir un "bon" partitionnement des tableaux.

HPF propose que le programmeur exprime ce partitionnement au moyen de directives incluses dans le source du programme sous forme de "commentaires structurés" introduit par les signes `!HPF$`.

Ce placement des données est réalisé par le regroupement de tableaux sur des *template* (entités abstraites définies par un rang et une taille) via les directives `ALIGN`. Ces templates sont ensuite distribués sur des arrangements rectilinéaires de processeurs virtuels (les `PROCESSORS`) par les directives `DISTRIBUTE` (figure 11.1).

La distribution des données spécifiée par les directives HPF vise à aider le compilateur à produire de meilleures performances sur les machines parallèles. Ces directives sont sans effet sur la sémantique des programmes.

L'intuition sous-jacente à la spécification du placement des données est la suivante :

- une opération sur différents objets est plus efficace si les objets sont sur le même processeur. Il convient donc d'aligner entre-eux les tableaux qui vont interagir pour mettre face à face les éléments correspondant ;

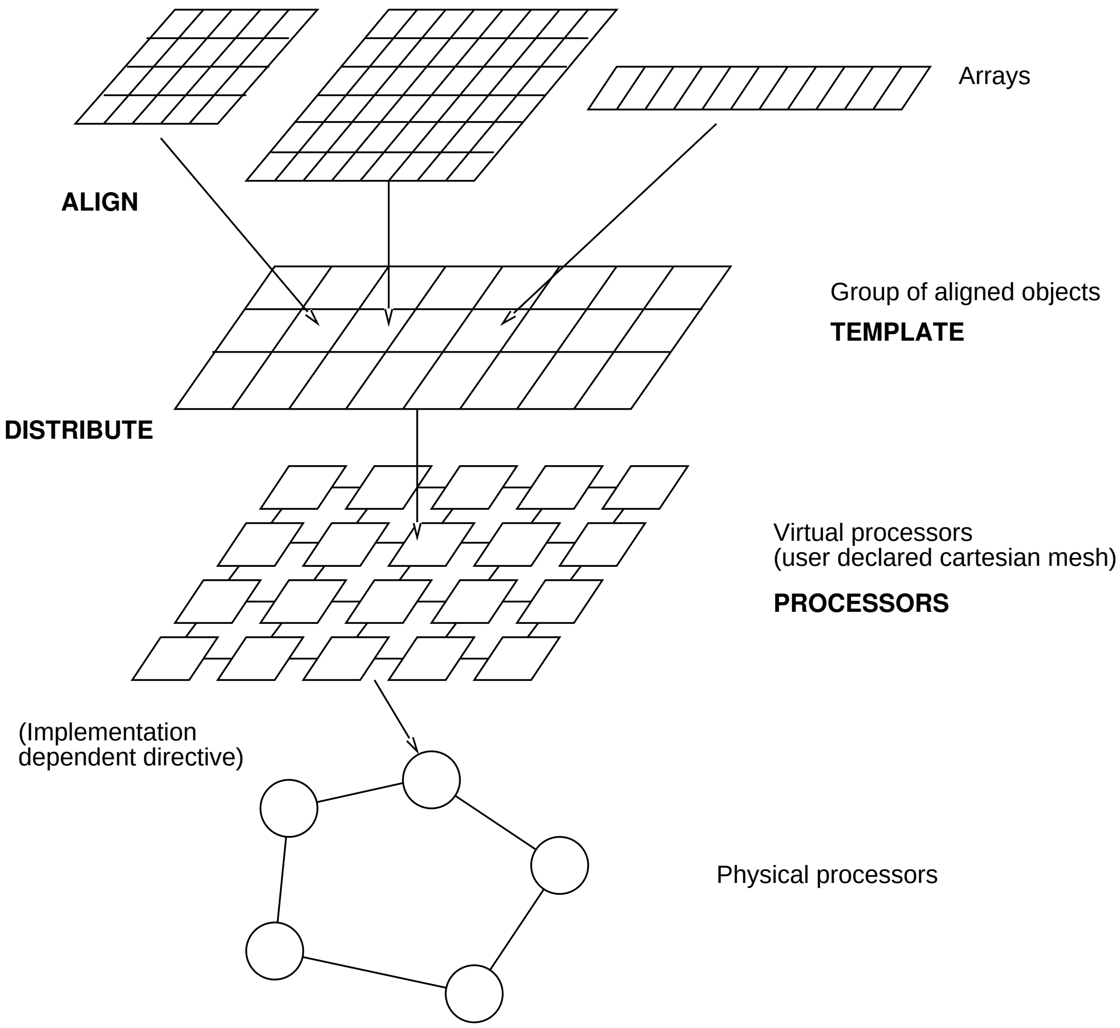


Figure 11.1 : Modèle HPF 2.0 de placement des données.

- de telles opérations peuvent être exécutées en parallèle si elles peuvent être réalisées sur différents processeurs. Il est donc nécessaire de distribuer les entités parallèles sur les processeurs.

11.3.1 Alignement de tableaux

L'alignement des données visent à rapprocher les données devant interagir. La directive d'alignement `ALIGN` d'HPF autorise le placement de l'ensemble des éléments d'un tableau. Typiquement, les directives d'alignement rapprochent des tableaux de templates précédement définis. Nous commentons l'usage de ces directives sur quelques exemples illustratifs.

L'extrait de source HPF

```
!HPF$ TEMPLATE D2 (N+1,N+1)
      REAL, DIMENSION (N,N) :: X, Y
!HPF$ ALIGN X(I,J) WITH D2(I,J)
!HPF$ ALIGN Y(I,J) WITH D2(I+1,J+1)
```

déclare un template `D2` sur lequel sont alignés deux tableaux `X` et `Y` comme illustrés par la figure 11.2

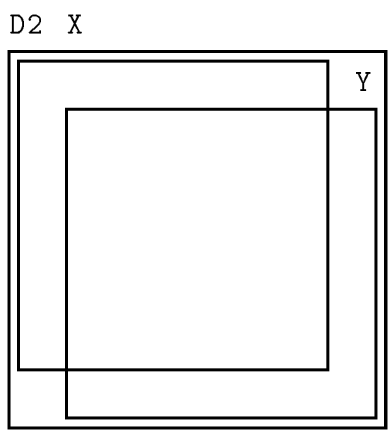


Figure 11.2 : Exemple d'alignement de deux tableaux `X` et `Y` sur un template `D2`.

La spécification d'alignement permet la réduction du parallélisme à un axe comme sur l'exemple suivant (figure 11.3) :

```
!HPF$ TEMPLATE D1 (N)
      REAL X (N, 34)
      REAL Y (N, 3)
      REAL Z (N, 4)
```

```
!HPF$ ALIGN X(J,K) WITH D1(J)
!HPF$ ALIGN Y(J,K) WITH D1(J)
!HPF$ ALIGN Z(J,K) WITH D1(J)
```

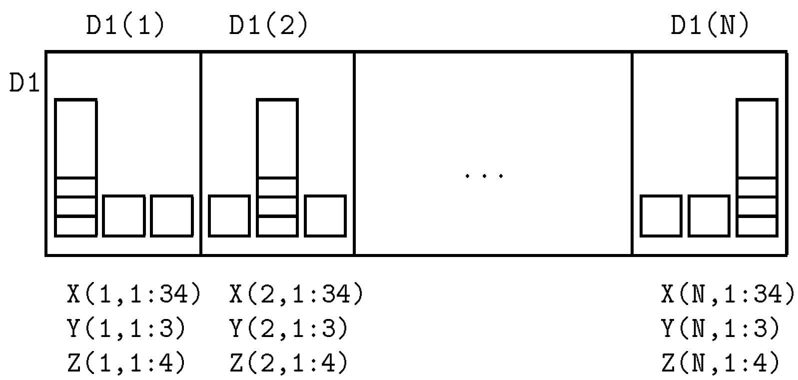


Figure 11.3 : Alignement : réduction du parallélisme à un axe.

Chacun des éléments du template `D1` reçoit 34 éléments de `X`, les 3 éléments de `Y` et les 4 éléments de `Z` correspondants.

Il est possible de transposer les axes du tableau et du template (figure 11.4) :

```
!HPF$ TEMPLATE D2(N,N)
      REAL     X (N,N)
!HPF$ ALIGN    X (I,J) WITH D2 (J,I)
```

ou d'inverser les axes du tableau et du template (figure 11.5) :

```
!HPF$ TEMPLATE D2(N,M)
      REAL     X (N,M)
!HPF$ ALIGN    X (J,K) WITH D2(N-J+1, M-K+1)
```

Les fonctions qui lient les indices des éléments du tableau à aligner et du template sont limitées aux fonctions affines pour faciliter les implantations des alignements HPF.

11.3.2 Distribution des entités parallèles

La seconde phase du placement des données HPF consiste en la définition des distributions des tableaux ou template sur les arrangements de processeurs virtuels

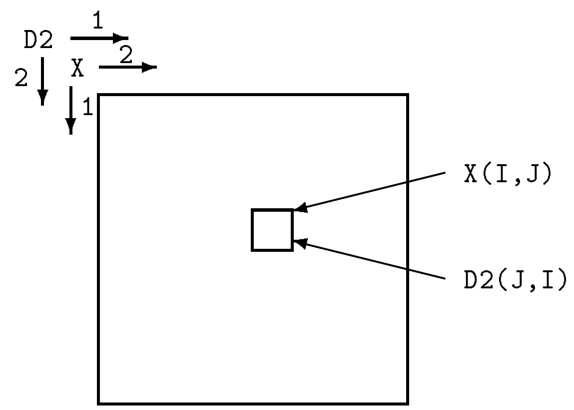


Figure 11.4 : Transposition des axes du tableau et du template.

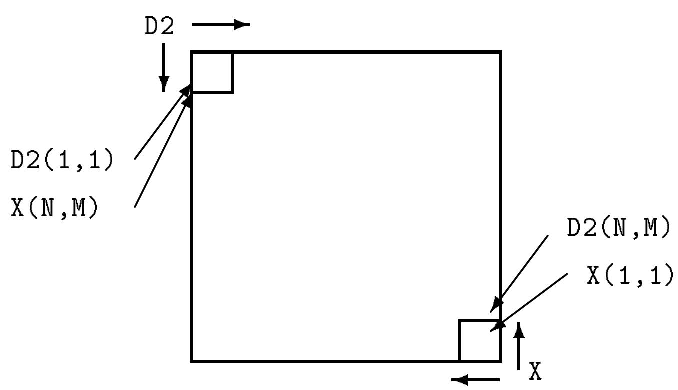


Figure 11.5 : Inversion des axes du tableau et du template.

PROCESSORS. L'idée sous-jacente est que deux objets placés sur le même processeur virtuel seront alloués sur le même processeur physique. Ces arrangements de processeurs virtuels sont définis par le programmeur de telle manière que le nombre de processeurs virtuels soit égal au nombre de processeurs physiques. Les fonctions intrinsèques NUMBER_OF_PROCESSORS et PROCESSORS_SHAPE peuvent être utilisées pour paramétrer les déclarations d'arrangements de processeurs virtuels :

```
!HPF$ PROCESORS PHUIT (8, NUMBER_OF_PROCESSORS()/8)
```

Deux distributions de base sont disponibles en HPF : la distribution par blocs et la distribution cyclique. Prennons comme exemple la distribution par blocs du template TT sur l'arrangement de processeurs PP :

```
!HPF$ TEMPLATE   TT (N)
!HPF$ PROCESSORS PP (P)
!HPF$ DISTRIBUTE TT(BLOCK) ONTO PP
```

On considère des blocs de $B = \lceil \frac{N}{P} \rceil$ éléments de TT. Ces blocs sont distribués, un sur chacun des processeurs de PP. Sur une hypothétique machine de $\texttt{P} = 5$ processeurs, un template de $\texttt{N} = 14$ éléments sera distribué ainsi :

Processeurs	1	2	...	5
	TT(1)	TT(4)		TT(13)
	TT(2)	TT(5)		TT(14)
	TT(3)	TT(6)		

Cette distribution par blocs autorise l'exécution du code

```
      REAL T(14)
!HPF$ ALIGN T WITH TT
      T(2:14:3) = T(1:13:3)
```

sans que des communications inter-processeurs soient nécessaires. Il est à noter que les distributions par blocs peuvent engendrer des pertes de place mémoire, voire l'inactivité de certains processeurs si la taille des objets distribués n'est pas multiple du nombre de processeurs.

Une distribution cyclique des éléments d'un objet est possible. Le programmeur spécifie une taille de blocs (par défaut 1) constitués d'éléments consécutifs de l'objet. Ces blocs sont cycliquement distribués sur les processeurs assurant ainsi généralement une meilleure répartition de la charge d'allocation. Par exemple la spécification

```
!HPF$ DISTRIBUTE TT (CYCLIC(2))
```

distribue les éléments de TT sur notre même machine de la manière suivante :

Processeurs	1	2	...	5
	TT(1)	TT(2)		TT(9)
	TT(2)	TT(3)		TT(10)
	TT(11)	TT(13)		
	TT(12)	TT(14)		

11.4 Extensions d'HPF-2 pour la distribution des données

La proposition HPF 2.0 présentée dans les sections précédentes restreint les constructions proposées dans un premier temps dans HPF 1. Par exemple les alignements et distributions sont désormais des informations statiques associées aux tableaux et templates. De même de nombreuses restrictions sur le passage en paramètre de tableau alignés et distribués assurent aux compilateurs la connaissance statique des alignements et distributions des structures parallèles. La mise en place de cette démarche espère découler rapidement sur la disponibilité effective de compilateur du langage.

Par ailleurs, la proposition HPF 2.0 inclut une section présentant les extensions approuvées de HPF. Ces extensions répondent à des besoins spécifiques en matière de distribution par exemple mais ne sont pas destinées à être supportées par les premières versions des compilateurs.

Nous présentons dans la suite les aspects dynamiques et irréguliers introduits par ces extensions et regarderons aussi les propositions d'autres langages extensions de Fortran en ce domaine.

11.4.1 Dynamicité et irrégularité en HPF

11.4.1.a Dynamicité des objets parallèles HPF

L'introduction de la dynamicité en HPF repose sur la dynamicité des tableaux présentée en section 11.2.2.b d'une part et sur la dynamicité des directives de placement. Le placement dynamique d'un objet nécessite l'identification de celui-ci comme *dynamique* par l'utilisation de l'attribut `DYNAMIC` lors de sa définition. Il est ensuite possible d'utiliser les pseudo-instructions `REALIGN` et `REDISTRIBUTE` pour modifier dynamiquement le placement de ces objets. Le modèle de placement des données est alors celui présenté à la figure 11.6.

Il apparaît naturel de combiner la dynamicité des tableaux avec celle du placement de ces tableaux. Lors de l'utilisation de tableaux dont la taille n'est pas connue à la compilation, il est en effet plus efficace de réaliser dynamiquement la distribution en ayant connaissance de cette taille. Suite à la déclaration d'un tableau

```
REAL, ALLOCATABLE, DIMENSION (:) :: TAB
```

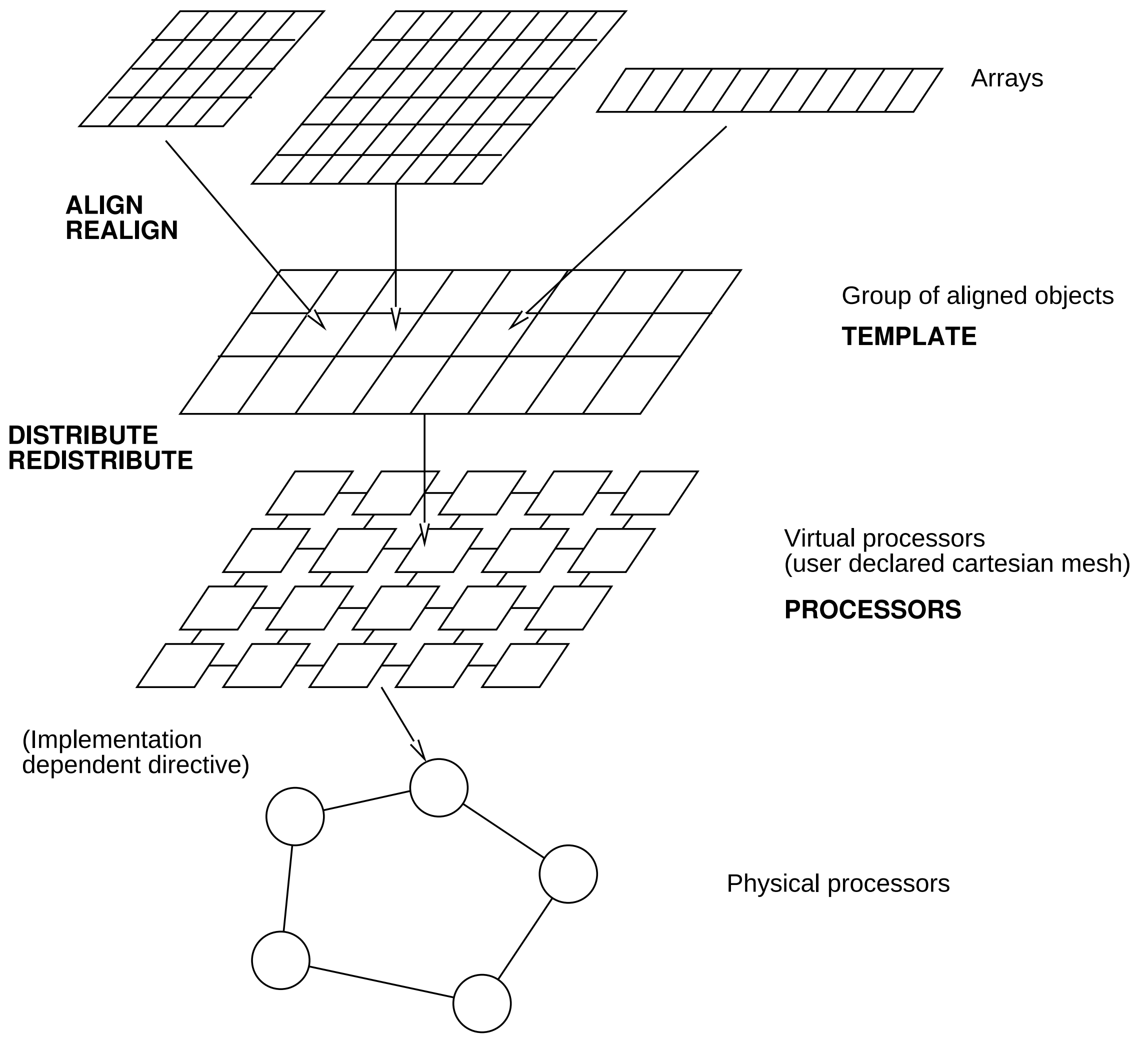


Figure 11.6 : Modèle étendu de placement des données de HPF 2.0.

```
!HPF$ DISTRIBUTE (BLOCK), DYNAMIC :: TAB
```

la séquence d'instructions

```
      ALLOCATE (TAB (N))
      ...
      N = ...
      ALLOCATE (TAB (N))
!HPF$ REDISTRIBUTE TAB (CYCLIC)
```

spécifiant une nouvelle allocation et une nouvelle distribution du tableau `TAB` se doit d'être réalisée de manière indissociable. En effet l'instruction d'allocation n'initialise pas les valeurs du tableau. L'instruction de redistribution ne devrait donc pas opérer au déplacement de ces valeurs pour effectuer le nouveau placement de `TAB` sur les processeurs. Cette réalisation de la redistribution est proposée par le document HPF-2.

Le placement dynamique des objets associé à une relâche des contraintes sur le passage en paramètre de tableaux distribués compliquent la tâche du compilateur qui ne peut déterminer avant l'exécution la distribution d'un tableau donné. Afin de réduire l'ensemble des distributions possibles pour un objet dynamique, le programmeur liste l'ensemble des formats de distribution que pourrait effectivement prendre l'objet par une déclaration `RANGE`, par exemple :

```
      SUBROUTINE SUB (X)
      REAL X (:, :)
!HPF$ RANGE X (BLOCK, CYCLIC) (BLOCK, BLOCK)
```

11.4.1.b Irrégularité des distributions

Les langages data-parallèles extensions de Fortran se limitent à l'utilisation de structures de données régulières telles les tableaux. La programmation d'applications irrégulières nécessite donc la projection par le programmeur de ses structures irrégulières sur les structures régulières disponibles dans le langage. Dans ce cadre, une distribution des données sur les processeurs cohérente avec la structure irrégulière nécessite des mécanismes irréguliers de distribution des données régulières. Les propositions de placement des données issues de HPF-2 apparaissent alors limitées.

Soit une application de calculs des différences finies dans laquelle une structure irrégulière en deux dimensions est composées de triangles représentant le domaine. Les nœuds de la structure sont représentés par un tableau à une dimension et les arêtes par un autre tableau. Une distribution de ces structures limitée aux distributions par blocs et cycliques amène des éléments nullement liés entre-eux à être alloués sur le même processeur et ne permet pas de regrouper sur un même processeur les éléments qui devront interagir. L'extension des directives de placement, en particulier la directive `INDIRECT`, assure une solution dans ce cas.

11.4.2 Distributions irrégulières en Fortran

Nous décrivons ici de nouvelles propositions étendant les placements de données. Les extensions d'HPF-2 et les directives de placement de HPF+ [3] et Vienna Fortran [4, 16], ou de PST [12] seront présentées.

11.4.2.a Distribution sur des sous-ensembles de processeurs

Les applications scientifiques reposent sur plusieurs structures de données parallèles. Il apparaît indispensable de permettre la distribution des différentes structures parallèles de manière indépendante. En particulier de “petites” structures parallèles n'ont pas à être distribuées sur l'ensemble de la machine parallèle. Pour ce faire, Vienna Fortran propose la distribution de tableaux sur un sous-ensemble de la grille de processeurs virtuels définie par le programmeur. Cette possibilité est reprise par les extensions d'HPF-2 par exemple pour distribuer un tableau `A` sur les cinq premiers processeurs de l'arrangement de dix processeurs `P` :

```
!HPF$ PROCESSORS P(10)
      REAL A (100)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P(1:5)
```

11.4.2.b Généralisation de la distribution par bloc

Les placements fournis par HPF permettent de regrouper les données par blocs (éventuellement restreint à une donnée) et de distribuer ces blocs de manière cyclique sur les processeurs. Cependant, la taille de chacun des blocs est identique. De tels placements limitent les possibilités d'équilibrage des donnés pour des structures irrégulières rangées dans des structures de données régulières. Une distribution par blocs ne peut garantir une bonne répartition des données sur les processeurs (figure 11.7).

Les tableaux Vienna Fortran peuvent être associés à une distribution par blocs de tailles non constantes ; deux spécifications de ces tailles sont disponibles : 1–Le programmeur donne une liste des tailles des blocs (mot-clé `S_BLOCK`) ; 2–Le programmeur donne une liste des indices de début des blocs (mot-clé `B_BLOCK`). L'extension `GEN_BLOCK` d'HPF-2 reprend avec une petite variante sémantique la directive `S_BLOCK` de Vienna Fortran. La figure 11.8 illustre la distribution obtenue par le code Vienna Fortran :

```
PROCESSORS R(4,4)
INTEGER, DIMENSION(3), PARAMETER :: BB = (/ 10,50,90 /)
INTEGER, DIMENSION(3), PARAMETER :: SB = (/ 10,40,40 /)
REAL, DISTRIBUTED(B_BLOCK(BB),S_BLOCK(SB)) :: A(100,100)
```

Quelque soit la distribution par blocs adoptée, Vienna Fortran assure, pour des raisons d'efficacité, la continuité des blocs sur les processeurs virtuels. En PST, lors

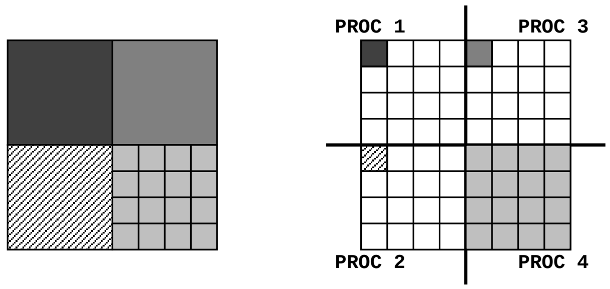


Figure 11.7 : Lors du placement de la structure de données irrégulières de la figure gauche sur un ensemble de 4 processeurs (figure de droite), la distribution par blocs de taille constante amène un déséquilibre.

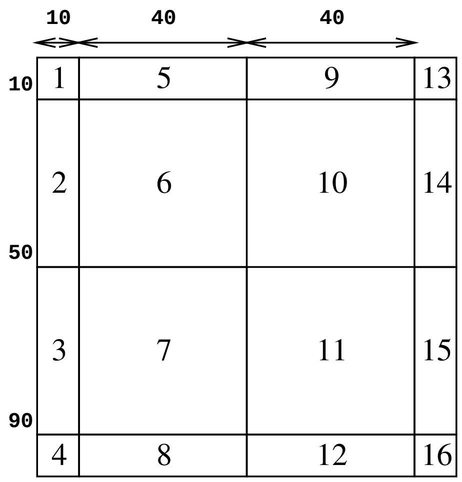


Figure 11.8 : Généralisation de la distribution par blocs en Vienna Fortran.
La figure représente la matrice `A(100,100)`. Chacun des sous-ensembles de valeurs de `A` est affecté au processeur dont le numéro est indiqué.

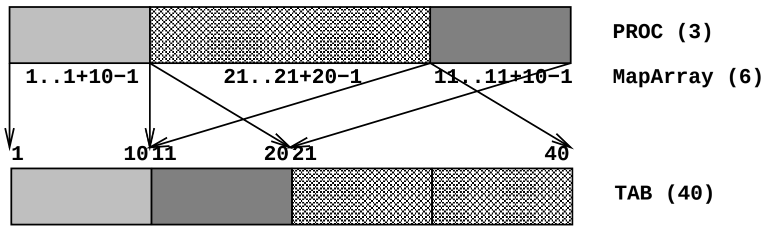


Figure 11.9 : Généralisation de la distribution par blocs en PST.
Chaque processeur `PROC` identifie par la donnée de deux valeurs du tableau `MapArray` la tranche du tableau `TAB` qui lui sera allouée.

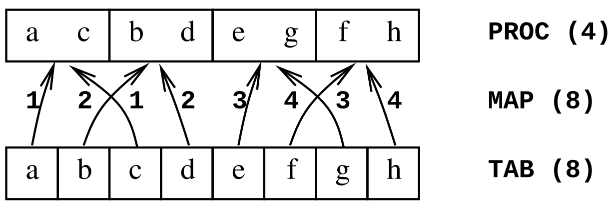


Figure 11.10 : Distribution indirect du tableau `TAB` sur les processeurs `PROC` en Vienna Fortran.

de l'utilisation de la distribution `BLOCK_GENERAL`, chacun des blocs est spécifié par deux valeurs : l'indice de base du bloc et la taille de celui-ci. La figure 11.9 illustre la déclaration PST du tableau `TAB` suivante :

```
      PARAMETER, INTEGER :: M = 3
!PST$ PROCESSORS PROC (M)
      INTEGER, DIMENSION (2*M) :: MapArray
      MapArray = (/1, 10, 21, 20, 11, 10 /)
      REAL, DIMENSION (40) :: TAB
!PST$ DISTRIBUTE TAB (BLOCK_GENERAL(MapArray)) ONTO PROC
```

11.4.2.c Distribution par tableau d'indirection

L'utilisation de la distribution indirecte de Vienna Fortran assure une distribution tout à fait générale des éléments d'un tableau sur les processeurs. Pour ce faire, on spécifie, via un tableau d'entiers, l'index du processeur sur lequel allouer chacun des éléments du tableau. L'exemple suivant est illustré par la figure 11.10 :

```
    PROCESSORS PROC(4)
    REAL TAB (8)
```

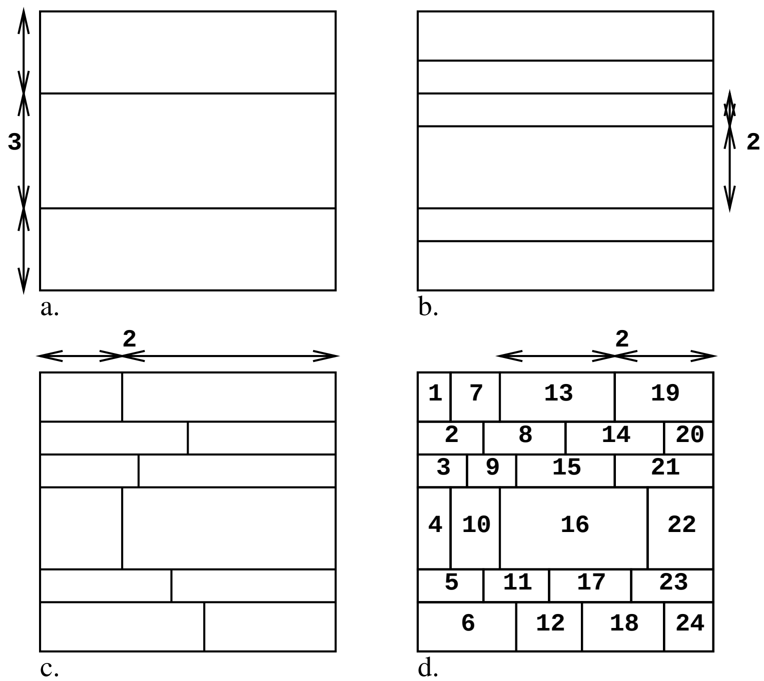


Figure 11.11 : Distribution par partionnement suivant une fonction en Vienna Fortran 90.

La fonction de mapping `MRD` réalise la distribution de `A` suivant une distribution calculée récursivement. Les paramètres de la fonction lui permettent de connaître les caractéristiques du tableau à distribuer et le nombre d'appels récursifs à réaliser. Cette fonction est définie par le programmeur.

```
INTEGER MAP (8)
MAP = (/ 1,2,1,2,3,4,3,4 /)
DISTRIBUTE TAB :: (INDIRECT(MAP)) TO PROC
```

Cette fonctionnalité est reprise par la directive `INDIRECT` de HPF-2.

11.4.2.d Fonction de distribution

L'utilisation de la distribution indirecte repose de manière générale sur un calcul dynamique du vecteur d'indices. Certains langages permettent de spécifier la distribution par une fonction de placement dont le résultat indiquera le placement à effectuer. C'est le cas des fonctions de mapping de Vienna Fortran 90 [2] illustrées par l'exemple suivant (figure 11.11) :

```
PROCESSORS R(6,4)
RECURSIVE MAPPING MRD(A,PF1,PF2)
```

```
...
END MAPPING MRD

DISTRIBUTED(MRD(A, (/ 3,2 /), (/ 2,2 /))) :: A(100,100)
```

De manière tout à fait générale, un algorithme de distribution dynamique peut être mis en place par l'appel à un "*mapper*" externe, tel le compilateur de communications sur la CM-2 [5] ou les primitives disponibles sur l'Intel iPSC [15] ou le Cray T3D [1].

11.5 Conclusion

La nouvelle norme Fortran 95 est un langage data-parallèle au sein duquel les strutures de données parallèles sont confondues avec les tableaux. La dernière proposition du Forum HPF, HPF-2, étend cette norme avec l'ajout de directives de placement des données exprimées par le programmeur pour faciliter l'obtention de bonnes performances des programmes sur les machines parallèles.

Dans un premier temps et afin d'assurer une disponibilité rapide des compilateurs HPF, les informations associées aux distributions des données doivent pouvoir être évaluées dès la compilation. HPF-2 définit ensuite des extensions qui autorisent des distributions irrégulières des données. Ces distributions irrégulières sont le moyen pour le programmeur d'exprimer l'irrégularité des structures de données représentées par les tableaux Fortran.

Les langages supports du parallélisme de données ne sont pas tous bâtis sur le modèle de Fortran. D'autres langages proposent des structures de données parallèles différentes des tableaux Fortran [10]. Dans ce cadre plus général, l'expression de la dynamicité et de l'irrégularité est aussi différente [6].

Bibliographie

[1] Barnard (S. T.). – PMRSB: Software to support unstructured applications on the T3D. *In : 1st European Cray T3D Workshop.* – Lausanne, Switzerland, Sept. 1995.

[2] Benkner (S.). – *Vienna Fortran 90 and its Compilation.* – Thèse de PhD, University of Vienna, Sept. 1994.

[3] Chapman (B.), Zima (H.) et Mehrotra (P.). – Extending HPF for advanced data-parallel applications. *IEEE Parallel and Distributed Technology*, vol. 2, n° 3, Fall 1994, pp. 59–70.

[4] Chapman (B. M.), Mehrotra (P.) et Zima (H.). – Programing in Vienna Fortran. *Scientific Programming*, vol. 1, n° 1, Fall 1992, pp. 31–50.

[5] Dahl (E. D.). – Mapping and compiled communication on the Connection Machine system. *In : Proc. of the Fifth Distributed Memory Computing Conf.*, pp. 756–766.

[6] Dekeyser (J.-L.) et Marquet (P.). – Supporting irregular and dynamic computations in data-parallel languages. *In : The Data Parallel Programming Model.* pp. 197–219. – Lectures Notes in Computer Science, Tutorial Serie, vol. 1132.

[7] Fox (G.), Hiranandani (S.), Kennedy (K.), Koelbel (C.), Kremer (U.), Tseng (C.-W.) et Wu (M.-Y.). – *FORTRAN D Language Specification.* – Research Report n° TR90-141, Dept. of Computer Science, Rice University, Avr. 1991.

[8] High Performance Fortran Forum. – High Performance Fortran language specification. *Scientific Programming*, vol. 2, n° 1-2, 1993, pp. 1–170.

[9] High Performance Fortran Forum. – High Performance Fortran language specification, version 2.0. – Rice University, Houston, TX, Jan. 1997.

[10] Marquet (P.). – Langages et expression du parallélisme de données. *Technique et Science Informatiques*, vol. 12, n° 6, 1993, pp. 685–714.

[11] MasPar Computer Corp., Sunnyvale, CA. – *MasPar Fortran — Reference Manual, Software Version 1.0*, Mars 1991. Doc. 9303-0000, Rev. A1.

[12] Müller (A.) et Rühl (R.). – Extending High Performance Fortran for the support of unstructured computations. *In : Proc. Int'l Conf. on Supercomputing.* – Barcelona, Spain, Juil. 1995.

[13] Pase (D. M.), MacDonald (T.) et Meltzer (A.). – *MPP FORTRAN Programming Model.* – Rapport technique, Eagan, Minnesota, Cray Research Inc., Jan. 1992.

[14] Thinking Machines Corporation, Cambridge, MA. – *Getting Started in CM FORTRAN*, Fév. 1990. Version 5.2-0.6.

[15] Williams (R.). – *DIME: A Distributed Irregular Mesh Environemnt.* – Rapport technique, Caltech Concurrent Supercomputing Facilities, 1990.

[16] Zima (H.), Brezany (P.), Chapman (B.), Mehrotra (P.) et Schwald (A.). – *Vienna Fortran—A Language Specification, Version 1.1.* – Rapport technique n° TR 92-4, ACPC, University of Vienna, 1992.

Chapitre 12

Conception par objet et mise en œuvre d'applications irrégulières

Jean-Marc Jézéquel (Irisa/CNRS)

12.1 Introduction

De nombreux programmeurs sont intéressés par la puissance de calcul offerte par les architectures multi-processeurs mais restent réticents à l'idée de transférer leurs applications vers ces dernières. En effet, les outils logiciels associés à ces machines manquent cruellement de maturité et les méthodes et environnements de programmation adaptés aux machines séquentielles s'avèrent peu appropriés à la gestion du parallélisme. Pour bien exploiter ces architectures parallèles, le programmeur doit ainsi posséder une connaissance approfondie de l'algorithmique parallèle et des spécificités de l'architecture utilisée.

L'objectif de cette présentation est de montrer comment une approche résolument orientée objet et fondée sur l'utilisation de patrons de conceptions spécifiques permet d'exprimer des traitements à un niveau où l'irrégularité des structures sous-jacentes est abstraite; ce qui permet de les exécuter optimalement tant sur des architectures centralisées que parallèles à mémoire (virtuellement) partagée ou distribuée, ou même mixtes (méta-computing). Cette présentation s'appuyera sur l'exemple de l'environnement EPEE, qui offre un cadre de conception, basé sur le langage Eiffel, pour développer des composants logiciels réutilisables pour applications traitant de grandes quantités d'événements, de données ou de calculs. On montrera comment appliquer ces idées sur une étude de cs simple : le problème des N corps.

12.2 Approche par objets et parallélisme et répartition

12.2.1 Approches classiques

Si la construction de systèmes complexes à l'aide des technologies objets commence à être assez bien maîtrisée, en revanche le bât blesse encore pour tout ce qui touche au parallélisme et à la distribution. En effet, comme l'avait écrit dès 1993 B. Meyer dans [18],

> *To judge by the looks of the two parties, the marriage between concurrent computation and object-oriented programming appears an easy enough affair to arrange. This appearance is deceptive: the problem is a hard one.*

En effet, dans les langages à objets, un objet est une unité de structuration de programme encapsulant données et méthodes (procédures et fonctions) travaillant sur ces données. En utilisant la terminologie Smalltalk, on dit que des objets communiquent par envoi de messages. On voit donc apparaître un certain nombre de points communs avec la notion de processus. La tentation est donc forte d'intégrer ces deux notions dans celle d'objet actif. POOL-T [1] est certainement un des premiers exemples significatifs illustrant cette approche : une fois créé, un objet peut continuer à être actif après avoir rendu la main à son créateur. La communication entre objets est ici de type Remote Procedure Call (appel de procédure à distance), et le programmeur contrôle alors explicitement le parallélisme et l'accès aux données.

Un autre moyen pour introduire du parallélisme dans les langages à objets consiste à rendre asynchrones les appels aux méthodes des objets : un objet appelant la méthode d'un autre objet peut continuer son activité en parallèle avec l'objet exécutant la méthode appelée. Ceci est implanté par exemple dans ABCL/1 [21], ELLIE [2], Eiffel// [4], ou encore ConcurrentSmalltalk [20].

Ces deux idées d'objet actif et d'appels de procédure asynchrones sont utilisées pour créer des LAO parallèles soit en intégrant ce parallélisme dès la phase de conception d'un nouveau langage de programmation concurrent (ELLIE, POOL-T, ABCL/1, ou plus récemment Java), ou plus simplement en étendant des langages séquentiels déjà existants pour leur permettre de gérer le parallélisme, comme dans DistributedSmalltalk [3] qui étend Smalltalk ou COOL [5] qui étend C++, ou encore dans CORBA, qui fourni les mécanismes par lesquels des objets (implantés dans différents langages) peuvent émettre des requêtes et recevoir des réponses de façon transparente.

12.2.2 Problématique

Le modèle de parallélisme sous-jacent à ces langages (fondamentalement les processus séquentiels communicants de Hoare [9]) pose de nombreux problèmes liés à sa nature, à la fois très primitif et très (trop) général.

Tout d'abord, il se concilie difficilement avec le mécanisme d'héritage inhérent aux langages à objets, en particulier en ce qui concerne les contraintes de synchronisation. Ce problème, connu sous le nom d'*anomalie d'héritage* [16] ne semble pas pouvoir être facilement contournable sans restreindre l'expressivité du modèle [17]. En effet ce problème est étroitement lié à la nécessité d'imposer des contraintes sur l'acceptation d'un message par un objet (exemple du tampon borné, dans [17]). Dans la plupart des langages à objets prenant en compte la concurrence, ceci est réalisé par du code de synchronisation contrôlant l'acceptation d'un message en fonction de l'état de l'objet récepteur. Ces codes de synchronisation sont difficiles à hériter et demandent généralement une redéfinition *in extenso*. L'anomalie d'héritage [16] est un problème causé par la présence de ce code de synchronisation, qu'il faut donc chercher à éliminer. Une approche possible consiste en l'intégration étroite d'une sémantique particulière du parallélisme dans un langage. C'est l'exemple de Parallel Eiffel [18], où il n'y a plus de distinction entre objets “actifs” et “passifs”, mais seulement la possibilité qu'un objet soit “séparé” (*separate*), c'est à dire traité par un autre processeur (réel ou virtuel). Cependant Parallel Eiffel s'appuie sur un mécanisme de multi rendez-vous réparti, qui est bien entendu suffisamment puissant pour permettre d'apporter des solutions simples, concises et élégantes aux problèmes classiques de concurrence (e.g. le problème dit *des philosophes*), mais qui limite drastiquement le domaine d'intérêt de l'approche.

D'autre part, comme les communications (et les synchronisations) entre processus sont entièrement sous la responsabilité du programmeur, celui-ci est confronté de plein fouet à la complexité des systèmes répartis : interblocages, absence d'état global observable rendant difficile le débogage réparti ou même simplement la détection de la terminaison, etc.

12.2.3 Entre langages et systèmes, la notion de “framework”

S'il n'est donc pas raisonnable, d'un point de vue génie logiciel, de programmer “à la main” (c'est à dire à un niveau sémantique aussi primitif que celui des processus communicants) des systèmes “critiques” de grande taille, subissant ou utilisant la répartition, il paraît tout aussi difficile, et surtout peu rentable, de rechercher un langage universel prenant en compte tous les aspects possibles de leur programmation. Ne serait-ce qu'en termes de sémantique de la communication entre processus (rendez-vous, files, RPC, hypothèses de fiabilité, d'ordonnancement, d'isochronisme, etc.) quelque soit le choix effectué, il serait forcément fermé (puisque intégré dans un langage), donc trop limitatif pour certaines applications, et par essence inapproprié pour des structures de donnée irrégulières.

Il parait en revanche beaucoup plus prometteur de définir (à l'aide de la technologie objet, qui est particulièrement bien adaptée pour cela) des *modèles* spécifiques à des domaines d'applications particuliers, comme par exemple le modèle d'exécution SPMD associé à la distribution de données pour le calcul scientifique intensif; et de fournir des cadres de conception, de réalisation et de validation adaptés à ces modèles : c'est la notion de "framework".

Un *framework* fournit un ensemble intégré (tout en restant ouvert et extensible) de fonctionnalités spécifiques à un domaine. Il consiste en une collection de classes liées entre elles par de multiples schémas (*patterns*) de collaboration statiques et dynamiques [6]. Il fournit un modèle d'interaction entre les différents objets instances des classes définies (ou seulement spécifiées pour les classes abstraites) dans le framework. Celui-ci présente en général une inversion du contrôle à l'exécution : alors qu'une application utilisant une bibliothèque s'appuie sur celle ci, dans le cas d'une application utilisant un framework, c'est le framework qui effectue l'essentiel du travail et appelle "de temps en temps" un composant spécifique réalisé par l'implanteur de l'application. Un framework peut donc être vu comme une application semi-complète. Des applications complètes sont développées en héritant et en instantiant des composants paramétrés de frameworks. Il suffit donc en quelque sorte d'enficher dans un framework les composants spécifiques de son application pour obtenir une application complète.

Nous explorons nous-mêmes depuis quelques années certains de ces problèmes, avec des domaines d'application variés, avec en particulier le calcul scientifique intensif [8, 11, 13, 15]. L'objectif était d'étudier, concevoir et valider des modèles et des méthodes de construction de logiciels pour architectures réparties par composition de composants logiciels dans un contexte de programmation par objets [10].

12.3 Application à la mise en oeuvre d'applications irrégulières

12.3.1 Le cadre général de EPEE

Notre approche s'appuie sur le principe de l'encapsulation des aspects liés au parallélisme dans des classes d'un langage à objets séquentiel. Pour cela, nous avons développé l'environnement EPEE (Environnement Parallèle d'Exécution de Eiffel), adoptant un modèle d'exécution SPMD qui permet de décomposer naturellement une application en un entrelacement de phases séquentielles et de phases parallèles. Ces dernières étaient initialement obtenues par réutilisation de composants logiciels encapsulant un parallélisme de données.

La solution que nous proposons, et que nous continuons d'expérimenter sur des problèmes concrets, dans l'environnement EPEE s'appuie sur des travaux à trois niveaux :

Exécutif [14] : L'introduction d'objets partagés dans un langage muni d'un ramasse miette au sein de son exécutif nous a demandé de modifier son comportement afin de disposer du ramasse miettes pour les objets partagés de la même façon que pour les objets alloués dans l'espace d'adressage local de chaque processus.

Boite à outils [19] : La migration du code séquentiel vers du code réparti se fait à l'aide de composants réutilisables intégrant la notion de distribution de données et de contrôle, tout en la masquant par une interface identique à leurs versions séquentielles.

Patrons de conception [15, 12] : Les patrons de conception, ou *design patterns*, introduits dans [6], sont des modèles de solutions permettant de résoudre des problèmes de conception particuliers, notamment dans un contexte de conception par objets de logiciels. Les *design patterns* permettent l'identification et la réutilisation de micro-architectures logicielles, qui sont décrites tant du point de vue statique que dynamique sous la forme d'un ensemble de classes et de leurs relations structurelles et contextuelles (i.e. leurs collaborations). Dans le cadre du parallélisme de données, auquel nous nous attachons ici, nous avons proposé le patron de conception des OPÉRATEURS permettant de construire des bibliothèques de composants qui soient à la fois réutilisables, extensibles et permettant une exécution parallèle (voir ci-dessous).

À travers cette maquette EPEE, nous avons montré l'intérêt de notre approche pour faciliter la programmation d'architectures parallèles réparties : elle permet de présenter un modèle de programmation séquentiel au programmeur d'application, qui ne voit alors la machine parallèle que comme un processeur de calcul plus puissant. Ce modèle s'adapte bien à l'expression d'un parallélisme massif pourvu que les problèmes à résoudre soient de taille assez grande. Nous utilisons maintenant EPEE surtout comme une plate-forme de réflexion et d'intégration pour mener nos recherches sur la notion de modèles de conception s'abstrayant des particularités de l'architecture répartie sous-jacente.

12.3.2 Les opérateurs parallèles

L'adoption de technologies objets dans la communauté de la programmation parallèle doit permettre à cette dernière de bénéficier des qualités de la conception orientée objet, en particulier la réutilisabilité et la facilité d'extension. Cependant, une approche traditionnelle de la conception des librairies parallèles ne permet pas de satisfaire pleinement ces objectifs. En effet, un principe fréquemment utilisé en conception orientée objet est celui de l'encapsulation qui préconise que des méthodes manipulant des données doivent être encapsulées avec elles dans des classes. Par exemple, lors de la conception d'une classe MATRICE, il est couramment admis qu'une opération comme le produit de matrices doit être définie dans l'interface de la classe MATRICE.

Utilisant ce principe, il est par exemple possible, en utilisant le polymorphisme entre différentes implantations (matrices denses ou creuses, ...) de la classe MATRICE et la liaison dynamique de la méthode *produit()*, réalisant le produit de matrices, de créer des bibliothèques complètes et optimisées, présentant des méthodes adaptées à chaque type d'implantation. Ainsi, la librairie d'algèbre linéaire répartie Paladin [7], développée dans le cadre de l'expérimentation du projet EPEE, a été conçue selon ce principe. Cependant, l'utilisation de cette approche pose des problèmes d'extensibilité (cf. [15]). Ces problèmes vont se poser en particulier lorsque l'utilisateur d'une telle bibliothèque veut ajouter une nouvelle implantation concrète, adaptée à ses besoins, de la classe MATRICE. En effet seules les méthodes génériques seront utilisables lorsque cette nouvelle classe sera utilisée comme argument des méthodes des classes déjà définies.

Pour résoudre ce problème d'extensibilité, il importe de séparer de la classe qui sert de "conteneur" toutes les opérations qui ne dépendent pas de l'implantation de de cette dernière. Ainsi, une méthode *produit* n'a aucun besoin de connaître la façon dont est implantée une matrice du moment qu'elle dispose d'un moyen de lire ses éléments. Les seules opérations qu'une classe conteneur devrait en fait fournir sont celles relatives à la gestion de la structure de données utilisée pour stocker ses éléments. La solution proposée dans [15] est d'opérer une séparation entre les classes liées au domaine de l'application et celles liées au domaine de l'implantation. Pour cela, il importe que les opérations extrinsèques appliquées aux structures de données soient *réifiées*, c'est à dire définies comme des classes propres et non plus comme des méthodes des agrégats manipulés. L'utilisation du *design pattern* de l'opérateur permet de réaliser cette séparation entre les deux domaines cités précédemment.

Ainsi, le *design pattern* de l'opérateur fait intervenir quatre entités de base, deux dépendant du domaine de l'application : les OPÉRATEURs et les ÉLÉMENTs; et deux autres du domaine de l'implantation: les CONTENEURS et les ITÉRATEURs. Les rôles de ces quatre abstractions sont les suivants :

Opérateur : Un OPÉRATEUR représente l'opération régulière appliquée à un agrégats d'ÉLÉMENTs.

Élément : Les ÉLÉMENTs sont les abstractions manipulées par l'application. Ils sont la cible des OPÉRATEURs.

Conteneur : Les CONTENEURs sont chargés de stocker, de modifier et de fournir les ÉLÉMENTs stockés dans une structure de données particulière.

Itérateur : Un ITÉRATEUR permet aux OPÉRATEURS de parcourir les ÉLÉMENTs stockés dans les CONTENEURs. Le rôle de l'ITÉRATEUR est de masquer la structure concrète des agrégats ainsi que le domaine d'itération. Certains ITÉRATEURs peuvent également être capable de générer eux-mêmes les éléments qu'ils fournissent sans être attachés à un conteneur (e.g. générateur pseudo aléatoire).

Ainsi, un OPÉRATEUR accède aux éléments stockés dans un CONTENEUR par l'intermédiaire d'un ITÉRATEUR. Il est possible pour un OPÉRATEUR, en utilisant le polymorphisme de son ITÉRATEUR, de parcourir un CONTENEUR de différentes façons et d'accéder à ses ÉLÉMENTS indépendemment de la structure interne du CONTENEUR.

Du fait du découplage opéré entre entre les abstractions du domaine de l'application et celles du domaines de l'implantation, la distribution des données va se faire de façon transparente vis-à-vis de l'utilisateur. Ainsi, les applications ne vont pas dépendre d'une architecture particulière et vont être facilement extensibles ou modifiables, toutes les fonctions de gestion du parallélisme (synchronisations, etc.) étant intégrées dans les méthodes d'accès aux données réparties ou partagées.

La migration vers le parallélisme de données d'applications séquentielles utilisant le *design pattern* de l'opérateur va donc se faire simplement en remplaçant les composants séquentiels par des composants polymorphiques incluant la gestion du parallélisme. Ainsi, les opérateurs utilisés sont changés en opérateurs parallèles héritant à la fois des opérateurs concrets et d'un opérateur parallèle générique. De même, le PROVIDER est transformé en PARALLEL PROVIDER adapté à la distribution des données utilisée.

L'utilisation du schéma de conception de l'opérateur dans le cadre d'EPEE permet donc de paralléliser des applications existantes de façon simple et efficace, sans l'utilisation de compilateur spécialisé ou de macro-instructions mais uniquement en tirant en tirant parti des propriétés des langages à objets telles que l'héritage multiple, le polymorphisme ou la liaison dynamique.

12.3.3 Les conteneurs hiérarchiques et le Méta-Computing

En collaboration avec l'Université de Tokyo (Naohito Sato et les Pr. Yonezawa et Matsuoka), nous avons proposé de nouvelles abstractions permettant d'assurer un découplage entre la distribution des données et le parallélisme [13]. Partant de l'observation qu'un calcul basé sur un modèle d'exécution SPMD peut être considéré comme l'application ordonnée d'une fonction (ou opérateur) sur une collection d'éléments, l'idée est d'abstraire les partitions de données et l'accès à ces données par l'utilisation de structures de données hiérarchiques, *hierarchical containers*, et d'itérateurs composables, *composable traversers*.

L'introduction des collections hiérarchiques doit permettre d'étendre de façon orthogonale le parallélisme et la distribution afin de pouvoir utiliser facilement et efficacement des schémas de distribution et de parallélisme complexes sans modification du code du client. Les collections hiérarchiques possèdent des relations d'ordre partiel entre les objets situés à un même niveau hiérarchique et des relations d'appartenance entre des collections adjacentes dans la hiérarchie. Les calculs sur des collections hiérarchiques sont effectués à l'aide d'itérateurs qui parcourent les éléments en respectant les relations d'ordre ainsi définies, les objets n'étant pas ordonnés peuvent être traités en parallèle.

Formellement, une collection hiérarchique est définie par l'application sur les éléments d'une partition, notée Π, d'une collection E d'un constructeur C qui groupe ces éléments pour créer un nouveau degré (plus élevé) dans la hiérarchie. Notons $\widehat{C}$, le constructeur composé qui applique C à chaque partition, définie par Π, de E ; une collection hiérarchique peut alors être définie par des applications successives de $\widehat{C}$ à partir d'une collection atomique E^0:

$$E^0 \rightarrow \widehat{C_1} E^1 \rightarrow \widehat{C_2} \cdots$$

De façon similaire, on peut décomposer les itérateurs parcourant les structures de données de l'application à l'aide d'itérateurs de base. A partir de la définition d'une librairie d'itérateurs simples, il devient alors possible en les composant de parcourir de diverses manières les différentes couches des collections hiérarchiques tout en satisfaisant les contraintes de distribution et en maximisant le parallélisme.

Ainsi, selon ces principes, un calcul basé sur l'application d'un opérateur sur une collection d'éléments se fait en décomposant de façon hiérarchique cette dernière afin de séparer parallélisme et distribution et en effectuant une composition d'itérateurs élémentaires pour pouvoir s'adapter à cette configuration particulière.

Les abstractions présentées dans cette section ont été implantées dans le cadre d'EPEE en étendant les notions de *conteneurs* et d' *itérateurs* avec les composants suivants:

Composable Traverser Cette classe permet d'implanter les principes présentés dans le paragraphe précédent.

Partially Ordered Provider Cette extension des *itérateurs* bénéficie des propriétés de composition héritées de la classe *Composable Traverser*.

Hierarchical Container Ce type de conteneur permet de réaliser les translations d'index entre les différents niveaux de la hiérarchie des collections.

Distributed_Container Ce type de conteneur permet d'encapsuler les les détails du modèle de mémoire (mécanisme de gestion du parallélisme et de la distribution, synchronisations, etc.).

L'exécution parallèle du programme est gérée par par l'abstraction *PO_Provider* qui s'occupe des dépendances de données imposées par l'ordre partiel défini sur les éléments à traiter en utilisant les possibilités de synchronisation et de communication offertes par les *Distributed Containers*. Du point de vue de l'utilisateur, l'exécution parallèle du programme est cachée grâce à l'utilisation de composants encapsulant le parallélisme. Le programmeur n'a qu'à manipuler des abstractions de parallélisme. La conception d'un programme parallèle s'effectue alors en trois étapes:

1. définition des éléments et des opérateurs agissant sur ces éléments
2. spécification de la distribution des données à l'aide de collections hiérarchiques
3. compositions d'itérateurs afin de prendre en compte à la fois les dépendances de données et leur distribution physique

Cette méthode permet ainsi au programmeur de définir facilement des schémas relativement complexes de distribution. Ce dernier n'a en effet qu'à choisir les collections et les itérateurs de base définis dans l'extension d'EPEE et de les composer afin de les adapter à son problème. Une application envisagée des collections hiérarchiques de données est le méta-computing, c'est à dire l'utilisation combinée de ressources de calculs hétérogènes (réseaux de stations de travail, calculateurs parallèles, etc.) couplés avec des réseaux à très haut débit. Une première étape concerne la programmation d'architectures à plusieurs niveaux mémoires. Un exemple d'une telle architecture est le *Power Challenge Array* de Silicon Graphics, constitué de l'interconnexion à travers un commutateur HiPPI de système multi-processeurs à mémoire partagée *Power Challenge*. On dispose donc de deux niveaux de mémoire :

- répartie entre les nœuds constitués du système Power Challenge
- et partagée à l'intérieur d'un nœud.

Un autre exemple de ce type d'architecture est la connexion à l'aide d'un réseau Myrinet de serveurs Sparc (ou PC) multi-processeurs. Le réseau Myrinet est un réseau à haut débit (1.28 Gigabits/s), particulièrement bien adapté à l'interconnexion de machines parallèles.

Les collections hiérarchiques ainsi que les itérateurs qui leur sont associés présentent l'avantage de pouvoir refléter cette hiérarchie mémoire et sont donc bien adaptés à la programmation de telles architectures, comme le montrent différentes études de cas que nous avons menées.

12.4 Exemple d'utilisation : le problème des N corps

Nous présentons dans cette section et la suivante un exemple d'application des idées développées dans ce rapport à travers l'étude de la parallélisation du problème des n-corps. Nous considérons dans cet exemple un ensemble de masses situées dans univers tridimensionnel et qui interagissent entre elles. Chaque masse possède une position ainsi qu'une vitesse et une accélération.

Le but du problème est alors de calculer l'évolution dans le temps de cet ensemble de masses. La simulation est divisée en intervalles de temps. Le calcul de l'évolution pour chaque intervalle se fait en deux étapes. La contribution de chaque masse sur chacun des autres corps est tout d'abord calculée. Ensuite, la position et la vitesse de chaque corps est mise à jour, en tenant compte de toutes les interactions.

12.4.1 Modélisation du problème

L'univers est défini comme une abstraction capable de stocker des corps ainsi que de fournir des accesseurs de base permettant de les accéder en lecture et écriture. Ces spécifications sont déclarées comme différées dans la classes abstraite UNIVERSE et sont implémentées par la collection instanciable utilisée pour le stockage des corps. La classe abstraite UNIVERSE encapsule également des opérateurs agissant sur tout l'univers, comme celui permettant d'initialiser l'universe de manière aléatoire. Comme le préconise le patron de conception de l'opérateur, ces opérateurs sont inclus sous la forme de *Factory Methods* (voir [6]), afin de pouvoir être rédéfinis facilement.

Les univers instantiables sont alors construits en héritant à la fois de cet univers ainsi que d'une structure de données locale, distribué ou partagée permettant de stocker les corps.

12.4.2 La mise à jour des corps

La classe UPDATOR est responsable de la mise à jour des paramètres des corps. Cette mise à jour se fait en parcourant l'ensemble des corps et en invoquant la méthode *update* sur chacun d'entre eux. La classe UPDATOR hérite donc de l'opérateur FORALL, la mise à jour constituant la méthode *forall_operation*.

12.4.3 Le calcul des contributions

Le calcul des interactions entre les différents corps se fait en deux étapes. Tout d'abord, nous calculons la contribution d'une simple masse sur l'ensemble des autres corps. Ce premier calcul est effectué par l'opérateur BODY_CONTRIBUTOR. Ce dernier est ainsi défini comme un opeŕateur de type FORALL qui ajoute la contribution de la masse fournie à tous les corps fourni par son itérateur ; cette opération étant effectuée grâce à la méthode *add_the_contribution_of* de la classe Body.

La deuxième étape consiste à calculer la contribution d'un ensemble de masses sur un ensemble de corps. Cette opération est effectuée par un deuxième opérateur : Univ_Contributor, héritant également de Forall, et qui calcule la contribution de chaque masse fournie par son itérateur sur chaque corps d'un univers donné. Pour chaque masse, l'opérateur Univ_Contributor utilise donc un Body_Contributor dans sa méthode *Forall_operation*, comme le montre la figure 12.1

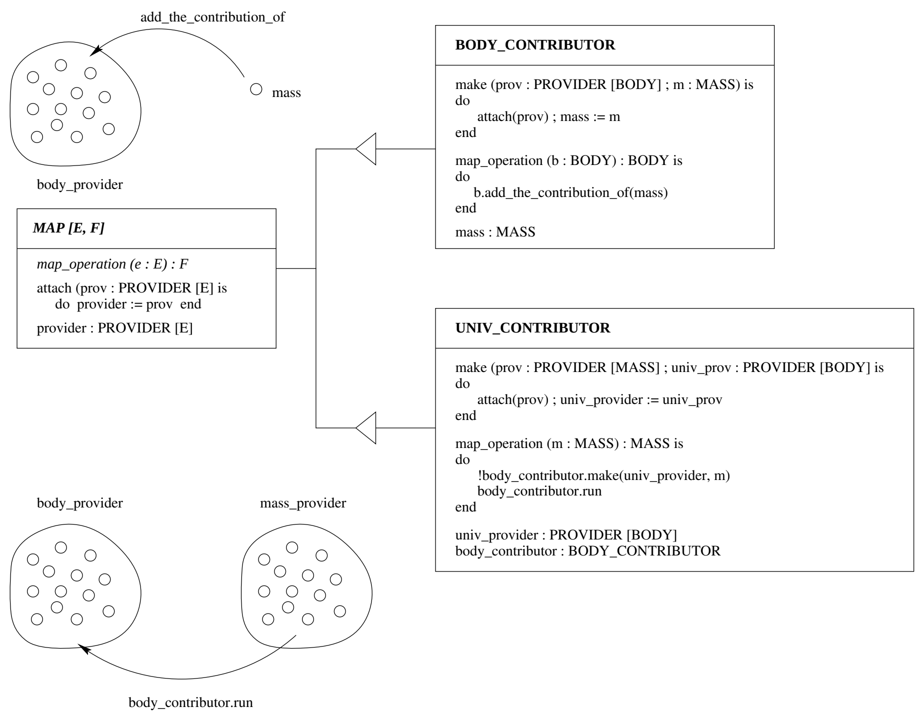


Figure 12.1 : Les opérateurs utilisés pour le calcul des contributions

12.4.4 Simulation

Tous les composants nécessaires à la simulation ont maintenant été décrits, il ne reste plus qu'a les assembler et à les initialiser, ce qui est le rôle de la classe Simulator. Pour utiliser les opérateurs de la simulation, nous avons besoin de deux itérateurs. Le premier itérateur fournit l'ensemble des masses contenues dans l'univers et est utilisé par la classe Univ_Contributor. Le second fournit les corps de l'univers et sert aux classes Body_Contributor et Updator. Comme la classe Body hérite de Mass, nous pouvons grâce au polymorphisme initialiser ces deux itérateurs de la même manière, en utilisant l'accesseur *items* de l'univers.

12.5 Parallélisation de la simulation

Nous allons maintenant décrire comment il est possible, en utilisant les abstractions de parallélismes décrites dans le chapitre précédent, de paralléliser la simulation des n-corps sans avoir à réécrire les collections et les opérateurs séquentiels.

12.5.1 L'univers en mémoire multi-niveaux

La première étape dans la parallélisation de la simulation consiste en la création d'un univers non plus local mais stocké en mémoire multi-niveaux. La définition d'un tel univers ML_UNIVERS se fait simplement en héritant de la classe UNIVERS, qui représente la spécification abstraite de l'univers, ainsi que d'une structure de stockage en mémoire multi-niveaux.

Les corps étant crées en mémoire partagée, l'univers ne contient plus des objets de type BODY mais de type SHD_BODY. Comme la classe SHD_BODY hérite de BODY, il n'y a pas besoin, grâce au polymorphisme, de redéfinir les opérations effectuées sur des objets de type BODY.

12.5.2 Parallélisation de la simulation

Une fois la distribution de l'univers achevée, la parallélisation de la simulation des n-corps se fait en deux étapes.

La première étape consiste à définir de nouveaux opérateurs parallèles qui remplaceront ceux séquentiels. Dans le cas de la simulation des n-corps, les opérateurs étant relativement simples leur parallélisation se fait tout simplement en héritant à la fois de l'opérateur séquentiel et de la classe abstraite PARALLEL_OPERATOR.

La deuxième étape concerne l'utilisation des ces opérateurs. Afin de fonctionner de manière parallèle, il faut en effet les utiliser avec les itérateurs appropriés. Cette utilisation dépend ainsi du fait que les opérateurs font intervenir des éléments locaux ou distants. Nous allons donc examiner comment s'effectue la parallélisation des différents opérateurs de la simulation.

12.5.2.a Parallélisation de la mise à jour des corps

Cet opérateur est chargé de mettre à jour les corps stockés dans l'univers. Lors d'une exécution parallèle, chaque processeur devra alors uniquement mettre à jour ses éléments locaux. La définition de la classe PARALLEL_UPDATOR se fait simplement par héritage de l'opérateur séquentiel UPDATOR et de l'abstraction PARALLEL_OPERATOR qui prend en charge les problèmes de synchronisation.

L'utilisation de l'opérateur UPDATOR se fait alors simplement en remplaçant l'ancienne phase d'initialisation :

!!updator.make(universe.items, slice)

par :

!!updator.make(universe.local_items, slice)

Les opérations de mise à jour définies dans l'opérateur séquentiel ne se font ainsi sur les corps dont le processeur est responsable et non plus sur l'ensemble des corps stockés dans l'univers.

12.5.2.b Parallélisation du calcul des contributions

Dans le calcul des interactions entre les corps, seul l'opérateur UNIV_CONTRIBUTOR nécessite d'être parallélisé. En effet, l'opérateur BODY_CONTRIBUTOR n'est utilisé que comme opération effectuée par la méthode *forall_operation* de BODY_CONTRIBUTOR et est donc utilisé de façon séquentielle par chaque processeur.

La version parallèle de UNIV_CONTRIBUTOR est définie uniquement par héritage de la version séquentielle ainsi que de l'abstraction PARALLEL_OPERATOR. La figure 12.5.2.b montre les relations entre les différentes classes.

Lors de son initialisation, la classe UNIV_CONTRIBUTOR nécessite deux itérateurs. Il calcule alors la contribution de l'ensemble des masses fournies par le premier itérateur sur l'ensemble des corps fournis par le second itérateur. Pour chaque processeur, le UNIV_CONTRIBUTOR parallèle calcule toujours la contribution de tout l'univers. En revanche, il ne calcule cette contribution que sur un sous-ensemble de l'univers. Le parallélisme provient ainsi du fait que pour chaque processeur, on effectue le calcul de la contribution de l'ensemble de l'univers uniquement sur les corps appartenant à ce processeur, comme le montre la figure 12.5.2.b. L'initialisation du PARALLEL_UNIV_CONTRIBUTOR se fait donc en remplaçant la phase d'initialisation séquentielle :

!!univ_contributor.make(universe.items, universe.items)

par :

!!univ_contributor.make(universe.items, universe.local_items)

L'itérateur sur l'ensemble des masses est donc inchangé et parcourt l'ensemble de l'univers tandis que l'itérateur sur les corps retourne uniquement les corps dont le processeur est responsable.

12.5.3 La version parallèle de la simulation des n-corps

Une fois que tous les composants parallèles décrits dans les paragraphes précédents ont été écrits, la parallélisation de la simulation se fait uniquement en remplaçant les collections et opérateurs séquentiels par leurs équivalents parallèles. Les modifications sont ainsi minimes et la majeur partie du code de la simulation est réutilisée.

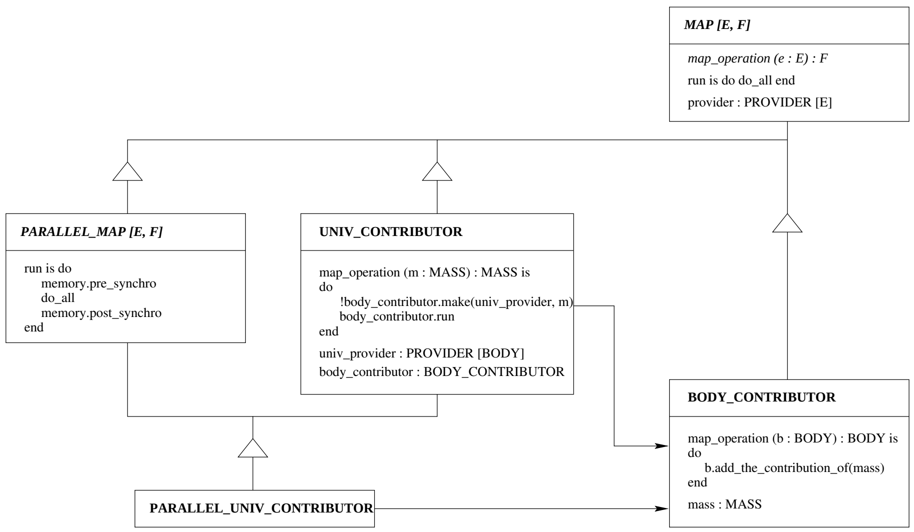


Figure 12.2 : Schéma de parallélisation de la contribution

12.6 Conclusion

Nous avons montré comment une approche résolument orientée objet et fondée sur l'utilisation de patrons de conceptions spécifiques permet d'exprimer des traitements à un niveau où l'irrégularité des structures sous-jacentes est abstraite; ce qui permet de les exécuter optimalement tant sur des architectures centralisées que parallèles à mémoire (virtuellement) partagée ou distribuée, ou même mixtes (méta-computing). Cette présentation s'est appuyée sur l'exemple de l'environnement EPEE, qui offre un cadre de conception, basé sur le langage Eiffel, pour développer des composants logiciels réutilisables pour applications traitant de grandes quantités d'événements, de données ou de calculs.

Remerciements

L'étude de cas des N corps a été développée et rédigée par Thomas Leseney pendant son stage de DEA (Ifsic, 1997).

Bibliographie

[1] America (P.). – Pool-T: A parallel object-oriented programming. *In: Object-Oriented Concurrent Programming*, éd. par Yonezawa (A.). pp. 199–220. – The MIT Press, 1987.

[2] Andersen (B.). – Ellie language definition report. *Sigplan Notices*, vol. 25, n° 11, November 1990, pp. 45–64.

[3] Bennett (J. K.). – The design and implementation of DistributedSmalltalk. *In : OOPSLA'87 Proceedings*. – October 1987.

[4] Caromel (D.). – Towards a method of object-oriented concurrent programming. *Communications of the ACM*, vol. 36, n° 9, September 1993, pp. 90–102.

[5] Chandra (R.), Gupta (A.) et Hennessy (J. L.). – Cool: a language for parallel programming. *In : Languages and Compilers for Parallel Computing*, éd. par et al. (D. G.). – MIT Press, 1990.

[6] Gamma (E.), Helm (R.), Johnson (R.) et Vlissides (J.). – *Design Patterns: Elements of Reusable Object-Oriented Software*. – Addison Wesley, 1994.

[7] Guidec (F.). – *Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'algèbre linéaire*. – Thèse de doctorat, IFSIC / Université de Rennes 1, juin 1995.

[8] Guidec (F.), Jézéquel (J.-M.) et Pacherie (J.-L.). – An object oriented framework for supercomputing. *Journal of Systems and Software*, vol. Special Issue on *Software Engineering for Distributed Computing*, Juin 1996.

[9] Hoare (C. A. R.). – Communicating sequential processes. *Comm. of the ACM*, vol. 21, n° 8, August 1978, pp. 666–677.

[10] Jézéquel (J.-M.). – *Object Oriented Software Engineering with Eiffel*. – Addison-Wesley, Mars 1996. ISBN 1-201-63381-7.

[11] Jézéquel (J.-M.), Guidec (F.) et Hamelin (F.). – Parallelizing Object Oriented Software Through the Reuse of Parallel Components. *Object-Oriented Systems*, vol. 1, 1994, pp. 149–170.

[12] Jézéquel (J.-M.) et J.-L. (P.). – Operator design pattern, application to parallel computation. *In : Collected papers from the PLoP '96 and EuroPLoP '96 Conferences*. – Washington University Department of Computer Science, Fév. 1997.

[13] Jézéquel (J.-M.), Matsuoka (S.), Sato (N.) et Yonezawa (A.). – A methodology for specifying data distribution using only standard object-oriented features. *In : Proc. of International Conference on Supercomputing*. – Vienna, Austria, Juil. 1997.

[14] Jézéquel (J.-M.) et Pacherie (J.-L.). – *Shared Objects in KOOPE*. – Rapport technique n° 2.3, Irisa-Intel ERDP, Juin 1995.

[15] Jézéquel (J.-M.) et Pacherie (J.-L.). – Parallel Operators. *In : ECOOP'96 proceedings*, éd. par Cointe (P.). pp. 384–405. – Lecture Notes in Computer Science, Springer Verlag, Juil. 1996.

[16] Matsuoka (S.) et Yonezawa (A.). – Analysis of inheritance anomaly in object-oriented concurrent programming languages. *In : Research Directions in Concurrent Object Oriented Programming*, éd. par Agha (G.), Wegner (P.) et Yonezawa (A.). – MIT Press, 1993.

[17] Meseguer (J.). – Solving the inheritance anomaly in concurrent object-oriented programming. *In : Proceedings ECOOP'93*, éd. par Nierstrasz (O.). pp. 220–246. – Kaiserslautern, Germany, Juil. 1993.

[18] Meyer (B.). – Systematic concurrent object-oriented programming. *Communications of the ACM*, vol. 36, n° 9, September 1993.

[19] Pacherie (J.-L.). – Modèle et environnemnt de programmation parallèle par objets. *In : Huitièmes rencontres francophones du parallélisme, RenPar'8*, éd. par Castanet (R.) et Roman (J.). p. 9. – Bordeaux, France, Mai 1996.

[20] Yokote (Y.) et Tokoro (M.). – The design and implementation of ConcurrentSmalltalk. *In : OOPSLA'86 Proceedings*. – 1986.

[21] Yonezawa (A.), Briot (J.-P.) et Shibayama (E.). – Object-oriented concurrent programming in ABCL/1. *In : OOPSLA'86 Proceedings*. – September 1986.

Chapitre 13

Mise au point d'applications parallèles irrégulières

Jacques Chassin de Kergommeaux, Benhur de Oliveira Stein, Philippe Waille (LMC-IMAG)

Ce chapitre est consacré aux outils d'aide à la mise au point d'applications parallèles. Les outils considérés ont pour objectif de faciliter la compréhension des exécutions de programmes parallèles pour en détecter les erreurs de logique ou d'implantation ainsi que les erreurs de performance. L'irrégularité sera considérée comme la contrainte d'être capable d'observer des programmes comportant un nombre potentiellement important de fils d'exécutions (threads), créés et détruits dynamiquement. Les problèmes qui en résultent ainsi que certaines solutions proposées pour les résoudre sont abordés dans le chapitre.

13.1 Introduction

Dans ce chapitre, nous supposerons que les modèles de programmation destinés aux applications irrégulières combinent des fonctionnalités de communication et de multiprogrammation légère et offrent des primitives de création dynamique de fils d'exécution (ou processus légers ou *threads*) à distance. Nous nous intéressons aux outils d'aide à la mise au point de telles applications, dont le degré de multiprogrammation légère (nombre de fils d'exécution par nœud) peut être important. Les modèles de programmation considérés sont habituellement implémentés en couplant une bibliothèque d'échange de messages telle que PVM [25] ou MPI [19] avec une bibliothèque de processus légers, le plus souvent à la norme POSIX [10]. Athapascan [2] et PM^2 [20] sont deux instances des modèles de programmation considérés.

Par mise au point on entend d'une part élimination des erreurs de logique ou d'implantation (« *correctness debugging* ») — qui font qu'un programme parallèle ne produit pas le résultat escompté — et d'autre part élimination des erreurs de performance (« *performance debugging* ») — qui empêchent un programme de tirer parti de la puissance de l'architecture parallèle utilisée. Aide à la mise au point signifie que les outils considérés n'ont pas pour objectif de détecter les erreurs automatiquement mais d'aider les programmeurs à identifier ces erreurs par observation et analyse de l'exécution de leurs applications parallèles.

La mise au point de programmes parallèles a suscité le développement d'outils adaptés aux difficultés spécifiques que suscite l'observation des exécutions de ces programmes, à savoir : très grand nombre d'états possibles, sensibilité aux perturbations externes susceptibles de modifier de façon profonde leur comportement et enfin nombre important d'entités mises en œuvre : processeurs, processus, liens de communication, etc. Ces difficultés sont accrues par la mise en œuvre d'un grand nombre de fils d'exécutions aux interactions fines.

Les outils de mesure de performances auxquels nous nous intéresserons étendent les techniques utilisées par les outils classiques de mesure de performances d'applications parallèles, basées sur le traçage et la visualisation. Les outils de débogage qui nous intéresserons ont pour objectif de généraliser l'usage des metteurs au point symboliques aux programmes considérés.

Ce chapitre expose les difficultés supplémentaires provenant des modèles de l'utilisation de nombreux fils d'exécution créés et détruits dynamiquement ainsi que des propositions de solutions, en considérant successivement les outils de traçage, visualisation et enfin de débogage.

13.2 Traçage logiciel d'applications parallèles

Le traçage logiciel est la technique d'observation des programmes parallèles la plus répandue. Après un rappel des raisons qui motivent ce choix, des problèmes classiques qu'il induit et de leurs solutions, nous présentons les problèmes spécifiques du traçage de programmes comportant de nombreux fils d'exécution ainsi que les solutions qui y ont été apportées pour tracer ATHAPASCAN.

13.2.1 Choix d'une technique d'observation

Plusieurs techniques peuvent être utilisées pour observer les exécutions de programmes parallèles afin d'en mesurer les performances [21] : l'échantillonnage, le comptage ou le traçage.

L'échantillonnage consiste à faire enregistrer la valeur du compteur ordinal, par un processus indépendant du programme observé, à intervalles de temps fixes, généralement dépendants du système d'exploitation (typiquement 20 milli-secondes sous Unix). Le temps passé dans une partie du programme (procédure) est supposé proportionnel à la fréquence d'apparition de la procédure dans les échantillons enregistrés. Très utilisé pour la mesure de performances de programmes séquentiels [5], ce type d'outil est en revanche peu adapté à la mise en évidence des obstacles à l'efficacité des programmes parallèles : les mesures rendent mal compte des dépendances qui limitent le parallélisme effectif et par ailleurs, les techniques d'échantillonnage ne permettent pas toujours l'évaluation des temps de communication, tels que l'attente d'un message (qui ne serait détectable que dans le cas d'une attente active).

Le comptage et le traçage enregistrent l'occurrence d'*événements*. On suppose que les processus des applications émettent des événements observables dont l'enregistrement constitue une trace d'exécution. Les événements auxquels on s'intéresse dépendent du type d'observation effectuée mais comportent le plus souvent des émissions et réceptions de messages par les processus ainsi que des événements «définis par les utilisateurs». Le comptage consiste donc à enregistrer le nombre d'événements de chaque type alors que chacun des événements d'une exécution parallèle est enregistré durant un traçage.

Le traçage est la technique de mesure de performances la plus générale. Si on désire obtenir des informations sur le temps passé dans les différentes parties des programmes (procédures), on enregistre débuts et fins de ces procédures. Pour détecter le temps passé en communications, on enregistre les émissions et réceptions de messages. La plupart des outils développés spécifiquement pour la mesure des performances des programmes parallèles sont basés sur la collecte et l'analyse de traces d'exécution ainsi que sur la visualisation des résultats d'analyse.

Il existe plusieurs types de traceurs d'exécutions parallèles [7] : traceurs matériels, logiciels ou hybrides. Les traceurs logiciels sont les plus portables et les moins coûteux. Leur utilisation peut être entièrement transparente au programmeur lorsqu'ils font partie d'une bibliothèque de communication pouvant être exécutée dans un mode «traçage» [4]. Le traceur peut également être composé d'une bibliothèque appelable par le programme tracé, les appels étant insérés par le programmeur [22] ou automatiquement par un pré-processeur [15]. L'utilisation de ces traceurs pose des problèmes de datation des événements, sur les systèmes matériels ne disposant pas de dispositif de datation globale, nécessitant l'utilisation de dispositifs logiciels de datation globale des événements enregistrés [12, 17]. Ce type de traceur pose également des problèmes d'intrusion, l'enregistrement et le transport des événements se faisant en concurrence avec l'exécution du programme parallèle par les processeurs et le réseau de communication du système parallèle utilisé. Des travaux sont en cours pour éliminer *post mortem* l'intrusion due au traçage, dans les cas déterministes où le traçage ne modifie pas les événements tracés [18, 27, 16], ainsi que dans le cas non déterministe [26].

13.2.2 Difficultés provenant de l'utilisation de fils d'exécution

La prise de trace concerne essentiellement l'ordonnancement des fils d'exécution, soit pour chaque fil son identité, le début, la fin et la cause des périodes de suspension de son exécution. S'y ajoutent des informations permettant de reconstituer l'historique des communications.

La multiprogrammation des nœuds soulève divers problèmes d'identification des fils d'exécution, d'observabilité de leur ordonnancement, d'atomicité des événements et de gestion des tampons de trace.

13.2.2.a Identification des fils d'exécution

Il convient d'identifier les fils d'exécution partageant un nœud multiprogrammé. Les noyaux de multiprogrammation légère gèrent de tels identificateurs, malheureusement inexploitables pour le développement d'outils portables de prise et de visualisation de traces d'exécution. La normalisation [10] spécifie en effet le nom du type identificateur, mais pas sa représentation interne, fortement dépendante de l'implantation.

La possibilité d'associer aux fils d'exécution une variable statique privée (*get* et *set specific* dans la norme POSIX), permet cependant de contourner le problème de représentation. L'adresse de cette variable peut servir d'identificateur de fil d'exécution et sur toutes les machines, une adresse peut être représentée par une constante entière.

13.2.2.b Identification des correspondants

Les bibliothèques de communication telles que PVM ou MPI indiquent le nœud source des messages reçus, ce qui permet d'apairer les émissions et réceptions de messages des traces des différents nœuds. La seule identité du nœud source est insuffisante lorsque les nœuds sont multiprogrammés : l'identification du fil émetteur nécessite une information complémentaire.

Le traçage des communications avec multiprogrammation nécessite donc une intervention sur la bibliothèque de communication pour lever l'ambiguïté des traces sur le fil émetteur. Deux stratégies sont envisageables :

- Ajouter une identification du fil émetteur aux informations (telles que le nœud source) déjà transportées avec le contenu du message ; cette solution implique de disposer du code source de la bibliothèque de communication.

- Sérialiser et numéroter sur chaque nœud les soumissions d'ordres de communication. Si les messages sont délivrés dans l'ordre où ils ont été émis (propriété FIFO du noyau de communication), la chronologie des soumissions permet de choisir la bonne association *post mortem*. C'est la solution retenue pour tracer ATHAPASCAN.

13.2.2.c Observabilité de l'ordonnancement

L'activité d'un unique fil d'exécution sur un nœud séquentiel est facile à tracer : il ne peut se bloquer que via des primitives explicites de communication dont il suffit d'enregistrer le début et la fin. Avec la multiprogrammation, la progression d'un fil est également suspendue lorsque le fil perd le processeur au profit d'un autre fil prêt à poursuivre son exécution, ce qui conduit naturellement au traçage de l'ordonnancement des fils (transitions entre les états actif, prêt et bloqué).

La réalisation d'un tel traçage bute sur deux obstacles : la disponibilité de noyaux de multiprogrammation légères instrumentés – on rejoint ici un problème général d'observation lorsque le comportement d'un niveau du système observé dépend de celui des niveaux sous-jacents qu'on n'est pas forcément capable d'observer – et la fréquence des transitions (problème d'intrusion et de taille des traces). Il en résulte, et c'est le cas pour ATHAPASCAN, que le traçage ne pourra porter que sur les périodes de blocage des fils et que la répartition du temps du (des) processeur(s) entre les différents fils ne pourra être déduite de la trace que sous des hypothèses restrictives particulières (nœud monoprocesseur, ordonnancement FIFO, absence de réquisition, etc). Ainsi sur le diagramme de la figure 13.2, plusieurs fils d'exécution peuvent être simultanément non bloqués, sur chacun des nœuds monoprocesseurs de la machine parallèle, sans qu'il soit possible de distinguer lequel d'entr'eux est actif à un instant donné.

13.2.2.d Atomicité et datation des événements

La multiprogrammation pose un problème classique d'atomicité susceptible de fausser la datation des événements. L'enregistrement d'un événement met en jeu une séquence d'opérations (réservation de place dans le tampon, lecture de la date de début, appel de la primitive à instrumenter, lecture de la date de fin, écriture de l'événement dans le tampon) au cours de laquelle le processeur peut être réquisitionné pour un autre fil.

Il en résulte une incertitude sur la datation des événements, dans la mesure où toute commutation de fil d'exécution entre la prise de date et l'appel de la primitive faussera d'autant cette date (cette incertitude réflète l'inobservabilité de l'ordonnancement des fils d'exécution).

13.2.2.e Gestion des tampons de trace

La trace est typiquement stockée dans un tampon en mémoire, recopié ensuite sur disque (sur débordement ou terminaison).

Le partage par plusieurs fils d'un tampon de traces commun accroît l'intrusion liée à la prise de trace. Il implique en effet une exclusion mutuelle lors de chaque accès (problème classique de rédacteurs multiples) qui affecte l'ordonnancement des fils d'exécution. L'exclusion affecte également l'exploitation du parallélisme interne de nœuds multiprocesseurs.

Doter les fils de tampons individuels substitue au problème du partage celui du dimensionnement des tampons : certains fils d'exécution auront une vie éphémère alors que d'autres généreront des traces conséquentes.

Le partage d'une collection de blocs de mémoire de taille fixe banalisés alloués à la demande constitue un compromis intéressant entre les deux alternatives. Un fil n'accède plus à la structure de données partagée à chaque événement, mais lors des débordements nécessitant l'allocation d'un tampon supplémentaire à son espace de stockage de trace. Les tampons pleins peuvent être recylés par un «démon» de vidage après recopie de leur contenu dans le fichier de trace.

13.3 Visualisation

13.3.1 Inadéquation des outils existants

Le grand nombre de fils d'exécution pose certains problèmes pour leur visualisation, comme la limitation d'espace disponible sur un écran pour les afficher, et la difficulté de compréhension d'un tel affichage. Les outils de visualisation de programmes parallèles existants comme Paragraph[6] et Pablo[22], n'ont pas été conçus pour visualiser l'activité de nœuds multiprogrammés. Leurs vues montrent l'activité des nœuds du système, ainsi que leurs communications. Il est bien sûr possible d'utiliser ces systèmes pour visualiser l'activité de fils d'exécution en représentant ces derniers comme des processeurs. Dans ce cas, la durée de vie des fils d'exécution doit être celle du programme, et leur nombre doit être (relativement) limité et doit rester constant durant l'exécution. Cette solution n'est donc pas adaptée à la visualisation de fils d'exécution dont le nombre varie continuellement et la durée de vie est souvent faible. Le problème qui se pose ici est similaire au problème de «scalabilité» qui se pose dans les outils «classiques» qui souhaitent représenter un nombre important de processeurs. Ici, même avec un nombre limité de nœuds on peut avoir une quantité importante de fils d'exécution.

Le système *Gthread*[28] permet la visualisation de plusieurs fils d'exécution pouvant éventuellement être créés et détruits dynamiquement. Cependant, il est conçu pour un système de programmation ne comportant qu'un seul niveau de parallélisme sur un même nœud, ce nœud étant en général un multiprocesseur symétrique à mémoire partagée.

13.3.2 Présentation de l'environnement Pajé

Pajé est l'environnement de visualisation de l'exécution des programmes ATHAPASCAN, à partir de traces d'exécution. Pour faciliter l'extensibilité de l'environnement et la réutilisation de ses modules, Pajé a été réalisé sous forme de composants interconnectés sur le modèle de l'environnement Pablo [22]. Chaque composant (module) est un objet indépendant, qui communique avec les autres à travers de ports de communication, de façon à constituer un graphe flot de données, dont les sommets sont les modules d'analyse et les arêtes sont les liens de communication. Les données qui circulent sur les arêtes sont des objets qui représentent les entités du programme analysé. L'indépendance des modules facilite le développement d'un environnement comportant des modules réutilisables. Un environnement de visualisation particulier est alors construit en connectant certains de ces composants.

La figure 13.1 représente un exemple d'un diagramme flot de données simple, avec un lecteur de traces, un simulateur, un module de statistiques et un module de visualisation (diagramme espace-temps). Le disque contient la trace, qui est lue et interprétée par le lecteur de traces, qui produit des objets qui représentent les événements qui y sont stockés. Le simulateur reçoit ces événements, simule les activités du programme et produit des objets qui représentent les états des fils d'exécution, les communications, les états des sémaphores. Ces objets sont utilisés par le module de statistiques ainsi que par le module de visualisation.

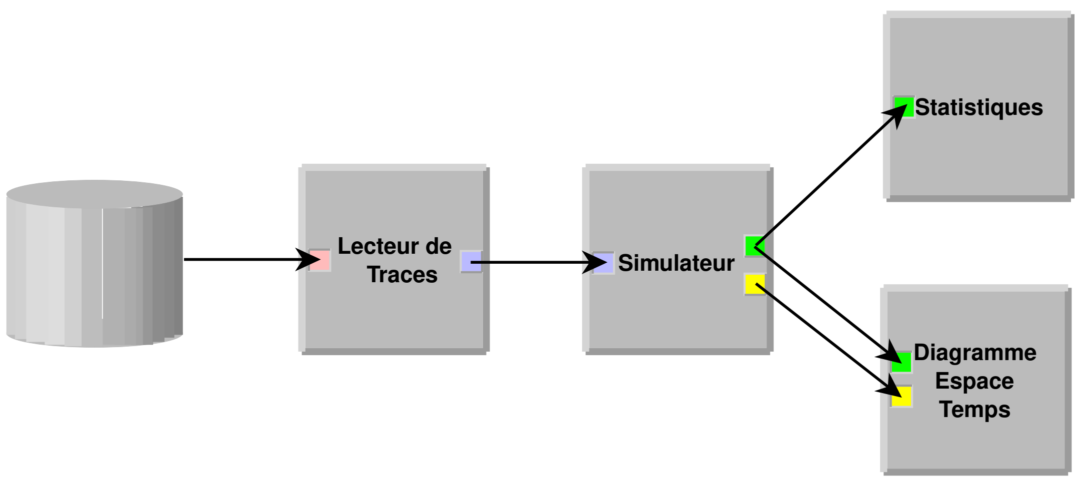


Figure 13.1 : Exemple de diagramme d'analyse

13.3.3 Visualisation des fils d'exécution dans Pajé

Des composants spécifiques ont été développés pour visualiser l'exécution de programmes comportant un nombre fixe de nœuds dont chacun exécute un nombre non borné de fils d'exécution. Le premier composant de visualisation réalisé est une combinaison étendue des diagrammes de Gantt et de Feynman. Ce composant est couplé à des filtres spécifiques pour prendre en compte les problèmes liés au grand nombre de fils d'exécution.

13.3.3.a Diagramme espace-temps

Ce module affiche sur l'écran un diagramme qui combine en une seule représentation les états successifs, les communications et les événements de chaque fil d'exécution, les opérations et les états des sémaphores. L'espace alloué à chaque nœud s'adapte dynamiquement au nombre de fils d'exécution qui s'y exécutent (voir figure 13.2).

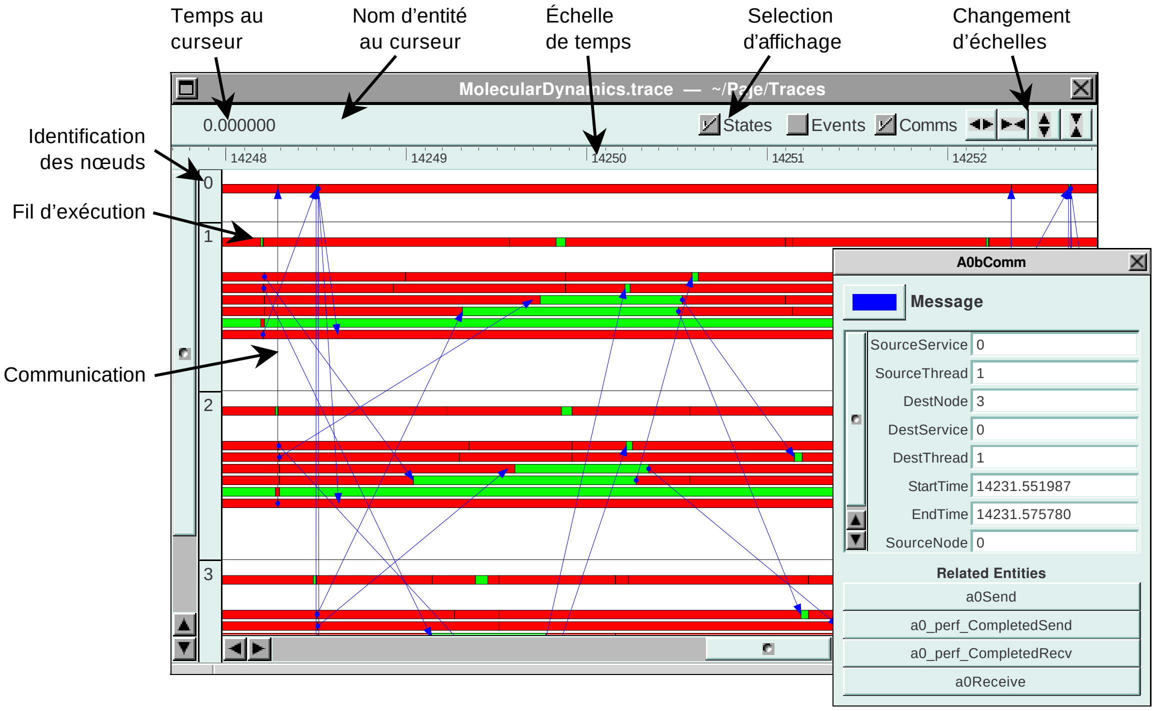


Figure 13.2 : Diagramme espace-temps

L'axe horizontal représente le temps, et l'axe vertical représente les fils d'exécution, groupés par nœud. Les communications sont représentées par des flèches, les états des fils d'exécution par des rectangles, et les événements par des triangles. Les différentes couleurs représentent le type de communication, d'activité des fils d'exécution ou d'événement. L'affichage de détails supplémentaires sur ces objets est fait en sélectionnant une de ces représentations.

13.3.3.b Filtres

Des filtres originaux ont été définis pour faciliter la représentation d'un grand nombre de fils d'exécution, d'une part en réutilisant l'espace libéré par les fils d'exécution qui ont achevé leur exécution et d'autre part en synthétisant les informations relatives à plusieurs fils d'exécution.

13.3.3.c Placeur de fils d'exécution

Les programmes ATHAPASCAN créent typiquement un grand nombre de fils d'exécution de durée relativement courte. L'objectif de ce filtre est de permettre une meilleure utilisation de la place disponible sur l'écran et une meilleure visualisation des fils, en réutilisant les positions des fils déjà terminés. Chaque fil d'exécution qui commence est placé en fonction des positions disponibles. Quand un fil termine, sa place est considérée comme disponible. Les figures 13.3(a) et 13.3(b) montrent un ensemble de fils d'exécution avant et après l'action du placeur.

Le placeur supporte la notion de service d'ATHAPASCAN[2] ; un fil d'exécution ne peut réutiliser que la position d'un autre fil d'exécution appartenant au même service du même nœud. Ainsi, les fils d'exécution d'un même service restent groupés, pour faciliter leur identification par l'utilisateur.

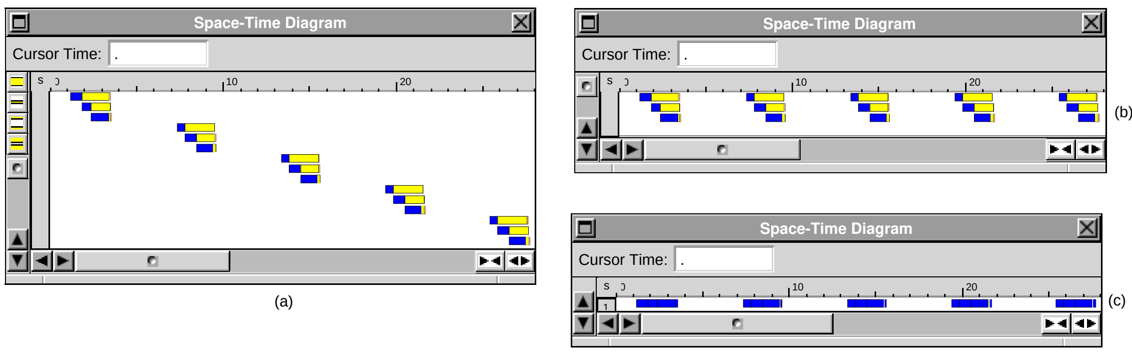


Figure 13.3 : Exemples de filtres

13.3.3.d Regroupement de fils d'exécution

Le module de regroupement génère, à partir de l'activité des fils d'exécution de la fenêtre courante, une vision plus globale de cette fenêtre qui ne contient qu'un seul fil d'exécution par nœud ou par service créé sur chaque nœud et dont les états peuvent être «en exécution» ou «en attente». Dans la vue groupée, il est plus facile de détecter si un nœud n'a pas assez de travail (voir figure 13.3(c)). Le passage de la vue détaillée à la vue groupée et réciproquement permet un effet de «zoom» sur la visualisation.

13.4 Outils de débogage

13.4.1 Débogueurs parallèles symboliques

Les débogueurs parallèles [9, 1] généralisent à plusieurs processus les fonctionalités classiques des débogueurs séquentiels. Ces débogueurs donnent aux programmeurs la possibilité de contrôler le déroulement d'un programme parallèle : pose de points d'arrêt sur un ou un ensemble de processus, exécution pas à pas d'un processus, etc. Ils offrent également la possibilité d'observer l'état de la pile de chacun des processus ainsi que les valeurs des variables, la visualisation se faisant à l'aide de systèmes de multifenêtrages. La principale différence relativement aux débogueurs séquentiels consiste en la notion de *contexte* : les fonctionalités héritées des débogueurs séquentiels n'affectent en effet que le ou les processus qui font partie du contexte courant.

Les outils de ce type facilitent beaucoup la mise au point d'applications parallèles et sont d'ailleurs au catalogue de la plupart des constructeurs de machines parallèles. En l'absence d'un tel outil, il est souvent possible d'en simuler le fonctionnement en «connectant» un débogueur séquentiel «a la *dbx*» à chacun des processus de l'application.

Les débogueurs parallèles existants ne gèrent pas tous la multiprogrammation des nœuds du système parallèle. Dans le pire des cas ils sont incompatibles avec l'existence de plusieurs fils d'exécution sur chacun des nœuds (ils ne sont pas «*thread safe*») ce qui rend nécessaire l'utilisation de techniques plus «rustiques» pour le débogage. Si le débogueur est compatible avec l'utilisation de fils d'exécution, il n'en supporte pas nécessairement la mise au point. Dans ce cas, le suivi de l'exécution sur un nœud peut se révéler complexe en raison des changements de contexte entre fils qui introduisent des ruptures dans contrôle non visibles dans le programme source.

Pour déboguer confortablement les applications considérées dans ce chapitre, il est nécessaire que le débogueur symbolique fournisse un support explicite à la notion de fil d'exécution. C'est le cas par exemple du débogueur *xpdbx* des machines IBM SP [9] qui permet au programmeur de suivre explicitement le passage d'un fil à l'autre à l'intérieur du nœud courant (voir figure 13.4).

Quel que soit leur degré de sophistication, les débogueurs parallèles sont cependant insuffisants pour lutter contre les erreurs fugitives apparaissant dans les programmes indéterministes.

13.4.2 Problème de l'indéterminisme

Un grand nombre de programmes parallèles présentent un comportement indéterministe, même s'ils produisent des résultats déterministes. L'indéterminisme est rendu possible lorsque le modèle de programmation permet l'existence de compétitions (*races*) entre messages (voir figure 13.5) ou pour l'accès à une mémoire

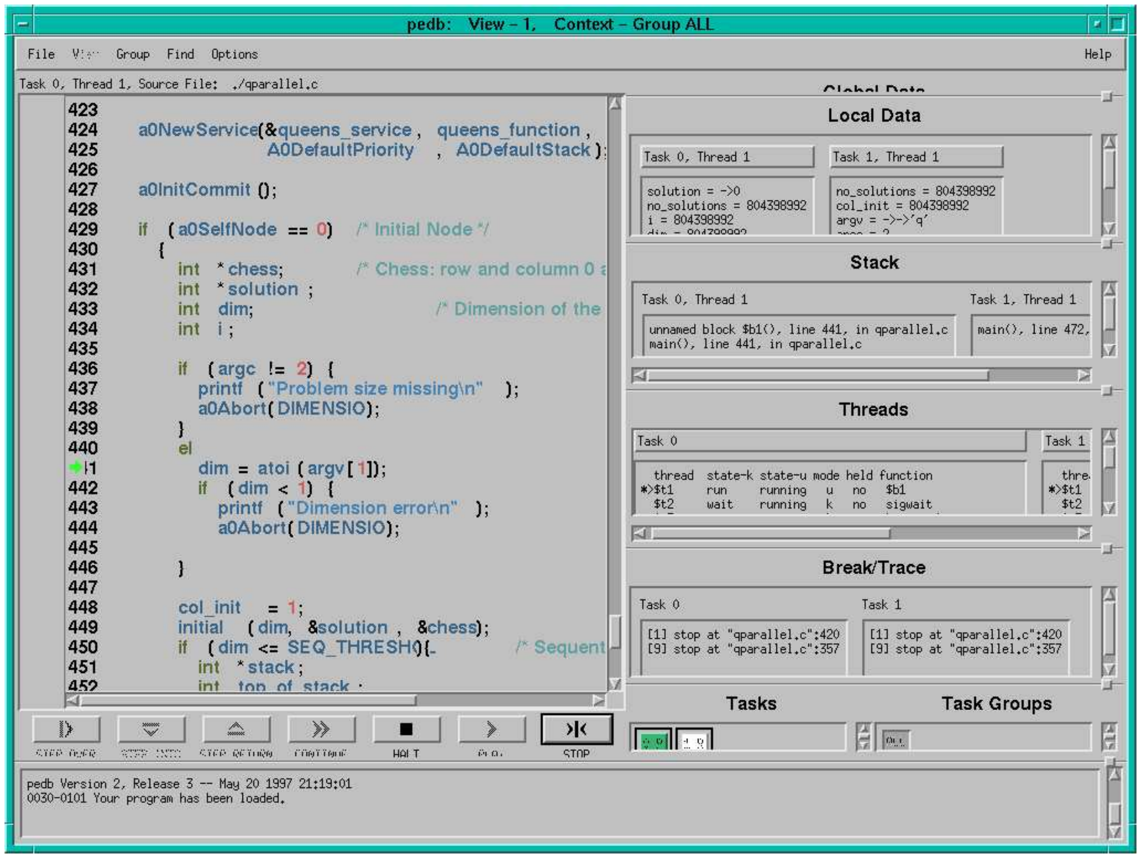


Figure 13.4 : Exemple d'utilisation du débogueur symbolique *xpdbx* de l'IBM SP (AIX 4.1)

partagée. À partir de paramètres initiaux semblables, un comportement indéterministe peut alors être provoqué par un grand nombre de facteurs incontrôlables par le programmeur d'une application, tels que les contenus initiaux des caches mémoires, l'état initial du système d'exploitation ou encore l'exécution simultanée d'autres processus que ceux de l'application sur le système parallèle.

Le comportement indéterministe d'un programme erroné peut être à l'origine d'erreurs fugitives, qui apparaissent peu fréquemment ou qui disparaissent dès qu'une instruction de traçage est insérée ou qu'un metteur au point parallèle est utilisé, en raison des changements de comportement des programmes parallèles induits par ces outils. Ce type de comportement rend impossible la mise au point «cyclique» utilisée habituellement pour les programmes séquentiels où l'on réexécute le programme fautif autant de fois qu'il est nécessaire pour localiser une erreur, en mettant des traces et des points d'arrêt de plus en plus près de l'erreur observée.

La technique la plus classiquement utilisée pour détecter les erreurs fugitives apparaissant durant les exécutions des programmes parallèles consiste à **enregistrer**

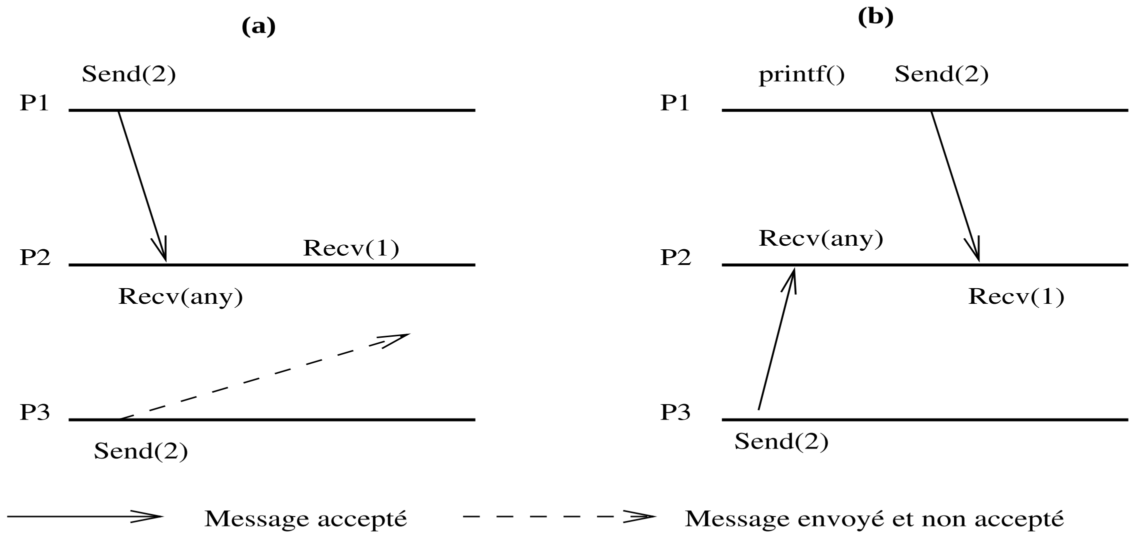


Figure 13.5 : Compétition entre messages reçus

L'erreur de programmation qui apparaît dans l'exécution (a) est masquée dans l'exécution (b) par la perturbation causée par l'ajout d'un ordre de traçage (*printf*).

une exécution initiale et à **forcer** les exécutions suivantes à se comporter de façon déterministe, relativement à cette exécution initiale, en utilisant les informations enregistrées. Le débogage d'un programme erroné revient alors à enregistrer une exécution erronée puis à appliquer les techniques de mise au point «cycliques», utilisées classiquement pour la mise au point de programmes séquentiels, durant les réexécutions successives du programme. Si le surcoût dû à l'enregistrement est suffisamment faible, il est possible de laisser l'enregistrement actif durant chaque exécution de programme parallèle, en sorte que toute erreur, même peu fréquente, puisse être enregistrée et reproduite à volonté.

La plupart des techniques d'enregistrement et de réexécution sont basées sur le mécanisme d'*Instant Replay* de Leblanc et Mellor-Crummey [13] qui ne nécessite que l'enregistrement de l'ordre des interactions entre processus. Cette technique a été adaptée aux modèles de programmation parallèle basés sur le partage de mémoire [23], le passage de messages [14, 8], la programmation objet [11, 24] et l'appel de procédure à distance (modèles client - serveur) [3].

Mis à part un cas particulier simple [3], nous ne connaissons pas de mise en œuvre de la réexécution déterministe pour un modèle de programmation combinant les fonctionnalités des librairies de fils d'exécution avec celles de communication. Les problèmes que pose cette mise en œuvre semblent de deux types :

- d'une part il s'agit de la difficulté d'implanter un enregistrement efficace (RE-

CORD) des informations de contrôle nécessaires à la réexécution de programmes. Ce problème est similaire à celui de l'implantation efficace d'un traceur (voir paragraphe 13.2.2).

- d'autre part, les modèles de programmation considérés offrent des possibilités telles que la migration [20] et un grand nombre d'interactions possibles entre fils d'exécution [2] : communications par mémoire commune sur un même nœud couplée avec différents mécanismes de synchronisation par sémaphore ou verrou, communications par messages entre nœuds. Ce grand nombre de possibilités rend plus complexe la définition d'un modèle d'exécution permettant de garantir l'équivalence entre exécutions parallèle et sa mise en œuvre efficace.

13.4.3 Complexité des exécutions parallèles

En présence d'un nombre important de processus parallèles, les outils de débogage n'offrent pas d'aide à la compréhension globale de l'application. Une approche intéressante suivie par Leu et Schiper [14] consiste à coupler la réexécution déterministe à l'outil de visualisation ParaGraph, utilisé habituellement pour la mesure de performances : les vues ParaGraph offrent une visualisation de haut niveau de l'exécution du programme parallèle (processus actifs, communications inter-processus, etc.) tandis que le débogueur parallèle de la machine permet l'examen de «bas niveau» (code source) de l'application.

Les difficultés posées par le traçage d'applications parallèles irrégulières , rendent ce type de couplage plus difficile à réaliser si les traces ne sont pas triées (voir paragraphe 13.2.2) et si les dates des événements enregistrés doivent être corrigées *post mortem* [17].

13.5 Conclusion

La mise au point de programmes parallèles corrects et efficaces a toujours été considérée comme un exercice difficile et a de ce fait suscité un nombre important de travaux. Dans ce foisonnement, les outils d'aide à la mise au point de programmes parallèles ont pour objectif d'aider les programmeurs à identifier les erreurs de correction et de performances de leurs programmes. Ces deux types d'outils ont en commun la sensibilité à l'intrusion et tous deux doivent représenter de façon intelligible et cohérente une grande quantité d'information.

Ces problèmes sont aggravés lorsqu'il s'agit d'applications irrégulières comportant un nombre important de fils d'exécution créés et détruits dynamiquement. Les tentatives de prise en compte de ces difficultés présentées ici consistent en des adaptations ou des extensions des outils développés pour la mise au point de programmes parallèles «traditionnels».

Bibliographie

[1] BBN Systems and Technologies. – *TotalView multiprocess debugger. User's Guide, version 3.4*, 1995.

[2] Briat (J.), Ginzburg (I.), Pasin (M.) et Plateau (B.). – Athapascan runtime: Efficiency for irregular problems. *In : Proceedings of EuroPar'97*. pp. 591–600. – Springer-Verlag.

[3] Fagot (A.) et Chassin de Kergommeaux (J.). – Formal and experimental validation of a low-overhead execution replay mechanism. *In : Euro-Par'95 Parallel Processing*. pp. 167–178. – Springer-Verlag.

[4] Geist (G. A.), Heath (M.), Peyton (B.) et Worley (P.). – *PICL, a Portable Instrumented Communication Library*. – TN n° 37831-8083, Oak Ridge, USA, Oak Ridge National Laboratory, 1991.

[5] Graham (S.), Kessler (P.) et McKusik (M.). – gprof: A call graph execution profiler. *In : Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*. pp. 120–126. – ACM.

[6] Heath (M. T.) et Finger (J. E.). – *ParaGraph: A Tool for Vizualizing Performance of Parallel Programs*. – Rapport technique n° ORNL/TM-11813, Oak Ridge National Laboratory, Oak Ridge, TN, 1991.

[7] Hofmann (R.), Klar (R.), Mohr (B.), Quick (A.) et Siegle (M.). – Distributed performance monitoring: Methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n° 6, Juin 1994, pp. 585–598.

[8] Hurfin (M.), Plouzeau (N.) et Raynal (M.). – EREBUS A debugger for asynchronous distributed computing systems. – *3rd IEEE Workshop on Future Trends in Distributed Computing Systems*, Taiwan, Avr. 1992.

[9] IBM. – *IBM Parallel Environment for AIX: Operation and Use, Volume 2, Tools Reference Version 2, Release 2*. Document Number SC28-1980-00.

[10] IEEE Standard for Multithreaded Programming. – *Posix 1.c Standard*. – IEEE Press, 1995.

[11] Jamrozik (H.). – *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants*. – Grenoble, Thèse de PhD, Université Joseph Fourier, Mai 1993.

[12] Jézéquel (J.-M.). – Building a global time on parallel machines. *In : Proc. of the* 3^{rd} *International Workshop on Distributed Algorithms*. pp. 136–147. – Lecture Notes in Computer Science, Springer Verlag.

[13] LeBlanc (T.) et Mellor-Crummey (J.). – Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, vol. C-36, n° 4, Avr. 1987, pp. 471–481.

[14] Leu (E.) et Schiper (A.). – Execution replay : a mechanism for integrating a visualization tool with a symbolic debugger. *In : Parallel Processing: Conpar92-VAPP V*. pp. 55–66. – Springer-Verlag.

[15] Maillet (E.). – *Tape/PVM: An efficient Performance Monitor for PVM Applications*. – User guide, B.P. 53, F-38041 Grenoble Cedex 9, France, LMC-IMAG, 1994. Available at ftp://ftp.imag.fr/imag/APACHE/TAPE.

[16] Maillet (E.). – *Traçage de logiciel d'applications parallèles : conception et ajustement de qualité*. – Thèse de PhD, Institut National Polytechnique de Grenoble, Sept. 1996.

[17] Maillet (E.) et Tron (C.). – On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, vol. 28, Juil. 1995, pp. 84–93.

[18] Malony (A. D.), Reed (A.) et Wijshoff (H.). – Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on parallel and distriuted systems*, vol. 3, n° 4, Juil. 1992.

[19] MPI Forum. – *MPI: a Message-Passing Interface Standard*. – Rapport technique, University of Tennessee, Knoxville, USA, 1995.

[20] Namyst (R.) et Méhaut (J.-F.). – Pm^2 parallel multithreaded machine: a multithreaded environment on top of pvm. *In : Proceedings of EuroPVM'95*. pp. 179–184. – HERMES (ISBN 2-86601-497-9).

[21] Reed (D. A.). – Experimental analysis of parallel systems: Techniques and open problems. *In : Proc. 7th Int. Conference on Computer Performance Evaluation*, éd. par Haring (G.) et Kotsis (G.). – Vienna, Austria, Mai 1994.

[22] Reed (D. A.), Aydt (R. A.), Madhyastha (T. M.), Noe (R. J.), Shields (K. A.) et Schwartz (B. W.). – *An Overview of the Pablo Performance Analysis Environment*. – Rapport technique, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1992.

[23] Ronsse (M.) et De Bosschere (K.). – *RecPlay: a fully integrated practical Record/Replay system*. – Rapport technique n° ELIS TR-DG 97 - 06, Department of Electronics and Information Systems, Universiteit Gent, Belgium, 1997.

[24] Roos (J.), Courtrai (L.) et Méhaut (J.). – Execution replay of parallel programs. *In : Proceedings Euromicro Workshop on Parallel and Distributed Processing.* – IEEE/Computer Society Press.

[25] Sunderam (V.). – PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, vol. 2, n° 4, Déc. 1990, pp. 315–339.

[26] Teodorescu (F.) et Chassin de Kergommeaux (J.). – On correcting the intrusion of tracing non deterministic programs by software. *In : Proceedings of EuroPar'97.* pp. 94–101. – Springer-Verlag.

[27] Yan (J. C.). – Performance tuning with aims — an automated instrumentation and monitoring system for multicomputers. *In : Proc. of the Twenty-Seventh Annual Hawai Conference on System Sciences.* pp. 625–633. – IEEE Computer Society Press.

[28] Zhao (Q. A.) et Stasko (J. T.). – *Visualizing the Execution of Threads-based Parallel Programs.* – Rapport technique n° GIT-GVU-95-01, Georgia Institute of Technology, 1995.

Quatrième partie

Gestion de données partagées

Chapitre 14

Des structures de données parallèles

Bertrand Le cun (PRiSM-Université de Versailles) Catherine Roucairol ((PRiSM-Université de Versailles)

14.1 Introduction

En séquentiel, une application utilise une ou plusieurs structures de données pour stocker et accéder aux informations. Ces structures de données peuvent être **simples**, comme les variables, les tableaux, les enregistrements, mais aussi très **complexes** comme les structures de recherches, les files de priorités, etc.

Pour une simple variable, une opération peut donc être la lecture, l'écriture ou le test sur un petit bloc mémoire. Ces opérations sont simples à mettre en œuvre et à utiliser. Leur complexité est constante en temps et en espace.

En revanche, si la SD est complexe, sa représentation c.à.d. l'organisation des données est une composition de structures simples appelées éléments. Une opération sur une structure complexe engendre donc un parcours de la mémoire selon la représentation, avec éventuellement des modifications de cette représentation. La complexité des opérations dépend alors de la taille et de la forme de la représentation.

Il existe beaucoup de catégories de SD complexes en séquentiel. Elles peuvent être classées selon les types d'opérations qu'elles supportent; un sous-classement peut être obtenu selon leur représentation en mémoire. Nous citons ci-dessous quelques exemples de telles structures.

- Les **structures de recherche** gèrent des éléments composés d'une clef et d'une information.

les opérations portent sur la clef pour : *rechercher un élément, insérer un élément, supprimer un élément.*

Les représentations classiques sont : les listes (simplement chaînées [1], doublement chaînées [1], les Skip-list [57], ...), les arbres binaires (AVL [1], Red-Black [64], Splay [67], Random search tree ou Treap [47, 2], ...), les arbres n-aires (les arbres 2-3-, les B-arbres, les B+-arbres, ...), les tables de hachage, etc.

- Les **files de priorités** gèrent des éléments composés d'une priorité et d'une information.

  Les opérations sont les suivantes : *insertion* d'un élément de priorité quelconque, *suppression* de l'élément ayant la plus forte priorité, *suppression* de tous les éléments de priorité inférieure à une valeur (opération spécifique à certaines applications).

  Les représentations possibles sont : les tas (d-Heap ou implicit-Heap [71, 1], leftist-Heap [1], Skew-Heap [65], Pairing-Heap [66], binomial-queue [6, 70], etc), toutes les structures de recherche détournées de leurs fonctions initiales, les structures spécifiques (Funnel-Tree [46], Funnel-Table [46], Calendar-queue [7]), etc.

Si on considère une application comme un *traitement* itératif sur des données, la procédure de traitement obtient des données par des demandes d'opérations à la structure, puis les modifie ou les combine. Les données nouvellement créées peuvent être réinsérées dans la SD.

Le **contexte d'exécution de la SD** (valeur possible de clefs pour les structures de recherche, valeurs possibles des priorités pour les files de priorité, nombre et ordres des différentes opérations, nombre d'éléments) est donc défini par le **comportement du traitement**. Ce dernier est très important car il va orienter le choix vers telle ou telle SD offrant les opérations les mieux adaptées à l'application.

14.2 Introduction du parallélisme dans les structures de données

Deux types de parallélisations du couple traitement+structure sont envisageables.

Une première consiste à paralléliser le traitement et la structure en "interne". Chaque traitement et chaque opération sont exécutés par plusieurs processus coopérants.

Une deuxième dite "externe" consiste à modifier le couple (traitement, structure) en (P traitements, structure). Les P procédures de traitement sont exécutées par P processus, chacune d'elles travaille sur des données différentes provenant d'une structure de données globale.

14.2.1 Parallélisme interne : Structures de données parallèles accélérées

Du point de vue de la structure, ce modèle entraîne une volonté d'accélérer chacune des opérations. par l'utilisation de plusieurs processus.

Pinotti et Pucci [55] ont présenté une file de priorité parallèle de ce type, le Min-Path-heap. Cette structure, est une version modifiée de la plus classique des files de priorité, le d-heap [71]. Elle utilise un algorithme permettant d'assurer la connaissance a priori du chemin que doivent prendre les opérations dans le tas. Une opération se réduit à l'insertion ou la suppression d'un élément dans un "tableau" de longueur $\log n$ (n est le nombre d'éléments). L'algorithme peut être efficace (complexité en $\log \log n$ avec $\log n$ processus) ou optimal (complexité en $\log \log n$ avec $\frac{\log n}{\log \log n}$ processus).

La connaissance a priori du chemin que doit prendre une opération est déterminé par l'opération précédente. La complexité d'une opération devient donc $O(\frac{\log n}{p} + \log \log n)$ [55].

D'autres parallélisations de ce type ont été proposées pour le HeapSort [71]. Les travaux de Olariu et Wen [50], Rao et Zhang [60], Pinotti et Pucci [55] et Dietz [15] concernent alors la construction du d-heap en parallèle.

L'approche SDPA définit généralement une parallélisation à grain fin. De plus, elle n'est applicable qu'à très peu de SD, car, bien souvent, la suite d'opérations élémentaires composant une opération doit être exécutée «séquentiellement». Pour une structure de recherche sous forme d'arbre par exemple, le chemin n'est pas connu à l'avance mais est construit tout au long du parcours.

14.2.2 Parallélisme "externe"

Une seconde approche consiste à exécuter P procédures de traitement de l'application sur P processus, chacun d'eux accédant à une SD "globale". Cela signifie que chacun des P processus exécutant la procédure de traitement, réalise ses opérations sur une seule SD. Une opération sur la SD doit avoir potentiellement accés à tout élément de la structure. Cette approche est actuellement la plus utilisée. La raison majeure réside dans le modèle de parallélisme sur lequel sont fondées les machines parallèles commerciales actuelles. Ce modèle étant dit à gros grain, cette seconde approche y est plus adaptée.

Dans la suite, nous considérons que l'accélération obtenue ne dépend que de l'accélération de la SDP, c.à.d. l'accélération de la partie du temps prise par l'exécution successive de la procédure de traitement est considérée comme linéaire.

14.2.3 Les structures de données à exécution séquentielle

Rendre parallèle une structure est chose aisée sur les machines MIMD actuelles. Si la mémoire de la machine parallèle est distribuée, il suffit de spécialiser un processus dans la gestion de la structure. Les P processus exécutant les traitements demandent au processus spécialisé d'exécuter leurs opérations pour eux. En revanche, si la machine dispose d'une mémoire partagée, la SDP réside en mémoire commune; chacun des P processus réalise ses opérations en section critique en utilisant une primitive d'exclusion mutuelle. Cette forme de parallélisation peut être vue comme un serveur d'opérations sur une SD séquentielle.

Nous appelons cette forme de SDP, **structure de données parallèles à exécution séquentielle (SDPES)**, Il n'y a donc aucun parallélisme possible du point de vue de la structure. L'accélération obtenue n'est donc due qu'à l'accélération de la procédure de traitement. Le surcoût engendré par cette structure est en distribué, le temps pris par tous les envois de message; en mémoire partagée, le temps d'exécution des primitives d'exclusion mutuelle.

Si le comportement de l'application parallèle définit un faible taux d'utilisation de la SD par rapport aux calculs locaux, le temps total pris par les opérations sur la SDP est négligeable par rapport à la somme des calculs locaux (l'exécution des procédures de traitement). Cette forme de SDP suffit donc, pour ce comportement, à obtenir une accélération linéaire.

Dans le cadre des files de priorité, l'intérêt de cette méthode est d'assurer que chaque processus obtienne l'élément de plus forte priorité lors des opérations de *suppression*. Ainsi, Roucairol [62, 63], Plateau and Roucairol [56], Quinn [58], McKeown [48], et Gendron et Crainic [25], Kindervater [32, 31], Eckstein [20, 19], Authié [4] ont proposé des gestions de SDP sur ce modèle, dans le contexte d'algorithmes de Branch and Bound en Optimisation.

Malheureusement, les SDPES ne répondent pas aux comportements de la plupart des applications parallèles. Car si le temps d'exécution des opérations de la structure n'est plus négligeable par rapport à la somme des calculs locaux, les processus demandeur sont la plupart de temps arrêtés dans l'attente d'éléments. De plus, en distribué, la SDPES étant gérée par un seul processeur, sa taille est limitée par les mêmes contraintes qu'en séquentiel.

Dès que l'application nécessite un grand nombre d'opérations, ou dès que le nombre de processus clients est trop important, les opérations sur la SDP doivent pouvoir être exécutées en parallèle.

14.2.4 Les structures de données à capacité parallèle

Les SDPES ne pouvant pas supporter toutes les demandes d'opérations en un temps raisonnable, les opérations doivent être parallélisées entre elles. A un instant donné, plusieurs opérations sont réalisées par plusieurs processus sur la SDP. Ce nombre d'opérations simultanées définit la *capacité* de la SDP.

Une SD étant composée de sa representation et de ces opérations, nous pouvons introduire le parallélisme dans ces deux composantes. Une SDP dont la représentation est parallélisée est appelée **structure de données à données parallèles (SDDP)**. Celle dont les opérations sont parallélisées est appelée **structure de données à opérations parallèles (SDOP)**, tandis que celle dont à la fois la représentation et les opérations sont parallélisées prend le nom de **structure de données à opérations et à données parallèles (SDODP)**.

Les trois paragraphes suivants expliquent et illustrent chacune de ces trois approches en présentant les parallélisations d'une structure de recherche représentée par une liste chaînée. Les problèmes spécifiques aux méthodes de parallélisation, que devraient gérer les algorithmes parallèles sur les opérations sont ainsi dégagés, ainsi que leur performances (capacité et coûts). Finalement, nous illustrons ce classement par des exemples de SDP tirées de la littérature.

14.3 Structures de données à données parallèles

Nous considérons une SD séquentielle notée S, de représentation R_s.

Paralléliser S par l'approche SDDP, consiste à créer D représentations R_s distinctes. Les n éléments de la structure séquentielle originelle sont répartis sur les D représentations R_s.

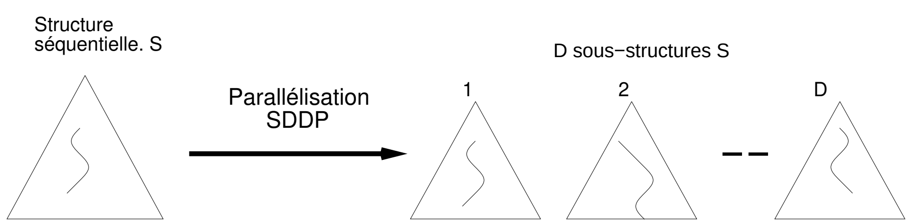


Figure 14.1 : Parallélisation par l'approche SDDP

14.3.1 Exemple : la liste chaînée à données parallèles

Dans le contexte d'une structure de recherche sous forme de liste chaînée, cette approche consiste à découper l'intervalle de recherche en D sous-intervalles. Une liste chaînée est associée à chacun d'eux. Les D listes sont notées L_i, $i \in 0..D-1$.

Un processus qui désire réaliser une opération d'*insertion*, de *suppression* ou de *recherche* d'un élément de clef v, accède à la liste L_i susceptible de contenir v.

14.3.2 Propriétés et coûts de la méthode

Théoriquement, la capacité C de la structure parallèle est égale à D (si il y a au moins D éléments), car D opérations peuvent être exécutées en parallèle (1 opération par R_s).

Chacune des D R_s doit contenir le même nombre d'éléments, pour assurer un nombre minimal de structures vides si $n \leq D$, un équilibre du temps nécessaire à la réalisation d'une opération, un équilibre du nombre d'opérations par représentation.

En effet, dans le cas idéal, chaque R_s contient $\frac{n}{D}$ éléments. La somme des opérations en séquentiel prend donc plus de temps que la somme des opération en parallèle. Ce qui tendrait à prouver que cette forme de parallélisation aboutit à une accélération surlinéaire.

Le problème est de maintenir le même nombre d'éléments dans chacune des R_s. Les opérations de modification tendent en effet à déséquilibrer la quantité d'éléments entre les D R_s. Un algorithme de gestion du parallélisme doit donc garder cet équilibre.

L'approche SDDP modifiant la représentation de la structure, l'algorithme de gestion du parallélisme doit aussi assurer la **même sémantique** des opérations en parallèle qu'en séquentiel.

En ce qui concerne les files de priorité, l'opération de *suppression* doit mettre en jeu un algorithme permettant d'assurer la même sémantique qu'en séquentiel.

Maintenir un équilibrage parfait et assurer pour les opérations parallèles la même sémantique qu'en séquentiel revient donc à maintenir une connaissance globale. Or, la complexité d'un algorithme maintenant une connaissance globale est une fonction dépendante de D et des caractéristiques de la machine.

Pour illustrer la difficulté entraînée par la gestion de la sémantique d'une opération, nous commençons par exposer la parallélisation d'une file d'attente proposée par Cheng [9].

14.3.3 File d'attente de Cheng

Cette parallélisation donne un très bon exemple de ce qu'il faut mettre en œuvre pour assurer la sémantique des opérations. Car bien qu'en séquentiel la complexité des opérations d'*insertion* et de *suppression* dans une file d'attente soit $O(1)$, en parallèle il n'en est pas de même.

Dans la description de Cheng [9], D processus peuvent accéder à D files d'attente séquentielles simultanément par l'utilisation d'un réseau multi-couches d'interconnexion.

Le réseau est composé de $\log D$ couches, chaque couche étant gérée par D processus spécialisés. Les processus spécialisés sont donc au nombre de $D \log D$.

L'algorithme consiste à trouver un chemin dans le réseau d'un point d'accès d'un processus vers une des files séquentielles, en prenant en compte le nombre de valeurs restantes dans les files, le nombre d'*insertions* et le nombre de *suppressions* pour un ensemble donné de demandes.

Le calcul est effectué par vagues successives. Les processus d'une couche travaillent sur un ensemble d'opérations de façon synchrone, ils aiguillent leur demande vers un processus particulier du niveau supérieur en fonction des résultats du niveau inférieur.

La complexité de cette structure pour l'*insertion* et la *suppression* est $O(1)$ avec $D \log D$ processus spécialisés.

L'intérêt de cette structure est d'illustrer la problématique posée par la sémantique des opérations et donc celle de la connaissance globale. Si en séquentiel, la complexité des opérations sur une file d'attente est $O(1)$, en parallèle, elle peut exécuter D opérations en $O(1)$ mais avec $D \log D$ processus.

14.3.4 Les files de priorité

Paralléliser une file de priorité par l'approche SDDP pose deux problèmes : l'équilibrage de charge, la sémantique de l'opération de *suppression*.

Nous exposons les SDP en commençant par celles qui assurent en parallèle une sémantique stricte, puis celles assurant une sémantique de plus en plus relâchée.

14.3.4.a Structures de données à opérations synchrones

Afin de garder les sous-structures parfaitement équilibrées, chaque opération de modification doit réaliser un équilibrage à chacune de ses exécutions. Les D opérations sur les D sous-structures doivent donc être réalisées de façon synchrone, d'où le nom de *structures de données à opérations synchrones*. Bien souvent, même s'il n'y a que $D - j$ $(1 < j < D)$ opérations, D processus doivent être présents.

Deo et Prasad [13], ont présenté une file de priorité parallèle à opérations synchrones fondée sur la structure de 2-Heap ou implicit-Heap. Nous en avons proposée une généralisation à toute les files représentées par des tas [36].

Une représentation élégante de la structure consiste à ce que chaque nœud du tas contienne D éléments, maintenus triés dans l'ordre décroissant des priorités. L'invariant du tas séquentiel est étendu à ce "tas parallèle".

Définition 14.3.1 *(L'Invariant du tas parallèle) les D éléments d'un nœud i ont une priorité inférieure aux D éléments du nœud j, père de i.*

En fait, on considère le même algorithme qu'en séquentiel, où chaque nœud du tas ne contient non plus 1 seul élément mais une liste de D éléments. Le problème à résoudre maintenant est de rendre comparable deux listes d'éléments. En effet, les D éléments d'une liste peuvent ne pas être supérieurs ou inférieurs aux D éléments d'une autre liste.

Nous avons introduit la fonction *TestPriForList* qui par redistribution des éléments, rend deux listes comparables et conserve l'invariant du tas.

Cette fonction utilise un algorithme de fusion puis recopie. Avec D processus, cet algorithme a une complexité de $O(\log D)$ (en version efficace). Le coût généré pour une opération parallèle c.à.d. D opérations est donc de :

$$\frac{n}{D} * \log D.$$

Cet algorithme permet la parallélisation de tous les tas de la littérature (D-Heap, Skew-Heap, binomial-queue, pairing-heap, etc). Le maintien de la connaissance globale et de l'équilibrage parfait coûte donc ici très cher.

14.3.4.b Structures de données à opérations macro-synchrones

Certaines applications comme le Branch and Bound en Optimisation, ont un comportement particulier, qui permet de proposer une solution moins «chère» que le 2-Heap parallèle de Deo et Prasad. En effet, les priorités des éléments générés par ces applications ont des propriétés intéressantes. D'une part, beaucoup d'éléments ont une priorité égale, et d'autre part, si tous les éléments de priorité v ont été supprimés, la priorité des éléments qui seront insérés par la suite, ne peut être qu'inférieure à v.

Dans ce cas, seule l'opération de *suppression* lorsqu'un changement de valeur de priorité apparait, peut faire l'objet de coopérations [14]. Un groupe de D *suppressions* retire D éléments de priorité égale en $\log D$ avec D processeurs. Les opérations d'*insertion* sont asynchrones, chaque processus insère ses éléments dans sa file locale.

Cette structure parallèle respecte strictement la sémantique de l'opération de *suppression* parallèle, et suffit à assurer du travail à tous les processus.

14.3.4.c Structures de données à opérations asynchrones

Dans les deux solutions précédentes, la sémantique stricte de l'opération de *suppression* est assurée soit au prix d'un coût élevé, soit en utilisant un comportement spécifique et connu de l'application. De plus, ces deux solutions nécessitent une exécution synchrone de groupes de D opérations de *suppression*.

Si l'application accepte un relâchement de la sémantique de l'opération de *suppression*, les opérations peuvent s'exécuter de manière asynchrone. Une stratégie d'équilibrage de charge est alors mise en œuvre pour équilibrer le nombre et la qualité des nœuds entre les différentes files.

Beaucoup de propositions ont été faites, sur machine à mémoire distribuée. Elles diffèrent par la stratégie d'équilibrage de charge qui est conçue : Lüling et Monien [41, 43, 42], Mans [44], Hajian, Hai et Mitra [26], Gendron et Crainic [25], Dowaji et Roucairol [17].

Bien que cette approche ait la plupart du temps été utilisée dans un contexte de machines distribuées, il existe des parallélisations de cette forme sur des machines à mémoire partagée : Pardalos [52], Rao et Kumar [61] Huang [27].

Proposer un équilibrage de charge efficace demande une caractérisation de l'application qui utilise la SDP.

14.3.5 Les structures de recherche

Deux domaines utilisent des structures de recherche parallèles : celui des Bases de Données avec les structures utilisées pour stocker les index ou les relations, et celui des machines dictionnaires.

14.3.5.a Les Bases de Données

Dans le contexte des Bases de Données, la parallélisation des SD dans un environnement distribué utilise beaucoup l'approche SDDP. Elle est alors appelée, *partitionnement* des données. L'intérêt d'une SDDP est d'accroître la quantité d'information stockée, tout en répartissant la charge sur différents processus.

L'introduction du parallélisme dans les SD pose des problèmes liés aux transactions. En effet, une transaction est une composition de plusieurs opérations sur plusieurs (ou une seule) SD. Or si une des opérations échoue, il faut pouvoir rendre inopérantes les opérations déjà effectuées. L'effet de toute opération doit donc pouvoir être annulé, pour «récupérer» l'ancienne version de la SD.

Deux méthodes résolvent ce problème. La première consiste à introduire de nouvelles opérations : *abort*, *commit*. Elles ont pour effet de rendre valide ou de faire échouer une opération. La deuxième est de considérer des SD multi-versions. Une SD stocke un historique de ces versions.

La structure étant représentée par plusieurs sous-représentations distinctes, chaque valeur de clef doit avoir une sous-représentation associée.

- Si le comportement de l'application est suffisamment connu, une association peut être décidée a priori. Les données sont placées statiquement ("Declustering" [33], Li, Srivastava et Rotem [40]).
- Si aucune hypothèse sur les valeurs des clefs ne peut être faite, la mise en œuvre d'équilibrage de charge dynamique est nécessaire.

14.3.5.b Les machines dictionnaires

L'approche SDDP est utilisée pour l'implémentation d'une machine dictionnaire sur machine parallèle SIMD [24], puis sur machine MIMD Volvox [18] fondée sur les arbres 2-3-4.

Des groupes d'instructions sont envoyés sur la machine. L'auteur ne faisant aucune hypothèse sur les valeurs des clefs, il devient donc obligatoire de mettre en place un algorithme d'équilibrage de charge entre les différentes sous-représentations.

14.4 Structures de données à opérations parallèles

Paralléliser une structure par l'approche "opérations parallèles" consiste à utiliser une seule représentation de la structure, et à effectuer les opérations en parallèle. Deux solutions peuvent être alors envisagées lorsque P opérations sont exécutées en synchrone ou asynchrone.

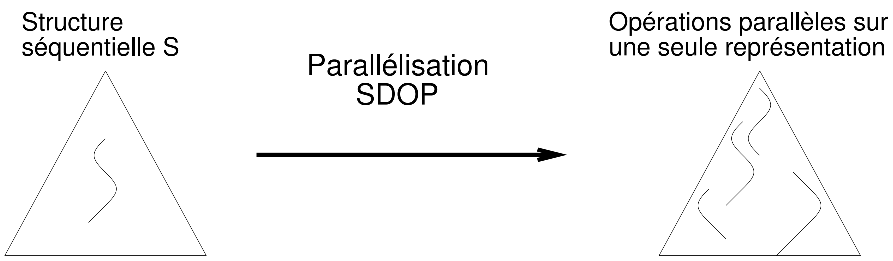


Figure 14.2 : Parallélisation par l'approche SDOP

Dans la suite, nous détaillons ces deux solutions et les illustrons par l'exemple d'une liste chaînée. Nous discutons ensuite des coûts et des limitations de la méthode synchrone, en prennant des exemples de parallélisation dans la littérature. Finalement, nous classons les parallélisations asynchrones, en fonction de la type de la structure, et de l'architecture de la machine.

14.4.1 Méthode synchrones

Un groupe d'opérations est exécuté de façon synchrone. Les parties communes de parcours sont réalisées par un seul processus. Lorsque les parcours divergent, le groupe d'opérations est séparé en deux sous-groupes. Chacun d'eux est exécuté par un processus.

Soient D opérations sur la liste L portant sur D éléments (*insertion*, *suppression*, *recherche*). Les D éléments sont triés dans un tableau t par ordre croissant (respectivement décroissant) si L est triée par ordre croissant (respectivement décroissant). Ils sont notés t_i.

Les D processus notés P_i effectuent les D opérations synchrones.

P_0 effectue l'opération sur t_0. Il parcourt alors L jusqu'à atteindre la position de t_0 dans L. Puis P_1 continue le parcours pour atteindre la position de t_1 et réalise l'opération sur t_1. Ainsi de suite, tous les P_i réalisent leurs opérations sur leur élément t_i. Les D opérations synchrones sont donc réalisées en un seul parcours.

Si la parallélisation est synchrone, la capacité est aussi grande que le permet la structure. Il est possible d'accéder aux n éléments simultanément. Cette technique est très intéressante car elle permet d'obtenir une grande capacité. Toutefois, cette méthode impose une synchronicité des opérations ce qui dans beaucoup d'applications, n'est pas souhaitable.

Plusieurs structures ont été proposées sur ce modèle de parallélisation. Une méthode générale appelée «cascade en hypercube» [3] peut être appliquées aux structures représentées par des graphes orientés sans cycle monotone sur machine SIMD organisée en Hypercube.

Dixon [16] a proposé, sur ce modèle, la parallélisation de la file de priorité de Van Emde Boas [69].

Un autre exemple est celui des arbres 2-3 proposée par Paul, Vishkin et Wagner [53]. La multi-recherche ne pose pas de problème particulier. En revanche, si la multi-insertion (D insertions simultanées) donne lieu à des opérations élémentaires de "split" et "join", un processus doit modifier des parties qu'il pourrait avoir en commun avec un autre processus. Deux opérations voulant effectuer un Split ou un Join sur un même nœud doivent donc être pipelinées.

Une limite de la méthode apparaît lorsque la SD à des opérations qui modifient la représentation. Dans l'exemple précédent de listes chaînées, considérons deux opérations de *suppression* portant sur deux nœuds x et y de la liste. Si x et y ne sont pas consécutifs, cela ne pose pas de problèmes : ils peuvent être supprimés en même temps. En revanche, si ils sont consécutifs dans la liste, les deux suppressions effectives des nœuds doivent être sérialisées. Les modifications se retrouvent donc pipelinées.

En fait, si les opérations doivent modifier des parties de la représentation, qui sont communes à deux ou plusieurs d'entre elles, les modifications doivent être sérialisées ou pipelinées, pour garder la consistance de la structure.

14.4.2 la méthode asynchrone

Une opération est composée d'une suite d'opérations élémentaires sur des parties distinctes de la représentation. Plusieurs opérations peuvent donc être exécutées en pipeline sur la représentation. Sur une machine à mémoire partagée (MMP), une opération est réalisée par un processus. Sur une machine à mémoire distribuée (MMD), elle le sera par plusieurs. Cette technique est issue de la théorie de la «sérialisation» (Calhoun et Ford [8]), où d'un point de vue extérieur à la structure de données, les opérations sont exécutées séquentiellement.

Sur une MMD, la liste L est distribuée sur P processus (idéalement P est égal au nombre d'éléments). L'opération se propage de processus en processus. Si une opération est à un instant donné sur le processus i $(0 < i < n)$, le processus $i-1$ peut travailler à la réalisation d'une autre opération.

sur une MMP, la liste réside en mémoire commune, les processus voulant réaliser une opération, accèdent à chacun des éléments en utilisant des primitives d'exclusion mutuelle, pour réaliser les modifications en section critique.

La capacité de la SDP dépend alors du nombre d'opérations élémentaires, du nombre d'éléments utilisés à chaque pas, et de la possibilité d'exécuter plusieurs opérations sur une même partie de la structure.

Si par exemple, chaque opération se décompose en h opérations élémentaires concernant deux éléments consécutifs, la capacité de la structure ne pourra être que de $\frac{h}{2}$.

Il est bien évident que si la structure en question est un arbre binaire, la capacité dépendra de la hauteur de l'arbre et sera donc limitée par le logarithme à base deux du nombre de nœuds.

Notons que les SDPES peuvent être prises comme un cas particulier de SDOP. Elles peuvent être accédées par plusieurs processus, avec une capacité limitée à un.

Nous dissocions dans la suite les algorithmes de ce type suivant qu'ils soient proposés sur MMP ou sur MMD, car bien que l'approche de parallélisation soit la même, les contraintes dues à l'environnement impliquent de grandes différences quant aux résultats obtenus.

14.4.2.a Mémoire partagée

Comme nous l'avons dit précédemment, la SDP réside en mémoire partagée. Nous dissocions maintenant les structures de recherche des files de priorité.

14.4.2.a.1 Les files de priorité Une opération étant composée d'une suite d'opérations élémentaires, un processus n'a besoin que d'une petite partie de la structure à un instant donné. L'utilisation de primitives d'exclusion mutuelle pour chaque entité modifiable de la SDP, rend possible des opérations concurrentes sur la structure. Les primitives d'exclusion mutuelle sont plus communément appelées verrous ou sémaphores. La technique introduite dite de *verrouillage partiel* (Jones [30], Deo [12], Mans et Roucairol [46], Le cun, Mans et Roucairol [37]), a été utilisée pour plusieurs files de priorité comme le Skew-Heap [30, 46], le d-Heap [59, 61, 5], le Funnel-Tree [46]. Un schéma de verrouillage permet d'interdire tout interblocage.

Les opérations de base nécessitent à chaque étape deux nœuds consécutifs (un nœud et son fils). Si la hauteur de l'arbre est $\log n$, la capacité est donc $C = \frac{\log n}{2}$.

Un surcoût de l'algorithme est dû aux appels de primitives d'exclusion mutuelle. Or, comme il en existe une par nœud de l'arbre, le nombre d'appels dépend aussi de la hauteur de l'arbre (une fonction de $\log n$).

Un autre surcoût est généré lorsque la mémoire de la machine n'est pas physiquement partagée mais virtuellement partagée. Chaque nœud pouvant être transféré d'un processus à un autre, ce coût est aussi une fonction de $\log n$.

Ces coûts peuvent être réduit en utilisant certains schéma de verrouillage [38].

14.4.2.a.2 Les arbres de recherche Dans le cas de structures de recherche (B-Arbre, Ellis [21], Kung et Lehman [34], Lehman et Yao [39], Kwong et Wood [35], Paul Vishkin et Wagener [53], Ishak [28], Johnson et Shasha [29], B+Arbre, ou les arbres binaires de recherche arbre AVL [22]), les opérations les plus utilisées sont les opérations de lecture. Elles ne modifient pas la représentation de la structure. Il est donc légitime d'utiliser plusieurs niveaux de verrous (verrou de lecture et verrou d'ecriture). Les protocoles de verrouillage sont appelés *Verrouillage d'arbre* (Tree-Locking). Les opération d'écriture utilisent les verrous exclusifs et les opérations de lecture les verrous de lecture.

Le problème est d'avoir le minimum de nœuds verrouillés à un instant donné, afin d'obtenir le maximum de processus travaillant en même temps dans la structure, et donc une capacité maximale.
Il est très difficile d'évaluer exactement la capacité. En effet, si toutes les opérations sont des opérations de lecture, la capacité est infinie.

De plus, les mêmes surcoûts que précédemment subsistent. A ceux dus aux verrous, s'ajoutent ceux dus aux transferts de données sur une machine à mémoire virtuellement partagée.

14.4.2.b Mémoire distribuée

Paralléliser une structure de données selon l'approche SDOP sur MMD consiste alors à placer la SD sur un nombre fixe ou variable de processus.

Le placement dépend des critères à optimiser qui sont au nombre de trois :

- maximiser la capacité de la structure, Un processus est associé ň niveau de l'arbre. La capacité est donc égale à la hauteur de l'arbre. Colbrook, Brewer, Dellarocas et Weihl [10] ont proposé un algorithme d'arbre n-aire de recherche de ce type, ainsi que Meybodyi [49] mais pour une FP.

- minimiser les temps de communications. Une opération doit traverser le minimum de processus. Un processus détient la racine, chaque sous-arbre de la racine est détenu par un processus (voir [54]).

- répartir l'occupation mémoire, Chaque processus stocke un sous-arbre (voir [45]).

En fait, il est impossible d'optimiser tous les critères en même temps dans le cas d'un arbre. Obtenir une bonne capacité conduit à associer un processus à chaque niveau de l'arbre, impliquant obligatoirement une mauvaise répartition de la mémoire. Alors que pour obtenir une bonne répartition mémoire, il faut grouper un même nombre d'éléments par processus. La capacité de la structure quant au nombre de processus utilisés est alors dégradée.

Dans tout ce que nous venons de voir, la représentation de l'arbre est placée statiquement sur les processus. La racine de l'arbre est associée à un processus. Un niveau ou un élément de l'arbre est affecté à un processus particulier. Nous avons proposé dans [36], une unification de toutes ces méthodes de parallélisation sur MMD, et MMP en simulant une mémoire partagée, l'accès aux données étant réalisé par des appels de procédure à distance.

14.5 Structures de données à opérations et à données parallèles

Les deux approches citées précédemment, ne sont pas incompatibles. Ce paragraphe montre comment elles peuvent être conjuguées.

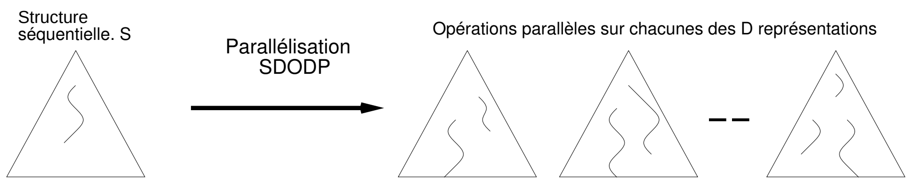


Figure 14.3 : Parallélisation par l'approche SDODP

14.5.1 Exemple : la liste chaînée

La SDP est représentée par D listes. Un groupe de $\frac{P}{D}$ processus est associé à une sous-liste parallélisée. sont parallélisées selon l'approche SDOP.

14.5.2 Limitation et coût de la méthode

L'approche SDDP avait l'intérêt de proposer une capacité "à la carte". En effet, il était possible de choisir un nombre D de sous-représentations. Toutefois, d'une part le coût engendré qui est fonction de D, peut devenir prohibitif par rapport au gain obtenu par la représentation de la SD globale, lorsque D croit. D'autre part, cette représentation pose aussi des problèmes de sémantique sur les opérations et donc de connaissance globale. Cette dernière, si elle doit être assurée dans son sens le plus strict, doit être gérée par un algorithme dont la complexité dépend aussi du nombre D de sous-représentations.

Par contre, l'approche SDOP ne pose pas de problèmes d'équilibrage de charge ou de connaissance globale, mais les SD étant souvent représentées par des arbres, leur capacité s'en trouve limitée.

L'approche SDODP, combinant les deux approches précédentes, a pour buts d'augmenter la capacité et de réduire le surcoût.

Soient D le nombre de sous-structures et h la capacité de chacune d'entre elles. Dans le cas où h dépend du nombre d'éléments, et où le nombre d'éléments contenus dans chacune des D sous-représentations est de $\frac{n}{D}$, la capacité d'une structure est égale à $h(\frac{n}{D})$.

La capacité C de la structure globale est :

$$C = D * h(\frac{n}{D}).$$

Si h est une fonction linéaire de n, la capacité obtenue est la même que pour l'approche SDOP.

Le gain s'exprime en terme de temps pour exécuter une opération. Le nombre d'éléments gérés par une opération n'est plus en effet en $O(n)$ mais en $O(\frac{n}{D})$.

De plus, pour réduire le nombre d'éléments par sous-représentations, il faut aussi réduire le surcoût de parallélisation des opérations. Si le surcoût s'exprime en fonction de h, le surcoût global s'en trouve donc réduit.

Dans l'approche SDDP, si la capacité est de P (le nombre de processus susceptibles d'accéder à la SDP), D doit être égal à P. Il suffit donc de choisir D tel que :

$$P = D * h(\frac{n}{D}).$$

Le nombre D obtenu est plus petit que P. Les surcoûts fonction du nombre D de sous-représentations sont donc réduit comparé à une SDDP.

Nous présentons dans la suite des SDODP tirées de la littérature.

14.5.3 Files de priorité

Si l'on considère les SDPES comme un cas particulier de SDOP, Trienekens et de Bruin [68], ansi que Kindervater [31] ont proposé une SDODP, qui pourrait être appelée SDESDP *structures de données à exécution séquentielle et données parallèles*.

La file de priorité globale est représentée par D sous-structures à exécution séquentielle. Chacune d'entre elles peut être accédée par un groupe de processus. Un processus particulier, le maître, de chaque groupe réalise l'algorithme SDDP.

Fan et Cheng [23] présentent une file de priorité parallèle, qui dans notre classement peut être vue comme une file de priorité représentée par une liste chaînée, parallélisée par l'approche SDODP. Cette file de priorité peut être vue comme la combinaison de la FP parallèle de Deo et Prasad représentée non plus par un 2-Heap mais par un 1-Heap en SDDP, et de la parallélisation des opérations en SDOP de Rao et Kumar.

Das et Horn [11] ont présenté une file de priorité SDODP fondée sur la file de Deo et Prasad [13]. L'algorithme de Deo et Prasad est une parallélisation par approche SDDP du 2-Heap. Or cette même structure a été parallélisée en SDOP par Bistwas et Brown [5] puis par Rao et Kumar [61]. Il est donc naturel qu'une parallélisation reprenant ces deux approches ait été proposée!

Fort de notre étude sur la capacité des structures parallèles, nous avons proposé [36] une FP SDODP, où le nombre D de représentations est variable en fonction du nombre d'éléments présents dans la SDP. Nous avons utiliser le Skew-Heap comme structures de base car elle permet des opérations de Split et Join entre deux skew-Heaps.

14.5.4 Les structures de recherche

Dans le cadre de l'étude d'une machine dictionnaire spécifiée en terme de VLSI [18], une SDODP a été proposée en employant comme sous-structure concurrente la machine dictionnaire de Ottman, Rosenberg et Stockmeyer [51]. Les D "sous-machines dictionnaires" recoivent les opération par l'intermédiaire d'un réseau diffusant. Pour résoudre les problèmes d'équilibrage de charge entre les différentes sous-structures, les PE sont aussi connectés par un arbre de calcul de préfixe, permettant de calculer la somme partielle du nombre d'éléments contenus dans les sous-structures et donc d'équilibrer la charge entre les D sous-structures.

Dans la cadre des Bases de données, l'utilisation d'un réseaux de MMP, peut entrer dans cette classe de structure dans la mesure où une «relation» est distribuée sur ces machines.

14.6 Conclusion

Nous avons proposé dans ce chapitre, une taxonomie de structures de données parallèles existantes. Cette taxonomie est fondée sur un concept englobant la représentation des données en mémoire et les opérations qui sont susceptibles d'y être effectuées. Le parallélisme introduit dans ces deux composantes permet de distinguer les structures. Il est possible d'établir un parallèle entre cette taxonomie et celle des ordinateurs proposé par Flynn pour les machines parallèles.

SISD Single Instruction Single Data	SOSD (SDPES) Single Operation Single Data structure
SIMD Single Instruction Multiple Data	SOMD (SDDP) Single Operation Multiple Data structures
MISD Multiple Instruction Single Data	MOSD (SDOP) Multiple Operations Single Data structure
MIMD Multiple Instruction Multiple Data	MOMD (SDODP) Multiple Operations Multiple Data structures

Cette taxonomie fondée sur la capacité, nous a permis de mieux caractériser les performances grandes classes de SDP. En effet, la complexité des opérations "parallèles" de certaines de ces structures peuvent être caractérisée comme efficace ou optimale (FP de Deo et Prasad [13]. En revanche, d'autres ne peuvent l'être en raison de asynchronisme. La capacité est un critère complémentaire pour comprendre et choisir la classe de structure qui correspond la mieux a l'application et à l'architecture visée.

Bibliographie

[1] Aho (A.), Hopcroft (J.) et Ullman (J.). – *The Design and Analysis of Computer Algorithms.* – Addison-Wesley, New-York, 1974.

[2] Aragon (C.) et Seidel (R.). – Randomized search trees. *FOCS 30*, 1989, pp. 540–545.

[3] Authié (G.), Ferreira (A.), Roch (J.-L.), Villard (G.), Roman (J.), Roucairol (C.) et Virot (B.). – *Algorithmique Parallèle : Analyse et Conception.* – Hermès, Paris, 1994.

[4] Authié (G.), Garcia (J.-M.), Ferreira (A.), Roch (J.-L.), Villard (G.), Roman (J.), Roucairol (C.) et Virot (B.). – *Parallélisme et Applications Irrégulières.* – Hermès, Paris, 1995.

[5] Bitwas (J.) et Browne (J.). – Simultaneous update of priority structures. *IEEE International conference on parallel computing*, 1987, pp. 124–131.

[6] Brown (M.). – Implementation and analysis of binomial queue algorithmms. *SIAM Comput.*, vol. 7, 1978, pp. 298–319.

[7] Brown (R.). – Calendar queues: a fast o(1) priority queue implementation for the simulation event set problem. *Communication of the ACM*, vol. 31, n° 10, Oct. 1988, pp. 1220–1227.

[8] Calhoun (J.) et Ford (R.). – Concurrency control mechanisms and the serializability of concurrent tree algorithms. *In : the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.* – Waterloo Ontario, Avr. 1984.

[9] Cheng (K.). – A simultaneous access queue. *Journal of parallel and distributed computing*, vol. 9, 1990, pp. 83–86.

[10] Colbrook (A.), Brewer (E. A.), Dellarocas (C. N.) et Weihl (W. E.). – *Algorithms for search trees on message-passing architectures.* – Rapport technique n° MIT/LCS/TR-517, Massachuset Institute of Technology, 1991.

[11] Das (S.) et Horng (W.-B.). – Managing a parallel heap efficiently. *In : Proc. PARLE'91-Parallel Architectures and Languages Europe*, pp. 270–288.

[12] Deo (N.). – Data structures for parallel computation on shared-memory machine. *In : SuperComputing*, pp. 341–345.

[13] Deo (N.) et Prasad (S.). – Parallel heap: An optimal parallel priority queue. *Journal of Supercomputing*, vol. 6, n° 1, Mars 1992, pp. 87–98.

[14] Diderich (C.) et Gengler (M.). – Experimental results with synchronized parallel branch and bound algorithms. *In : IFIP WG 10.3, IRREGULAR94 : Parallel Algorithms for Irregular Structured Problems, Suisse*, éd. par A. Ferreira, J. Rolim (K. A.).

[15] Dietz (P. F.). – Heap construction in the parallel comparison tree model. *In : SWAT'92 Third Scandinavian Workshop on Algorithm Theory, Scandinavie.*

[16] Dixon (B.). – Concurrency in a $o(loglogn)$ priority queue. *In : Lecture Notes in Computer Science, Parallel and Distributed computing : Theory and Practice, Canada-France Conference on Parallel Computing*, pp. 59–72.

[17] Dowaji (S.) et Roucairol (C.). – Influence of priority of tasks on load balancing strategies for distributed branch-and-bound algorithms. *In : 9th International Parallel Processing Symposium (IPPS'95) - Workshop on Solving Irregular Problems on Distributed Memory Machines*, pp. 83–90. – Santa Barbara (USA), Avr. 1995.

[18] Duboux (T.), Ferreira (A.) et Gastaldo (M.). – *A scalable design for VLSI dictionary machines.* – Rapport technique n° 94-33, LIP Ens-Lyon, France, Déc. 1994.

[19] Eckstein (J.). – *Parallel Branch-and-Bound algorithms for general mixed integer programming on the CM-5.* – Rapport technique n° TMC-257, Thinking Machines Corporation, 1994.

[20] Eckstein (J.). – Control strategies for parallel mixed integer branch and bound. *In : Supercomputing'94*, éd. par In I.C.S. Press (e.).

[21] Ellis (C.). – Concurrent search and insertion in 2-3 trees. *Acta Informatica*, vol. 14, 1980, pp. 63–86.

[22] Ellis (C.). – Concurrent search and insertion in avl trees. *IEEE Trans. on Cumputers*, vol. C-29, n° 9, Sept. 1981, pp. 811–817.

[23] Fan (Z.) et Cheng (K.). – A simultaneous access priority queue. *International conference on parallel processing*, 1989, pp. I–95–98.

[24] Gastaldo (M.). – *Contribution à l'algorithmique Parallèle des structures de données et des structures discrètes : machine dictionnaire et algorithmes pour les graphes.* – Thèse de doctorat, École Normale Supérieure de Lyon et l'Université Claude Bernard Lyon I, France, Déc. 1993.

[25] Gendron (B.) et Crainic (T.). – Parallel implementation of a branch-and-bound algorithm for multicommodity uncapacited location with balancing requirements. *In : INFOR*, pp. 151–165.

[26] Hajian (M.), Hai (I.) et Mitra (G.). – *A Distributed Processing Algorithm For Solving Integer Programs Using a Cluster Of Workstations.* – Rapport technique n° TR/14/94, Department of Mathematics and Statistics, 1994.

[27] Huang (Q.). – An evaluation of concurrent priority queue algorithms. *In : Third IEEE Symposium on Parallel and Distributed Processing*, pp. 518–525.

[28] Ishak (R.). – Semantically consistent schedules for efficient and concurrent b-tree restructuring. *In : Eight international Conference on Data Engineering*, pp. 184–199.

[29] Johnson (T.) et Shasha (D.). – The performance of concurrent b-tree algorithms. *ACM Transaction on DataBase System*, vol. 18, n° 1, Mars 1993, pp. 51–101.

[30] Jones (D.). – Concurrent operations on priority queues. *ACM*, vol. 32, n° 1, Jan. 1989, pp. 132–137.

[31] Kindervater (G.). – What is branch and bound ...? from folklore to a computational model and its implementation. *In : IFIP WG 10.3, IRREGULAR94 : Parallel Algorithms for Irregular Structured Problems*, éd. par A. Ferreira, J. Rolim (K. A.).

[32] Kindervater (G.) et Trienekens (H.). – Experiments with parallel algorithms for combinatorial problems. *EJOR, European Journal of Operational Research*, no33, 1988, pp. 65–81.

[33] Kitsuregawa (M.), Tanaks (H.) et Moto-Oka (Y.). – Architecture and performance of relational algebra machine grace. *In : the International Conference on Parallel Processing.*

[34] Kung (H.) et Lehman (P.). – Concurrent manipulation of binary search trees. *ACM Trans. on Database Systems*, vol. 5, n° 3, 1980, pp. 354–382.

[35] Kwong (Y.) et Wood (D.). – A new method for concurrency in b-tree. *IEEE Trans. Software Engr*, vol. SE-8, n° 3, Mai 1983, pp. 211–221.

[36] Le cun (B.). – *Des structures de données parallèles.* – 4, Place Jussieu, 75252 Paris Cedex 05, FRANCE, Thèse de PhD, Université Pierre et Marie Curie – Paris VI, Jan. 1996. In French.

[37] Le Cun (B.), Mans (B.) et Roucairol (C.). – *Comparison of concurrent priority queues for branch and bound Algorithms.* – RR n° 92-65, MASI Universite Pierre et Marie Curie, 1992.

[38] Le Cun (B.) et Roucairol (C.). – Concurrent data structures for tree search algorithms. *In : IFIP WG 10.3, IRREGULAR94 : Parallel Algorithms for Irregular Structured Problems*, éd. par A. Ferreira, J. Rolim (K. A.), pp. 135–155.

[39] Lehman (P.) et Yao (S.). – Efficient locking for concurrent operation on b-tree. *ACM trans. on Database Systems*, vol. 6, n° 4, Déc. 1981, pp. 650–670.

[40] Li (J.), Srivastava (J.) et Rotem (D.). – Cmd: A multidimensional declustering method for parallel database systems. *In : the 18th VLDB Conference, Vancouver, Canada.*

[41] Lüling (R.) et Monien (B.). – Two strategies for solving the vertex cover problem on a transputer network. *In : The 3rd Int. Workshop On Distributed Algorithms, number 392 in Lecture Notes Of Computer Sciences.*, pp. 160–171.

[42] Lüling (R.), Monien (B.), Räcke (M.) et Tschöke (S.). – *Solving the TSP with a distributed Branch-and-Bound Algorithms on a 1024 processor network.* – Rapport technique n° 160, University of Paderborn, 1995.

[43] Lüling (R.), Monien (B.) et Ramme (F.). – Load balancing in large network: A comparative study. *In : 3rd IEEE Symposium on Parallel and Distributed Processing.*

[44] Mans (B.). – *Contribution à l'Algorithmique Non Numérique Parallèle : Parallélisation de Méthodes de Recherche Arborescente.* – Thèse de doctorat, Université Paris VI, Paris, France, Juin 1992.

[45] Mans (B.). – Portable distributed priority queues with MPI. *In : Web server.*

[46] Mans (B.) et Roucairol (C.). – *Concurrency in Priority Queues for Branch and Bound algorithms.* – Rapport technique n° 1311, INRIA, Rocquencourt, France, 1990.

[47] McCreight (E. M.). – Priority search trees. *SIAM J Computing*, vol. 14, n° 2, Mai 1985, pp. 257–276.

[48] McKeown (G.), Rayward-Smith (V.), Rush (S.) et Turpin (H.). – Using a transputer network to solve branch and bound problems. *In : Transputing 91, Proceedings of the world Transputer User Group Conference*, pp. 781–800.

[49] Meybodi (M.). – Concurrent data structures for hypercube machine. *In : Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'92).*

[50] Olariu (S.) et Wen (Z.). – Optimal parallel initialization algorithms for a class of priority queues. *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, n° 4, Oct. 1991, pp. 423–429.

[51] Ottman (T.), Rosenberg (A.) et L.Stockmeyer. – A dictionary machine (for vlsi). *In : IEEE Trans. on Computers*, pp. 892–897.

[52] Pardalos (P.). – Parallel algorithms for nonlinear assignment problems. *In : IFIP WG 10.3, IRREGULAR94 : Parallel Algorithms for Irregular Structured Problems*, éd. par A. Ferreira, J. Rolim (K. A. P.).

[53] Paul (W.), Vishkin (U.) et Wagener (H.). – Parallel computation on 2-3 tree. *RAIRO informatique théorique, France*, vol. 17, n° 4, 1983, pp. 397–404.

[54] Peleg (D.). – Distributed data structures: a complexity-oriented view. *In : Proceedings of the 4th International Worshop on Distributed Algorithms (WDAG'90)*, pp. 71–89.

[55] Pinotti (M.) et Pucci (G.). – Parallel algorithm for priority queue operations. *In : SWAT'92 Third Scandinavian Workshop on Algorithm Thoery.*

[56] Plateau (G.) et Roucairol (C.). – A supercomputer algorithm for the 0-1 multiknapsack problem. *Impacts of Recent Computer Advances on Operations Research*, 1989, pp. 144–157.

[57] Pugh (W.). – Skip lists: A probabilistic alternative to balanced trees. *Communication of ACM*, vol. 33, n° 6, Juin 1990, pp. 686–676.

[58] Quinn (M. J.). – Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer. *IEEE Transaction*, vol. 39, n° 3, March 1990.

[59] Quinn (M. J.) et Yoo (Y. B.). – data structure for the efficient solution of graph theoretic problems on tighly-coupled mimd computers. *In : Proceedings of the International Conference on Parallel Processing*, pp. 431–438.

[60] Rao (N.) et Zhang (W.). – Building heaps in parallel. *Information Processing Letter*, no37, 1991, pp. 355–358.

[61] Rao (V.) et Kumar (V.). – Concurrent insertions and deletions in a priority queue. *IEEE proceedings of International Conference on Parallel Processing*, 1988, pp. 207–211.

[62] Roucairol (C.). – *Du séquentiel au parallèle: la recherche arborescente et son application à la programmation quadratique en variable 0-1, Thèse d'état.* – Thèse de PhD, Université Paris VI, Juin 1987.

[63] Roucairol (C.). – A parallel branch and bound algorithm for the quadratic assignment problem. *Discrete Applied Mathematics*, vol. 18, 1987, pp. 211–225.

[64] Sleator (D.) et Tarjan (R.). – Self-adjusting trees. *In: 15th ACM Symposium on theory of computing*, pp. 235–246.

[65] Sleator (D.) et Tarjan (R.). – Self-adjusting heaps. *SIAM J. Comput.*, vol. 15, n° 1, Fév. 1986, pp. 52–69.

[66] Stasko (J.) et Vitter (J.). – *Pairing Heap: Experiments and Analysis.* – Rapport technique n° 600, INRIA Rocquencourt, France, Fév. 1987.

[67] Tarjan (R.) et Sleator (D.). – Self-adjusting binary search trees. *Journal of ACM*, vol. 32, n° 3, 1985, pp. 652–686.

[68] Trienekens (H.) et de Bruin (A.). – *Towards a Taxonomy of Parallel Branch and Bound algorihms.* – RR n° EUR-CS-92-01, Rotterdam, The Netherlands, Erasmus University, Computer Science Departement, 1992.

[69] Van Emde Boas (P.), Kaas (R.) et Zilstra (E.). – Design and implementation of an efficient priority queue. *Mathematical System Theory*, vol. 10, 1977, pp. 99–127.

[70] Vuillemin (J.). – A data structure for manipulating priority queues. *Comm. ACM*, vol. 21, 1978, pp. 309–314.

[71] Williams (J.). – Algorithm 232: Heapsort. *Communications of the ACM*, vol. 7, 1964, pp. 347–348.

Chapitre 15

Critères de cohérence pour les systèmes à mémoire partagée

Michel Raynal (IRISA/INRIA)

15.1 Introduction

Since the end of the eighties, the Distributed Shared Memory abstraction (a shared memory built on top of a distributed system) is receiving more and more attention. One of its very first implementation has been done in the IVY system designed by Li and Hudak [19]. The distributed shared memory abstraction (DSM for short) has many advantages. First, at the application level, DSM frees the programmer from the underlying support as he has to consider only the well known *shared variables programming* paradigm to design a solution to his problem, independently of the system that will run his program (be it a centralized shared memory or a distributed one). Additionally, this facilitates his programming task as a lot of problems (especially related to numerical analysis or image processing) are easier to solve by using the shared variables paradigm than by using the message passing one. Second, at the system level, DSM makes transparent transport of programs, load balancing and process migration.

So, numerous protocols implementing a DSM on top of a distributed memory parallel machine or on top of a distributed system have been proposed. References [24] and [26] survey systems offering a DSM to their users. DSM implementations have common points with multiprocessor caches, networked file systems and distributed databases. Basically, the shared memory is supported by local memories of processors and copies of a data item can simultaneously be present in several local memories. Due to characteristics of the distributed context (asynchronous communications, existence of several copies,etc), some protocols implementing a shared memory on top of a distributed system offer to users a shared memory whose semantics is lightly different (sometimes in a very subtle way) from the classic semantics associated with a centralized shared memory, namely the *atomic* semantics.

Semantics of a shared memory is expressed by a *consistency criterion*. Such a criterion defines the value returned by every read operation invoked by a process. In nearly all DSMs [24, 20, 26], this consistency criterion is not formally defined and has to be deduced from the protocol implementing the shared memory. This makes study of properties of DSMs difficult and facilitates neither their understanding nor their comparison.

This tutorial[1] presents a set of formal definitions for the following consistency criteria: atomic consistency, sequential consistency, causal consistency, PRAM consistency and a few others. These definitions consider a shared memory computation as a partial order on the set of read and write operations issued by processes, and a particular consistency criterion is expressed as a constraint that the partial order has to satisfy[2]. A protocol implementing a DSM with some consistency criterion C has to ensure all computations will satisfy the associated constraint. Such an approach has several advantages. First, as these definitions are independent of particular implementations, they exhibit intrinsic properties associated with consistency criteria; so, this approach follows the abstract data type one by clearly distinguishing the semantics of the "object" offered to users (a shared memory with some semantics) from particular implementations. Second, the set of definitions given in this paper constitutes a hierarchical suite in the following sense: as they all are expressed by using the same formalism, it is possible to order them (from the more to the less constrained); consequently it is easy to see what are the additional constraints required by one consistency criterion with respect to another by comparing their positions within the hierarchy.

The paper is divided into 6 main sections. Section 2 presents the basic shared memory model. Then, Sections 3, 4, 5, and 6 give formal definitions for sequential, atomic, causal and PRAM consistency, respectively. Basic principles of protocols implementing these criteria are also given. Section 7 completes the panorama by examining other consistency criteria, namely hybrid, mixed, release and entry consistencies.

[1]Ce texte est extrait é d'un article (écrit avec André Schiper) accepté pour publication et à paraitre dans la revue *Annales des Télécommunications.*

[2]Moreover, it is worth noting that this set of formal definitions is based on very few (and simple) mathematical notions, namely: partial order, linear extension, suborder and legality (of read operations).

15.2 Shared Memory Model

15.2.1 Notations

A shared memory system is composed of a finite set of sequential processes $P_1, \ldots, P_n$ that interact via a finite set X of shared objects. Each object $x \in X$ can be accessed by read and write operations. A write into an object defines a new value for the object; a read allows to obtain a value of the object. A write of value v into object x by process P_i is denoted $w_i(x)v$; similarly a read of x by process P_j is denoted $r_j(x)v$ where v is the value returned by the read operation; op will denote either r (read) or w (write). For simplicity, as in [22, 4, 27], we assume all values written into an object x are distinct[3]. Moreover, the parameters of an operation are omitted when they are not important. Each object has an initial value; it is assumed that this value has been assigned by an initial fictitious write operation.

15.2.2 Histories

Histories are introduced to model the execution of shared memory parallel programs. The *local history* (or local computation) $\widehat{h}_i$ of P_i is the sequence of operations issued by P_i. If $op1$ and $op2$ are issued by P_i and $op1$ is issued first, then we say $op1$ precedes $op2$ in P_i's process-order, which is noted $op1 \rightarrow_i op2$. Let h_i denote the set of operations executed by P_i; the local history $\widehat{h}_i$ is the total order $(h_i, \rightarrow_i)$.

An *execution history* (or simply a history, or a computation) $\widehat{H}$ of a shared memory system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that[4] :

- $H = \bigcup_i h_i$
- $op1 \rightarrow_H op2$ if :

  i) $\exists\ P_i\ : op1 \rightarrow_i op2$ (in that case, $\rightarrow_H$ is called *process-order* relation),

  or *ii)* $op1 = w_i(x)v$ and $op2 = r_j(x)v$ (in that case $\rightarrow_H$ is called *read-from* relation),

  or *iii)* $\exists op3\ : op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$.

Two operations $op1$ and $op2$ are *concurrent* in $\widehat{H}$ if we have neither $op1 \rightarrow_H op2$ nor $op2 \rightarrow_H op1$.

[3]This hypothesis is usual in the domain of database where it is given greater place to *conflict equivalence* than to *view equivalence* when defining *serializability* [25]. Intuitively, it can be seen as an implicit tagging of each each value by a pair composed of the identity of the process that issued the write plus a sequence number.

[4]Section 8 briefly compares definition of computations in the shared memory model and in the message-passing model.

15.2.3 Legality

A read operation $r(x)v$ is *legal* if: (i) $\exists\, w(x)v\ : w(x)v \rightarrow_H r(x)v)$ and (ii) $\nexists\, op(x)u\ :$ $(u \neq v) \wedge (w(x)v \rightarrow_H op(x)u \rightarrow_H r(x)v)$. A history $\widehat{H}$ is legal if all its read operations are legal.

The legality concept is the key notion on which are based our definitions of shared memory consistency criteria. In a legal history no read operation can get an overwritten value. In the following sections, the definition of every consistency criterion follows the same pattern:

- First, according to the consistency criterion considered, one or several histories are defined from the computation $\widehat{H}$,
- Then, $\widehat{H}$ is claimed to satisfy the consistency criterion if and only if this (these) associated history (-ies) is (are) legal.

15.3 Sequential Consistency

15.3.1 Definition

Sequential consistency has been proposed by Lamport in 1979 to define a correctness criterion for multiprocessor shared memory systems [18]. A system is sequentially consistent with respect to a multiprocess program, if "*the result of any execution is the same as if (1) the operations of all the processors were executed in some sequential order, and (2) the operations of each individual processor appear in this sequence in the order specified by its program*".

This informal definition states that the execution of a program is sequentially consistent if it could have been produced by executing this program on a monoprocessor system[5]. More formally, we define sequential consistency in the following way.

Definition. Sequential consistency. A history $\widehat{H} = (H, \rightarrow_H)$ is *sequentially consistent* if it admits a linear extension[6] in which all reads are legal.

[5] In his definition, Lamport assumes that the *process-order* relation defined by the program (see point *(2)* of the definition) is maintained in the equivalent sequential execution, but not necessarily in the execution itself. As we do not consider programs but only executions, we implicitly assume that the *process-order* relation displayed by the execution histories are the ones specified by the programs which gave rise to these execution histories.

[6] A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{H} = (H, \rightarrow_H)$ is a topological sort of this partial order, *i.e.*, (i) $S = H$, (ii) $op_1 \rightarrow_H op_2 \Rightarrow op_1 \rightarrow_S op_2$ ($\widehat{S}$ maintains the order of all ordered pairs of $\widehat{H}$) and (iii) $\rightarrow_S$ defines a total order.

As an example let us consider the history $\widehat{H_1}$ (Figure 1)[7]. Each process P_i, (i=1,2), has issued three operations on the shared objects x and y. The write operations $w_1(x)0$ and $w_2(x)1$ are concurrent. It is easy to see that $\widehat{H_1}$ is sequentially consistent by building a legal linear extension $\widehat{S}$ including first the operations issued by P_2 and then the ones issued by P_1. It is also easy to see that the history $\widehat{H_2}$ (Figure 2) is not sequentially consistent, as no legal linear extension of $\widehat{H_2}$ can be built.

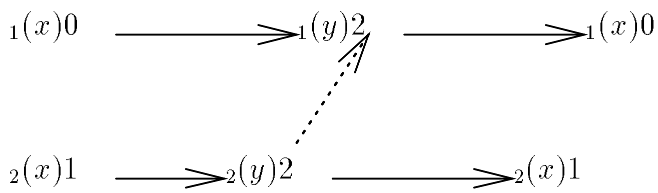


Figure 15.1 : A sequentially consistent history $\widehat{H_1}$

15.3.2 Protocols

Various cache-based protocols implementing sequential consistency have been proposed in the context of parallel machines [2, 7, 23]. In most of these protocols, every processor local memory contains a copy of the whole shared memory. So, each read operation is executed locally, while write operations issued by processes are globally synchronized to get a total order.

In [2] and in the *fast read* protocol of [7], this total order is built by an underlying *atomic broadcast* primitive (messages sent with this primitive are delivered in the same order to each processor). Read operations issued by a process are appropriately scheduled by its processor in order to ensure their legality.

In the protocol presented in [23], a process P_i issuing a write operation sends a write message to a central manager and waits for an answer. The central manager totally orders write operations. After receiving a write message from a process P_i, the central manager sends back an answer informing P_i about its set of copies of variables whose values are out of date and consequently whose future reads will no more be legal. Two versions of the protocol are described; in the first one, variables whose future reads by P_i will be illegal are invalidated in its local memory; in the second one, the manager informs process P_i of the current values associated with these variables.

[7] In all figures, only the edges that are not due to transitivity are indicated (transitivity edges come from *process-order* and *read-from* relations). Moreover, (intra-process) *process-order* edges are denoted by continuous arrows and (inter-process) *read-from* edges by dotted arrows.

In the context of distributed systems, where each object is supported by several permanent copies, non cache-based protocols implementing sequential consistency have been proposed. Usually these protocols use votes [28] or quorums [14] mechanisms and, consequently, implement actually atomic consistency which is stronger than sequential consistency (see Section 4). Some systems [11] consider copies as cached values and employ techniques (invalidation *vs* update) similar to the ones used in the management of cache mechanisms [6] ([20] provides an empirical comparison of these techniques).

15.4 Atomic Consistency

15.4.1 Definition

Atomic consistency is the "oldest" consistency criterion and the one that is the most encountered in distributed systems. While sequential consistency does not consider real-time, atomic consistency does. So, the underlying model for atomic consistency is asynchrony+real-time. Informally, atomic consistency adds to sequential consistency the following constraint: any two non-overlapping operations must appear in their real-time order within $\widehat{H}$.

Expressed in the previous model this means that executions of operations can no longer be considered as instantaneous. In order to take into account the real-time occurrence of operations, a *real-time precedence* relation, denoted $\prec_{RT}$, is defined in the following way. Let e_i^s and e_j^t be two operations belonging to H; if e_i^s was terminated before (with respect to physical time) e_j^t began, then we have, by definition: $e_i^s \prec_{RT} e_j^t$. Relation $\prec_{RT}$ is a partial order relation: two operations overlapping in real-time are not ordered.

Definition. Atomic Consistency. A history $\widehat{H} = (H, \rightarrow_H)$ is *atomically consistent* if it admits a linear extension $\widehat{S} = (H, \rightarrow_S)$ (i) whose all reads are legal (i.e., $\widehat{S}$ is sequentially consistent) and (ii) which is a linear extension of $(H, \rightarrow_{RT})$ (i.e., $e_i^s \prec_{RT} e_j^t \Rightarrow e_i^s \rightarrow_S e_j^t$).

As soon as reads are legal (point i of the definition), they return the "last" value of a variable. The fundamental difference between sequential consistency and atomic consistency lies in the meaning of the word "last". In the case of sequential consistency "last" refers to logical time, while it refers to physical time in the case of atomic consistency (point ii of the definition).

The interested reader will find in [22, 7] a theory of atomic consistency. In [15], under the name of *linearizability*, atomic consistency theory is generalized to objects.

15.4.2 Protocols

The most representative protocol implementing atomic consistency on top of distributed memory parallel machines is the Li-Hudak's one [19]. This protocol uses an invalidation approach. Each data (a page in this protocol) is owned by a process, namely the last process that wrote into it. When a process, wants to read a page for which it has not a copy, it sends a request to the manager of this page that forwards this request to the current owner. When the owner receives such a request, it sends a copy of the page to the requesting process and invalidates its write access right associated with the page. When a process wants to write a page, it sends, through the manager of the corresponding page, a request to the current owner; when receiving such a request, the owner first invalidates all -except his own- copies it has previously disseminated, and then sends its copy to the requesting process. After this, the requesting process is the new owner of the page, and no one else has a copy of the page. These mechanisms ensure atomicity (*i.e.* mutual exclusion) between any couple of read and write operations, and any couple of write operations.

Operating systems, especially distributed file systems, have mainly considered atomic consistency. This criterion is implemented by using a majority voting protocol [28], or a more general quorum protocol [14]. A quorum can be seen as a set of permissions owned by processes and granted to a requesting process. After having executed the operation for which the quorum was necessary, the requesting process gives back permissions to their owners. To read (write) a data x, a process P_i must get a read quorum $QR_{i,x}$ (write quorum $QW_{i,x}$). Read and write quorums guaranteing atomic consistency are defined by the two following rules which implement the classic readers-writers discipline:

$$\forall i \neq j : \forall x \in X : QR_{i,x} \cap QW_{j,x} \neq \emptyset \tag{15.1}$$

$$\forall i, j : \forall x \in X : QW_{i,x} \cap QW_{j,x} \neq \emptyset \tag{15.2}$$

Rule (1) realizes readers-writers mutual exclusion. It states that if P_i wants to read x, it must get permissions from all processes belonging to $QR_{i,x}$. Similarly, when P_j wants to write x, it must get permissions from all processes belonging to $QW_{j,x}$. As any process in $QR_{i,x} \cap QW_{j,x}$ can grant its permission either to P_i or to P_j (i.e., it can not satisfy simultaneously both of them), the desired exclusion follows. In the sam way rule (2) realizes writer-writer mutual exclusion.

15.5 Causal Consistency

15.5.1 Definition

Causal consistency has first been introduced by Ahamad *et al.* in 1991 [4], and then studied by several authors [5, 3, 27]. It defines a consistency criterion strictly weaker than sequential consistency, and allows for a wait-free implementation of read and write operations in a distributed environment (*i.e.* causal consistency allows for cheap read/write operations).

With sequential consistency, all processes agree on a same legal linear extension $\widehat{S}$. The agreement defined by causal consistency is weaker. Given a history $\widehat{H}$, it is not required that two processes P_i and P_j agree on the same ordering for the write operations which are not ordered in $\widehat{H}$. The reads are however required to be legal.

The set of operations that may affect a process P_i are the operations of P_i plus the set of all write operations issued by other processes. Let $\widehat{H_i}$ be the sub-history of $\widehat{H}$ from which all read operations not issued by P_i have been removed[8].

Definition. Causal consistency. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. $\widehat{H}$ is *causally consistent* if, for each process P_i, all the read operations of $\widehat{H_i}$ are legal.

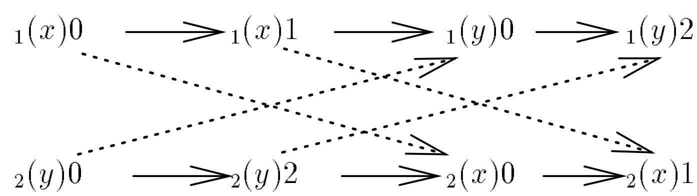


Figure 15.2 : A causally consistent history $\widehat{H_2}$

Said another way, in a causally consistent history, all processes see the same partial order on operations but, as processes are sequential, each of them might see a different legal linear extension of this partial order.

So, in a causally consistent history, no read operation of a process P_i can get a value that, from P_i's point of view, has been overwritten by a more "recent" write. As an example consider history $\widehat{H_2}$ (Figure 2). This history is causally consistent as all its read operations are legal. The history $\widehat{H_3}$ (Figure 3) is not causally consistent as the read operation $r_3(x)1$ issued by P_3 is not legal: $w_1(x)1 \rightarrow_H r_3(x)2 \rightarrow_H r_3(x)1$.

[8] More formally, $\widehat{H_i}$ is the sub-relation of $\widehat{H}$ *induced* by the set of all the writes of H and all the reads issued by P_i.

Said another way: when P_3 has issued its first read operation on x (namely $r(x)2$), it has got the value 2, and consequently for this process, the value 1 of x has logically been overwritten.

Actually, when considering read and write operations as equivalent to receive and send operations in message passing systems, causal consistency is equivalent, in the shared memory model, to causal ordering for the delivery of messages in the message passing model (other "similarities" between both models are addressed in Section 8).

15.5.2 Protocols

[5] presents an implementation of causal memory and studies programming with such a memory. It is shown that, when executed on a causally consistent memory, concurrent-write free[9] or data-race free[10] parallel programs behave as if the underlying shared memory was sequentially consistent. This is particularly interesting as these programs, intended to be executed on a sequentially consistent memory, remain correct when executed on a "less synchronized memory".

This approach has been generalized in [27] where a synchronization condition, called MSC, less restrictive than concurrent-write freeness or data-race freeness, is proposed. Informally, MSC introduces a new constraint called *concurrent-read freeness* and states a combination of (concurrent-write freeness, concurrent-read freeness and data-race freeness) constraints which allows concurrent writes on a same object while ensuring sequential consistency.

This condition can be used when executing, on a causally consistent memory, a program designed for a sequentially consistent memory. For the computation to be correct, it is only necessary to add on top of the causally consistent memory, a protocol implementing the MSC condition. Such a condition is interesting as it provides a layered approach to design a sequentially consistent memory: the library can contain (i) a first protocol implementing a causally consistent memory, and (ii) a second one implementing the MSC condition.

15.6 PRAM Consistency

PRAM (Pipelined RAM) consistency [21] is a consistency criterion weaker than causal consistency. The difference, in the shared memory model, between PRAM consistency and causal consistency is the same as the one between FIFO ordering and causal ordering for message deliveries in the message passing model. PRAM

[9] A program is *concurrent-write free* if, in all its executions, all its write operations are totally ordered.

[10] A program is *data-race free* if all accesses to each variable follow the readers-writer discipline [1].

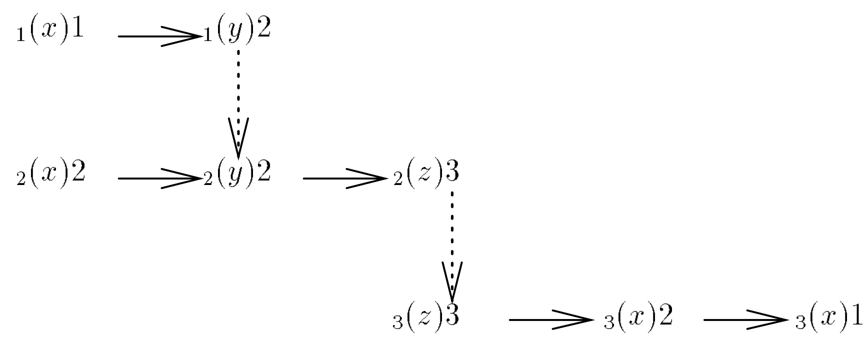


Figure 15.3 : A non causally (but PRAM) consistent history $\widehat{H_3}$

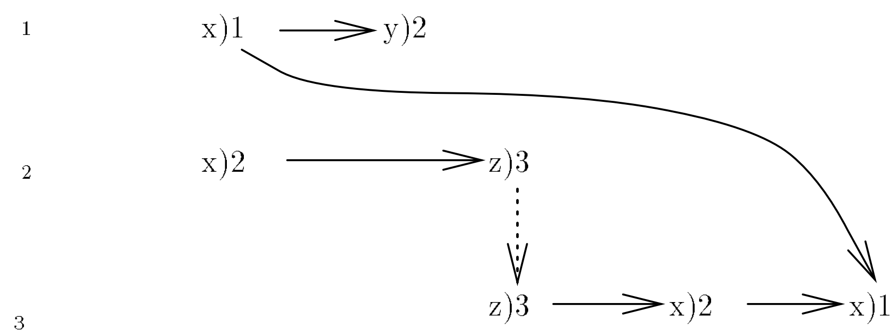


Figure 15.4 : The sub-history $\widehat{H_3'}$

and FIFO are only concerned by "direct relations" between pairs of "adjacents" processes and do not take into account transitivity due to intermediary processes. More precisely, in a message passing system with FIFO ordering, two messages sent to a same process by two distinct senders can be delivered in any order, even if the send events are causally related [17] (this is not the case with causal ordering: if the send events are causally related, messages must be delivered in their sending order to the destination process). In the same way, in a PRAM consistent shared memory system, two updates of objects by two distinct processes can be known in any order by a third one[11] (this is not the case in a causally consistent shared memory : if $w_k(x)u \rightarrow_H w_j(x)v$, a process P_i reading x can never get v and then

[11]Of course, only one of the updates can be known, if updates overwrite the value previously written by other processes.

u). As an example consider history $\widehat{H_3}$ which is not causally consistent (Figure 3). This history is PRAM consistent: as $w_1(x)1$ and $w_2(x)2$ have been issued by distinct processes, values 1 and 2 of x can be known by P_3 in any order.

Let $\widehat{H}$ be a history and let $\widehat{H'} = (H', \rightarrow_{H'})$ be a history defined from $\widehat{H}$ in the following way ($\widehat{H'}$ differs from $\widehat{H}$ only in point *iii* defining transitivity -see Section 2.2- where $\rightarrow_i$ is used instead of $\rightarrow_H$)[12]:

- $H' = H$ (so $H' = \bigcup_i h_i$)
- $op1 \rightarrow_{H'} op2$ if :

  i) $\exists\ P_i\ : op1 \rightarrow_i op2$ (*process-order* relation),

  or *ii)* $op1 = w_i(x)v$ and $op2 = r_j(x)v$ (*read-form* relation),

  or *iii)* $\exists op3\ : op1 \rightarrow_i op3$ and $op3 \rightarrow_i op2$.

Let $\widehat{H'_i}$ be the sub-history of $\widehat{H'}$ from which all read operations not issued by P_i have been removed (Figure 4 depicts the sub-history $\widehat{H'_3}$ associated with the computation $\widehat{H_3}$ described in Figure 3). $\widehat{H}$ is PRAM consistent if, for each P_i, all read operations of $\widehat{H'_i}$ are legal. This definition of PRAM consistency shows that its difference, with respect to causal consistency, lies only in the nature of the transitivity considered (point *iii* of their definitions).

15.7 Other Consistency Criteria

15.7.1 Mixed Consistency

Mixed consistency has been introduced by Agrawal *et al.* in [3]. This consistency criterion on one side considers histories including memory operations (read and write) and synchronization operations (lock, barrier and await), and on the other side combines PRAM consistency with causal consistency; namely every read operation is tagged either *PRAM* or *causal*.

A history $\widehat{H}$ is mixed consistent if it is:

- causally consistent when considering only the legality of read operations tagged *causal*, and
- PRAM consistent when considering only the legality of read operations tagged *PRAM*.

[12] As $(e \rightarrow_i f) \Rightarrow (e \rightarrow_H f)$, we have $\widehat{H'} \subset \widehat{H}$. When compared to $\widehat{H}$, some transitivity part of the causality relation are missing in $\widehat{H'}$.

The following result is shown in [3]. A mixed consistent history $\widehat{H}$ in which all read operations are tagged *causal*[13] and in which every pair of concurrent operations commute[14], is sequentially consistent.

15.7.2 Hybrid Consistency

Hybrid consistency has been introduced by Attiya and Friedman in [8]. This consistency criterion guarantees properties on the order in which operations appear to be executed at the program level. Operations are labeled either *strong* or *weak*. By defining which operations are strong and which are weak, a user can tune the consistency criterion to his own need. Informally hybrid consistency guarantees the following two properties:

- all strong operations appear to be executed in some sequential order,
- if two operations are invoked by the same process and at least one of them is strong, then they appear to be executed in their invocation order to all processes.

Hence all processes agree on a total order for all strong operations, and on the same order for any pair of strong and weak operations issued by the same process. They can disagree on the relative order of any pair of weak operations issued by a process between two strong operations. Let us consider the two following constraints:

- constraint SW: all writes are strong and all reads are weak.
- constraint SR: all writes are weak and all reads are strong.

The following result is proved in [9]: every hybrid consistent history that satisfies either the constraint SW or the constraint SR is sequentially consistent. When considering only constraint SW, this result is similar to the one implied by the concurrent-write freeness synchronization constraint (see Section 5.2): to get sequential consistency, SW and concurrent-write freeness order all write operations.

15.7.3 Non Primitive Read and Write Operations

Till now we have supposed that read and write operations offered to users are primitive operations. Some authors have considered to provide users with mechanisms allowing them to define non primitive read and write operations on a set of shared data objects (notation: READ, WRITE). Each such READ or WRITE operation is actually a procedure bracketed by two synchronization operations (*release*

[13]Note $\widehat{H}$ is then causally consistent.

[14]Two concurrent operations commute if their execution order is irrelevant (this is not the case for two concurrent writes on a same object).

and *acquire*). A non-primitive READ is composed of non synchronized primitive read operations while a non-primitive WRITE can include read and write primitive operations. Both *release* consistency [13] and *entry* consistency [12] address such non primitive READ and WRITE operations, and provide sequential consistency when acquire and release operations guarantee the readers-writers discipline. Concerning protocols implementing these consistency criteria, eager *vs* lazy [16] is an implementation issue whose aim is to reduce the number of messages and the amount of data exchanged; invalidation *vs* update [11] is another implementation issue addressing the management of multiple copies of objects when a cached-based approach is used.

15.8 Conclusion

Numerous protocols implementing distributed shared memory systems have been designed. In the most of them, the semantics (consistency criterion) they offer to users is defined only by the description of the protocol and not in an abstract way. This makes it difficult the study of their properties and the appreciation of their differences. In this paper we provided a suite of formal definitions for the most encountered consistency criteria, namely atomic, sequential, causal, release and entry consistencies. These definitions are not bound to particular implementations and are based on a unique framework. This, not only eases their understanding and their comparison, but should facilitate the design of a generic protocol which could be customized to the specific need of each user.

Last but not least, strong "similarities" between the shared memory model and the message passing model have been exhibited.

Bibliographie

[1] S.V. Adve and M.D. Hill. Weak ordering - a new definition. *Proc. 17th Annual ISCA*, pp.2-20, 1990.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM TOPLAS*, 15(1):182–205, 1993.

[3] D. Agrawal, M. Choy, H.V. Leong and A. Singh. Mixed consistency: a model for parallel programming. *In Proc. 13th ACM Symposium on PODC*, pages 101–110, 1994.

[4] M. Ahamad, J.E. Burns, P.W. Hutto, and G. Neiger. Causal memory. in *Proc. 5th Int. Workshop on Distributed Algorithms (WDAG-5)*, pages 9–30. Springer Verlag, LNCS 579, 1991.

[5] M. Ahamad, P.W. Hutto, G. Neiger, J.E. Burns, and P. Kohli. Causal memory: definitions, implementations and programming. *Distributed Computing*, 9:37-49, 1995.

[6] J.L. Archibald and J.L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM TOCS,* 4(4):276-298, 1986.

[7] H. Attiya and J.L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2):91–122, 1994.

[8] H. Attiya and R. Friedman. A correctness condition for high performance multiprocessors. in *Proc. 24th ACM Annual Symposium on the Theory of Computing*, pages 679–690, 1992.

[9] H. Attiya, S. Chaudhuri, Friedman R. and J.L. Welch. Shared memory consistency conditions for non sequential executions: definitions and programming strategies. in *Proc. 5th ACM SPAA*, july 1993.

[10] Bagrodia R. L. Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Languages and Systems*, 11(4):1053-1065, 1989.

[11] H.E. Bal, F. Kaashoek, A.S. Tanenbaum and J. Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency : Practice and Experience*, 4(5):337-355, 1992.

[12] B.N. Bershad, M.J. Zekauskas and W.A. Sawdon. The Midway distributed shared memory system. *Proc. of the Compcon 93 Conference*, pages 528–537, Feb. 1993.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessey. Memory consistency and event ordering in scalable shared memory multiprocessors. *Proc. 17th Annual ISCA*, Seattle, WA, pages 15–26, 1990.

[14] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–850, 1985.

[15] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[16] P. Keleher, A.L. Cox and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *Computer Architecture News*, 22(2):13–21, 1992.

[17] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE TC*, C28(9):690–691, 1979.

[19] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4),:321-359, 1989.

[20] W.G. Levelt, M.F. Kaashoek, H.E. Bal and A.S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software Practice and Experience*, 22(11):985-1010, 1992.

[21] R.J. Lipton and J.S. Sandberg. PRAM: a scalable shared memory. Tech. Report CS-TR-180-88, Princeton University, Sept. 1988.

[22] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM TOPLAS*, 8(1):142–153, 1986.

[23] M. Mizuno, M. Raynal, and J.Z. Zhou. Sequential Consistency in Distributed Systems. Springer-Verlag LNCS 938, pp.227-241, 1994.

[24] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52-60, 1991.

[25] C. Papadimitriou, The theory of concurrency control. *Computer Science Press*, 1986.

[26] J. Protic, M. Tomašević and V. Milutinović. A survey of distributed shared memory systems. *Proc. 28th Annual HICSS*, Vol. I, pp. 74-84, 1995.

[27] M. Raynal and A. Schiper. From causal consistency to sequential consistency in shared memory systems. *Proc. 15th Int. Conf. FST&TCS*, Bangalore, Sringer-Verlag LNCS Series 1026, pp.180-194, dec. 1995.

[28] R.H. Thomas. A majority consensus approach to concurrency control for multiple copies databases. *ACM TODS*, 4(2):180–209, 1979.

Chapitre 16

Conception et réalisation de Mémoires Virtuelles Partagées

Thierry Priol (IRISA/INRIA)

16.1 Introduction

La gestion des données dans une architecture parallèle à mémoire distribuée (APMD) est rendue complexe par la distribution physique des mémoires. Ce caractère distribué de la mémoire oblige l'utilisateur ou un compilateur "intelligent" a distribuer les données de l'algorithme devant s'exécuter en parallèle. Cette distribution des données est une opération complexe qui demande une très bonne connaissance de l'application à paralléliser. Dans le cas des applications régulières, la distribution des données peut être effectuée de manière statique sur des structures régulières (vecteurs, matrices). Cependant, cette opération est souvent une source d'erreurs car il est nécessaire de modifier les accès aux tableaux en utilisant des fonctions de conversion des indices globaux (algorithme original) en indices locaux (algorithme parallélisé). Dans le cas des applications irrégulières, manipulant soit des structures régulières, mais par des accès indirects, ou soit des structures irrégulières (liste, graphe,...), la distribution des données ne peut pas être effectuée de manière statique. Il faut donc prévoir des mécanismes de gestion de données permettant la migration de celles-ci en fonction des calculs effectués par chaque processeur. Ces mécanismes peuvent être de différentes natures en fonction de l'application. Nous nous intéressons ici au concept de mémoire virtuelle partagée (MVP) qui est un exemple de mécanisme de gestion de données. Un tel concept offre un espace d'adressage global pour une architecture parallèle ayant un ensemble d'espaces d'adressage disjoints (APMD). Nous avons expérimenté ce concept dans le cadre de la programmation des calculateurs parallèles. Les MVP KOAN[12] et MYOAN[3] ont été conçues respectivement pour l'iPSC/2 et le Paragon XP/S d'Intel. De nombreuses autres implémentations ont été proposées à l'aide de dispositifs matériels spécialisés ou bien par l'ajout de couches logiciels soit au sein du système d'exploitation ou soit comme un "middleware". On trouvera dans [16] une présentation des différentes approches.

Dans le cadre de cet article, nous nous intéressons à une forme particulière d'implémentation fondée sur l'utilisation des mécanismes de gestion mémoire virtuelle au sein ou au dessus d'un système d'exploitation. L'espace d'adressage global est une région de mémoire virtuelle composée de pages migrant à la demande entre les processeurs selon les accès mémoires. Chaque mémoire locale agit comme un large cache, ou mémoire attractive, contenant les pages précédemment accédées. Comme tout dispositif fondé sur l'utilisation de caches, le problème de la cohérence de ces caches se pose. L'objectif de cet article est de présenter les résultats les plus intéressants dans le domaine de la conception et de la réalisation de mécanismes de mémoire virtuelle partagée.

Dans le paragraphe 16.2, nous présentons un modèle de cohérence relâché qui est reconnu comme étant un des plus efficaces. Il s'agit du modèle de cohérence à la libération (*release consistency*). Nous illustrons ce modèle par une des toutes premières mises en œuvre: la MVP TreadMarks. Nous terminons cette présentation par quelques commentaires sur les améliorations possibles. Les paragraphes 16.3 et 16.4 présentent deux solutions pour améliorer la mise en œuvre de mémoire virtuelle partagée utilisant le modèle de cohérence relâché. La première solution, implémentée par la MVP Midway, permet d'éviter la comparaison explicite de pages pour localiser les modifications. La deuxième solution est fondée sur l'utilisation d'une nouvelle génération de réseau d'interconnexion haute-performance. Il s'agit de réseaux offrant la capacité d'adresser la mémoire des calculateurs à distance. Nous présentons deux technologies: Memory Channel de DEC et SCI de Dolphin Interconnect Solutions. Nous présentons la MVP Cashmere qui exploite la technologie Memory Channel. Enfin nous concluons en présentant quelques réflexions sur les perspectives associées au concept de MVP.

16.2 Modèle de cohérence à la libération

Le concept de MVP fournit une vision globale de la mémoire dans laquelle les calculateurs peuvent lire ou écrire. Vis à vis de l'utilisateur, il offre également un modèle mémoire qui caractérise le comportement de la mémoire lorsque plusieurs calculateurs effectuent des accès simultanés. De façon intuitive, l'utilisateur souhaite que la mémoire fournisse toujours le dernier résultat qui a été écrit dans la mémoire. Cependant, dans un système parallèle, la notion de "dernier accès" est ambiguë. Il oblige à définir un ordre total sur tous les accès mémoire, ce qui n'est pas souvent nécessaire. Le modèle de cohérence séquentielle est un exemple de modèle mémoire dont les accès sont consistants avec un ordre total. Un système mémoire possède la propriété de cohérence séquentielle si tous les processus voient les accès mémoire comme si ils avaient été exécutés sur un calculateur séquentiel multiprogrammé. Du point de la vue de la mise en œuvre d'une MVP, un tel modèle impose de nombreuses communications (accès aux pages, invalidation, etc.). Plusieurs travaux ont été réalisés afin de concevoir de nouveaux modèles mémoires pouvant être implémentés plus efficacement sur des systèmes parallèles. Parmi ceux-ci, le modèle de cohérence à la libération [4] a été un des plus étudiés. Nous en rappelons le principe dans le paragraphe suivant.

16.2.1 Principe

Le principe de ce modèle mémoire repose sur le constat que les accès aux données, effectués par un programme parallèle, sont souvent synchronisés. C'est le cas, par exemple, des codes de calcul scientifique. Il est rare que plusieurs processeurs effectuent des accès à une même donnée sans que ceux-ci soient synchronisés. Le modèle mémoire à consistance à la libération est fondé sur l'utilisation de deux classes d'opérations sur la mémoire. La première classe regroupe les opérations classiques de lecture et d'écriture tandis que la deuxième classe contient les opérations de synchronisation : libération et acquisition. Le rôle de ces deux opérations est de propager les modifications qui ont été réalisées par les opérations d'écriture. Une opération de libération indique qu'un processeur a effectué des modifications et que celles-ci doivent être communiquées à tout processeur qui effectuera une opération d'acquisition. De même, une opération d'acquisition indique qu'un processeur va exécuter des opérations qui nécessitent la connaissance des modifications effectuées par les processeurs ayant exécuté une opération de libération.

Deux formes de cohérence à la libération ont été proposées [8]. La première forme est appelée cohérence à la libération impatiente. Les modifications, réalisées depuis la dernière opération d'acquisition, sont propagées à tous les autres processeurs lors de la libération. Une des limites de cette forme de cohérence est l'obligation de diffusion des modifications à tous les processeurs même si certains n'ont pas besoin tout de suite de celles-ci. Elle nécessite un grand nombre de messages dans le réseau. La deuxième forme, appelée cohérence à la libération paresseuse, diffère de la précédente par le moment choisi pour diffuser les modifications. Plutôt que de le faire à la libération, les modifications sont propagées lors de l'opération d'acquisition. Lors de l'acquisition, le processeur détermine quelles sont les modifications valides dont il a besoin en fonction de la définition du modèle de cohérence à la libération. Cette approche permet ainsi de réduire fortement le nombre de messages

16.2.2 Contraintes de mise en œuvre

Le principe du modèle de cohérence à la libération semble intuitivement simple cependant sa mise en œuvre au sein d'une MVP est complexe. Tout d'abord, la mise en œuvre efficace d'une MVP s'appuie le plus souvent sur les mécanismes de gestion de la mémoire virtuelle. Ceci afin d'utiliser, de façon transparente, les opérations de lecture et d'écriture du processeur lors de l'accès à la MVP. L'unité d'accès à la MVP n'est pas le mot mémoire mais la page. Le système de MVP n'a pas le contrôle direct sur les écritures/lectures dans la page. Il est informé uniquement lorsqu'il se produit une violation des droits d'accès. La détermination des modifications effectuées entre une acquisition et une libération passe par l'utilisation d'une copie de page (page jumelle) puis par une comparaison de la page jumelle et de la page courante. Cette dernière technique est appelée "diffing". La création d'une page jumelle s'effectue lors de la première modification d'une page depuis l'exécution de la dernière opération de synchronisation. Lors de l'opération d'acquisition par un processeur, le droit d'accès des pages valides est en lecture uniquement. Ainsi, à la première écriture, une violation du droit d'accès entraîne l'exécution d'une routine qui effectue la création de la page jumelle. Le droit d'accès à la page courante est rétabli en mode lecture/écriture. Il n'y aura donc plus d'appel à la routine pour les écritures suivantes jusqu'à la prochaine opération de synchronisation.

16.2.3 La MVP TreadMarks

Une première mise en œuvre de la cohérence à la libération paresseuse a été effectuée au sein de la MVP TreadMarks [9, 7]. TreadMarks est une MVP implémentée au dessus du système d'exploitation. Elle s'appuie sur les mécanisme de gestion mémoire offert par Unix dont la projection de fichiers en mémoire virtuelle (*mmap*) et le contrôle sur les droits d'accès des pages (*mprotect*) constituant l'espace de mémoire virtuelle utilisée lors de la projection du fichier. Des routines sont activées lorsqu'il se produit une violation des droits d'accès aux pages ou bien lorsqu'il y a réception d'une requête envoyée par un processeur (accès aux pages, aux répertoires, etc...). Ce choix de conception à l'avantage d'offrir une grande portabilité. La MVP TreadMarks a été portée sur de nombreuses architectures (SUN, IBM, HP, SGI, DEC, ...).

L'originalité de TreadMarks provient de la mise en œuvre du modèle de cohérence à la libération paresseuse. Cette mise en œuvre se révèle d'une grande complexité. Lors de l'exécution d'une opération d'acquisition par un processeur, il doit demander, à celui ayant effectué la dernière libération, quelles sont les pages qu'il a modifiées ainsi que celles modifiées par tous les autres processeurs ayant effectuée auparavant une opération d'acquisition. La définition du "auparavant" s'appuie sur l'établissement d'un ordre partiel sur les accès à la mémoire partagée. Cet ordre partiel correspond à la fermeture transitive entre l'ordre des accès mémoires effectués au sein d'un même processus et l'ordre des synchronisations (acquisition/libération). Ainsi un accès a_1 est dit devant a_2 si a_1 apparaît avant a_2 dans l'ordre du programme. De même, si a_1 est une opération de libération au sein d'un processus p_1 et a_2 est une opération d'acquisition au sein du processus p_2 protégeant la même adresse mémoire et retournant la valeur écrite par a_1, alors a_1 est dit devant a_2. D'autre part, si a_1 est devant a_2 et a_2 est devant a_3 alors a_1 est devant a_3.

Plutôt que d'appliquer cet ordre partiel à tous les accès mémoire, ce qui serait coûteux à la fois en temps et en mémoire, les concepteurs de TreadMarks proposent de l'appliquer à des intervalles de temps entre deux opérations de synchronisation. Ceci permet de définir un ordre partiel non pas sur un seul accès mémoire mais sur un ensemble d'accès mémoire réalisés pendant un intervalle de temps. Chaque processeur maintient un vecteur d'horloge, fondé sur le principe des horloges de Mattern[14]. Ce vecteur permet à chaque processeur de connaître précisément les modifications à demander lors d'une opération d'acquisition. Si VV_p^i est le vecteur d'horloge du processeur p pour l'intervalle i, l'entrée $VV_p^i[q]$ indique au processeur p quel le dernier intervalle de temps du processeur q dont il a eu connaissance et par la même, quelles sont les modifications aux données partagées, réalisées par le processeur q qui ont déjà été effectuées par le processeur p. Ceci permet au processeur p de ne demander que les modifications qui ont été réalisées depuis cet intervalle de temps. La mise à jour des vecteurs d'horloge s'effectue lors de l'exécution des opérations de synchronisation (acquisition/libération).

Lorsqu'un processeur exécute une opération d'acquisition, il reçoit non pas les modifications effectuées par les autres processeurs mais simplement la liste des pages qui ont été modifiées. A partir de cette liste, le processeur ayant exécuté l'acquisition, invalide les pages. Cette invalidation ne consiste pas à détruire les pages mais simplement à provoquer l'exécution d'une routine lors d'un futur accès afin de demander les modifications au processeur ayant effectué l'opération de libération.

Plusieurs expérimentations ont été réalisées afin de comparer les performances de deux versions de codes parallèles [13] sur un réseau de stations de travail. Ces expérimentations exploitent le jeu de test SPLASH. La première version utilise l'échange de messages (PVM) tandis que la deuxième utilise TreadMarks. Les résultats montrent que les performances des codes parallèles qui utilisent TreadMarks sont inférieures de 10% par rapport aux versions qui utilisent PVM. Cette baisse de performance est tout à fait acceptable compte tenu de la simplicité d'utilisation d'une MVP par rapport à l'échange de messages.

16.2.4 Optimisations

Bien que les performances obtenues avec TreadMarks soient remarquables, de nombreuses optimisations peuvent permettre encore d'améliorer les performances des programmes utilisant une MVP et accroître ainsi le spectre des applications pour lesquelles une MVP peut être une alternative efficace à la programmation par échange de messages. Parmi les optimisations possibles, nous en avons relevé deux: l'élimination de l'opération de comparaison de pages et la réduction du nombre de serveurs présents dans chaque processeur pour servir les requêtes extérieures.

Nous présentons dans les paragraphes suivants deux solutions en les illustrant par des implémentations de MVP. La première solution montre une technique évitant la comparaison explicite de page grâce à une technique d'instrumentation de code. La deuxième solution est fondée sur l'utilisation d'un réseau d'interconnexion à capacité d'adressage.

16.3 Technique de “diffing” sans comparaison explicite

La mise en œuvre de MVP, qui implémentent un protocole de cohérence relâché (cohérence à la libération, multiples écrivains, ...), s'appuie sur l'utilisation des mécanismes de défaut de page (violation des droits d'accès) lors de l'écriture des pages. Le déclenchement de la routine de traitement de l'exception est une opération coûteuse. D'autre part, il est nécessaire d'utiliser, par la suite, une technique de comparaison de pages (diffing) pour localiser les modifications effectuées depuis la dernière opération de synchronisation. Cette comparaison de page génère un grand nombre de références mémoires interférant avec les données contenues dans les caches.

16.3.1 Principe

La comparaison de page peut être évitée si l'on sait déterminer ce qui a été modifié lors d'un accès en écriture dans l'espace d'adressage global. Une solution consiste à instrumenter le code du programme afin qu'à chaque opération d'écriture soit associée une mise à jour d'un bit d'état associé à chaque mot mémoire de l'espace d'adressage global. Si ce bit est positionné, le mot mémoire correspondant a été modifié. Cette technique a été proposée et expérimentée par les auteurs de la MVP Midway.

16.3.2 La MVP Midway

La MVP Midway [2] propose plusieurs modèles de cohérence dont notamment la cohérence à la libération. Cependant l'originalité de Midway réside dans la mise en œuvre du modèle de cohérence à l'entrée (*entry consistency*). Ce modèle consiste à propager les modifications des données partagées au moment de la synchronisation. Ce modèle fait les mêmes hypothèses que le celui à la libération: séparation entre les accès aux objets partagées de l'utilisateur et ceux aux objets de synchronisation.

Par rapport au modèle de cohérence à la libération impatiente, les modifications sont propagée lors de l'acquisition d'un objet de synchronisation et non lors de la libération, d'où son appellation cohérence à l'entrée. En fait, le modèle de cohérence à l'entrée est très proche conceptuellement de celui à la libération paresseuse. La différence réside dans l'association explicite d'objets de synchronisation (verrou, barrière) aux objets de l'utilisateur. Cette information supplémentaire est exploitée par la MVP pour en améliorer ses performances. Comme dans TreadMarks, Midway utilise un ordre partiel pour déterminer quelles sont les dernières modifications que doit demander un processeur lors de l'acquisition d'un objet de synchronisation. A chaque objet de synchronisation est associé une valeur d'horloge qui est incrémentée à chaque synchronisation avec un autre processeur. La valeur de l'horloge est communiquée au processeur destinataire qui met à jour sa propre horloge en prenant la valeur maximale entre son horloge et celle du processeur ayant envoyé le message.

Les concepteurs de Midway se sont confrontés au même problème que ceux de TreadMarks. Il est en effet nécessaire, dans une MVP utilisant à la fois un mécanisme de pagination et un modèle de cohérence relâché, de déterminer les modifications effectuées depuis la dernière opération de synchronisation. L'approche proposée dans Midway repose sur une instrumentation du code [17]. Cette technique permet d'ajouter des instructions dont le rôle est de modifier des bits d'états lors d'une écriture dans la MVP. Ces bits d'états sont utilisés par la suite afin de propager le contenu des adresses qui ont été modifiées entre deux opérations de synchronisation. A chaque écriture dans la MVP, des instructions supplémentaires sont exécutées. On peut donc penser qu'une telle technique est coûteuse. En fait, il faut la comparer au coût d'un traitement de défaut de page. Les expériences réalisées montrent que sur un processeur Mips R3000 (25 Mhz), avec le micro-noyau Mach et une taille de page de 4 Koctets, le coût d'un défaut de page est de l'ordre de 1200 micro-secondes (30000 cycles) alors que la mise à jour d'un bit d'état nécessite 0.360 micro-secondes (9 cycles). Même si tous les mots mémoire d'une page sont modifiés, la technique d'instrumentation est moins coûteuse que la technique fondée sur l'utilisation des mécanismes de protection de pages. Les expérimentations, réalisées à l'aide de jeux de test SPLASH, ont montré que la technique d'instrumentation de code permet de réduire les temps d'exécution de 10% à 80% selon les cas étudiés. Il faut cependant relativiser ces résultats car les expériences ont été réalisées avec un micro-noyau Mach qui est connu pour ses surcoûts dans la gestion de la mémoire virtuelle.

16.4 Impact des réseaux d'interconnexion à capacité d'adressage sur les MVP

Dans la plupart des MVP, chaque processeur est en charge de gérer un sous-ensemble des pages utilisées pour l'espace d'adressage global. Lorsqu'un processeur souhaite obtenir des informations sur une page, il s'adresse au processeur gestionnaire de celle-ci. Cette communication provoque l'exécution à distance d'une routine qui va répondre à la requête. Il y a interruption du processus utilisateur afin de permettre le traitement de la requête. Cette interruption est donc coûteuse en temps.

Une nouvelle technologie réseau, à capacité d'adressage à distance, émerge. Cette technologie devrait permettre d'interconnecter plusieurs dizaines voire milliers de calculateurs. La différence fondamentale avec les technologies classiques (FDDI, ATM, Gigabit Ethernet, Myrinet) est la possibilité d'écrire et/ou lire la mémoire d'un calculateur sans intervention d'une couche logicielle lors de la communication. La communication est entièrement prise en charge par le matériel. Ceci a une incidence immédiate sur une diminution de la latence et l'accroissement important du débit offert par ces réseaux. Nous appellerons ce type de réseau: réseau d'interconnexion à capacité d'adressage. Cette nouvelle technologie permet à la fois de simplifier et d'améliorer l'efficacité des MVP. En effet, elle permet de réduire le nombre de serveurs nécessaires à l'implémentation d'une MVP. Ces serveurs, sous forme de processus, répondent aux requêtes telles que la demande de page ou l'accès aux données des répertoires associés à la gestion des pages. Un simple accès par lecture permet d'obtenir l'information sans interrompre le processeur disposant de celle-ci.

Deux technologies sont actuellement disponibles. La première est une technologie propriétaire développée par DEC à partir de la technologie mémoire réflective[15] de la société Encore tandis que la deuxième est une mise en œuvre de SCI (Scalable Coherent Interface) qui est une norme adoptée par l'IEEE. A noter que pour ces deux technologies, la connexion au réseau s'effectue par des interfaces connectées au bus d'entrée/sortie (SBUS, PCI, ...). Cela ne nécessite pas l'accès au bus mémoire et permet ainsi une plus large diffusion de cette technologie. Les paragraphes suivants présentent les principales caractéristiques de ces deux technologies. Nous présentons ensuite la MVP Cashmere qui utilise le Memory Channel de DEC.

16.4.1 Interface Memory Channel de DEC

L'interface Memory Channel de DEC [5, 6] permet la projection, dans l'espace d'adressage d'un calculateur, d'une région mémoire d'un autre calculateur. Le réseau d'interconnexion se présente sous la forme d'un "hub" permettant la communication entre tous les calculateurs. Chaque interface est dotée d'une table de contrôle de pages (PCT) permettant d'associer un cadre de page dans l'espace mémoire physique d'un calculateur à un cadre de page dans l'espace mémoire physique d'un autre calculateur. Une région mémoire gérée par l'interface peut être de deux type: émission ou réception à la fois. Dans le cas d'une région mémoire de type émission, l'espace de mémoire virtuelle associé à cette région est projeté dans l'espace physique d'E/S associé à l'interface Memory Channel sur le bus PCI. Lorsque la région mémoire est de type réception, l'espace de mémoire virtuelle associé à cette région est projeté dans la mémoire physique du calculateur. Lors d'une écriture dans une région mémoire de type émission, l'interface Memory Channel intercepte l'écriture et la diffuse à toutes les interfaces Memory Channel des calculateurs ayant projeté cette région mémoire en mode réception. Ces interfaces effectuent par la suite la mise à jour de la mémoire locale par DMA. L'interface Memory Channel permet ainsi une communication (écriture) de type point à point ou par diffusion. La taille maximale de l'ensemble des régions est fixée à 512 Moctets (64k pages de 8 koctets). Pour permettre la projection de régions mémoire dans une architecture distribuée, chaque région possède un identificateur. La protection s'effectue au moment de la projection. Les expériences réalisées avec cette technologie ont permis d'obtenir une latence de l'ordre de 5 micro-secondes et un débit asymptotique de 30 Moctet/secondes.

16.4.2 Interface PCI/SCI de Dolphin

Tout comme la technologie Memory Channel de DEC, l'interface PCI/SCI de Dolphin Interconnect Solutions permet la projection de régions mémoire dans l'espace d'adressage d'un autre calculateur. Il s'agit d'une mise en œuvre de la norme IEEE P1596 sans le support de la cohérence de caches. Chaque interface dispose d'une table (ATT) de traduction d'adresses PCI (32 bits) vers SCI (64 bits). Cette table est le mécanisme de base pour la projection de régions mémoire. Un calculateur crée un segment de mémoire partagée alloué dans son espace physique. Les autres calculateurs peuvent alors projeter ce segment dans leur espace d'adressage virtuel. Dans ce cas, la région de mémoire virtuelle est projeté dans l'espace physique d'E/S associé à l'interface PCI/SCI. Une fois projeté, les calculateurs peuvent écrire ou lire à distance. Dans les deux cas, les données ne sont pas enregistrées dans les caches. Chaque accès sera alors transmis au calculateur ayant créé le segment mémoire. Cette approche est différente de celle proposée par l'interface Memory Channel. En effet, elle ne permet pas la diffusion d'une écriture vers plusieurs calculateurs. La diffusion d'une donnée peut cependant être réalisée en trois temps: écriture locale, synchronisation puis lecture par tous les calculateurs. Cette approche semble cependant moins performante que celle offerte par l'interface Memory Channel. L'interface PCI/SCI offre des mécanismes permettant de réduire le nombre de transaction sur le réseau en fonction des accès réalisés. Si un processeur effectue des accès contiguës à la mémoire, un système de tampon permet de regrouper ces accès mémoire en une seule transaction SCI au lieu d'une transaction par accès. Ceci permet d'obtenir un meilleur débit. La taille maximale de l'ensemble des régions créées ou projetées est fixé à 128 Moctets (256 pages de 512 koctets). Chaque région possède un identificateur utilisé lors de la création et la projection. Les expériences réalisées avec ces interfaces ont permis d'obtenir une latence de l'ordre 2.5 micro-secondes et un débit asymptotique de 45 Moctet/secondes. Ces résultats varient fortement en fonction des circuits PCI (chipset).

16.4.3 La MVP Cashmere

La MVP Cashmere [11, 10], tout comme TreadMarks, offre un modèle de cohérence à la libération paresseuse. Elle est implémentée sur un ensemble de machines DEC de type SMP (*Symmetric Multi-Processing*) c'est à dire disposant de plusieurs processeurs par nœud. Les nœuds sont interconnectés entre eux par l'interface Memory Channel. Cette interface permet une mise en œuvre efficace des communications au sein de la MVP Cashmere. Elle permet la synchronisation ainsi que la mise à jour des pages de l'espace d'adressage global et du répertoire donnant l'état des pages sans intervention de mécanismes logiciels.

Pour chaque page, de la MVP, accédée par un processeur, celui-ci possède une copie locale dans sa mémoire physique et une autre dans l'espace d'adressage de l'interface Memory Channel (MC). Cette page est contenue dans une région MC projetée dans l'espace d'adressage du processus en mode transmission. Elle correspond à la page originale gérée par le processeur propriétaire (*home node*) qui est elle même projeté en mode réception dans l'espace virtuel du processus s'exécutant sur ce processeur. La région MC projetée en mode réception, donc allouée dans la mémoire physique du processeur, contient plusieurs pages (superpage). Ce choix est en fait dicté par les limitations du nombre de régions MC pouvant être projetées. L'utilisation de la MVP Cashmere nécessite une instrumentation du code afin d'écrire à la fois dans la page allouée physiquement dans la mémoire locale et dans la page de la région MC projetée en mode transmission. Cette instrumentation est effectuée lors de la phase l'assemblage du code. Ceci permet au processeur propriétaire de la page de voir toutes les modifications effectuées par les processeurs. Il n'y a donc plus besoin d'utiliser une technique de comparaison de pages.

Le répertoire des pages est présent dans chaque nœud de la machine sous forme de projection d'une région MC en mode transmission et réception. Une modification du répertoire entraîne ainsi une diffusion de la modification à tous les nœuds de la machine. L'accès au répertoire s'effectue en section critique grâce à des verrous dont le coût est de l'ordre de 10 micro-secondes.

Chaque page de l'espace d'adressage global peut être dans trois états différents: non-caché, partagé ou inconsistant (*weak*). Une page est dans l'état non-caché si aucun des processeurs à un accès à cette page. Une page est dans l'état partagé si un ou plusieurs processeurs ont un accès à cette page mais n'ont pas effectué des modifications devant être visibles par tous les autres processeurs. Enfin l'état inconsistant caractérise une page accédée par plusieurs processeurs ayant modifié celle-ci et dont les modifications doivent être propagées lors d'une prochaine opération de libération.

Le protocole de Cashmere répond à quatre type d'événements: défaut de page en lecture ou en écriture, opération d'acquisition et opération de libération. Lors d'un défaut de page en lecture ou en écriture, le processeur en défaut accède au répertoire local en section critique, celui-ci étant constamment mis à jour par l'interface MC. Il change l'état de la page en mode partagé si l'état précédent était non-caché. Il alloue une page dans sa mémoire locale et obtient le contenu de la page du processeur propriétaire. Cette dernière opération correspond à une lecture à distance qui n'est pas supporter par l'interface MC. Les auteurs de Cashmere ont implémenté plusieurs mécanismes pour émuler une opération de lecture à distance: par interruption, par scrutation périodique du réseau ou bien utilisation d'un processeur d'un nœud SMP. Compte tenu du coût du traitement d'une interruption (1 milli-seconde), la technique de scrutation est celle qui donne la meilleure performance. La troisième technique, la plus simple, ne se révèle pas plus performante que celle par scrutation. Le défaut d'écriture est traité de la même façon qu'un défaut d'écriture. Elle s'accompagne d'une mise à jour locale d'une liste indiquant les pages modifiées depuis la dernière opération d'écriture.

Lors d'une opération d'acquisition, le processeur examine quelles sont les pages qui ont été modifiées par d'autres processeurs. Il parcourt une liste, accessible globalement, indiquant quelles sont les pages dans un état inconsistant. Il invalide localement ces pages et met à jour, par une écriture à distance, l'entrée du répertoire correspondant en s'assurant l'accès exclusif. Le prochain accès à la page provoquera un défaut de page qui sera traité comme indiqué précédemment. L'opération de libération consiste à informer les processeurs des modifications effectuée par le processeur exécutant cette opération de libération. Le processeur en défaut parcourt la liste contenant les pages modifiées qui est mise à jour lors du traitement d'un défaut en écriture. Pour chaque page, le processeur vérifie sont état en accédant au répertoire de manière exclusive. Si la page est en mode partagé, son état est changé en mode inconsistant et la page est ajoutée dans la liste des pages inconsistantes de chaque processeur ayant une copie de la page.

Les expériences réalisées avec des jeux tests provenant de SPLASH ont montré des performances équivalentes à celles obtenues avec une mise en œuvre de TreadMarks utilisant l'interface MC pour la communication par échange de messages. Les performances sont rarement meilleures pour plusieurs raisons: le coût de l'accès à une page qui s'effectue par l'activation à distance d'une routine. L'absence d'une opération de lecture à distance dans l'interface MC est donc pénalisante. La deuxième raison provient de la technique de la double écriture qui provoque des interférences dans le cache du processeur DEC 21064.

16.5 Perspectives et conclusions

Les évolutions récentes des calculateurs parallèles montrent bien l'intérêt croissant des utilisateurs au concept de MVP. En effet, les mécanismes matériels d'adressage global sont de plus en plus présents dans les nouveaux calculateurs parallèles tels que les HP/Convex Exemplar, SGI Origin, Data General Aviion ou Sequent NUMA-Q. Les calculateurs parallèles à mémoire distribuée ne proposant que l'échange de message ont tendance à disparaître au profit de ces nouvelles machines. Parallèlement à cette évolution, une autre est commence à émerger. Il s'agit d'utiliser les systèmes distribués, par exemple les réseaux de stations de travail (ou NOW en anglais), comme calculateurs parallèles. Cette évolution est fortement dictée par des impératifs économiques. Les réseaux de stations de travail équipent d'ores et déjà la plupart des sociétés ayant des besoins de calcul. L'utilisation d'une technologie d'interconnexion à capacité d'adressage peut transformer un tel réseau de stations de travail en un calculateur parallèle à mémoire logiquement partagée grâce à l'utilisation d'une MVP. Une initiative récente, OpenMP[1], soutenue par plusieurs acteurs industriels du calcul haute-performance, vise à standardiser un modèle de programmation fondé sur une mémoire partagée. Grâce à l'utilisation d'une MVP efficace, on peut envisager, dans un avenir proche, une compatibilité entre serveurs de calcul à mémoire partagée et réseaux de stations de travail munis d'une MVP. Cependant, cette approche n'est viable que si la perte de performance due à l'utilisation d'une MVP est "modérée". L'expérimentation des réseaux d'interconnexion à capacité d'adressage n'est qu'à son début. L'utilisation de réseaux, tel que SCI offrant à la fois la lecture et l'écriture à distance, peut permettre de nouvelles approches dans la conception de MVP et permettre ainsi à ce concept de se diffuser plus largement.

Bibliographie

[1] OpenMP: A Proposed Industry Standard API for Shared Memory Programming. – http://www.openmp.org.

[2] Bershad (B.), Zekauskas (M.) et Sawdon (W. A.). – The Midway Distributed Shared Memory System. *In : COMPCON 1993*.

[3] Cabillic (G.), Priol (T.) et Puaut (I.). – *MYOAN: an Implementation of the KOAN Shared Virtual Memory on the Intel Paragon.* – Rapport technique n° 812, IRISA, avril 1994.

[4] Gharachorloo (K.), Lenoski (D.), Laudon (J.), Gibbons (P.), Gupta (A.) et Henessy (J.). – Memory Consistency and event ordering in scalable shared memory multiprocessors. *In : 17th Annual International Symposium on Computer Architectures.* ACM, pp. 15–26.

[5] Gillet (R.). – *Memory Channel: An Optimized Cluster Interconnect.* – Technical journal, Digital, automne 1995.

[6] Gillett (R.). – Memory Channel Network for PCI. *IEEE Micro*, vol. 16, n° 2, février 1996.

[7] Keleher (P.). – *Lazy Release Consistency for Distributed Shared Memory.* – Thèse de PhD, Rice University, janvier 1995.

[8] Keleher (P.), Cox (A.) et Zwaenepoel (W.). – Lazy Release Consistency for Software Distributed Shared Memory. *In : 19th International Symposium on Computer Architecture*, pp. 13–21.

[9] Keleher (P.), Dwarkadas (D.), Cox (A.) et Zwaenepoel (W.). – TreadMarks: Distributed Shared Memory on standard workstations and operating systems. *In : Proceedings of the 1994 Winter Usenix Conference*, pp. 115–131.

[10] Kontothanassis (L.), Hunt (G.), Stets (R.), Hardavellas (N.), Cierniak (M.), Parthasarathy (S.), Jr. (W. M.), Dwarkadas (S.) et Scott (M.). – *VM-Based Shared Memory on Low-Latency, Remote-Memory-Acess Networks.* – Technical Report n° 643, University of Rochester, novembre 1996.

[11] Kontothanassis (L.) et Scott (M.). – *Distributed Shared Memory for New Generation Networks.* – Technical Report n° 578, University of Rochester, mars 1995.

[12] Lahjomri (Z.) et Priol (T.). – KOAN: a Shared Virtual Memory for the iPSC/2 hypercube. *In : CONPAR/VAPP92.*

[13] Lu (H.), Dwarkadas (S.), Cox (A.) et Zwaenepoel (W.). – Message Passing Versus Distributed Shared Memory on Networks of Workstations. *In : Supercomputing '95.*

[14] Mattern (F.). – Virtual Time and Global States of Distributed Systems. *In : Parallel & Distributed Algorithms*, éd. par Cosnard (M.), Robert (Y.), Quinton (P.) et Raynal (M.). pp. 215–226. – Elsevier Science Publishers.

[15] Meares (G.). – Reflective Memory Network. *Real-Time Magazine*, février 1996.

[16] Priol (T.). – Mémoire Virtuellement Partagée pour le calcul de haute performance : évolutions et tendances. *TSI*, vol. a paraître, 1997.

[17] Zekauskas (M.), Sawdon (W.) et Bershad (B.). – Software Write Detection for a Distributed Shared Memory. *In : First USENIX Symposium on Operating System Design and Implementation*, pp. 87–100.

Chapitre 17

Utilisation de structures de données parallèles dans des applications

Bertrand Le cun (PR*i*SM-Université de Versailles)

17.1 Introduction

L'objet de cet article est de présenter deux travaux de parallélisation d'algorithmes de résolution de problèmes combinatoires NP-Difficiles : A* et la résolution du problème de découpe à 2 dimensions. Hormis l'utilisation d'un parcours d'espace de recherche, ils ont en commun une spécificité qui les différencient des classiques Branch and Bound. En effet, ils utilisent non pas une seule structure de données complexe (la file de priorité pour le Branch and Bound) mais deux structures de données complexes : une file de priorité et une structure de recherche.

Cette spécificité rend la parallélisation difficile et donc intéressante.

17.2 La parallélisation de l'algorithme A*

Ce travail est un travail joint avec Van-Dat Cung.

L'algorithme A* [11, 12] effectue une recherche d'une solution optimale avec une stratégie meilleur d'abord dans un espace d'états. L'espace d'états est représenté par un graphe noté $G = (V, E, C)$, où

E est l'ensemble des arcs du graphe représentant les transitions possibles entre états,

c_{ij} représente le coût du passage de l'état v_i à l'état v_j,

V est l'ensemble des états/sommets.

Trouver une solution optimale du problème est équivalent à trouver le chemin minimal d'un état initial v_0 à un état final v_n.

Dans la suite, nous confondons les termes état, et sommet du graphe.

Chaque sommet a une valeur assignée, calculée par une fonction d'évaluation noté f. Pour un sommet v_i, elle vaut :

$$f(v_i) = g(v_i) + h(v_i)$$

où g est la valeur du chemin de v_0 à v_i, et h est une fonction heuristique évaluant le chemin de v_i à v_n.

L'algorithme A* utilise le même schéma d'exécution que l'algorithme Branch and Bound avec une stratégie meilleur d'abord. A chaque pas de l'algorithme, le sommet ayant la meilleure valeur de h est sélectionné pour être exploré.

17.2.1 Problématique des structures de données pour l'algorithme

La structure utilisée pour stocker les sommets non explorés est appelée la *liste ouverte*.

Si l'espace de recherche est un arbre, la *liste ouverte* est une simple file de priorité. L'algorithme A* est dans ce cas équivalent au Branch and Bound meilleur d'abord.

En revanche, si l'espace de recherche est un graphe, chaque sommet peut être référencé par une clef, représentant la configuration de l'état. Nous notons $k(v_i)$ la configuration de l'état v_i.

Dans ce cas, plusieurs chemins peuvent exister d'un sommet v_i à un sommet v_j Le sommet v_j peut donc être exploré plusieurs fois, des parcours redondants de l'espace de recherche peuvent donc être effectués. Pour les éviter, l'algorithme doit gérer l'historique de la recherche. Une structure appelée liste *fermée* stocke tous les sommets déjà explorés.

Un nouveau sommet v_i généré est inséré dans la liste *ouverte*, s'il n'existe pas un élément de clef égale à v_i, ou si un tel élément x existe mais avec une priorité inférieure à celle de v_i. Dans ce dernier cas, v_i est inséré et x doit être supprimé.

La sémantique des opérations de la *liste ouverte* est celle des files de priorité mais la contrainte d'unicité de clef doit être pris en compte.

Le structure doit donc gérer le **double critère** (priorité, clef).

Beaucoup des parallélisations de l'algorithme A* présentées dans la littérature considèrent qu'un espace de recherche sous forme d'arbre, réduisant le problème de la parallélisation de l'algorithme A* à celui d'un Branch and Bound [13].

Toutefois, un précédent travail [4] sur la recherche dans un graphe par l'algorithme A* en parallèle, implémente la file ouverte par deux structures distinctes (une files de priorité et une structure de recherche avec des références croisées).

Une parallélisation tendant à rendre les opérations concurrentes étaient très difficile. En effet, pour interdire les interblocages, une des deux structures devait être accédée de manière exclusive.

Nous montrons qu'il est intéressant d'utiliser une version modifiée de la Treap (Tree-Heap) pour implémenter la liste ouverte.

17.2.2 La Treap

La gestion de la file ouverte, est donc typiquement celle d'une structure de données à double critère. La file ouverte doit offrir les opérations d'une file de priorité avec une contrainte d'unicité de clef.

Nous proposons d'utiliser une version modifiée d'une structure appelée Treap.

17.2.2.a Représentation et algorithme des opérations séquentielles

La représentation de la treap est un arbre binaire, où chaque nœud x stocke un élément qui contient deux informations : une priorité, une clef.

L'arbre est tel que :

1. les clefs respectent l'invariant des arbres binaires de recherche.

2. les priorités respectent l'invariant des tas.

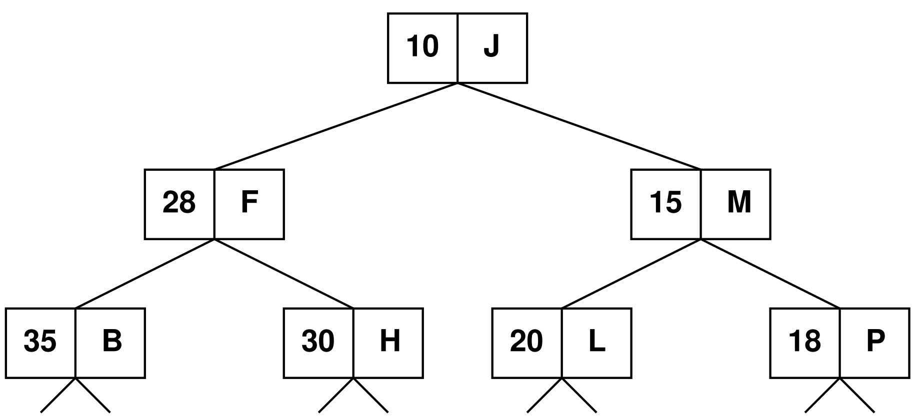


Figure 17.1 : Un exemple de Treap

Dans la suite, nous confondons un nœud et l'élément qu'il contient. Il est clair que, quelques soient les valeurs des couples (priorité,clef) un tel arbre existe (vois la figure 17.1. Mais, la disposition des nœuds dépend de ces valeurs. Si la valeur des couples fait que les nœuds ont le même ordre selon leur priorité et leur clef, l'arbre est réduit à une simple liste.

L'ordre induit par les priorités et les clefs définissent la longueur des branches et par suite l'équilibrage de l'arbre. Nous reviendrons sur ce problème, après avoir expliqué les sémantiques des opérations sur la *file ouverte* de l'algorithme A*.

Rechercher(clef) cherche un nœud x dans la treap tel que la clef de x est égal à clef. Le nœud est recherché comme dans un arbre binaire classique, en suivant les valeurs de clefs des nœuds rencontrés.

Insérer(x) ajoute un nœud x dans la treap, en s'assurant que x est unique.

Soit x' un nœud de même clef que x, existant dans la treap. si x' est moins prioritaire que x,: x est inséré et x' est supprimé, sinon x n'est pas ajouté.

Supprimer(c) retire l'unique nœud x de clef c.

SupprimerMin() supprime l'élément contenu dans la racine de la treap.

Le détail des algorithmes des opérations peut être trouvé dans [3]. Brièvement, les algorithmes utilisés sont dérivés des splay-tree. L'intérêt principal est que toutes ces opérations ne portent maintenant que sur un seule arbre, rendant ainsi applicables les techniques de verrouillage partiel présentées dans [9].

Comme nous l'avons dit, l'équilibrage de l'arbre dépend des valeurs des couples (clef, priorité). Il est donc clair que la complexité des opérations en dépend aussi.

Aragon et Seidel [1] ont proposé une treap pour implémenter une structure de recherche où la valeur de priorité de chacun des éléments est utilisée pour garder l'arbre équilibré. Ils proposent de choisir aléatoirement la valeur de priorité de chacun des nœuds. Ils démontrent que si ces valeurs sont indépendantes et équitablement distribuées, la profondeur de l'arbre est $O(\log\ n)$

En fait, il faut que l'ordre des priorités soient indépendants de celui des clefs. La valeur de priorité ne peut être changée. En revanche, l'ordre des clef peut être modifié par une fonction s'appliquant sur la configuration de l'état.

17.2.3 Expérimentations

Le problème pris comme exemple, est le jeu du taquin (15-puzzle). Ce jeu est considéré comme le problème test pour l'algorithme A*. Le jeu de taille 15 est une grille de 4x4 emplacements possibles, contenant quinze tuiles et un emplacement vide (le trou).

Chaque tuile est numérotée, une permutation des tuiles sur les emplacements, est appelée configuration. Le but du jeu est d'obtenir, à partir d'une configuration initiale, une configuration finale, en faisant glisser les tuiles.

L'algorithme A* est utilisé pour trouver le minimum de glissements, (appelés mouvements), pour atteindre la configuration finale.

La clef est la configuration du jeu. La valeur de g est le nombre de mouvement déjà effectués. Nous avons utilisé comme fonction h, la somme des distances de Manhattan pour chacune des tuiles, entre sa position actuelle dans la grille et sa position finale.

Korf [8] a donné une liste d'instances de problèmes. Nous en avons choisi quatre de tailles différentes. Le nombre d'états engendrés varie approximativement de 60 milles (Korf12) à 1,5 million (Korf78). La longueur du chemin de l'état initial à un état final sont respectivement de 45 (Korf12), 41 (Korf55), 53 (Korf78 et Korf94). Les accélérations sont illustrées par la figure 17.2.

L'allure générale des courbes montre que de bonnes performances sont obtenues avec peu de processeurs, mais plus le nombre de processeurs augmente plus les performances se dégradent.

La génération et l'évaluation des états fils d'un état nécessitent pour ce problème très peu de temps par rapport au temps d'accès à la treap. L'arbre de recherche développé au cours de l'exploration a un facteur de branchement très faible, chaque état n'ayant que trois états fils possibles. Ceci explique pourquoi les accélérations ne sont pas linéaires par rapport au nombre de processeurs utilisés. De meilleures accélérations pourraient être obtenues si les calculs pour l'exploration d'un état étaient plus importants, par exemple plus d'états fils à engendrer et plus de temps pour calculer leur évaluation. Rao et al. [14] ont mis en évidence expérimentalement que les accélérations de ce type de parallélisation centralisée sont meilleures quand l'arbre de recherche développé a un facteur de branchement b plus important impliquant plus de calculs locaux.

Les différences d'accélération entre les instances sont dues à la différences de taille des instances. L'instance Korf78 induit une arborescemce plus importante que les autres.

D'autre part, les tests ont été effectués sur une ksr1, qui simule une mémoire partagée au dessus d'une mémoire distribuée. Les temps de transferts des données entre processus justifient d'autant plus cette dégradation.

Nous observons une anomalie d'accélération sur-linéaire pour l'instance Korf78 lorsque 2 ou 3 processeurs sont employés. Cela est provoqué par le fait qu'un état final peut être trouvé plus rapidement avec plusieurs processeurs qu'avec un seul. Des études détaillées de ce phénomènes peuvent être trouvées dans [15, 10].

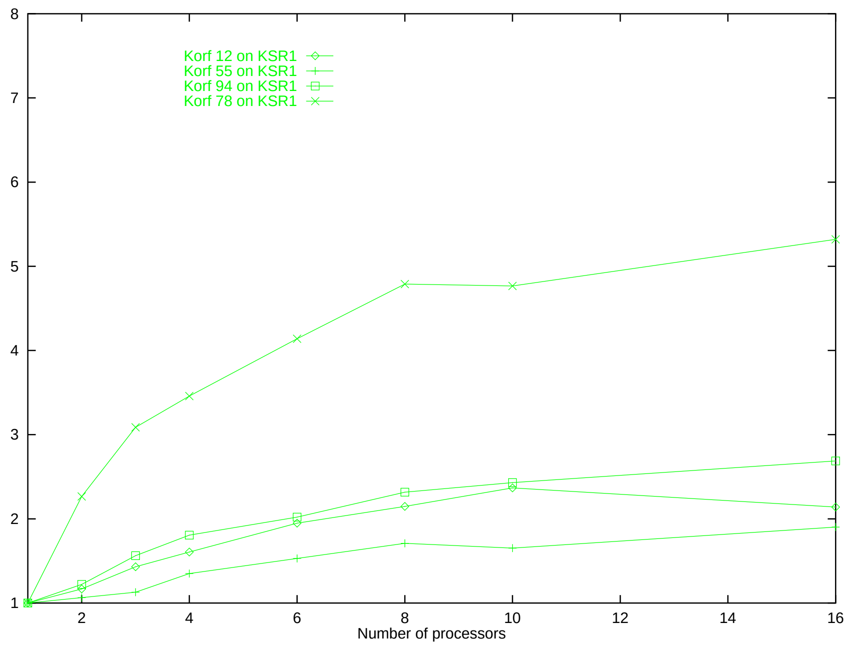


Figure 17.2 : Taquin: Accélérations sur la KSR1.

17.3 La parallélisation du 2-D CSP

Ce travail est un travail joint avec Mahnd Hifi et Van-Dat Cung.

17.3.1 Introduction

Le problème de découpe est un problème NP-complet avec de nombreuses applications en industrie, en systèmes multiprogrammés et multiprocesseurs, en placement de circuits intégrés.

Nous nous intéressons au cas où l'on veut optimiser l'utilisation d'une entité disponible en taille limitée en y plaçant des sous-entités prédéterminées. On considère que l'entité disponible est donnée sous forme rectangulaire de dimensions fixées, qu'on appelle rectangle initial. Les sous-entités ont aussi des formes rectangulaires qu'on appelle pièces. On se propose de découper le rectangle initial en pièces appartenant à l'ensemble des pièces disponibles $\mathcal{S}$ tout en *minimisant* la surface des pièces produites n'appartenant pas à cet ensemble, appelée *chute*, ou bien en maximisant le profit des pièces placées dans la plaque initiale. Une politique de choix sur le matériel de découpe est indispensable. Dans notre étude, nous supposons qu'on utilise des découpes du type *guillotine* qui consiste à prendre la découpe d'un côté à son opposé parallèlement aux deux autres (voir figure 17.3).

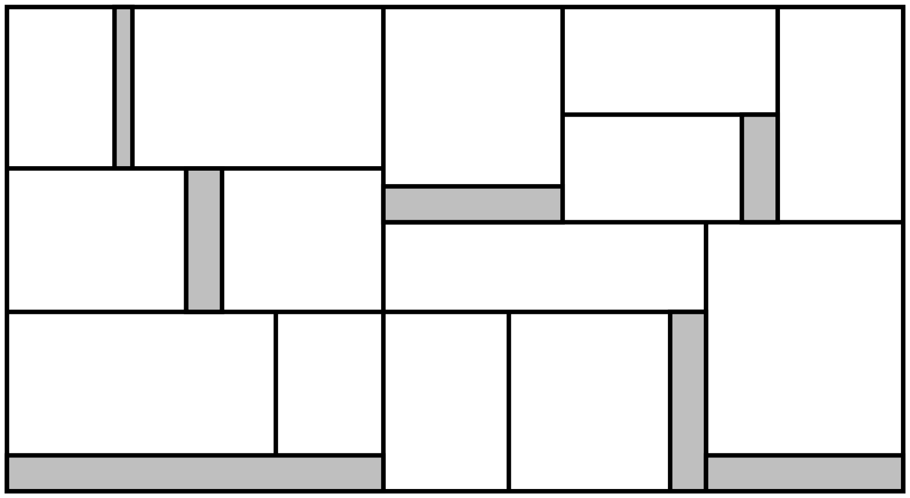


Figure 17.3 : Solution à un problème de découpe guillotin à deux dimensions

17.3.2 Problème de découpe contraint

Une instance du problème de découpe contraint à deux dimensions est définie par le quadruplet $(\mathcal{R}, \mathcal{S}, c, b)$. $\mathcal{R}$ représente le rectangle initial, $\mathcal{S}$ l'ensemble des pièces, c le vecteur profit associé aux pièces et $b = (b_i)_{i=1,\ldots,n}$ une borne supérieure pour le nombre d'occurrences de chaque type de pièces dans un plan de découpe réalisable. Ainsi, pour plus de clarté, ce problème peut être formulé comme le programme suivant:

$$\begin{cases} F = & \max \sum_{i=1}^{n} c_i \xi_i \\ & \xi_i \leq b_i,\ \xi_i \text{ entier pour } i = 1, \ldots, n,\ \xi \in \mathcal{P} \end{cases}$$

où ξ_i est le nombre d'occurrences de la pièce du type i dans le plan de découpe réalisable $\xi \in \mathcal{P}$ contraint par la borne supérieure b_i.

De la même manière, une solution optimale pour ce problème est représentée par le couple (ξ^*, F^*), et telle que ξ^* donne le meilleur plan de découpe de valeur *maximum* F^* sous les contraintes $\xi_i \leq b_i$, pour $i = 1, \ldots, n$.

17.3.3 Algorithme de résolution

Viswanathan et Bagchi [17] ont proposés une méthode de résolution fondée sur les méthodes de séparation et d'évaluation. Cette méthode consacre un temps considérable pour l'évaluation d'une fonction donnant une solution pour le problème de découpe contraint. La méthode consiste dans l'utilisation de toutes les possibilités de découpes guillotine sur le rectangle initial qui peuvent être obtenues par la reconnaissance des découpes verticales et horizontales. Afin de mieux comprendre les différentes constructions utilisées, nous donnons les définitions suivantes pour la construction des différents blocs formants des plans de découpe réalisables.

Définitions:

1. On dit qu'un plan de découpe forme une *construction horizontale* si la construction des deux rectangles (x_1, y_1) et (x_2, y_2) engendre un rectangle de dimensions $(x_1 + x_2, \max\{y_1, y_2\})$.

2. Un plan de découpe est une *construction verticale* si le regroupement de deux rectangles (x_1, y_1) et (x_2, y_2) donne un rectangle résultant $(\max\{x_1, x_2\}, y_1 + y_2)$.

3. On dit que le rectangle résultant (des deux constructions précédentes) est un *rectangle guillotin* si les contraintes du plan de découpe adéquat sont satisfaites.

Nous exposons dans les paragraphes suivants les principes des composantes de notre branch and bound qui sont en grande partie des dériviés de l'algorithme de Viswanathan et Bagchi : la borne inférieure (valeur de la meilleur solution connue, la borne supérieure (évaluation d'un sous-problème) et le critère de séparation.

17.3.3..1 La borne inférieure Le 2-D CSP est un problème de maximisation, la borne inférieure représente donc la valeur de la meilleur solution connue. La méthode de la calcul de cette borne consiste en la décomposition du rectangle initiale en sous-bandes optimales *verticales* et *horizontales*, tout en respectant les contraintes sur le pièces initiales. Ces sous-bandes sont calculées par Programmation dynamique. Dans chaque sous-bandes, un knapsac est réalisé [7, 6].

17.3.3..2 La borne supérieure La borne supérieure de l'algorithme pour un rectangle R, est donnée par le coût réèl de R (fonction $g(R)$) plus une évaluation du coût de la surface restante (fonction $h(\mathrm{R})$).. L'algorithme décrit par Viswanathan et Bagchi, calcule $h(R)$ en s'appuyant principalement sur la résolution du problème de découpe sans contraintes à deux dimensions. Cette valeur est obtenue par la suppression des bornes de capacité sur les pièces. Cette résolution est donnée par l'application de la procédure de Gilmore et Gomory [5] qui consiste à résoudre le problème sur la plaque initiale (L, H) par la programmation dynamique, et par la suite on récupère la valeur optimale d'un sous-rectangle représentant un nœud intérieur à l'arborescence créée.

L'algorithme de Viswanathan et Bagchi [17] utilise ces valeurs comme étant la borne supérieure à chaque nœud de l'arborescence, avec une petite amélioration. Nous utilisons une deuxième borne supérieure donnée par la résolution du problème de sac dos unidimensionnel borné suivant:

$$(K_u)\begin{cases} \max & \sum_{j\in R_{\alpha\beta}} c_j x_j \\ \text{s.c.} & \sum_{j\in R_{\alpha\beta}} (l_j h_j) x_j \leq (\alpha\beta) \\ & x_j \leq \min\{b_j, \lfloor \frac{\alpha}{l_j} \rfloor \lfloor \frac{\beta}{h_j} \rfloor\},\ j \in R_{\alpha\beta} \end{cases}$$

où $R_{\alpha\beta}$ est l'ensemble des pièces qui rentrent dans le (sous-)rectangle (α, β), x_j désigne le nombre d'apparitions de la $j^{ème}$ pièce dans le (sous-)rectangle (α, β). Par ailleurs, la résolution du problème (K_u), en utilisant les techniques de la programmation dynamique et en posant $\alpha = L$ et $\beta = H$, fournit toutes les solutions optimales du problème pour une longueur $\alpha \leq L$ et une hauteur $\beta \leq H$.
Notons que cette représentation considère que le rectangle initial est donné sous forme d'une bande de longueur $\alpha\beta$ et de hauteur 1, et chaque pièce, pour $i = 1, ..., n$, est représentée sur une longueur $l_i h_i$ et de hauteur 1. Nous notons la valeur de la solution du problème (K_u) par V^{sup}, ainsi cette valeur sera comparée à chaque nœud à la valeur supérieure donnée par la procédure de Gilmore et Gomory. En d'autres termes, nous gardons toujours la plus petite valeur entre les deux et elle sera considérée comme borne supérieure pour notre algorithme.

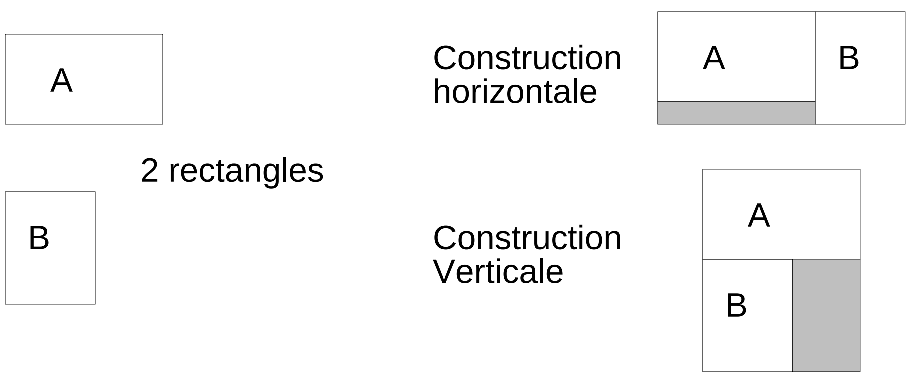


Figure 17.4 : Construction pour le 2D-CSP

17.3.3..3 Critère de séparation La critère de séparation de l'algorithme de Viswanathan et Bagchi se fonde sur l'utilisation de deux listes respectivement notées $Ouverte$ et $Fermée$. La liste $Ouverte$ comporte, au départ, une copie de chaque type de pièces de la liste $\mathcal{S}$. La liste $Fermée$ est initialement vide. A chaque étape de l'algorithme, un rectangle R_o de valeur f maximum est choisi de la liste $Ouverte$. Ce rectangle est immédiatement transféré de la liste $Ouverte$ vers la deuxième liste $Fermée$. Par la suite, on construit les nouveaux rectangles par constructions verticales et horizontales (voir figure 17.4) obtenues en combinant le rectangle R_o avec tout les rectangles de la liste *Fermée* (y compris le rectangle R_o). Chaque rectangle R résultant est inséré dans *Ouvert* s'il vérifie les conditions suivantes

1. les dimensions du rectangle ne dépassent pas les dimensions du rectangle initial i.e. si $R = (l_R, h_R) > (L, H)$ alors il est évident que R ne peut être introduit dans le rectangle initial.

2. Le rectangle R ne viole aucune des contraintes sur les bornes de la liste des pièces.

3. L'évaluation est supérieure à la valeur de meilleur solution connue.

L'algorithme s'arrête lorsque le rectangle R choisi dans l'ensemble *Ouvert*, réalise la valeur $h(R) = 0$. Dans ce cas, la valeur intermédiaire $g(R)$ réalise la valeur de la solution optimale dont la composition est donnée par le rectangle guillotine R.

17.3.4 Les sous-problèmes symétriques

Cet algorithme ou plutôt le critère de séparation, génère beaucoup de sous-problèmes symétriques.

Par exemple, un rectangle qui a été formé par la construction verticale de deux sous-rectangles A et B, où A (resp: B) est le résultat d'une construction horizontale de deux pièces identiques a (resp: b), est équivalent à un rectangle qui est résultat de la construction horizontale de deux rectangles identiques A', ce dernier étant une construction verticale de a et b. Même si les places prises par les pièces dans les rectangles ne sont pas tout à fait les mêmes, les tailles des rectangles ainsi que leurs valeurs sont égales.

Une façon de gérer le problème est de faire une recherche dans la liste fermée à chaque génération de nœud [16]. La complexité devient donc exponentiel en la taille du problème, car d'une part le nombre d'éléments de la liste fermée croit exponentiellement avec l'algorithme, et d'autre part le nombre de recherche dans cette liste est aussi exponentiel.

Nous proposons trois permettant d'interdire la plus part des constructions symétriques.

1. Les symétries élagantes.

   Ce très simple test permet de ne pas générer des éléments qui ne seront pas utilisés plus tard. En effet, la fonction de génération peut induire des rectangles contenant des chutes. C'est à dire que dans un rectangle généré des zones non utilisées existent. Ce test consiste à verifier s'il n'existe pas de pièces initiales respectant les contraintes, et qui soit plus petite, que la chute du rectangle considéré (voir figure 17.5).

Figure 17.5 : Symétrie élagante

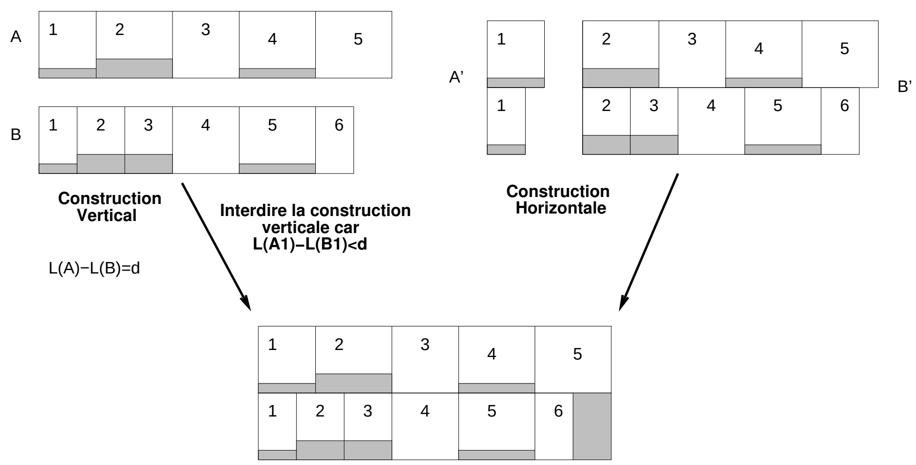


Figure 17.6 : Symétrie de sens opposé

2. Les symétries par opposition de sens.

   Ce type de symétrie peut être formulé comme suit (voir figure 17.6) : Soit A un rectangle résultant d'une construction horizontal des rectangles $A_1, A_2, ...A_n$ et un rectangle B résultant d'une construction également horizontal des rectangles $B_1, B_2, ...B_m$. On pose $d = |A| - |B|$ (nous considérons que $d \leq 0$) où $|A_i|$ (resp: $|B_i|$) la longueur du rectangle A_i (resp: B_i).

   La construction verticale des rectangles A et B est une construction symétrique si il existe un vecteur x_i et un vecteur y_i tel que

$$\sum_i^n |A_i| x_i - \sum_j^m |B_j| y_j \leq d$$

   s.c. $\sum_i^n x_i < n \; \sum_i^n y_i \leq m \; x_i = 0,1 \; y_i = 0,1$

   Complexité du problème: Résoudre ce problème exactement consiste à résoudre $(n-1)!$ knapsac successifs avec les rectangles B_i avec comme taille limites celles des $(n-1)!$ combinaisons linéaires des rectangles A_i. Pour chaque knapsac, nous devons vérifier si la différence entre la valeur trouvée et la limite est inférieure à d.

Ce calcul est très long à effectuer, nous proposons donc une heuristique très simple, qui consiste à rechercher un A_i et un B_j tels que $0 \leq |A_i| - |B_j| \leq d$.

3. Les symétries de même sens.

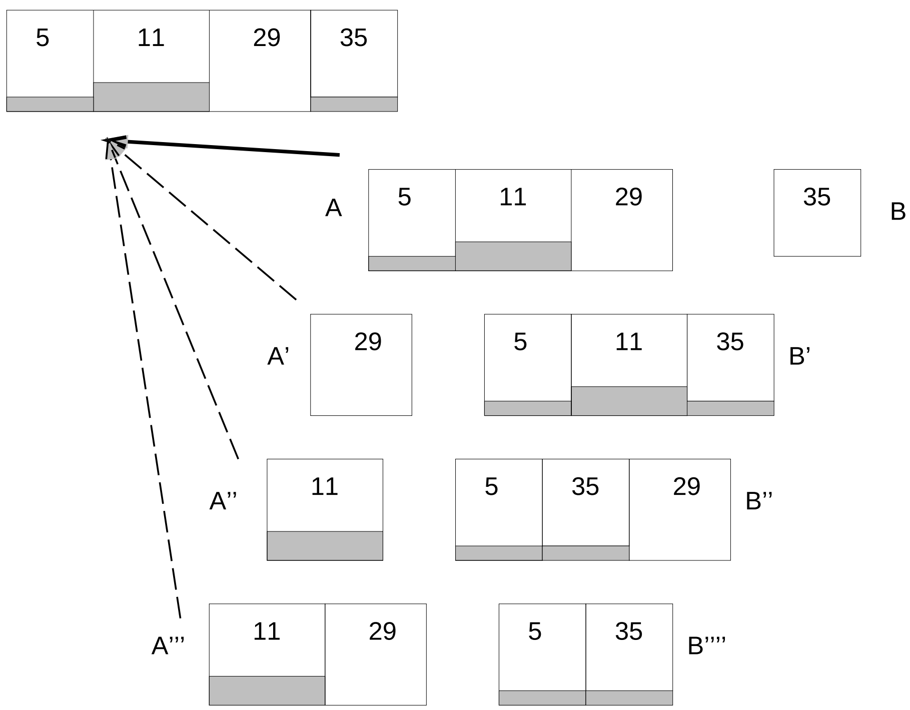


Figure 17.7 : Symétrie de même sens

Ce test de symétrie veut interdire la construction de rectangles formées par construction horizontale ou verticales de plusieurs sous-rectangles (voir figure 17.7). Pour un ensemble de sous-rectangle donnée, une seule construction doit être acceptée. Dans la suite, nous ne présentons que le cas horizontal, le cas vertical se faisant de même. Nous posons $E_{diff}(A)$ l'ensemble des identificateurs des rectangles distinctes qui par construction horizontale forme A et $N_{diff}(A)$ leur nombre. Dans le cas, où $N_{diff}(A) = 1$, nous posons $N_{id}(A)$ le nombre de rectangles identiques formant A.

La construction horizontale de A provenant de la liste ouverte et B est autorisé dans les deux cas suivants :

(a) si $N_{diff}(A) = 1$ et $N_{diff}(B) = 1$ et $L_{diff}(A) = L_{diff}(B)$ et si $-1 \leq N_{id}(A) - N_{id}(B) \leq 1$.

(b) si $N_{diff}(A) = 1$ et $E_{diff}(A) \not\subset E_{diff}(B)$.

Les démonstrations de ces tests de symétries sont dans [2].

17.3.5 La structure de la liste fermée

Cet algorithme utilisent deux listes : la liste ouverte et la liste fermée. Les structures de données doivent être adaptées afin d'obtenir les meilleurs performances. La liste ouverte ne pose pas de réelle problème. Une simple file de priorité est tout à fait efficace.

En revanche, la liste fermée nécessite plus de réflexion. En effet, comme il est dit précédemment, un élément retiré de la liste ouverte doit être combiné horizontalement et verticalement avec tout les éléments de la liste fermée. le résultat de cette construction doit respectée les contraintes du problème. Représenter la liste fermée par une simple liste chaînée d'éléments induit un parcours exhaustif de la liste fermée. Nous proposons donc l'utilisation d'une structure spécifique, offrant la possibilité de sélectionné rapidement les éléments de la liste fermée qui respectera au moins les contraintes de tailles. Cette sélection doit être faite, pour une construction horizontale ainsi que pour une construction verticale. La structure doit supporter ces opérations de sélection selon deux critères distinctes : taille verticale et taille horizontale. Les éléments doivent restés triés selon ces deux critères. Une propriété intéressante de ce problème est que les domaines des tailles possibles de coupes sont bornés par la taille du rectangle initial.

La structure est représenté comme suit : Deux tableaux L et W de listes simplement chaînées d'éléments, la taille de L (cep W) correspond à la longueur (resp : la largeur) du rectangle initial. A la case i du tableau L (resp W), seront les éléments de longueur (resp : largeur) i. La structure de nœud du sous-problème contient donc maintenant deux pointeurs, chaque couple permettant le chaînage avant pour chacune des deux dimensions.

17.3.6 La parallélisation

De par l'utilisation de deux structures de données, la parallélisation de cet algorithme n'est pas habituelle comparé aux Branch and Bound classiques. Chaque éléments de la file *Ouverte* doit être combiné avec tous les éléments de la liste Fermée. Sur machine à mémoire distribuée, l'une des deux files doit donc être partagée et l'autre distribuée. Notre solution reprend celle choisie par Tschöke et Holthöfer [16] la file Ouverte est partagée et la file fermée est distribuée.

17.3.6..4 Algorithme Parallèle Dans les faits, les deux files sont distribuées. Le partage de la liste Ouverte est géré comme suit : chaque nœud supprimé de la file Ouverte, est diffusé sur tous les processus afin d'être combiné avec les nœuds des différentes listes fermée locales. Évidemment, un processus n'ajoute à sa liste fermée que les nœuds qui proviennent de sa file Ouverte locale, et non pas ceux qui lui ont été envoyés. Ceci permet de réutiliser les algorithmes classique d'équilibrage de charge pour gérer le nombre d'éléments dans les listes Ouvertes locales.

Nous avons choisi d'utiliser un anneau virtuel avec jeton comme principe de diffusion, ceci permet de réduire le nombre de messages de diffusion existant sur le réseau.

17.3.6..5 Symétries dû à la parallélisation Un problème de construction dupliquée apparaît alors. En effet, si le processus i travaille sur un rectangle A, et si un processus j travaille sur un rectangle B. Puis que chacun des deux processus diffusent ces rectangles aux autres, le processus i va combiner B nouvellement arrivé avec A qui est dans la liste fermée, et de même le processus j va combine A avec B. Il vient donc que deux éléments identiques sont générés.

La solution employée pour interdire ce type de duplication, consiste à associer à un rectangle le tour du jeton où il a été inséré dans la liste fermée ainsi que l'identificateur du processus stockant la file fermée dans lequel il réside. La construction par un processus i du rectangle A reçu d'un processus j et de B de la liste fermée est interdit lorsque i est inférieure à j.

Ceci marcherait très bien si on n'utilise pas les tests de symétries dus au critère de séparation, présentés précédemment. En effet, le troisième test de symétrie utilise un ordre d'insertion dans la liste fermée. Il se trouve donc certains cas où d'une part le test de symétrie du coté critère de séparation et le test de symétrie du coté parallélisation interdise toute possibilité de générer un rectangle. Tous les sous-problèmes ne sont donc pas explorés. Le test de symétrie dû à la parallélisation doit donc être modifié pour prendre en compte ce phénomène. Le détail des preuves de ces algorithmes sont dans [2].

17.4 Conclusion

Dans cette article, nous avons expliqué la problématique de la parallélisation de la méthode A* appliquée au taquin, et celle d'un algorithme de résolution du problème de découpe contraint à deux dimensions. Ces algorithmes sont difficiles à paralléliser car en plus de leurs caractères irréguliers (comme toutes recherches arborescentes) ils utilisent non pas une structure de données mais deux.

Dans A*, la problématique vient d'une part de la sémantique qui est rattachée à la file ouverte, et d'autre part à la nécessité d'avoir une recherche efficace dans la file fermée. Nous avons proposé la treap pour implanter éléguamment la file ouverte.

Pour le 2-D CSP, nous avons d'une part, proposé des tests de symétries interdisant la construction de rectangles equivalents. D'autre part, la forte dépendance des deux structures entraîne aussi lors de la parallélisation des constructions de sous-problèmes symétriques. Nous y avons aussi apporté des solutions, pour les cas où l'algorithme utilise ou non les tests de symétries générées par le critère de branchement.

Bibliographie

[1] Aragon (C.) et Seidel (R.). – Randomized search trees. *FOCS 30*, 1989, pp. 540–545.

[2] Cung (V.), Le cun (B.) et Hifi (M.). – *Constrained Two-Dimensional Cutting Stock Problems, A Best-First Branch-and-Bound Exact Algorithm.* – Rapport technique n° to appear, PR*i*SM Versailles, 1997.

[3] Cung (V.-D.) et Le Cun (B.). – A suitable data structure for parallel a*. *In : DIMACS workshop on Parallel Processing of Discrete Optimization Problems.*

[4] Fouché (D.). – Parcours parallèles de type a* pour différents problèmes combinatoires difficiles. – Mémoire d'ingénieur IIE-CNAM, Juin 1993.

[5] Gilmore (P.) et Gomory (R.). – The theory and computation of knapsack functions. *Operations Research*, no14, 1966, pp. 1045–1074.

[6] Hifi (M.). – *Study of some combinatorial optimization problems: Cutting stock, packing and set covering problems.* – Pantheon-Sorbonne, Thèse de PhD, University of Paris 1, Déc. 1994.

[7] Hifi (M.). – An improvement of viswanathan and bagchi's exact algorithm for cutting stock problems. *Computers and Operations Research*, vol. 24, n° 8, 1997, pp. 727–736.

[8] Korf (R. E.). – Depth-first iterative-deepening : An optimal admissible tree search. *Artificial Intelligence*, no27, 1985, pp. 97–109.

[9] Le Cun (B.) et Roucairol (C.). – Concurrent data structures for tree search algorithms. *In : IFIP WG 10.3, IRREGULAR94 : Parallel Algorithms for Irregular Structured Problems*, éd. par A. Ferreira, J. Rolim (K. A.), pp. 135–155.

[10] Mans (B.). – *Contribution à l'Algorithmique Non Numérique Parallèle : Parallélisation de Méthodes de Recherche Arborescente.* – Thèse de doctorat, Université Paris VI, Paris, France, Juin 1992.

[11] Nilsson (N. J.). – *Principles of Artificial Intelligence.* – Tioga Publishing Co., 1980.

[12] Pearl (J.). – *Heuristics.* – Addison-Wesley, 1984.

[13] Rao (V.) et Kumar (V.). – Concurrent insertions and deletions in a priority queue. *IEEE proceedings of International Conference on Parallel Processing*, 1988, pp. 207–211.

[14] Rao (V. N.), Kumar (V.) et Ramesh (K.). – *Parallel Heuristic Search on Shared Memory Multiprocessors : Preliminary Results.* – Rapport technique n° AI85-45, Artificial Intelligence Laboratory, The University of Texas at Austin, Juin 1987.

[15] Roucairol (C.). – *Recherche arborescente en parallèle.* – RR n° M.A.S.I. 90.4, Institut Blaise Pascal - Paris VI, 1990. In French.

[16] Tschöke (S.) et Holthöfer (N.). – A new parallel approach to the constrained two-dimensional cutting stock problem. *In : the Second International Workshop. Parallel Algorithms for Irregularly Structured Problems*, éd. par 980 (L.), pp. 285–300.

[17] Viswanathan (K.) et Bagchi (A.). – Best-first search methods for constrained two-dimensional cutting stock problems. *Operations Research*, vol. 41, n° 4, 1993, pp. 768–776.

Chapitre 18

Travail coopératif et mémoire partagée répartie

Hervé Guyennet (LIB Besançon)

Le but de ce papier est de montrer que l'utilisation de la mémoire partagée répartie peut être bénéfique pour construire des applications de travail coopératif. Dans un premier paragraphe nous caractérisons le travail coopératif, puis nous présentons succintement la mémoire partagée répartie. Dans de troisième paragraphe nous dégageons quelques critères de choix d'un type de cohérence pour une application coopérative donnée. Ensuite nous présentons un algorithme pour gérer la cohérence d'une mémoire partagée répartie. Enfin une plate-forme de développement d'application coopérative est présentée rapidement ainsi qu'une application de dessin cooprératif s'appuyant sur cette plate-forme.

18.1 Caractérisation du travail coopératif

La coopération permet à plusieurs utilisateurs de travailler conjointement sur un même projet. Nous pouvons voir un groupe coopératif comme un groupe d'intervenants qui partagent les responsabilités pour réaliser des tâches en commun.

Un logiciel coopératif est avant tout un système distribué. Il en utilise les différents mécanismes, comme par exemple la mémoire partagée, l'échange de messages ou encore la répartition de charge...

Le domaine auquel nous nous intéressons est celui de la coopération informelle. Les applications coopératives étudiées échangent des données sans travailler sur leur sémantique : échange d'objets graphiques, de mots, de paragraphes, de sons, d'images...

Nous décrivons dans la suite de ce chapitre les principales caractéristiques des applications coopératives.

18.1.1 Les architectures coopératives

A partir du modèle d'architecture traditionnel qui sépare une application en un noyau fonctionnel et une interface, on distingue trois approches pour l'architecture de systèmes collecticiels [Kar94] : les architectures centralisées, les architectures décentralisées et les architectures hybrides.

18.1.1.a Architecture centralisée

Un seul processus gère la cohérence des données et les actions survenues au niveau des fenêtres des différents utilisateurs.

L'architecture centralisée a l'avantage de la simplicité d'implémentation. En particulier, la synchronisation et le contrôle de la concurrence sont triviaux. Cependant, il existe deux inconvénients majeurs pour un système interactif. D'une part, le temps de réponse est fortement ralenti puisqu'un seul processus gère les actions de l'ensemble des utilisateurs. D'autre part, un système centralisé n'est pas tolérant aux fautes.

18.1.1.b Architecture répliquée

Un processus correspond à chaque utilisateur. Les données sont répliquées sur chaque site et leur cohérence sera maintenue grâce à des communications entre les différents processus.

L'architecture répliquée ne présente pas les inconvénients cités ci-dessus. Son principal inconvénient est la difficulté de mise en œuvre. En effet, la répartition du contrôle et des données rend plus complexe les opérations telles que la cohérence des données ou encore l'ordonnancement des actions

18.1.1.c Architecture hybride

Un processus central gère la cohérence des données et un processus par utilisateur gère les actions non sémantiques de l'utilisateur sur l'interface.

L'architecture hybride résout partiellement un des inconvénients de l'architecture centralisée : la dégradation du temps de réponse de l'interface. Les actions telles que par exemple le déroulement d'un menu ou encore la gestion des boites de dialogue sont traitées localement. En revanche, le problème de tolérance aux fautes n'est pas résolu.

18.1.2 La synchronisation

La coopération asynchrone correspond aux applications coopératives dont le support de communication est asynchrone. La présence effective des membres du groupe n'est pas nécessaire pour que la coopération asynchrone se réalise. Toutes les applications de type courrier électronique font partie de cette catégorie.

À l'inverse, la coopération synchrone comprend les applications pour lesquelles la présence de chacun des membres du groupe est obligatoire pour effectuer le travail coopératif. Elle s'appuie sur des supports de communication synchrone. Les conférences sont des applications types de la coopération synchrone. Il est également possible que les applications coopératives utilisent les deux modes ce qui est le cas de l'éditeur développé dans le cadre du projet Mjolner [MM93]. Certaines applications coopératives permettent plusieurs degrés de synchronisation.

18.1.3 L'implication des membres coopérants

Les systèmes peuvent être répartis en fonction du degré d'implication de l'utilisateur au sein de l'activité.

Nous avons d'une part les systèmes individuels qui fournissent des outils permettant de réaliser des activités de coopération à partir d'un plan de travail personnel. Les systèmes de messagerie, de téléphonie ou encore de visiophonie font partie de cette catégorie.

D'autre part les systèmes de groupe qui procurent une vision et éventuellement un accès au plan de travail du groupe coopérant.

18.1.4 La distance entre les membres coopérants

La distance entre les sites coopérants est un critère important dans l'évaluation du coût de communication. Lors du choix du degré de synchronisation de l'application coopérative, il est important de connaître ce coût. En effet, il faut pour le confort d'utilisation qu'un membre coopérant ne ressente pas la latence due aux échanges de données. Si les sites sont très éloignés géographiquement il sera difficile de gérer une application parfaitement synchrone : les échanges incessants devenant trop pénalisants.

Lorsque des travaux sont réalisés en coopération, il est intéressant afin d'optimiser l'application de créer des groupes de coopérants.

Ces groupes devront être gérés dynamiquement pendant la réalisation du travail. Les modifications des structures des groupes se font ou bien à chaque fois que cela est possible ou bien sous forme de vote [GMB85, AAC91]. Les groupes permettent d'optimiser le système coopératif pour une diffusion par exemple, ou encore pour un travail utilisant une mémoire partagée répartie, la création de groupe réduit le contexte partagé.

18.1.5 La granularité

La granularité caractérise la taille de l'entité élémentaire partagée par les membres coopérants. Nous pouvons différencier le grain des actions du grain des données.

- Le grain au niveau des actions :

  En WYSIWIS (*What You See Is Wat I See*) strict les actions sont synchronisées, elles apparaissent *en même temps* dans chaque fenêtre du groupe. Le temps permet de définir le grain de la coopération : par exemple dans GROVE qui est un éditeur de texte coopératif [EGR91], le grain est l'appui sur une touche. Un grain variable permet des moments privés durant lesquels aucune des actions de l'utilisateur n'est connue du reste du groupe, ceci n'est possible que dans les applications qui ne sont pas parfaitement synchrones.

- Le grain des données :

  Une application coopérative peut offrir un grain de donnée de taille fixe, par exemple il est possible de travailler à l'aide d'un buffer. Chaque fois que le *buffer* est plein, il est envoyé aux autres membres coopérants. Mais le plus souvent, le grain est variable et il est fonction du grain des actions.

18.1.6 Espaces partagés et privés

Le fait de créer puis de travailler dans un espace privé revient à *buffeuriser* les actions, et donc à augmenter le grain des actions. Un participant réalise une travail personnel sur son espace privé, une fois terminé ce travail ne sera pas nécessairement transmis au reste du groupe.

Cet espace est intéressant à deux titres :

- Au niveau des performances, le nombre des communications est diminué.
- Au niveau coopératif, la transmission immédiate des actions peut entraîner des réactions ou des critiques qui ne sont pas nécessaires et qui peuvent être nuisibles à l'efficacité du travail de groupe. Aussi l'espace privé permet à l'utilisateur de réaliser des travaux personnels.

18.1.7 Les protocoles coopératifs

Ils définissent les règles ou politiques gouvernant les interactions et les échanges entre les différents utilisateurs. On distingue deux types de protocoles : les protocoles technologiques implémentés au niveau du matériel ou au niveau du logiciel et les protocoles sociaux laissés au contrôle des participants.

```
...
draw(circle,R,Cx,Cy,linestyle,pattern);
create(MESSAGE,circle,R,Cx,Cy,linestyle,pattern);
multicast(MESSAGE,my_group);
wait_for_acknowledges;
...
```

a.

```
...
shared_draw(circle,R,Cx,Cy,linestyle,pattern,my_group);
...
```

b.

Figure 18.1 : Dessin d'un cercle

- Si le contrôle est laissé aux protocoles sociaux cela encourage la collaboration, en effet le groupe doit définir ses propres règles d'échanges qui sont plus proches des besoins du groupes mais sont soumises à la bonne volonté des participants. Dans GROVE, par exemple, [EGR91], la gestion des fenêtres publiques est un protocole social. Mais suivant les environnements, ces protocoles peuvent être injustes voir inefficaces.

- Au contraire si des protocoles technologiques sont implémentés dans le logiciel, ils permettent de structurer le groupe. Les protocoles technologiques peuvent être restrictifs, en particulier parce qu'ils limitent les possibilités d'utilisation.

18.2 La mémoire partagée répartie

18.2.1 Pourquoi utiliser une mémoire partagée répartie ?

L'implantation d'une mémoire partagée répartie offre une abstraction de mémoire commune et rend l'échange de messages transparent. L'utilisation de cette technique présente deux avantages majeurs :

- *la facilité de programmation*

  En effet, pour programmer une application répartie il est beaucoup plus simple d'utiliser des variables ou des objets partagés que de gérer explicitement l'échange de messages.

  Par exemple, pour programmer un éditeur graphique, le dessin d'un cercle nécessite l'écriture d'un code (figure 18.1a) en échange de messages qui sera simplifié (figure 18.1b) en utilisant une mémoire partagée. Dans le premier cas, il faut dessiner localement le cercle, formater un message comprenant toutes les données permettant de dessiner le cercle, diffuser par exemple au sein d'un groupe les données, et attendre de chaque processeur partenaire un accusé-réception. En utilisant une mémoire commune, tous les mécanismes sous-jacents sont transparents, pour le programmeur une seule commande de dessin partagé suffit.

- *La garantie de performance minimum et de fiabilité*

  Pour un système à mémoire partagée donné, les performances sont connues et garanties. Le développeur d'applications peut prétendre à une certaine fiabilité. Il est également sûr d'avoir des performances minimales, celles offertes par le système. Celles-ci ne dépendront pas de la qualité de sa programmation mais du système sous-jacent. Il y aura donc une moins grande dispersion des temps d'exécution. Bien sûr, comme beaucoup d'outils systèmes, nous risquons en offrant plus de confort de faire baisser les performances.

18.2.2 Choix d'une mémoire partagée

De nombreux travaux ont déjà présenté les différentes implantations de mémoire partagée ([Tan95]). Dans les architectures faiblement couplées, ou le partage mémoire n'est pas réalisé au niveau matériel, on utilise le concept de Mémoires Partagées Réparties (MPR). Deux approches de MPR sont proposées : la mémoire virtuelle partagée et le partage des données. La première méthode est une extension de la mémoire virtuelle des architectures monoprocesseurs [Pri94], c'est un espace d'adressage qui est partagé. La deuxième méthode travaille à un niveau supérieur, en partageant des données (non plus un espace) auxquelles on accède par des fonctions de haut niveau. La mémoire virtuelle partagée se situe au niveau système et le partage des données au niveau langage de programmation.

Pour un environnement coopératif, le but **n'est pas** d'avoir un espace de **grande taille**, mais un espace répliqué sur chaque processeur.

Les différents utilisateurs pourront ainsi accéder simultanément aux mêmes données. Une question se pose : *quel est l'état de la mémoire à un instant t* et plus particulièrement *pour une donnée x, à un instant t quelle est la valeur retournée en lecture ?*

Pour obtenir une réponse, il faut gérer très rigoureusement les accès, ce que l'on appelle ***gestion de la cohérence des données***.

18.2.3 Les différents modèles de cohérence

Nous rappelons simplement les différents types de cohérence les plus courants: la cohérence stricte, la cohérence séquentielle, la cohérence causale, la cohérence PRAM, la cohérence lente.

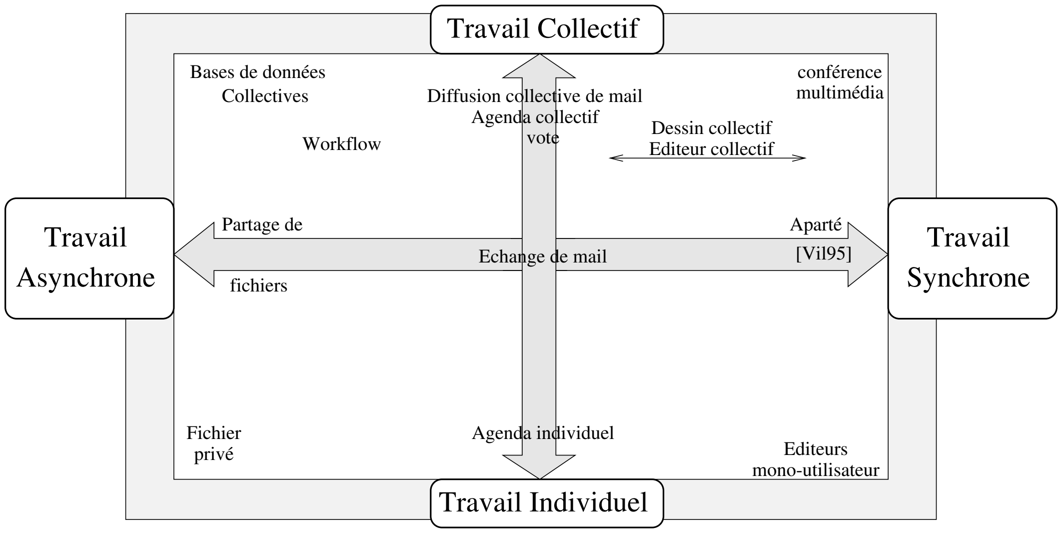


Figure 18.2 : Classification des applications coopératives

18.3 Choix d'un type de cohérence pour une application coopérative

Il s'agit de déterminer un modèle respectant les caractéristiques de l'application coopérative (synchronisation par exemple) mais qui ne soit pas trop pénalisant. Par exemple, la cohérence stricte est intéressante pour toute application coopérative, mais les protocoles l'implémentant sont trop coûteux. Il faut trouver le modèle de cohérence mémoire le moins contraignant qui respecte les caractéristiques coopératives de l'application.

18.3.1 Étude des paramètres qui déterminent ce choix

Pour [LH89] les performances d'un programme parallèle utilisant la mémoire partagée dépendent principalement : du nombre de processus et **du degré de mise à jour des données partagées**. Pour un programme coopératif, ce degré sera également très important, nous l'appelons degré de synchronisation.

Sur le dessin 18.2 nous avons placé en fonction des paramètres *nombre de participants* et *degré de synchronisation* les familles d'applications coopératives. La région représentant le travail asynchrone ne nous intéresse pas dans le cadre de cette étude, nous pouvons cependant citer les applications connues telles que les partages de fichiers ou encore le *WORFLOW* qui représente la gestion et la circulation d'informations sur un réseau : par exemple, des documents administratifs qui suivent la voie hiérarchique via le réseau informatique.

Toujours asynchrone, mais considérées comme plus *dynamiques* nous rencontrons les applications de type mail ou encore agendas.

La région droite du dessin, la région synchrone comprend à la fois les applications de type éditeurs coopératifs qui peuvent ne pas être totalement synchrones, et les applications qui intègrent des dialogues sonores avec ou sans images qui doivent être synchrones. Par exemple dans [Vil95] la notion d'aparté permet à deux membres coopérants de s'isoler pour dialoguer. Cette phase est parfaitement synchrone.

La synchronisation détermine à quel moment les opérations effectuées par un des membres coopérants sont *vues* par l'ensemble de ces membres. Mais un autre paramètre est très important pour le choix du modèle de cohérence, c'est le grain de l'application. Dans certains modèles de cohérence nous pouvons considérer le grain comme l'opération, dans d'autres le grain est déterminé par la synchronisation (nombre d'opérations entre deux synchronisations).

18.3.2 Choix du modèle de cohérence

Lorsque des membres d'une application coopérative travaillent conjointement, il n'est pas forcément nécessaire qu'ils voient en temps réel les travaux des autres participants. Il s'agit de trouver un compromis entre le confort des membres coopérants et les performances en terme de coût de communication.

18.3.2.a Les éditeurs coopératifs

L'éditeur de texte Prep [NKCM94] offre différents modes d'annotations qui correspondent à différents grains, c'est-à-dire à différents visualisation des modifications : le plus strict étant la visualisation de chaque correction, le plus relâché étant la seule vue du résultat final.

Plus généralement dans les éditeurs coopératifs, le grain est variable : dans un éditeur de dessin il peut être l'objet, ou le groupe d'objets. Dans ces applications l'ordre des actions est important comme dans les modèles de cohérence séquentielle ou encore causale.

Si nous raisonnons sur le degré de synchronisation, nous remarquons que par exemple pour un éditeur de dessin qui offre une possiblité de travailler dans une fenêtre privée comme DECO (cf section 18.5.2), deux degrés sont gérés : synchrone dans la fenêtre partagée et asynchrone dans la fenêtre privée.

Pour des mécanismes asynchrone, les modèles de cohérence séquentielle ou causale sont inutilement trop coûteux, un modèle de cohérence faible est suffisant. Il respecte l'ordre des opérations mais permet d'en *masquer* certaines.

18.3.2.b Les applications de conférence

Ces applications offrent des médias tels que le son et l'image, et dans ce cas une cohérence stricte est nécessaire.

18.3.2.c Les systèmes de partage d'applications et les plate-formes de développement pour applications coopératives

Ces systèmes ont des vocations très générales : implémentation ou partage de toutes applications coopératives. Il faudrait prévoir dans ce cas différents mode de cohérence ou encore un mode de cohérence faible dont la fréquence des synchronisations serait modulable.

Si nous voulons proposer un modèle qui puisse être utilisé pour le travail coopératif en général, il faut le choisir souple pour qu'il s'adapte à chaque application. Il doit accepter les modes synchrones comme asynchrones et un grain variable. Le modèle de cohérence faible est donc le plus approprié :

- il devient séquentiel si l'on synchronise après chaque action,
- il devient stricte, si en plus, la fréquence des synchronisations est importante.

18.4 Présentation du protocole du Pèlerin

Ce protocole permet de gérer la cohérence d'une mémoire partagée répartie de type mémoire répliquée à partage d'objets. **Chaque** objet est présent sur chaque site intégré à l'anneau qui l'**utilise**.

Chaque objet est à un instant t la propriété d'un seul site qui est le seul autorisé à écrire sur l'objet. Un objet peut changer de propriétaire.

Pour chaque objet, ce protocole est de type simple écrivain et lecteurs multiples. Une instance de chaque objet est présente sur chaque site. Les lectures donnent accès à la valeur locale de l'objet. Mais par rapport à l'ensemble de la mémoire, il est de type écrivain multiples et lecteurs multiples. Chaque site possède une partie des objets de la mémoire sur lesquels il est le seul à pouvoir écrire. Par exemple, un objet *dessin* est partagé par les membres d'une session coopérative. Tous les membres peuvent écrire sur le dessin en même temps, mais sur des objets graphiques différents. Un objet graphique est uniquement écrit ou modifié, à un instant t donné, par un seul dessinateur. L'objet *dessin* est finalement composé des différents objets graphiques.

En cours d'exécution les seuls messages émis sont les messages qui permettent la circulation du jeton : envoi du jeton et accusé-réception. D'autres messages circulent, non plus exclusivement sur l'anneau mais sur l'ensemble des canaux :

- pour gérer la structure de l'anneau (par exemple l'entrée d'un nouveau processeur dans l'anneau logique),
- pour permettre les dialogues audio en mode point à point.

Ce protocole utilise un anneau logique unidirectionnel. Les démons CAliF, nœuds du système, reçoivent lors de leur initialisation un numéro (identificateur). La numérotation est croissante, le démon le plus ancien possède l'identificateur le plus petit. Le sens de circulation du jeton sur l'anneau logique suit cette numérotation.

Au passage du jeton, deux types d'opérations sont possibles : les opérations sur les données, et les opérations sur les propriétés.

- *Les opérations sur les données :*

  Lorsque le jeton arrive sur un site i, il contient les mises à jour des données possédées par les sites autres que i. Les instances locales des données sont mises à jour : création, modification, suppression. Les données possédées par i, si elles ont été modifiées depuis le dernier passage du jeton, sont placées sur ce dernier qui propage la nouvelle valeur.

- *Les opérations sur la propriété :*

  Il est possible de modifier le propriétaire d'une donnée. La demande de récupération ainsi que la réponse circulent par l'intermédiaire du jeton.

18.4.1 La structure du jeton

Le pèlerin (jeton) est un tableau d'objets : les objets qu'il faut mettre à jour. Les objets des différents sites sont distingués par des séparateurs. Comme nous pouvons le remarquer sur la figure 18.3 qui présente un système à **quatre** nœuds, il y a dans ce tableau cinq séparateurs. Les premiers éléments, précédant les quatre premiers séparateurs, sont les éléments des sites 1,2,3 et 4 qui ont été soit modifiés depuis le dernier passage du jeton, soit font l'objet d'une demande de changement de propriétaire. Les derniers éléments sont ceux de la corbeille pour lesquels une demande de récupération est en cours.

18.4.2 La structure des objets

Un objet o_{ij} (figure 18.4) est composé de quatre champs : une commande, le paramètre de la commande, le rang j de l'objet dans la liste i (liste des objets possédés par le site i) et la donnée sur n octets.

18.5 Une plate-forme de développement d'applications coopératives

18.5.1 Présentation de la plateforme

Nous proposons une nouvelle plate-forme CALIF qui s'appuie sur une couche de communications MPI (*Message Passing Interface*), qui permet l'utilisation d'une mémoire partagée, qui gère des services tels que la gestion de groupes ou encore gestion

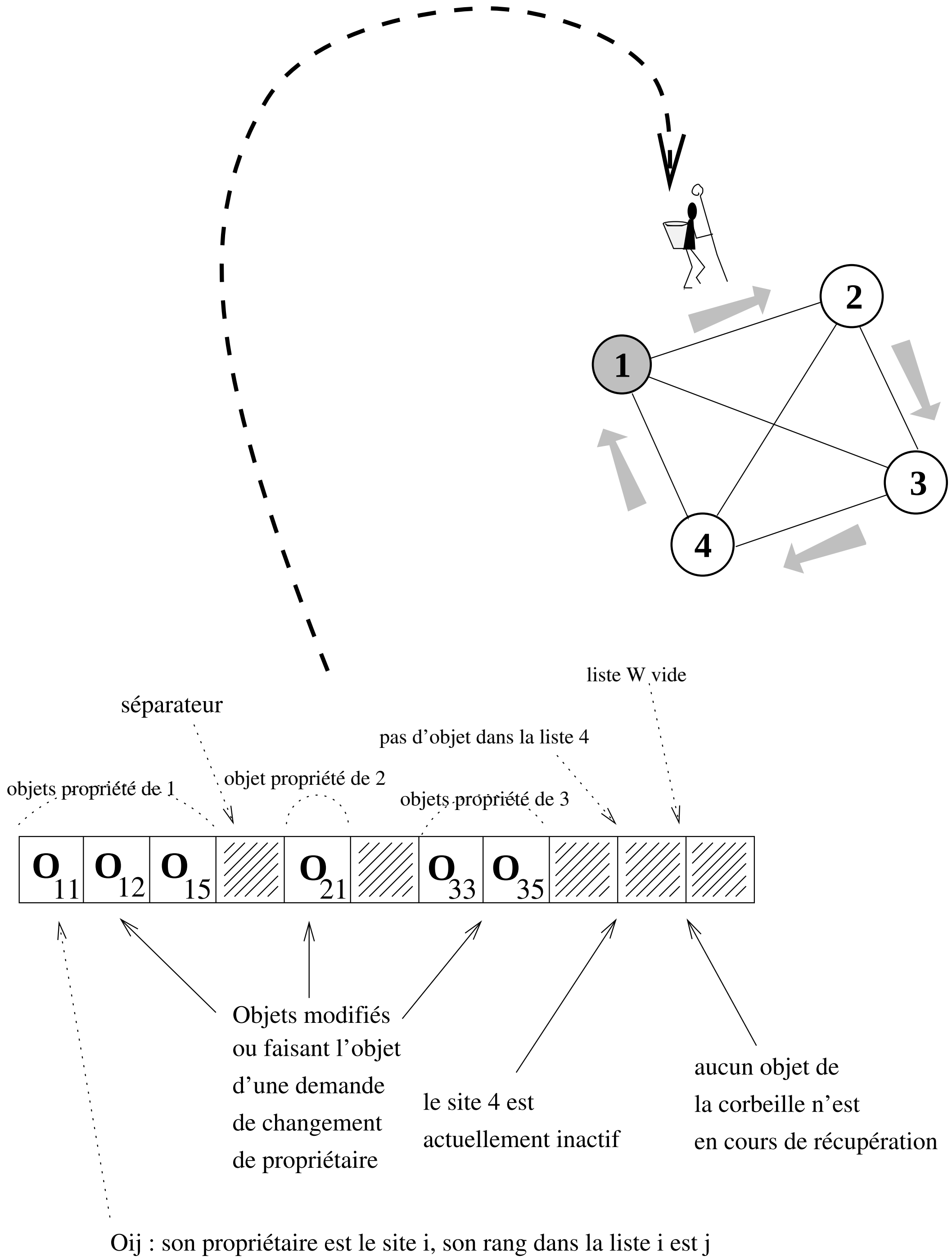


Figure 18.3 : La structure du Pèlerin

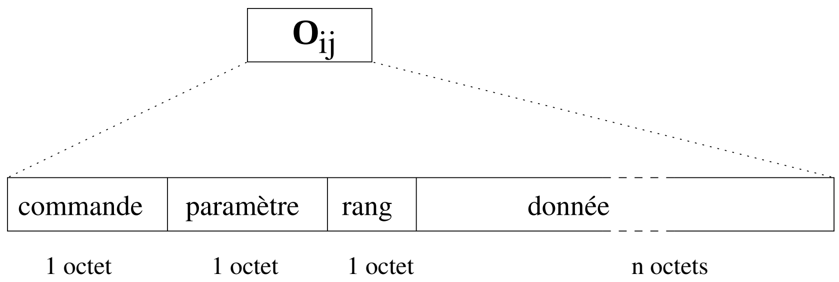


Figure 18.4 : Structure d'un objet

de la coopération et qui propose des bibliothèques de fonctions utiles au développement d'applications coopératives. Notre système devra être aussi souple et portable que le sont PVM (*Parallel Virtual Machine*) et MPI. Le programmeur d'applications coopératives utilisera CAliF comme une boîte à outils. Des fonctions telles que *pvm_send(...)* ou *MPI_Send(...)* seront proposées, par exemple CAliF_send(...).

La notion de partage omniprésente dans les types d'applications de travail coopératif nous conduit naturellement vers l'utilisation de la mémoire partagée. L'idée est de mapper cette couche sur un système de partage de mémoire qui reste transparent pour l'utilisateur. La gestion de la couche mémoire partagée permet d'obtenir des programmes performants tout en garantissant la rigueur au niveau de la gestion du partage. Afin de bien identifier les différents concepts communs à tout système coopératif, nous avons défini notre plate-forme sur quatre niveaux ([GL96, GLT97]) :

- Le niveau communication offre un réseau de canaux entre les nœuds. Ce niveau est développé sur MPI.

- Le niveau mémoire virtuelle partagée propose un espace virtuel de mémoire qui permet de gérer de manière transparente les variables partagées. Nous avons implémenté à ce niveau un nouveau protocole de cohérence de mémoire partagée appelé l'algorithme du Pèlerin.

- Le niveau services de CAliF, propose plusieurs types de service :

  - *Le service groupes* gère l'utilisation des notions de groupe de processeurs et de groupe de processus.
  - *Le service gestion de la coopération* propose différentes fonctions usuelles dans les applications coopératives : l'entrée en coopération (création dynamique de la machine), la connexion d'un démon, la connexion d'un membre d'une application, la sortie de coopération, la diffusion.
  - *Le service qualité de service* offre une garantie de qualité sur la gestion de la mémoire.

   – *Le service sécurité* permet de protéger l'espace partagé : seuls les membres d'un groupe coopératif pourront accéder à l'espace de données partagées du groupe.

- Le niveau application distribuée coopérative offre un grand nombre de librairies aidant au développement d'applications coopératives.

18.5.2 DECO

Le logiciel DECO [BGST94] est un logiciel de dessin coopératif. Il a été développé par l'équipe *Réseaux et Systèmes Distribués* de Besançon. Il est composé d'un éditeur textuel, d'un éditeur graphique et d'un canal sonore.

DECO propose une coopération synchrone au niveau des éditeurs textuels et graphiques. DECO offre également des espaces privés (textuels ou graphiques) qui permettent le support d'une édition plus asynchrone. En édition privée, les modifications d'un utilisateur sont uniquement locales.

La communication vocale permet d'échanger des idées sur un sujet donné entre deux participants.

Le concept WYSIWIS a été adopté pour l'interface de DECO. La relaxation est faite sur la congruence des vues et le temps (Figure 18.5).

Conclusion

Nous avons présenté une plate-forme de développement d'applications coopératives basée sur la mémoire partagée répartie. Avec la facilité de programmation, les performances mesurées lors de la diffusion de messages par l'ensemble des membres coopérants sont excellentes et valident ce travail.

Bibliographie

[AAC91] M. Ahamad, M.H. Ammar, and S.Y. Cheung. Multidimensionnal Voting. *ACM Transactions on Computers Systems*, 9(4):399–431, November 1991.

[BGST94] C. Balayer, H. Guyennet, F. Spies, and M. Tréhel. The Concepts of Distributed Groupware and Multimedia Applied to a Drawing Tool. *Proceedings of the Intelligent Multimedia Information Retrieval Systems and Management, RIAO 94, New York USA*, October 1994.

[EGR91] C. Ellis, S. Gibbs, and G. Rein. Groupware. Some Issues and Experiences. *Communication of ACM*, 34(1):35–58, January 1991.

[GL96] H. Guyennet and J-C. Lapayre. A Co-operative Application Management Platform Based on Shared Virtual Memory. *IEEE Proceedings*

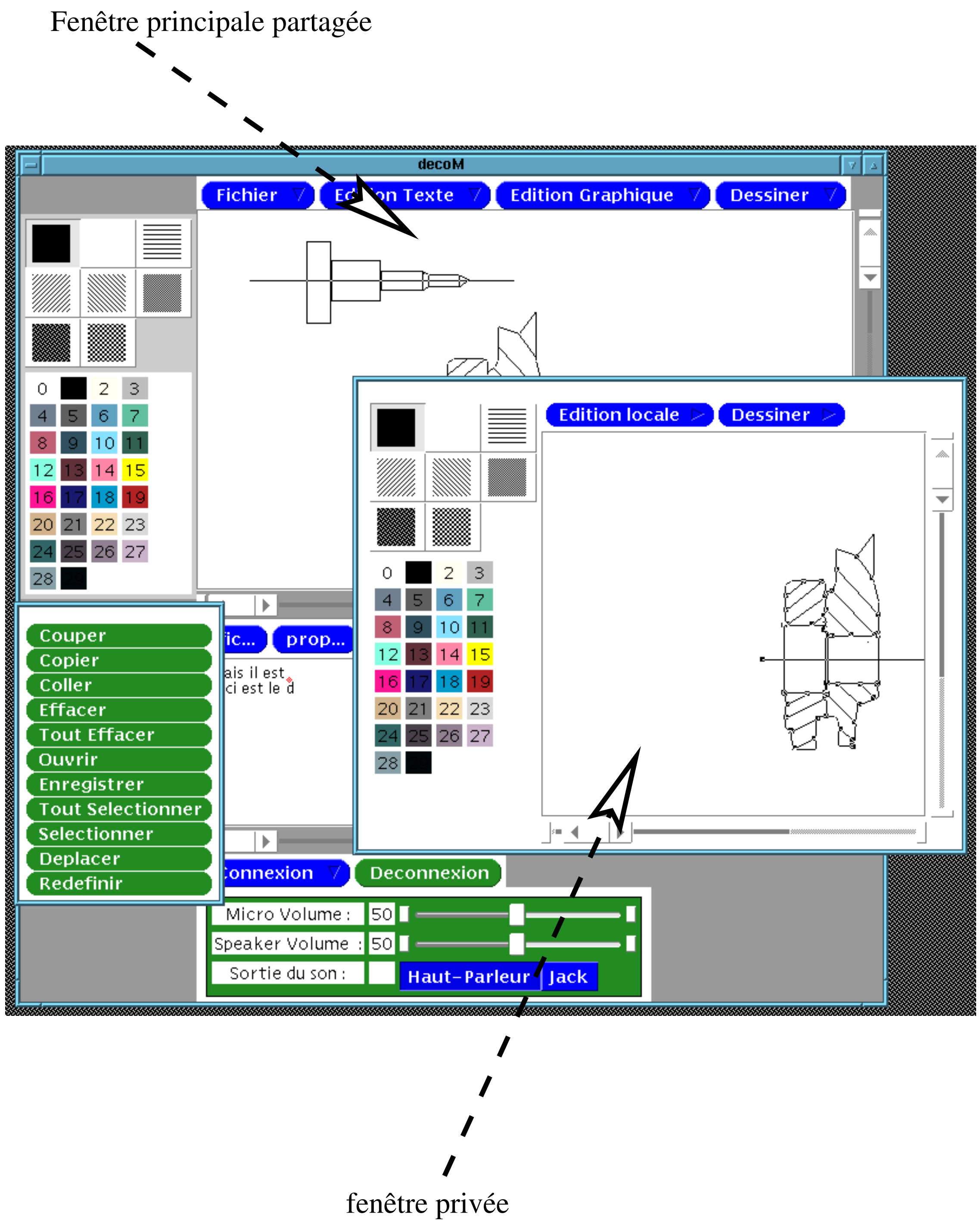


Figure 18.5 : Le dessin coopératif dans l'application DECO

of the HiPC'96 International Conference, Trivandrum Inde, December 1996.

[GLT97] H. Guyennet, J-C. Lapayre, and M. Tréhel. CAliF : une plate-forme de développement de collecticiels utilisant la mémoire partagé distribuée. *Calculateurs Parallèles*, 1997.

[GMB85] H. Garcia-Molina and D. Babara. How to assign votes in a distributed system. *Journal ACM*, 32(4):841–860, Octobrer 1985.

[Kar94] A. Karsenty. Le collecticiel : de l'interaction homme-machine à la communication homme-machine-homme. *TSI Technique et Science Informatiques*, 13(1):105–127, 1994.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[MM93] S. Minör and B. Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, September 1993.

[NKCM94] C. Neuwirth, D. Kaufer, R. Chandhok, and J. Morris. Providing Computer Support for Distributed Collaborative Writing. *ACM Proceedings of the Conference on Computer Supported Cooperative Work CSCW'94*, October 1994.

[Pri94] T. Priol. Conception d'un environnement de programmation pour les architectures parallèles à mémoire partagée. *Calculateur Parallèle*, 6(4):103–108, 1994.

[Tan95] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall International Editions, 1995.

[Vil95] T. Villemur. Conception de Services et de Protocoles pour la Gestion de Groupes Coopératifs. *Thèse de l'UPS Toulouse France*, January 1995.

Chapitre 19

Vers l'automatisation de la parallélisation pour les programmes irréguliers

LaMI

Mots clés : Parallélisation, Applications irrégulières, Matrices creuses, Analyse de dépendances

19.1 Introduction

Cat article présente un cadre de parallélisation pour une catégorie d'applications numériques manipulant des structures creuses. Cette approche conduit à étendre les méthodes de parallélisation pour tirer profit de la spécificité du traitement appliqué à ces structures ; cette extension porte sur l'analyse du programme en entrée, sur l'analyse de dépendances et sur la génération de code parallèle. Dans cet article, nous décrirons synthétiquement cette approche. Une étude plus approfondie peut être trouvée dans [2].

19.1.0..1 Analyse du programme en entrée : Une structure creuse correspond à un tableau dont le format de stockage élimine le grand nombre de 0. Traditionnellement la représentation de ces structures s'effectuent par deux composants : un tampon de valeurs (`YVAL`) et un tableau d'indices (`YIND`). Par exemple, le programme suivant (figure fd:fig1) représente le calcul de $\vec{x} = \vec{y} + \vec{x}$ où $\vec{y}$ est un vecteur creux.

Bien que le programme dense fasse apparaître un parallélisme simple à détecter, le format de stockage creux sous la forme d'un couple **valeur** (`YVAL`), **entrée** (`YIND`) restreint la puissance de l'analyse de dépendance qui se fonde sur des outils d'algèbre linéaire en nombre entier et qui par conséquent s'applique dans le cas où le référencement des tableaux est fait par des fonctions affines. Pour éviter ce problème, une approche consiste à introduire la notion de typage creux `SPARSE` et à rédiger le programme comme si l'on gérait des tableaux denses. Ce typage exprime le fait que les structures creuses sont *logiquement considérées comme étant des tableaux* mais physiquement les zéros seront éliminés. Charge donc au compilateur d'effectuer la sélection des structures typées creuses (`SPARSE Y(n)`) en fonction du programme écrit en dense. Le lecteur peut se référer aux articles de [1],[5], [6] pour une synthèse des travaux menés sur le typage creux pour les langages parallèles et séquentiels.

19.1.0..2 Analyse de dépendances : Un programme implicitement creux autorise l'analyse de dépendance classique. Toutefois, notre approche met à profit la spécificité du creux pour obtenir un parallélisme supplémentaire à celui obtenu par cette analyse. Cette approche étend la définition des dépendances en introduisant une analyse sur la sémantique des expressions en fonction des propriétés d'absorption et de neutralité de 0. Le programme scalaire suivant (figure fd:fig2) illustre cette extension.

L'analyse du graphe de dépendance conclut que l'exécution de ce programme ne peut être que séquentiel. En considérant à présent que 0 est neutre pour l'addition et absorbant pour la multiplication, l'exécution parallèle de s_1, s_2, s_3 peut être faite en ignorant le calcul des expressions (`2*a`) et (`c*a`). Intuitivement, une instruction s_2 est dépendante par flot de s_1 si *l'état de la variable écrite dans* s_1 *et lue dans* s_2 *conduit à une modification de l'état de la variable écrite dans* s_2. Ce mode d'analyse trouve ses origines dans la parallélisation de la factorisation de Cholesky creux [4]. Nos travaux visent à automatiser ce traitement. La section 19.2 décrit les différentes étapes conduisant à la construction du graphe de dépendance creux.

19.1.0..3 Génération de code : La génération de code peut se scinder en deux étapes : définir un schéma de génération qui tient compte de l'analyse de dépendances liées au creux, produire un programme parallèle utilisant les structures creuses à partir d'un programme écrit en dense. La section 3 traitera des solutions

```
do i=1,n                      do i=1,IMAX
  X(i)=X(i)+Y(i)                X(YIND(i))=X(YIND(i))+YVAL(i)
enddo                         enddo
```

Figure 19.1 : version dense & creuse de $\vec{x} = \vec{x} + \vec{y}$

s_1 `a = 0`
s_2 `c = 2*c+2*a`
s_3 `d = d+1+c*a`

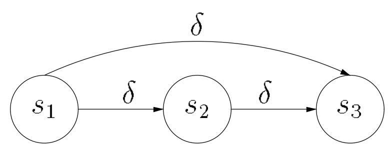


Figure 19.2 : exemple introductif

proposées pour ces problèmes.

19.2 Analyse de Dépendances creuses

Le schéma général de parallélisation couple un travail de compilation et un travail à l'exécution. Le travail de compilation consiste à "synthétiser" des fonctions représentant le calcul symbolique et les dépendances ; puis à générer le programme. Le travail à l'exécution est de calculer les résultats attendus à partir de ces fonctions.

19.2.1 Remplissage

En s'appuyant sur l'exemple précédent, on constate que le calcul des dépendances liées aux creux nécessite une connaissance de l'état des variables. Ceci implique de définir une abstraction du calcul de l'état mémoire opposant les cellules mémoire à zéro à celles qui ne le sont pas. Clairement, celle-ci est donnée par les entrées. Pour obtenir une caractérisation exacte des dépendances, il faut donc calculer symboliquement les entrées non nulles introduites par l'exécution réelle de l'algorithme. Le compilateur synthétise une *fonction de remplissage* { représentant l'exécution du programme pour une itération générique. Cette étape sera illustrée par l'exemple suivant.

En premier lieu, on caractérise l'ensemble *des itérations significatives* qui représente les itérations engendrant de nouvelles valeurs non nulles à partir des entrées non nulles . Ce calcul se définit par une mise en correspondance des opérateurs arithmétiques avec les opérateurs d'ensemble appliqués sur des ensembles d'entiers décrits possédant des paramètres symboliques représentant les entrées. Ces règles s'établissent pour l'exemple précédent de la façon suivante: Soit $E_A = \{a | A(a) \neq 0\}$ l'ensemble des entrées pour le tableau A et $D = \{j \in Z | 3 \leq j \leq 12\}$ le domaine d'itérations de la boucle. Pour les trois occurrences du tableau à droite de l'instruction s les ensembles d'itérations significatives sont :

$$I = \emptyset,\ I_{a_1} = \{j \in D | a_1 = j-1\}\ et\ I_{a_2} = \{j \in D | a_2 = j-2\}$$

Pour l'occurrence `A(j+1)`, $I = \emptyset$ car la partie droite étant la même que la partie gauche, cette occurrence ne crée aucun élément nouveau, quel que soit l'itération. Comme 0 est absorbant pour la multiplication, $(j-1)$ **et** $(j-2)$ doivent tous deux référencer une cellule non nulle; cela correspond à l'intersection de I_{a_1} et I_{a_2}. De même "`+`" correspond à l'union qui reflète la propriété de neutralité. L'ensemble des itérations significatives $I_{(a_1,a_2)}$ est donc:

$$I_{(a_1,a_2)} = \emptyset \cup (I_{a_1} \cap I_{a_2}) = \{j \in D | a_1 = a_2 - 1 = j - 1, \}, a_1 \in E^{max}, a_2 \in E^{max}$$

L'ensemble des cellules écrites en fonction de $I_{(a_1,a_2)}$ est son image par la fonction en écriture ($W_{(a_1,a_2)} = Img_{(j+1)}(I_{(a_1,a_2)})$). Sa forme optimisée est :

$$W_{(a_1,a_2)} = \{j \in D |\ j = a_1 + 2 \wedge ((a_2 - a_1 + 1) = 0) \wedge (2 \leq a_1 \leq 11)\}, a_1 \in E^{max}, a_2 \in E^{max}$$

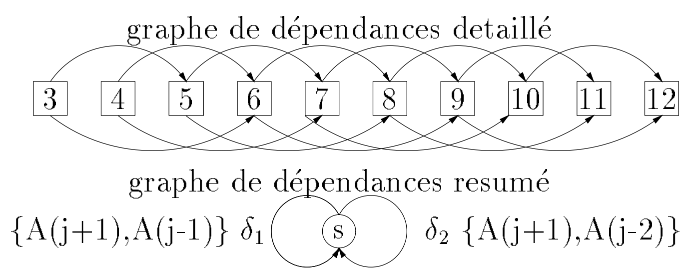
La "simulation" des itérations conduisant à l'état final s'obtient enfin par l'application itérée de la fonction $\mathcal{F}$ définie ci-après jusqu'à l'obtention d'un point fixe. L'ensemble initial des entrées (E_A^0) correspond à celui avant l'exécution de la boucle, soit :

$$E_A^{t+1} = \mathcal{F}(E_A^t) = E_A^t \cup \{a | a \in W_{(a_1,a_2)}, a_1 \in E_A^t, a_2 \in E_A^t\}, \ \bigcup_{t \geq 0} E_A^t = E_A^{max}$$

19.2.2 Caractérisation des dépendances

Les dépendances creuses étendent la définition des dépendances en intégrant deux propriétés supplémentaires : Les cellules référencées font partie de E_A^{max} car les autres ne sont jamais accédées ; le référencement implique une modification. Par

```
SPARSE,DOUBLE::A(13)
do j = 3,12
s  A(j+1)=A(j+1)+A(j-1)
                *A(j-2)
enddo
```



A Exemple B Graphe de dépendances

Figure 19.3 : Programme & Graphe de dépendance

exemple, une cellule pleine référencée par `(j-1)` peut ne pas engendrer de modification si `(j-2)` adresse une cellule vide. La modification se caractérise par rapport à *l'ensemble des itérations modificatrices* dont la construction s'établit par des règles similaires à celles décrites dans la sous-section précédente pour les itérations significatives. Ce raffinement conduit à caractériser les dépendances comme étant des fonctions des entrées vers des couples d'itérations les représentant. Intuitivement, on peut expliquer cela par le fait que les dépendances se calcule en fonction d'un un accès commun à une adresse. Si cette adresse ne figure pas dans les entrées E^{max}, la dépendance ainsi calculée ne traduit pas une dépendance effective car il n'y a pas de modification de l'état mémoire. Dans le cadre de l'exemple, les dépendances de flots creuses correspondant aux extensions des dépendances denses δ_1 et δ_2 sont définies par les fonctions suivantes :

$$\Delta_1(a_1, a_2) = \{(a_1 - 1, a_1 + 1) | a_1 = a_2 + 1 \wedge 4 \leq a_1 \leq 11\}$$

$$\Delta_2(a_1, a_2) = \{(a_2 - 1, a_2 + 2) | a_2 = a_1 - 1 \wedge 4 \leq a_2 \leq 10\}$$

La définition des dépendances concernent seulement celles "transportées" par le boucle car seules celles-ci sont interessante pour la parallélisation des programmes creux (voir section 19.3).

Définition 19.2.1 *Soit $\mathcal{M}(s, i)$, un prédicat qui est vérifié si la variable écrite à l'itération i est modifiée ; soit D_1, D_2 deux domaines d'itérations ; soit P_1, P_2 deux matrices de projection sur le domaine communs T à D_1 et D_2 ; soit f et g deux fonctions affines représentant respectivement la fonction en écriture et en lecture on a :*

- *dépendance de flot (lecture après écriture) : $t_1 \Delta t_2$ ssi*
  $\exists i_1 \in D_1, \exists i_2 \in D_2, g(i_2) = f(i_1) = a \wedge P_1 i_1 = t_1 \wedge P_2 i_2 = t_2 \wedge t_1 \ll t_2 \wedge \mathcal{M}(s_2, i_2) \wedge a \in E_A^{max}$
- *anti-dépendance (écriture après lecture) : $t_1 \bar{\Delta} t_2$ ssi*
  $\exists i_1 \in D_1, \exists i_2 \in D_2, g(i_1) = f(i_2) = a \wedge P_1 i_1 = t_1 \wedge P_2 t_2 = c_2 \wedge t_1 \ll t_2 \wedge \mathcal{M}(s_1, i_1) \wedge a \in E_A^{max}$
- *dépendance de sortie (écriture après écriture) : $t_1 \Delta^{\circ} t_2$ ssi*
  $\exists i_1 \in D_1, \exists i_2 \in D_2, f_1(i_1) = f_2(i_2) = a \wedge P_1 i_1 = t_1 \wedge P_2 i_2 = t_2 \wedge t_1 \ll t_2 \wedge a \in E_A^{max}$

Donnons le détail du calcul pour Δ_1. a_1 et a_2 correspondent à des paramètres qui font référence aux entrées adressées respectivement par $a_1 = A(j_2 - 1)$ et $a_2 = A(j_2 - 2)$ pour une itération j_2. Le calcul de cette dépendances a pour origine la résolution de l'égalité $j_1 - 1 = j_2 + 1 = a_1$

1. $\exists j_1 \in D_1, f(j_1) = a_1$ implique que $j_1 \in \{j_1 | a_1 = j_1 + 1 \wedge j_1 \in D_1\}$. en réécrivant cet ensemble pour orienter les équations afin que a_1 soit un paramètre on obtient l'ensemble suivant

$$S_{1(a_1)} = \{j_1 \mid j_1 = a_1 - 1 \wedge 2 \leq a_1 \leq 11\}$$

2. de manière similaire $\exists j_2 \in D_2, g(j_2) = a_1$ implique que $j_2 \in \{j_2 | a_1 = j_2 - 1 \wedge 3 \leq j_2 \leq 12\}$.

$$S_{2(a_1)} = \{j_2 \mid j_2 = a_1 + 1 \wedge 4 \leq a_2 \leq 13\}$$

3. $\mathcal{M}(s, j_2)$ Dans le cas présent, cela revient à déterminer si $j_2 \in I_{(a_1,a_2)}$ car la définition de l'ensemble des itérations modificatrices est ici identique à l'ensemble des itérations significative ; donc $j_2 \in \{j_2 | j_2 - 1 = a_1 \wedge j_2 - 2 = a_2 \wedge 3 \leq j_2 \leq 12\}$

$$S_{\mathcal{M}(a_1,a_2)} = \{j_2 | j_2 = a_1 + 1 = a_2 + 2 \wedge 4 \leq a_1 \leq 13\}$$

4. La dépendance Δ_1 correspond à l'intersection de ces ensembles. Etant donné que $j_1 = t_1$ et $j_2 = t_2$, l'ensemble résultant paramétré par a_1 et a_2 est donc :

$$\Delta_1(a_1, a_2) = \{(t_1, t_2) | t_1 = a_1 - 1, t_2 = a_1 + 1, a_1 + 1 = a_2 + 2, t_1 < t_2, 4 \leq a_1 \leq 11\}$$

19.3 Génération de code

En partant de la représentation générale d'un programme parallélisé donnée par P. Feautrier et [3] montrée à gauche de la figure ci après ; la génération de code produit le programme de droite. Prog représente un programme parallèle dont l'ordonnancement est fixé par l'exécution de la boucle `do`. L'index t représente la date d'exécution de plusieurs itérations exécutées en parallèle qui forment un front. $sparse$(Prog) représente le programme Prog restructuré pour convenir aux tableaux creux. En résumé, la parallélisation pour des structures creuses a pour but de trouver un nouvel ordonnancement de ces fronts en fonction du nouveau graphe de dépendances.

do $t \in T$
$\quad Prog(t)$
enddo
	`calculer le remplissage`
	`calculer le graphe de dépendance creux` $FrDG$
	Θ `=partition de` T
	`représentant un ordonnancement`
	do `t = 1,`$\vert\Theta\vert$
	doall $t' \in \Theta(t)$
	Sparse(P(t'))
	enddoall
	`synchronisation`
	enddo

19.4 Conclusion

Ce mode de parallélisation tente de capturer le parallélisme lié à des aspects dynamiques en simulant *en parallèle et symboliquement* l'exécution afin de déterminer une exécution numérique parallèle. Dans cette perspective, le rôle de l'analyse statique est de produire un programme d'analyse dynamique spécifique au programme traité. Concernant la génération du graphe de dépendances liée aux creux, l'interaction entre compilation et exécution est résumée par le schéma suivant :

	Remplissage		**Dépendances**		**Exécution**
Compilation	Synthèse de la fonction de remplissage	$\rightarrow$	Synthèse de la fonction de dépendance	$\rightarrow$	Génération de code
Exécution	Calcul du point fixe	$\rightarrow$	Calcul du graphe de dépendances	$\rightarrow$	Exécution numérique

Cette approche pose la question du coût de la phase de simulation et du gain apporté. Une première réponse concernant le coût peut être donnée par les résultats d'[1]expériences faites manuellement sur la factorisation Cholesky programmée à partir de la spécification produite par des outils d'analyses automatiques. En optimisant le remplissage pour réduire la redondance d'informations, les résultats montrent que cette phase est 100 fois plus rapide que celle de calcul. Concernant le gain, son efficacité est reconnue pour l'algorithme de factorisation creux de Cholesky. Aussi, notre objectif consiste à définir un cadre d'analyse automatique pour étendre à d'autres algorithmes cette approche et les optimisations attenantes.

[1]menées sur une SP2, 8 processeurs, 64 Mo/processeurs. $42913 \leq |E_A^0| \leq 1029655, 28627 \leq |E_A^{max}| \leq 6873127$

Bibliographie

[1] A.J.C Bik and H.A.G Wijshoff, *Automatic Data Structure Selection and Transformation for Sparse Matrix Computation*, in IEEE Transactions on Parallel and Distributed Systems, 1996, volume 7, pp. 1-19

[2] F.Delaplace and R. Adle, *Extension of the Dependence Analysis for the Sparse Computation*, PDCS97, 1997.

[3] P. Feautrier, *Some Efficient Solutions to the affine Scheduling Problem, I, one Dimensional time*, in International Journal of Parallel Programming, October 1992, volume 21, pp. 313-348

[4] M.T. Heath and E.Ng and B.W. Peyton, *Parallel Algorithm for Sparse Linear Systems*, in Siam Review, Sept 1991, volume 33, pp. 420-460

[5] P. Marquet and J.L. Dekeyser, *Irrégularité et dynamicité dans les langages à parallélisme de données*, in Spring School on Data Parallelism, 1996.

[6] M. Ujaldon and E. L. Zapata, *Efficient Resolution of Sparse Indirections in Data-Parallel Compilers*, in Proc. International Conference on Supercomputing, July 1995, pp. 117-126

Cinquième partie

Communications: optimisation et mise en oeuvre

Chapitre 20

Impact du modèle sur l'algorithmique

Afonso Ferreira (LIP) et Pierre Fraigniaud (LRI)

20.1 Introduction

Un des deux principaux objectifs de ce chapitre est une présentation de différents modèles du parallélisme, de la PRAM au modèle BSP. Cette présentation n'a pas pour objectif d'être exhaustive. La diversité des modèles présentés dans la littérature est en effet telle qu'une présentation exhaustive est à ce jour presque impossible. Il nous est apparu plus pédagogique de présenter les grandes classes de modèles, et d'axer cette présentation autour d'une classification. Nous pensons que la classification proposée s'avérera suffisamment efficace pour y faire entrer tout autre modèle de la littérature.

Le deuxième objectif de ce chapitre est de montrer l'influence du modèle sur les algorithmes développés. Ainsi, un algorithme de tri ou une inversion de matrices ne se traitera pas de la même manière sur le modèle PRAM et sur le modèle BSP. Dans le premier cas, le nombre de processeurs est lié à la taille du problème et les processeurs accèdent à une mémoire commune, alors que, dans le modèle BSP, le nombre de processeurs est une donnée du modèle et les processeurs procèdent par échanges de messages.

Bien sûr, nous discuterons de l'adéquation du modèle avec la "réalité" des machines parallèles existantes. Il convient cependant de ne pas se focaliser sur ce paramètre. D'une part, la technologie évolue si rapidement qu'il est loin d'être certain que les machines actuelles soient représentatives des machines de la prochaine génération. La définition de nouveaux modèles peut être en fait vue comme une façon de déterminer les fonctionnalités auxquelles les algorithmiciens aimeraient avoir accès dans le futur. D'autre part, il existe bien d'autres paramètres permettant de juger de la pertinence d'un modèle : sa portabilité, son aptitude à développer une théorie de la complexité, sa facilité d'utilisation, etc. Chacun de ces critères a plus ou moins d'importance selon l'utilisation faite du modèle. Un théoricien de la complexité préférera certainement la PRAM à un modèle plus proche des machines mais plus difficilement maniable. A l'inverse, un programmeur d'une application spécifique sur une catégorie de machines spécifique utilisera un modèle reflétant toutes les caractéristiques de la machine : entrées/sortie, cache, etc.

Afin d'insister encore sur la perte de temps que peut constituer une discussion sur une possible classification linéaire des modèles, prenons un exemple. Pour résoudre des problèmes tels que la prédiction météo ou le calcul de trajectoire d'une sonde sur Mars, il est important de se rapprocher de la machine. Ainsi, on adaptera un produit matriciel à la structure de la machine afin qu'il s'exécute le plus rapidement possible. Cependant, si on se pose la question de la *complexité* du produit matriciel, une telle approche ne permet pas de répondre car une machine est trop peu stable dans le temps pour que l'on puisse développer une théorie de la complexité reposant sur ses caractéristiques. C'est une des raisons pour lesquelles les théoriciens de la complexité préfèrent le modèle PRAM à beaucoup d'autres modèles. Un bon modèle est, selon nous, un modèle ayant un intérêt pour au moins un des points de vue exprimés ci-dessus : réalisme vis-à-vis des machines existantes, portabilité, possibilité de développements fondamentaux, facilité d'utilisation, prédictabilité, etc. Cette définition peut paraître laxiste. Elle offre cependant l'avantage d'éliminer quantité de variations autour de tel ou tel modèle.

Dans la section suivante, nous présenterons et classifierons les modèles majeurs du parallélisme. Dans la section 20.3, nous étudierons l'impact de ces modèles sur la manière de développer des algorithmes. Cet impact est en fait important. En effet, si les algorithmiciens arrivaient aux mêmes conclusions (c'est-à-dire aux mêmes algorithmes) quelque soit le modèle considéré, cela signifierait que tous les modèles sont identiques du point de vue de l'algorithmique qu'ils suggèrent. Ce n'est pas le cas : chaque modèle suggère des problèmes algorithmiques propres, et donc une charge de travail propre. Ne considérer qu'un seul modèle est donc dangereux pour un algorithmicien qui doit au contraire connaître, au moins dans les grandes lignes, les caractéristiques des principaux modèles afin de ne passer à côté d'aucun paradigme de programmation. Ceci nous amène d'ailleurs à faire une dernière remarque : il est à l'inverse possible que deux modèles paraissent totalement différents, l'un par exemple offrant des caractéristiques "appliquées", l'autre des aspects plus "fondamentaux", sans pour autant que les algorithmes développés sur ces deux modèles diffèrent significativement. Nous reviendrons sur ce point plus loin dans le texte.

La section 20.4 conclut le chapitre avec une discussion sur les perspectives des modèles de calcul parallèle.

20.2 Une classification des modèles du parallélisme

Comme nous l'avons dit précédemment, l'objectif de cette section n'est pas de lister l'ensemble des modèles du parallélisme présentés dans la littérature ces dernières années. Leur nombre est tel qu'une telle tâche demanderait un chapitre entier. D'autre part, la nuance entre différents modèles est si fine qu'il est préférable de se concentrer sur quelques modèles principaux afin d'en comprendre les caractéristiques majeures.

20.2.1 Présentation des principaux modèles

Par "principaux" modèles, nous entendons les modèles dont les spécifications sont très largement diffusées, et dont l'impact sur l'algorithmique est considérable, ne serait-ce qu'en terme du nombre de publications utilisant ces modèles comme base de la description et de l'étude de complexité des algorithmes proposés. Ce n'est en rien un jugement de valeur. Tous ces modèles ont des avantages et des inconvénients dont nous discuterons. Beaucoup de modèles sont d'ailleurs des variations autour des modèles décrits ci-dessous. Nous avons choisi une présentation historique plutôt que technique, et ce afin de bien faire saisir les raisons profondes qui ont provoquées l'émergence de tel ou tel modèle. Dans le chapitre suivant, nous classifierons les modèles en en dégageant les principales propriétés.

20.2.1.a Le modèle PRAM

Le modèle PRAM (pour *Parallel Random Access Machine*) est historiquement apparu le premier, alors que les super-calculateurs parallèles n'avaient pas encore vu le jour [6]. Il correspond à la vision d'une machine parallèle comme un ensemble de machines séquentielles (des machines RAM pour *Random Access Machine*) accédant à une mémoire commune. Plus formellement, le modèle PRAM se définit comme un ensemble arbitrairement grand de processeurs, chacun possédant un ensemble fini de registres et ayant accès à une mémoire commune, de taille elle aussi arbitraire. Le fait que le nombre de processeurs et la taille de la mémoire soient non bornés *a priori* ne doit pas choquer le lecteur ; simplement, ce sont des paramètres du modèle, et la complexité des problèmes sera exprimée en fonction de ces paramètres. En pratique, la mémoire est souvent considérée d'une taille du même ordre de grandeur que le nombre de processeurs. Une PRAM de n processeurs peut potentiellement exécuter n instructions simultanément, ces instructions pouvant être identiques ou non sur chaque processeur. Il n'y a pas de coût de communication : tous les processeurs accèdent à la mémoire commune en temps constant.

Le modèle PRAM s'est révélé être un outil formidable pour l'étude de la complexité parallèle des problèmes algorithmiques fondamentaux. Ce modèle a permis la définition formelle de classes de complexité comme la classe NC correspondant à l'ensemble des problèmes de taille n solvables en temps polylogarithmique sur un nombre polynomial de processeurs. Une des conjectures les plus fameuses de l'algorithmique parallèle est de savoir si $P = NC$, c'est-à-dire de savoir si l'ensemble des problèmes solvables séquentiellement en temps polynomial pouvaient être résolus en temps polylogarithmique sur une PRAM ayant un nombre polynomial de processeurs. Notez que la réciproque est triviale par simple simulation des n processeurs par un unique processeur : chaque étape parallèle nécessitant $O(n)$ étapes séquentielles.

20.2.1.b Les modèles à topologies explicites

D'un point de vue "pratique", le modèle PRAM est apparu critiquable à un grand nombre d'algorithmiciens parallèles dès le milieu des années 80. Le principal reproche fait à ce modèle était de ne pas tenir compte de la structure des machines parallèles alors disponibles : la majorité d'entre elles se révélaient être des machines à mémoire distribuée, bien loin des PRAM. L'accès à une donnée non résidante sur le processeur s'effectuait alors par un appel spécifique à une procédure de communication. La mémoire globale du modèle PRAM et l'hypothèse de lecture/écriture en temps constant semblaient ne pas correspondre aux machines d'alors pour lesquelles un accès à la mémoire locale était bien moins coûteux qu'un accès à la mémoire d'un processeur voisin, lui même moins coûteux qu'un accès à la mémoire d'un processeur lointain (i.e. non directement connecté).

Les algorithmiciens ont alors introduit de nouveaux modèles faisant intervenir la topologie de façon explicite : grille, hypercube, etc. Les architectures MIMD (pour *Multiple Instructions Multiple Data*) et SIMD (pour *Single Instruction Multiple Data*) ont mené naturellement à deux types de modèles : les modèles à *grains fins* modélisant les machines SIMD, en particulier les *Connection Machines* CM1 et CM2 [5], et les modèles à *gros grains* modélisant des machines telles que les *i*PCS d'Intel, ou les machines à base de *Transputers* en Europe [1].

Un grand nombre d'algorithmes ont été développés sur ces modèles. Ces modèles ont permis, non seulement de faire émerger des paradigmes de programmation parallèle (pipeline, regroupement de messages, partition des données, équilibrage de charge, etc.), mais aussi de mettre en évidence des techniques de programmation complexes et des problèmes fondamentaux non encore résolus. A titre d'exemple, le tri en $O(\log n\ (\log\log n)^2)$ sur hypercube permet de mesurer la complexité des techniques mises en œuvre. Le problème du tri en $O(\log n)$ sur hypercube est d'ailleurs toujours ouvert [7].

20.2.1.c Les modèles sans topologie

Deux facteurs ont modifié le comportement des algorithmiciens vis-à-vis des modèles à topologie explicite. D'une part, les communications des machines parallèles modernes tendent à masquer la distance entre les processus communiquant : que ceux-ci soient situés sur deux processeurs voisins ou distants ne modifie pas en première approximation le coût de la communication. D'autre part, le paradigme par passage de messages tend à être sérieusement concurrencé par d'autres paradigmes tels que la mémoire virtuellement partagée pour lesquels la topologie n'est plus un critère à prendre en compte par le concepteur de l'algorithme, ce critère devant être traité au niveau système. Ajoutons qu'aucune topologie n'a réellement émergé comme candidat "universel" d'interconnexion des processeurs d'une machine parallèle. Traiter d'une topologie particulière peut se révéler une très mauvaise approche si le programme conçu devait être amené à s'exécuter sur une autre topologie.

Ceci a donc amené les algorithmiciens à proposer des modèles ne tenant pas compte de la topologie, ou plutôt en tenant compte de façon implicite au travers d'autres paramètres. Un modèle fameux est le modèle BSP (pour *Bulk-Synchronous Parallel*) [11]. La machine parallèle est considérée comme un ensemble de p processeurs communicants disposant d'un processus de synchronisation. Les algorithmes s'effectuent en phases synchrones (*Synchronous*). Chaque phase se décompose à son tour en deux sous-phases : une sous-phase importante (*Bulk*) de calcul et une sous-phase de communication. A la fin de la sous-phase de calcul, chaque processeur envoie les résultats nécessaires aux autres processeurs et reçoit les données nécessaires à la phase de calcul suivante. Ce modèle ne considère pas la topologie du réseau, c'est seulement au travers des coûts de communication que cette topologie s'exprimera. Nous reviendrons en détail sur ce point dans la section suivante.

Les modèles tels que le modèle BSP, dont le CGM est un exemple, ont connu un grand succès [4]. Le fait de ne plus avoir à considérer la topologie s'est révélé un "plus" algorithmique considérable. La portabilité des algorithmes s'est considérablement améliorée de ce fait. De plus, les paramètres décrivant ce type de modèles permettent, s'ils sont choisis habilement, de coller à une certaine "réalité". Mais la richesse de ces modèles s'est aussi exprimée au travers des simulations formelles d'autres modèles, tels que les PRAM. On peut ainsi penser un algorithme dans le modèle PRAM, et le traduire en BSP par exemple. Prendre en compte de façon explicite la topologie nuit à ce genre de simulation : simuler une PRAM sur une architecture de topologie explicite G demande de simuler les lectures/écritures de la PRAM par des communications dans G. De telles communications peuvent s'avérer extrêmement complexes à mettre en œuvre.

20.2.2 Définition formelle et classification

Nous allons focaliser notre attention sur les modèles suivants : PRAM, BSP, CGM, G-RAM gros grain ou grain fin. Comme nous l'avons dit plus haut, ces modèles suffiront à faire apparaître les principales caractéristiques des modèles du parallélisme.

20.2.2.a Définitions formelles

PRAM. Dans le modèle PRAM, un ensemble arbitrairement grand de processeurs, chacun possédant un ensemble fini de registres, ont accès à une mémoire commune, de taille elle aussi arbitraire. Une *étape* de calcul PRAM correspond à l'exécution par l'ensemble des processeurs des séquences suivantes en parallèle :

- lecture d'une donnée dans la mémoire commune (la donnée lue est alors stockée dans un des registres locaux) ;
- calcul élémentaire (addition, multiplication, etc.) sur les données disponibles dans les registres locaux ;
- écriture en mémoire du contenu d'un registre local, par exemple celui stockant le résultat de l'opération élémentaire effectuée précédemment.

On distingue usuellement trois contraintes sur les accès des processeurs à la mémoire. La plus restrictive de ces contraintes suppose qu'une case mémoire ne peut être lue ou écrite que par au plus un processeur à la fois. C'est le modèle EREW pour *Exclusive Read, Exclusive Write*. Un premier relâchement de cette contrainte permet la lecture commune dans une même case mémoire, les écritures restant pour leur part exclusives. C'est le modèle CREW pour *Concurent Read, Exclusive Write*. Finalement, le modèle CRCW (pour *Concurent Read, Exclusive Write*) permet la lecture et l'écriture simultanées d'un nombre quelconque de processeurs dans une même case mémoire.

Remarque. Tout comme la RAM est une vision "haut niveau" de la machine de Turing, la PRAM peut être décrite en termes plus formels à partir des *circuits booléens* [10]. La définition ci-dessus reste cependant suffisante pour le propos de ce chapitre.

En un certain sens, le modèle PRAM permet de faire apparaître le parallélisme intrinsèque du problème puisqu'il est possible d'effectuer un nombre arbitrairement grand d'instructions en parallèle. De ce point de vue, les différentes variations EREW, CREW et CRCW ne sont souvent que des artifices facilitant la description des algorithmes. Ces modèles se simulent entre eux avec des facteurs $O(\log n)$, et sont donc virtuellement équivalents pour la plupart des problèmes considérés [6].

***G*-RAM.** Il s'agit d'un modèle pour machines parallèles à mémoire distribuée. Le réseau de la machine parallèle est représenté par un graphe G : les sommets représentent les processeurs et les arêtes représentent les canaux de communication. On distingue deux sous-modèles principaux. Le modèle à *gros grains* se base sur la donnée de deux paramètres distincts : le nombre de processeurs p et la taille du problème n. Des questions telles que le tri de n entiers sur p

processeurs, ou l'inversion d'une matrice $n \times n$ sur p processeurs ont un sens dans le modèle G-RAM à gros grains. Le modèle G-RAM à grains fins suppose sinon l'identité de ces deux paramètres, au moins une relation spécifique en terme d'ordre de grandeur. Dans ce type de modèle, on se pose par exemple la question du tri de n entiers en sous-entendant qu'il s'agira d'effectuer ce tri sur une "machine" avec $\Theta(n)$ processeurs. Tout comme pour la PRAM, le nombre de processeurs est arbitrairement grand dans le modèle G-RAM à grains fins.

Dans les deux cas, le modèle est *a priori* synchrone. Dans le modèle grains fins, chaque étape consiste en un calcul local et en un échange de données avec les processeurs voisins. Tout comme pour la PRAM, on peut distinguer plusieurs hypothèses selon la capacité des processeurs à communiquer simultanément ou non avec plusieurs voisins. Dans le modèle G-RAM à gros grains, chaque processeur effectue un plus grand nombre de calculs locaux avant de regrouper les données à envoyer durant la phase de communication [1].

Dans les deux types de "granularités", il n'y a pas de possibilité de communications distantes et/ou globales. Ainsi, si une donnée doit parvenir à un processeur non voisin, elle devra être relayée par des processeurs ou des routeurs intermédiaires. De même, si un processeur doit diffuser une information à l'ensemble des autres processeurs, tous les processeurs devront prendre part. Dans les deux cas, les communications nécessiteront plusieurs étapes (le coût de chaque étape pouvant varier selon la manière d'apprécier les communications – nous reviendrons sur cela dans la section suivante). Le nombre d'étapes dépend fortement de la topologie du réseau, et donc du graphe G.

Par exemple, dans la section 20.3, nous allons utiliser des G-RAM où G est en topologie en *hypercube* (voir la figure 20.2.2.a). Rappelons qu'un hypercube de dimension d, ou d-cube est composé de $n = 2^d$ processeurs P_i numérotés de 0 à $n-1$. Soit $i_{d-1}i_{d-2}\ldots i_1 i_0$ la représentation binaire de l'entier $i \in \{0, \ldots, n-1\}$. Les voisins de P_i sont slors les P_j dont la représentation binaire de j diffère de celle de i en exactement une position. Si elles diffèrent dans la position k, P_i et P_j sont dits voisins sur la dimension k. Ainsi, les processeurs sont les sommets d'un hypercube de dimension d, chacun relié aux d sommets voisins, sur d dimensions différentes. La plus grande distance dans un hypercube est celle entre deux processeurs dont les représentations binaires diffèrent dans toutes les positions. De ce fait, le *diamètre* d'un d-cube est d. Ceci implique en particulier que $\Omega(\log n)$ est une borne inférieure en temps pour la résolution de tout problème prenant en compte *toutes* les données dispersées sur *tout* le réseau [5].

BSP. Le modèle BSP cherche à abstraire la topologie du réseau et à ne la décrire qu'au travers de paramètres numériques. Ce modèle considère la machine

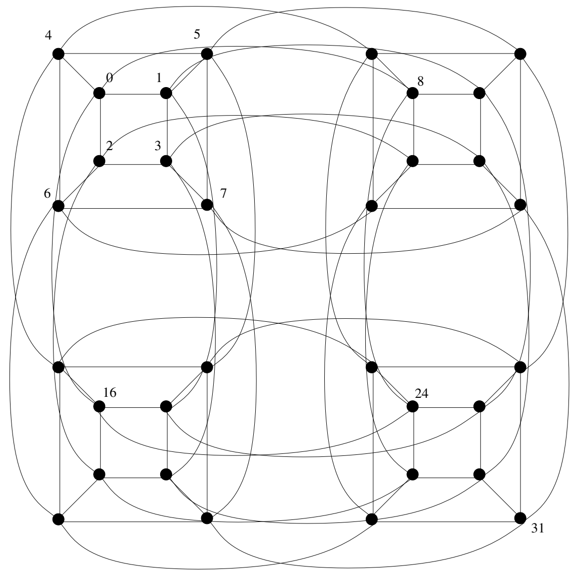


Figure 20.1 : *Un hypercube de 32 sommets.*

comme un ensemble de p processeurs reliés à un réseau d'interconnexion. Peu importe ici la structure du réseau. Il peut s'agir d'un réseau "point-à-point" comme dans les modèles G-RAM, mais on peut aussi supposer l'utilisation d'un réseau à bus, d'un réseau de stations de travail, voire d'une mémoire commune.

L'exécution d'un algorithme s'effectue par phases synchrones, chaque phase comprenant à son tour une phase de calcul et une phase de communication. Les coûts de calcul peuvent se définir de différentes manières mais il est communément admis de donner un coût unitaire à chaque opération élémentaire. Les coûts de communication dépendent de deux paramètres : la *latence* L et le *débit* g. Lorsque, durant la phase de communication, un processeur envoie h données élémentaires (des octets par exemple) et en reçoit h', le coût de cette communication sera de $L+max(h,h')g$. Le coût total de la phase de communication est $max_{i=1,\dots,p} L + max(h_i, h'_i)g$ où h_i (resp. h'_i) représente le nombre de données envoyées (resp. reçues) par le processeur i. Ce sont les paramètres L et g qui devront être suffisamment bien choisis pour refléter en partie la structure du réseau. Ainsi, le L d'un hypercube sera vraisemblablement plus petit que celui d'une grille à cause de son faible diamètre. De même, le fort degré et la large bissection de l'hypercube permettent de supposer que le paramètre g sera plus faible que pour une grille. Mais ceci dépend bien sûr aussi du mode de commutation et d'autres contraintes architecturales telles que la taille des caches ou la vitesse des bus internes et des DMA (pour *Direct Access Memory*, i.e., l'interface réseau).

On peut objecter qu'une communication entre voisins devrait coûter "moins cher" qu'une communication distante. Les utilisateurs du modèle BSP répondent par deux arguments. D'une part, les modes de commutation modernes (commutation de circuits, *wormhole*, accès à un bus *ethernet*, etc.) ne sont que très peu sensibles à la notion de localité (dans un réseau de stations de travail, il n'y a même pas du tout de notion de localité). D'autre part, il peut être plus important d'utiliser un modèle "portable" qu'un modèle plus précis mais ne modélisant qu'un type restreint de machines [11].

Dans le modèle BSP, il n'y a pas de notion spécifique de communications globales puisqu'en fait tout est global. Chaque phase de communication est une (h, h')-relation, une diffusion n'est donc qu'une communication parmi beaucoup d'autre, en fait une $(p-1, 1)$-relation.

Remarque. Le modèle LogP est très semblable au modèle BSP, nous n'en dirons donc pas plus sur ce modèle et renvoyons le lecteur à [3].

CGM. Le modèle CGM (pour *Coarse Grained Machine*) offre de grandes similarités avec le modèle BSP. Il en diffère principalement de par la forme des communications élémentaires. L'objectif du modèle CGM est de concevoir les algorithmes par briques base dont on peut espérer la mise en œuvre rapide sur chaque machine. De telles briques de base sont par exemple le tri. Beaucoup de problèmes discrets se résolvent ainsi à partir de l'utilisation répétée de tri. Des briques de base moins sophistiquées sont l'échange total ou la diffusion [4].

La classification de la section suivante va permettre de faire apparaître de façon encore plus nette les différences entre ces modèles.

20.2.2.b Classification

Nous avons choisi de classer les modèles en fonction de trois critères : la granularité, la prise en compte de la topologie et la manière dont le coût des communications est évalué.

- La granularité est, en première approximation, soit fine soit grossière. Dans le premier cas, le nombre de processeurs est arbitraire, et est généralement lié à la taille du problème. Dans le second cas, la taille du problème n et le nombre de processeurs p sont deux paramètres distincts. En général, les algorithmes jouent alors sur le rapport entre ces deux quantités : $n \simeq p$ permet des accès rapide en mémoire locale mais multiplie les communications, alors que $n \gg p$ limite le nombre de communications mais peut surcharger la mémoire locale d'un processeur. En fait, la granularité est une mesure du parallélisme disponible et/ou extractible d'un problème.

- La topologie de la machine semble un paramètre essentiel. Quoi de plus différent à première vue d'un chemin qu'un graphe complet ? Pourtant, comme le suggère le modèle BSP, il n'est pas forcément nécessaire d'expliciter la topologie, celle-ci pouvant apparaître de façon implicite au travers d'autres paramètres quantitatifs plus facilement manipulables. Un modèle à topologie *explicite* permet de raffiner l'algorithme mais peut le rendre trop dépendant de l'architecture. A l'inverse, un modèle à topologie *implicite* est portable, mais peut parfois se révéler peu fiable au sens où ses résultats pourront être peu en rapport avec l'expérience sur machine.

- Le coût des communications peut être apprécié de plusieurs manières [9]. Nous distinguerons les communications *fixes* des communications *paramétrées*. Dans le premier cas, une communication est une opération atomique au coût unitaire. Dans un modèle à grains fins par exemple, il pourra s'agir de l'échange d'une donnée entre deux processeurs voisins. Dans un modèle à gros grains, la notion de coûts fixes est plus délicate. Elle apparaît cependant sous-jacente dans le modèle G-RAM si l'on pense par exemple un algorithme en terme des phases d'échange de listes entre processeurs voisins. On cherche alors à minimiser le nombre de ces échanges sans qu'il soit nécessaire de modéliser le coût d'un échange proprement dit. Un modèle à communications paramétrées est un modèle dans lequel toutes les communications ne sont pas comptabilisées de façon identique. Une des mesures les plus utilisées est la quantité de données transmises. Mais, comme nous le verrons plus loin, d'autres mesures peuvent être prises en compte : temps d'initialisation, distance inter-processeurs, etc.

Avant de passer les différents modèles décrits ci-dessus au crible de ces trois critères, il convient de revenir encore sur un point crucial : notre objectif n'est pas de faire le procès de tel ou tel modèle. Il n'y a pas pour l'instant – et il n'y aura peut-être jamais –, de modèle universel satisfaisant tous les critères d'un bon modèle. D'un point de vue fondamental, un bon modèle est un modèle sur lequel on peut bâtir une théorie riche en ce qu'elle permet d'exprimer formellement un grand nombre de résultats considérés naturellement comme de première importance. Pour reprendre un exemple déjà cité, la question de $P = NC$ du modèle PRAM est cruciale et naturelle car il est légitime de se demander jusqu'à quel point un problème soluble séquentiellement en temps polynomial peut se paralléliser. D'un point de vue plus pratique, un bon modèle pourrait être un modèle dans lequel le comportent d'un très grand nombre de machines pourrait être exprimé. Mais même cette simple phrase prête à discussion. A quoi sert une description précise si le nombre et la complexité des paramètres ne permettent pas d'expliciter de façon simple le moindre algorithme ? Nous pensons qu'il est plus convenable de chercher à *qualifier* les modèles en fonction de leur utilité, et qu'il est illusoire de chercher à *quantifier* leur valeur intrinsèque sur une échelle linéaire forcément arbitraire.

La table 20.1 présente notre classification des modèles en fonction des trois critères listés ci-dessus. Le modèle PRAM est bien sûr à grains fins. La topologie est implicite, dans le sens ou elle n'est pas prise en compte. Notons qu'on pourrait estimer au contraire que la PRAM a intrinsèquement une topologie en graphe complet, ou encore que la mémoire partagée est une forme de topologie explicite. Ces arguments sont recevables. Nous préférons cependant considérer la PRAM comme à topologie implicite car, d'un point de vu algorithmique, la topologie n'intervient pas. Il n'est pas possible de jouer sur la topologie pour donner des bornes inférieures par exemple. La PRAM joue un rôle de point de référence dans les modèles à topologie explicite : on cherche souvent à simuler un algorithme PRAM sur une topologie donnée. Nous concédons cependant qu'il y a une part d'arbitraire dans notre discours.

Topologie	Gros grains		Grains fins	
	Comm. param.	Comm. fixes	Comm. param.	Comm. fixes
Explicite	G-**RAM** **"MIMD"**	G-**RAM** **"gros grain"**		G-**RAM** **"SIMD"**
Implicite	**BSP** (et **LogP**)	**CGM**		**PRAM**

Tableau 20.1 : Classification des modèles. On distingue les communications à coûts fixes ou paramétrées, les topologies implicites ou explicites, et la granularité.

Le modèle G-RAM SIMD est un modèle à grains fins, à communication fixes et a structure explicite. Le terme SIMD fait référence à toute une gamme de machines massivement parallèles ne possédant qu'un seul séquenceur distribuant les instructions à l'ensemble des processeurs. La contrainte SIMD est compensée sur ce type de machine par le nombre élevé de processeurs (> 65000 sur la CM2 de TMC), d'où le terme SIMD pour qualifier le modèle G-RAM grains fins. Dans ce modèle, la topologie est explicite : un hypercube, une grille, etc. La nature de grains fins implique quasi nécessairement de ne considérer que des communications atomiques à coût unitaire.

Les modèles BSP et CGM sont des modèles à gros grains et à topologie implicite. Ils diffèrent l'un de l'autre par les coûts de communication. BSP modélise ces coûts en les paramétrant par la latence et le débit. De ce fait, la topologie de la machine pourra être cachée derrière les paramètres L et g du modèle BSP. Le modèle

CGM, version simplificatrice de BSP, s'affranchit de ces paramètres. L'objectif des algorithmes CGM est de minimiser les phases de communication, phases elles-même potentiellement complexes.

Les modèles G-RAM "MIMD" et "gros grains" sont tous deux des modèles à topologie explicite et à granularité grossière. Ils se distinguent par les coûts de communication de la même façon que ce critère a permis de différencier BSP de CGM. Dans le modèle G-RAM MIMD, les communications sont modélisées très finement. Par exemple, dans un modèle à commutation de messages, le coût des communications de voisin à voisin est modélisé par une fonction affine de la taille L des messages : $\beta + L\tau$ où β représente un temps d'initialisation (logiciel et matériel) et $1/\tau$ représente la bande passante des liens. Dans un modèle par commutation de circuits, ce même temps est représenté par : $\beta + d(x,y)\delta + L\tau$ où $d(x,y)$ dénote la distance entre la source x et la destination y, et où δ est fonction du temps de commutation des routeurs intermédiaires. Le modèle "gros grains" ne précise pas aussi clairement les coûts de communication : on compte comme dans le cas des G-RAM SIMD, i.e., chaque communication atomique aura un coût unitaire.

Notons que deux cases restent vides : les modèles grains fins à communications paramétrées, et à topologie explicite ou implicite. Ceci provient simplement du fait qu'un modèle grains fins suppose souvent, voire toujours, des communications atomiques et donc des coûts unitaires. On compte les phases de communication, mais chaque phase a un coût constant, la quantité de données transférées étant constante à chaque phase.

20.3 Impact du modèle sur l'algorithmique : quelques exemples

Dans cette section, nous allons montrer, à travers quelques exemples, les principales caractéristiques de chaque modèle, et leur impact sur l'algorithmique. Nous commençons par les modèles aux communications à coût fixe. Dans ce cadre, nous avons considéré un problème simple mais fondamental : les sommes partielles préfixées. Nous allons voir comment les solutions à ce problème se déclinent selon le modèle. Nous traiterons dans un deuxième temps des modèles aux communications à coût paramétré. Pour cela, nous considérerons un problème pour lequel les coûts de communication et les coûts de calcul sont du même ordre : l'extraction de racines de polynômes (aucun pré-requis en calcul numérique ne sera nécessaire pour suivre cet exemple).

20.3.1 Un problème d'école : les sommes partielles préfixées

Soit un ensemble de n éléments $v_1, v_2, \ldots, v_n$ muni d'une loi associative $\oplus$. Les sommes partielles préfixées de ces n éléments sont les n sommes partielles $S_k = \oplus_{i=1}^{k} v_i$ pour $k = 1, \ldots, n$. Le problème à résoudre est donc le suivant :

> *Problème* Sommes partielles préfixées
>
> **Données :** un ensemble E muni d'une opération associative $\oplus$;
>
> **Entrée :** n éléments $v_1, v_2, \ldots, v_n$ de E ;
>
> **Sortie :** $S_k = v_1 \oplus \ldots \oplus v_k, k = 1, \ldots, n$.

Par exemple, imaginez que les entrées soient $v_1 = 1$, $v_2 = 2$, $v_3 = 4$, $v_4 = 3$, $v_5 = 5$, $v_6 = 1$ et $v_7 = 4$ dans l'ensemble des entiers munis de l'opération d'addition. Les sommes partielles préfixées seront donc $S_1 = 1$, $S_2 = 3$, $S_3 = 7$, $S_4 = 10$, $S_5 = 15$, $S_6 = 16$ et $S_7 = 20$.

20.3.1.a Les sommes partielles préfixées sur PRAM

Nous reprenons ici l'algorithme décrit dans [8]. Il est basé sur la technique, classique, du "Diviser pour Paralléliser", elle même basée sur la technique "Diviser pour Régner" séquentiel. Cette technique mène à la réduction du problème à des instances indépendantes plus petites qui sont résolues récursivement et en parallèle. La construction de la solution du problème initial est alors faite à partir des solutions de chacune des sous-instances. Plus formellement, un algorithme PRAM de calcul des sommes préfixées a la forme suivante (on suppose pour simplifier les notations que n est une puissance de 2).

1. Réduction : diviser la séquence $A = v_1, \ldots, v_n$ en deux sous-séquences de taille $n/2$, à savoir $A_1 = v_1, \ldots, v_{\frac{n}{2}}$ et $A_2 = v_{\frac{n}{2}+1}, \ldots, v_n$.
2. Récursion : calculer en parallèle les sommes préfixées $S_1, \ldots, S_{\frac{n}{2}}$ pour A_1 et $S'_1, \ldots, S'_{\frac{n}{2}}$ (c'est-à-dire en fait $S_{\frac{n}{2}+1}, \ldots, S_n$) pour A_2.
3. Fusion : obtenir les sommes préfixées manquantes en calculant $S_{\frac{n}{2}+j} = S'_j + S_{\frac{n}{2}}$, pour $j = 1, \ldots, n/2$.

Considérons une PRAM CREW de n processeurs. Les phases de réduction ont un coût nul. Etant donné deux résultats partiels obtenus par la phase de récursion, les fusionner requiert un temps constant. En effet, $\frac{n}{2}$ processeurs $p_1, \ldots, p_{\frac{n}{2}}$ lisent la même case mémoire où est stockée $S_{\frac{n}{2}}$ et ajoutent cette valeur à S'_j. On a donc un

coût t_n pour une séquence de n éléments qui satisfait : $t_n = t_{\frac{n}{2}} + O(1)$, c'est-à-dire en temps $O(\log n)$.
Remarque. Le temps est calculé ici en nombre d'opérations $\oplus$.

20.3.1.b Les sommes partielles préfixées sur l'hypercube SIMD

L'hypercube a été décrit dans la section 20.2.2.a. Nous aurons besoin des notations suivantes : soit $(i)_2$ la représentation binaire de i, et soit i_k le k-ième bit de $(i)_2$, de droite à gauche en commençant à $k = 0$. Si les processeurs P_i et P_j sont voisins dans la dimension k, nous noterons $j = i^{\underline{k}}$.

L'algorithme PRAM décrit plus haut ne peut s'appliquer directement sur l'hypercube pour deux raisons. D'une part, dans la PRAM, tous les processeurs manipulent une mémoire commune, ce qui n'est justement pas le cas de l'hypercube où la mémoire est répartie entre les processeurs. D'autre part, les processeurs d'une PRAM CREW accèdent de façon concurrente à la mémoire en lecture. Par conséquent, il faut concevoir un autre algorithme pour notre problème. La technique “Diviser pour Paralléliser” sera utilisée de nouveau grâce aux caractéristiques topologiques récursives de l'hypercube [5]. Les accès à la mémoire partagée de la PRAM seront remplacés par des échanges d'information sur des dimensions consécutives, et le besoin d'accès concurrent sera remplacé par l'existence d'une variable auxiliaire qui stockera la somme totale d'une sous-instance.

Nous supposons que chaque processeur P_i possède des variables $S(i)$, $T(i)$ et $ST(i)$, pour stocker, respectivement, la somme partielle préfixée, la somme totale et le sous-total. L'algorithme ci-dessous procède dimension par dimension. Il est optimal, i.e., il termine en temps $O(\log n)$. (Souvenez vous que tout algorithme prenant en compte toute les données réparties sur tout l'hypercube nécessite un temps $\Omega(\log n)$ puisque c'est le diamètre de l'hypercube. Ici, les données sont réparties une par processeur et la somme totale requiert la connaissance de toutes les données.)

Procédure Sommes partielles préfixées sur hypercube grains fins.
Entrée : une variable $v(i)$ par processeur P_i.
Sortie : pour chaque P_i la sortie est $(S(i) = \bigoplus_{j<i} v(j))$.

```
Chaque P_i execute :
  T(i):=S(i):= v(i) ;
  pour k := 0 jusqu'a d-1 faire
    (1) echanger T(i) avec T(i^k) et stocker dans ST(i) ;
    (2) si i_k = 1 alors S(i) := S(i) ⊕ ST(i) ;
    (3) T(i) := ST(i) ⊕ T(i) ;
```

Analysons cet algorithme. Il s'exécute en $d = \log_2 n$ étapes. A l'étape $k, k = 0, \ldots, d-1$, le processeur P_i communique avec le processeur $P_{i^{\underline{k}}}$. Montrons par récurrence qu'à l'étape k, un processeur P_i a calculé les sommes partielles préfixées des données de l'hypercube de dimension $k+1$ le contenant et ne faisant intervenir que les dimensions $0, 1, \ldots, k$. C'est vrai pour $k = 0$ (l'hypercube est réduit à un sommet). Si c'est vrai à l'étape k, alors la propriété reste vraie à l'étape $k+1$ car, après chaque échange (instruction (1)), et pour chaque paire de processeurs ayant échangé de l'information, le plus grand des deux rajoute la somme totale du sous-cube opposé dans la dimension k à sa somme partielle (instruction (2)). Tous les processeurs mettent ensuite à jour la somme totale (instruction (3)).

Remarque. Notez que l'opération `échange` de l'instruction (1) est effectuée en une seule étape de communications sur l'hypercube car les processeurs impliqués sont toujours voisins. Par conséquent, la complexité de l'algorithme de calcul des sommes partielles sur hypercube est, tout comme sur la PRAM CREW, en $O(\log n)$.

La PRAM CREW n'est pas plus "puissante" que l'hypercube SIMD pour le calcul des sommes partielles préfixées. Cela se vérifie d'ailleurs pour d'autres types de problèmes. C'est ce qui a fait, en partie, le succès de l'hypercube : sa nature récursive binaire intrinsèque en fait la structure idéale pour la mise en œuvre du paradigme "diviser pour régner", paradigme de base de l'algorithmique. Notons cependant que l'obtention d'algorithmes aussi performants que ceux de la PRAM nécessite souvent un travail plus important. Notons aussi qu'il y a des problèmes qui semblent plus difficiles sur hypercube que sur PRAM. C'est le cas du tri : existe-t-il un algorithme de tri en 0/1 sur hypercube SIMD ? Plus précisément, étant données $n = 2^d$ entrées binaires $b_i, i = 1, \ldots, n$ réparties arbitrairement sur les n processeurs (b_i sur P_i), existe-t-il un algorithme en $O(\log n)$ étapes d'échanges-comparaisons tel qu'après l'exécution de l'algorithme, on ait : pour tout $i \leq j$,

$$P_i \text{ possède } b_{i'} \text{ et } P_j \text{ possède } b_{j'} \Rightarrow b_{i'} \leq b_{j'}.$$

Ce problème est encore ouvert à ce jour (le meilleur algorithme est en $O(\log n\ (\log\log n)^2)$).

20.3.1.c Les sommes partielles préfixées sur l'hypercube à gros grains

Il n'y a pas de grandes différences entre l'algorithme conçu pour l'hypercube SIMD et celui que l'on peut mettre en œuvre sur un hypercube à gros grains. La seule modification intervient au niveau de la répartition des données : chaque processeur possède au départ n/p données. L'algorithme procède en $\log_2 p$ itérations. Initialement, chaque processeur effectue la somme partielle préfixée de ses données propres. Supposons alors pour simplifier que le rapport n/p soit une puissance de 2. On se retrouve donc presque dans la situation de l'hypercube grain fin de dimension $\log_2 n$ après $\log_2 n - \log_2 p$ itérations. La seule nuance est que lors des $\log_2 p$ itérations à suivre, chaque processeur simule en fait le comportement d'un hypercube de n/p processeurs. Après l'échange, le processeur d'indice le plus élevé met à jour toute sa liste de sommes partielles préfixées en leur ajoutant le sous-total de son voisin.

Le nombre d'itérations est donc $\log_2 p$. Par contre, ce type de modèle ne traite pas explicitement du rapport entre les calculs et les communications. On voit cependant apparaître une façon d'agglomérer les calculs en passant de grains fins à gros grains. C'est important si l'on désire mettre en évidence des propriétés d'extensibilité tant de l'algorithme (ici le paradigme "diviser pour régner") que de l'architecture (ici un hypercube). A l'inverse, le modèle ne fait pas directement apparaître l'équilibre entre calcul et communication. Dans le cas des sommes partielles préfixées, cela ne joue pas puisque les échanges consistent uniquement en données atomiques (le sous-total). Il existe cependant des cas pour lesquels plus le calcul est important moins on communique, et réciproquement. Nous étudierons cela dans le cas de l'hypercube MIMD.

20.3.1.d Les sommes partielles préfixées sur le modèle CGM

Les algorithmes écrits dans le modèle CGM sont typiquement décomposés en une succession de deux phases : une phase où chaque processeur effectue un calcul important sur ses données locales, et une phase où les processeurs échangent des données afin de les redistribuer. Pendant cette dernière phase chaque processeur envoie en général $O(\frac{n}{p})$ données et en reçoit $O(\frac{n}{p})$.

Dans le modèle CGM, deux contraintes sur la taille et le stockage des données sont souvent considérées. D'une part les mémoires des processeurs sont supposées bornées par $O(\frac{n}{p})$. Ceci provient d'un souci de travailler en faisant varier p tout en gardant une taille mémoire globale correspondant à la taille du problème, ici n. D'autre part, si l'on suppose par hypothèse que $n \gg p$, on souhaite aussi pouvoir pousser l'extensibilité des algorithmes au maximum, c'est-à-dire approcher le cas limite $n = p$. Pour cela, on fait une deuxième hypothèse sur le rapport n/p en supposant $\frac{n}{p} = \Omega(p^{\epsilon})$, où ϵ est une constante positive. Le souhait est de rendre ϵ le plus petit possible.

L'algorithme CGM de calcul des sommes préfixées sur p processeurs est décrit ci-dessous. Les $\frac{n}{p}$ valeurs stockées dans le processeur P_i seront notées $v_1(i), \ldots, v_{\frac{n}{p}}(i)$, $i = 1, \ldots, p$. L'algorithme est composé de quatre étapes :

1. chaque P_i calcule la somme totale locale $T(i)$;
2. les processeurs effectuent un échange total de leur sommes totales locales de façon à ce que chaque processeur P_i connaisse les sommes totales locales de tous les autres processeurs $P_j, j \neq i$;
3. chaque P_i calcule la somme des sommes locales des processeurs d'indice inférieur, i.e., $ST(i) = \bigoplus_{j<i} T(j)$;
4. chaque P_i calcule les sommes partielles locales $S_k(i) = ST(i) \oplus v_1(i) \oplus \ldots \oplus v_k(i), k = 1, \ldots, \frac{n}{p}$.

On reconnaît dans cet algorithme un des soucis du modèle CGM : limiter le nombre de phases de communication (ici une unique phase). Notons que la phase de communication par elle même peut être coûteuse (ici un échange total), mais l'objectif est justement de découper l'algorithme en phases de communications dont on peut supposer que la mise en œuvre soit optimisée pour chaque machine.

Quel est le degré maximum de parallélisme que l'on peut tirer de cet algorithme ? Lors de la phase de communication, chaque processeur reçoit p sommes locales. La taille mémoire étant bornée par $O(\frac{n}{p})$, on doit avoir $p = O(\sqrt{n})$. Cet algorithme n'est donc pas arbitrairement extensible et, en particulier, on ne peut pas faire tendre p vers n.

20.3.2 Un autre problème d'école : l'extraction de racines de polynômes

Soit $P(x) = \sum_{i=0}^{n} a_i x^i$ un polynôme de degré n à coefficients réels ou complexes. Un tel polynôme a exactement n racines complexes (potentiellement multiples) $\omega_1, \ldots, \omega_n$. Une méthode classique d'extraction de racine d'un tel polynôme repose sur l'itération de Newton appliquée au système de Viète, c'est-à-dire aux fonctions symétriques des racines. On obtient ainsi l'itération de Durand-Kerner [2]. Tout ce qu'il suffit de savoir pour la suite de ce chapitre est que cette méthode produit l'algorithme itératif suivant :

$$z_i^{(k+1)} = z_i^{(k)} - \frac{P(z_i^{(k)})}{\Pi_{j \neq i}(z_i^{(k)} - z_j^{(k)})}, \qquad i = 1, \ldots, n. \tag{20.1}$$

On initialise les variables $z_i^{(0)}$, $i = 1, \ldots, n$ de façon convenable (il existe des méthodes pour cela, mais ce n'est pas le propos de ce chapitre), et on effectue l'itération de l'équation 20.1 jusqu'à un certain critère d'arrêt (dont il serait de même hors de propos de discuter ici). Nous allons nous intéresser à la parallélisation d'une itération. Notre problème simplifié est donc le suivant :

Problème Itération de Durand-Kerner

Données : un polynôme $P(x) = \sum_{i=0}^{n} a_i x^i$;

Entrée : n réels distincts deux à deux $y_1, \ldots, y_n$;

Sortie : n réels $z_1, \ldots, z_n$ tels que $z_i = y_i - \frac{P(y_i)}{\Pi_{j \neq i}(y_i - y_j)}$ pour $i = 1, \ldots, n$.

20.3.2.a Extraction de racines de polynômes sur l'hypercube MIMD

Nous supposons ici que les n données y_i sont équi-réparties sur les p processeurs, avec au plus $\lceil \frac{n}{p} \rceil$ données par processeur. On souhaite que les résultats z_i soient de même équi-répartis entre les processeurs. Pour $j = 1, \ldots, p$, on note X_j l'ensemble des indices i tels que P_j détient la donnée y_i et calcule la valeur z_i. L'algorithme se décompose en deux phases :

1. échange-total des $y_i, i = 1, \ldots, n$ entre les p processeurs de façon à ce que chaque processeur connaisse l'ensemble des y_i ;
2. chaque processeur P_i effectue $z_i = y_i - \frac{P(y_i)}{\Pi_{j \neq i}(y_i - y_j)}$ pour $i \in X_j$.

Un algorithme possible d'échange total sur hypercube de dimension $d = \log_2 p$ procède en d phases numérotées de 0 à $d-1$. A la phase i, chaque processeur échange avec son voisin dans la dimension i toutes les informations qu'il a collectées lors des phases précédentes. A chaque phase, la quantité de données échangées double. Si $\beta + L\tau$ exprime le coût d'un échange de L données atomiques (un entier, un réel, etc.), alors le coût d'un tel échange total est de $\sum_{i=0}^{d-1}(\beta + 2^i \frac{n}{p}\tau)$ puisqu'à la première phase une donnée est échangée, à la deuxième deux, à la troisième quatre, etc. Cela donne un coût total de $\log_2 p\, \beta + (p-1)\frac{n}{p}\tau$. On ne peut malheureusement pas négliger le terme logarithmique car, d'un point de vue pratique, $\beta \gg \tau$ sur la plupart des machines distribuées.

En ce qui concerne les calculs, chaque affectation $z_i = y_i - \frac{P(y_i)}{\Pi_{j \neq i}(y_i - y_j)}$ pour un i donné s'effectue en $O(n)$ opérations élémentaires $(+, -, /, *)$. On a donc un coût de calcul sur chaque processeur de l'ordre de $\frac{n^2}{p}\alpha$ où α modélise le coût d'une opération arithmétique élémentaire. Le coût global de l'algorithme est donc de l'ordre de

$$\log_2 p\, \beta + n\tau + \frac{n^2}{p}\alpha.$$

Il convient alors de noter deux caractéristiques du modèle G-RAM MIMD, la première positive, et la seconde négative. D'une part, le modèle permet d'équilibrer calcul et communication. Ainsi, dans le coût total exprimé ci-dessus, le coût des communication croit en fonction de p (de façon logarithmique, mais avec un facteur constant important β), alors que le coût des calculs décroît linéairement (mais avec un facteur α faible). On peut donc calculer, étant données des valeurs de α, β et τ, le nombre optimal de processeurs permettant de minimiser le coût global des calculs plus des communications. C'est un point extrêmement positif du point de vue de la prédictabilité. A l'inverse, l'analyse de l'algorithme repose sur le coût de l'échange total. Ce coût peut être complexe à estimer analytiquement pour des architecture plus complexes que l'hypercube. Cela peut restreindre le domaine d'application du modèle.

20.3.2.b Extraction de racines de polynômes dans le modèle BSP

L'algorithme décrit dans la section précédente s'applique tout à fait au modèle BSP, seule son analyse, et donc sa mise au point, va être modifiée par le modèle.

Un échange total dans ce contexte est une $(\frac{n}{p}, (p-1)\frac{n}{p})$-relation puisque chaque processeur envoie $\frac{n}{p}$ données et en reçoit $(p-1)\frac{n}{p}$. Le coût de cette opération dans le modèle BSP est donc de l'ordre de $L + ng$. Les coûts de calcul restent identiques à ceux calculés dans la section précédente. Si α représente le coût d'une opération arithmétique élémentaire, on obtient un coût global dans le modèle BSP de

$$L + ng + \frac{n^2}{p}\alpha. \tag{20.2}$$

On peut donc noter ici une grande différence entre un modèle à topologie implicite (par exemple BSP) et un modèle à topologie explicite (par exemple G-RAM). Dans le premier cas, ce sont les paramètres numériques, en l'occurrence L et g dans le modèle BSP, qui caractérisent la topologie. Dans le second cas, les paramètres de communication (β et τ) n'expriment que le coût des communications entre voisins. Nous avons vu comment jongler avec ces paramètres dans la section précédente. Nous allons voir comment user des paramètres L et g d'une manière semblable dans le contexte BSP.

L'équation 20.2 peut se réécrire $L(G_p) + n\ g(G_p) + \frac{n^2}{p}\alpha$ pour insister sur le fait que L et g dépendent en réalité de la topologie G_p considérée. BSP permet de dériver un algorithme générique, et d'en exprimer le coût formel. C'est la force de ce modèle qui assure une certaine portabilité des solutions proposées. Il permet de distinguer la phase de conception de la phase de mise au point, à l'inverse des modèles à topologie explicite qui souvent mélangent ces deux phases (il faut par exemple assurer la communication entre voisins dans l'hypercube). Partant d'un algorithme générique, il est alors possible de l'optimiser en fonction des paramètres. Parmi une famille d'architectures $\{G_p\}_p$ paramétrées par le nombre de processeurs, il est possible de choisir l'architecture G_{p_0} telle que

$$L(G_{p_0}) + n\ g(G_{p_0}) + \frac{n^2}{p_0}\alpha = \min_p \left(L(G_p) + n\ g(G_p) + \frac{n^2}{p}\alpha \right).$$

20.4 Discussion

Nous pensons que la première notion à retenir, quand on parle de modèles algorithmiques, est qu'un modèle résulte de choix *a priori* arbitraires. Ainsi, la concepteur d'un modèle souhaite faire apparaître telles ou telles fonctionnalités, peut-être au détriment d'autres caractéristiques. Le modèle PRAM est simple et permet d'extraire le parallélisme d'un problème ; le modèle G-RAM à topologie explicite assure une bonne prédictabilité des coûts mesurés ; le modèle BSP permet le développement d'algorithmes génériques ; etc. Il convient donc qu'un algorithmicien connaisse toutes les grandes classes de modèles et soit apte à critiquer ses algorithmes non seulement d'un point de vue intra-modèle, mais aussi inter-modèles.

La deuxième notion à retenir est qu'il convient de ne pas balayer trop rapidement un modèle sous des prétextes conjoncturels. Par exemple, qui aurait travaillé sur un hypercube de dimension 16 dans les années 70 ? Pourtant la *Connection Machine* 2 comportait plus de 65000 processeurs en hypercube dès la fin des années 80. A la jonction des années 80 et 90 justement, la plupart des machines parallèles étaient d'ailleurs des hypercubes. Il en suivit un engouement considérable pour cette topologie et pour toutes les topologies concurrentes (de Bruijn, CCC, Mélange Parfait, etc). Les modèles à topologie explicite connurent alors un succès énorme. La fin des années 90 voit la difficulté des machines distribuées massivement parallèles à percer (problèmes de coûts, de support, de manque de logiciels et d'environnements de programmation, etc.), et leur remplacement par des machines à mémoire partagée, voire par des réseaux de stations de travail. Le modèle BSP connaît donc actuellement un succès important. Sera-t-il éphémère lui aussi ? Son approche haut niveau pourrait peut-être lui permettre de perdurer, même en cas de bouleversements technologiques. Notons que certains auteurs éminents continuent de penser que le modèle PRAM est représentatif de ce vers quoi tend le parallélisme (ce parallélisme pouvant s'exprimer au niveau des instructions, plus que des processeurs). A suivre...

Bibliographie

[1] Akl (S. G.). – *Design and Analysis of Parallel Algorithms.* – Prentice-Hall International, 1989.

[2] Cosnard (M.) et Fraigniaud (P.). – Analysis of aynchronous polynomial root finding me thods on a distributed memory multicomputer. *IEEE Transactions on Parallel and Distributed Syst ems*, vol. 5, n° 639-648, 1994.

[3] Culler, Karp (R.), Patterson (D.), Sahay (A.), Schauser (K.), Santos (E.), Subrarnonian (R.) et von Eicken (T.). – LogP: Towards a realistic model of parallel computation. *In : Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming.*

[4] Dehne (F.), Fabri (A.) et Rau-Chaplin (A.). – Scalable parallel geometric algorithms for coarse grained multicomputers. *In : ACM 9th symposium on computational geometry*, pp. 298–307.

[5] Ferreira (A.). – *In : Handbook of Parallel and Distributed Computing*, éd. par Zomaya (A.), chap. Parallel and Communication Algorithms for Hypercube Multiprocessors. – New York (USA), McGraw-Hill, 1996.

[6] Karp (R. M.) et Ramachandran (V.). – A survey of parallel algorithms for shared-memory machines. *In : Handbook of Theoretical Computer Science*, éd. par van Leeuwen (J.). – Elsevier/MIT Press, 1990.

[7] Leighton (F.). – *Introduction to Parallel Algorithms: Arrays, Trees, Hypercubes.* – San Mateo, CA, Morgan-Kaufmann, 1991.

[8] Roch (J.). – Algorithmique PRAM. *In : Algorithmes parallèles : analyse et conception*, chap. 5, pp. 105 – 128. – Hermes, 1994.

[9] Rumeur (J. D.). – *Communication dans les réseaux de processeurs.* – Masson, 1994, *Collection Etudes et Recherches en Informatique.*

[10] Ruzzo (W.). – On uniform circuit complexity. *J. Computer and System Sciences*, vol. 22, n° 3, 1981, pp. 365–383.

[11] Valiant (L.). – A bridging model for parallel computation. *Communication of ACM*, vol. 38, n° 8, 1990, pp. 103–111.

Chapitre 21

Placement statique et émulation

Dominique Barth (LRI), François Pellegrini (LaBRI)

21.1 Introduction

La mise au point d'applications efficaces s'exécutant sur des systèmes parallèles ou répartis nécessite tout d'abord une modélisation pertinente de l'algorithme parallèle à exécuter puis, à partir de cette modélisation, une implantation judicieuse sur l'architecture d'exécution (machine parallèle ou réseau de stations de travail). Dans ce cadre, nous présentons deux approches complémentaires. D'une part, nous étudions le *placement statique de tâches*, dont le but est de fournir, le plus souvent au moyen d'algorithmes heuristiques, des implantations efficaces d'algorithmes parallèles sur des architectures parallèles. D'autre part, nous étudions l'*émulation* qui, dans le cas d'applications parallèles classiques (transformée de Fourier rapide, calcul matriciel, communications globales), propose des modélisations des algorithmes sur des structures régulières efficacement simulées par l'architecture cible.

Dans ce cours, après avoir défini la notion de placement, outil théorique de base pour les deux approches, nous présentons différentes méthodes utilisées pour le placement de tâches et l'émulation.

21.1.1 Placement de graphes

Tous les travaux et études présentés par la suite utilisent la notion de *placement de graphes*. Un placement d'un graphe G sur un graphe H est la donnée de deux applications p et R, où p associe un sommet de H à chaque sommet de G, et où R associe à chaque arête (ou arc) (u, v) de G une chaîne entre $p(u)$ et $p(v)$ (ou de $p(u)$ à $p(v)$) dans H.

La nature des graphes G et H, et les contraintes et paramètres étudiés pour un placement sont fonction de l'étude visée. Le réseau H modélise toujours une architecture parallèle sur laquelle doit s'exécuter un programme. Le réseau G représente le graphe de tâches d'une application parallèle dans l'étude du placement de tâches, et un réseau d'interconnexion sur lequel s'exécute un algorithme parallèle dans le cas de l'émulation.

21.2 Placement de tâches

21.2.1 Placement statique

L'exécution efficace d'un programme parallèle sur une machine parallèle nécessite que les processus communicants du programme soient affectés aux processeurs de la machine de manière à minimiser le temps total d'exécution. Lorsque tous les processus coexistent simultanément pendant toute la durée d'exécution du programme, ce problème d'optimisation est appelé *placement*. Il revient à équilibrer le coût de calcul des processus sur les processeurs de la machine, tout en réduisant le coût de communication induit par le parallélisme en conservant les processus fortement inter-communicants sur des processeurs aussi voisins que possible. Un placement est dit *statique* s'il est précalculé avant le lancement du programme et n'est jamais remis en cause par la suite, et *dynamique* si les processus peuvent être déplacés en cours d'exécution ; dans ce dernier cas, il faut faire appel à des outils de *régulation de charge* pour effectuer la migration des processus.

Le problème du placement statique peut être modélisé en représentant à la fois le programme et l'architecture parallèles sous forme de graphes valués. Le programme parallèle est décrit par un *graphe source*, ou *graphe de processus*, dont les sommets représentent les processus et les arêtes les échanges de données entre processus voisins. Les sommets et arêtes du graphe peuvent être pondérés par des valeurs entières représentant respectivement le coût de calcul associé au processus et le volume de données à transmettre. De même, l'architecture parallèle est décrite par un *graphe cible*, ou *graphe d'architecture*, dont les sommets représentent les processeurs et les arêtes les liens de communication entre processeurs. Les sommets et arêtes du graphe peuvent également être pondérés par des valeurs entières représentant respectivement la puissance relative du processeur et le coût de traversée du lien.

Le placement du programme parallèle sur l'architecture parallèle est alors modélisé par le placement du graphe de processus sur le graphe d'architecture (voir section 21.1.1 ci-dessus), de telle sorte que tous les processus affectés au même processeur soient chargés et exécutés par celui-ci. Dans un contexte SPMD ("*Single Process, Multiple Data*"), ceci revient à *distribuer* les données sur les processeurs ;

dans ce cas, toutes les données affectées à un même processeur sont gérées par un unique processus s'y exécutant.

On peut remarquer que, suivant ce modèle, le placement statique se présente comme une variante de l'*ordonnancement*, puisque le graphe orienté modélisant les dépendances logiques et temporelles entre processus est remplacé par un graphe non-orienté, représentant des dépendances mutuelles (échanges de données).

21.2.2 Fonctions de coût

Le calcul de placements statiques efficaces nécessite une connaissance *a priori* du comportement dynamique de l'architecture parallèle vis-à-vis des programmes qui y sont exécutés. Cette connaissance est synthétisée dans une *fonction de coût* à minimiser, dont la nature détermine les caractéristiques des placements statiques optimaux souhaités. Les principaux critères cités dans la littérature [1, 41, 48, 59, et leurs références] et pouvant intervenir dans ces fonctions permettent d'évaluer soit la distribution des calculs, soit la réduction des communications. Ce sont, par exemple,

- la variance des charges des processeurs, c.-à.-d. de la somme des poids des processus qui y sont placés, qu'il est intéressant de minimiser puisque si l'on ne tient pas compte des communications, une charge équirépartie assure un temps d'exécution minimal ;
- la *congestion* des liens de l'architecture, c.-à.-d. la quantité maximale de communication transitant par un lien de H (voir section 21.3.1.b pour les définitions), qui conditionne le temps de transmission ;
- la variance de la congestion des liens de l'architecture, qui mesure la répartition de la charge de communication ;
- la somme des *dilatations* des arêtes de G plongées dans H, qui équivaut à la somme des congestions des arêtes de H (ce qui revient à parler de dilatation et congestion moyennes), qui mesure la minimisation globale des longueurs des routages.

Dans la majorité des cas, les fonctions de coût combinent plusieurs de ces critères par sommation, en les pondérant par des coefficients dépendant des caractéristiques de la machine cible. Ainsi, si une machine calcule rapidement mais communique lentement, on pénalisera plus la communication que le coût de calcul, afin de privilégier les placements minimisant la communication par rapport aux autres. L'inconvénient majeur de ces fonctions de coût, dites *unifiées*, est l'existence de ces coefficients de pondération. Outre le fait qu'ils doivent être mesurés pour chaque machine cible, l'amalgame de quantités de nature différente, même si on ne les manipule que sous forme de nombres sans dimension (variances, par exemple), n'est en pratique pas facile à réaliser.

Certains auteurs préfèrent donc définir des fonctions de coût *hiérarchisées,* dans lesquelles la validité de la minimisation d'une fonction continue est conditionnée par la vérification préalable d'autres conditions portant sur des paramètres autres que ceux de la fonction continue. Le plus souvent, il s'agit de minimiser une fonction de coût de communication, tout en conservant l'équilibrage de charge dans une tolérance prédéfinie [33, 17, 16, 44]. L'avantage de cette approche est qu'elle réduit le nombre de paramètres à considérer, et les rend plus naturels à manipuler. Ainsi, dans l'exemple ci-dessus, le seul paramètre manipulé par l'utilisateur du logiciel de placement est le taux maximum de déséquilibre de charge acceptable, qui définit un sous-espace de l'espace des solutions dans lequel il convient de minimiser la fonction de coût de communication.

Cette hiérarchisation pourrait se traduire dans une fonction de coût unifiée par le fait de rajouter à la fonction de communication un terme de valeur nulle si l'équilibrage est réalisé, et infinie dans le cas contraire. Cependant, certains algorithmes de placement tirent implicitement parti de la continuité de la fonction de coût pour en rechercher les minima locaux (les algorithmes de recuit simulé [10] et génétiques [42, 12] en particulier), et employer une fonction discontinue remettrait en cause leur efficacité. La hiérarchisation des fonctions de calcul et de communication n'apparaît donc principalement que dans les algorithmes heuristiques basés sur les graphes [17, 33].

À l'heure actuelle, du fait de l'arrivée sur le marché de générations de machines pour lesquelles le routage n'est plus statique mais géré dynamiquement par le système de communication (c'est en particulier le cas pour les machines basées sur des réseaux de communication multi-étages, de type Oméga, comme pour la SP-2 d'IBM), les nouveaux outils de placement tendent à ne plus calculer les routages (c.-à.-d. l'application R) ; les paramètres liés à la communication sont estimés à partir de routages de plus courts chemins dans H induits par le placement des sommets de G sur H (application p).

21.2.3 Algorithmes de placement statique

La recherche du placement optimal d'un programme à s processus sur une machine à t processeurs nécessite en théorie l'évaluation des t^s placements possibles. Bien que les dimensions du problème puissent être réduites du fait des contraintes s'appliquant aux placements effectivement réalisables, cette recherche reste NP-complète dans le cas général [20]. L'alternative est donc soit d'effectuer une recherche exhaustive en risquant l'explosion combinatoire, soit de calculer au moyen d'heuristiques un placement acceptable en un temps polynomial.

21.2.3.a Algorithmes exacts

Les algorithmes exacts permettent toujours de trouver la ou les solutions de coût minimal. Du fait de sa NP-complétude, la recherche d'une solution optimale dans le cas général nécessite l'exploration systématique de l'espace des solutions. Pour ce faire, on se base habituellement sur un arbre de recherche dont les nœuds correspondent à des sous-ensembles de solutions, le parcours des branches de la racine vers les feuilles représentant alors le fait de lier l'une après l'autre les variables du problème jusqu'à l'obtention de solutions complètes.

Les méthodes de type "*branch and bound*" permettent d'éviter de parcourir toutes les branches, en "élaguant" les sous-arbres menant à des solutions toutes moins bonnes qu'une solution donnée. Plusieurs variantes de l'algorithme existent, selon que l'on explore l'arbre en profondeur, par niveaux, ou par ordre croissant des valeurs des fils non encore traités (le parcours de l'arbre est alors analogue à celui effectué par un algorithme de type A* [54, 57]). Bien que la complexité de cet algorithme soit dans le pire des cas identique à celle d'une énumération explicite, il est en moyenne plus efficace.

Une autre approche est celle de la programmation dynamique. On représente alors le problème comme la minimisation avec contraintes d'une fonction de coût quadratique en variables $\{0, 1\}$ [26, 51].

Il est à noter que, dans certains cas, le problème de placement peut être réduit à un problème polynomial. Stone [58] a ainsi montré comment un placement optimal peut être obtenu en temps polynomial pour une machine bi-processeurs, au moyen d'un algorithme de flot maximal/coupe minimale. Ce résultat a été partiellement étendu à un nombre quelconque de processeurs par Lo [38], qui donne un algorithme polynomial pouvant conduire à une solution optimale prouvée, mais pouvant aussi échouer. Dans le même article, Lo fournit également un algorithme polynomial optimal dans le cas de processeurs identiques ne supportant qu'au plus deux processus chacun, basé sur l'utilisation d'un couplage parfait maximal.

21.2.3.b Méthodes approchées

De nombreuses heuristiques ont été proposées afin de trouver des solutions sous-optimales en un temps raisonnable [9, 10, 16, 39, 42]. Elles sont issues de différents domaines, tels la théorie des graphes (recherche d'homomorphismes faibles, partitions de coupe minimale, groupement de processus), les méthodes d'optimisation combinatoire liées à la simulation de phénomènes physiques (recuit simulé et algorithmes génétiques), l'étude spectrale des matrices, l'utilisation de l'information géométrique des graphes (méthodes inertielles et rectilinéaires), etc.

Les classifications des heuristiques [1, 41, 48] distinguent entre les algorithmes gloutons qui, initialisés avec une solution partielle, construisent une solution complète sans jamais remettre en cause un choix effectué, et les algorithmes itératifs, qui procèdent par raffinement d'une solution réalisable courante. Notons cependant que la limite n'est pas toujours nette, principalement avec les heuristiques de graphes : certains algorithmes peuvent être considérés comme itératifs au sens où ils remettent en cause des choix effectués, et gloutons parce que leur comportement est totalement déterministe.

Nous proposons un critère de classification supplémentaire, selon que la résolution du problème de placement est envisagée soit globalement, soit comme un ensemble de sous-problèmes plus ou moins dépendants les uns des autres, de façon ascendante (quotientement) ou descendante (décomposition), utilisant dans ces deux derniers cas une approche de type "diviser pour résoudre". Ceci nous permet de proposer la classification représentée en figure 21.1, page 413.

21.2.3.b.1 Méthodes globales Elles consistent à considérer le placement dans sa globalité, en manipulant simultanément l'ensemble des variables du problème. C'est le cas des méthodes itératives de type recuit simulé [10], des algorithmes génétiques [42, 12], et de certaines méthodes de graphes [9, 43]. L'inconvénient de ces approches est que la taille de l'espace des solutions augmente exponentiellement par rapport à la taille du problème, ce qui rend son exploration très coûteuse.

21.2.3.b.2 Méthodes de quotientement Elles ont pour but de réduire la taille du problème considéré, afin de mettre sa nouvelle formulation à la portée d'algorithmes qui ne peuvent efficacement traiter le problème initial, du fait de sa trop grande taille. On calcule ainsi une solution de placement sur la formulation réduite du problème, que l'on "remonte" ensuite pour l'appliquer au problème initial (d'où le nom d'approche ascendante). L'outil de contraction de graphes habituellement utilisé est le *quotientement*, qui permet de construire le graphe réduit à partir d'une partition des sommets du graphe initial en contractant en un unique sommet tous ceux appartenant à une même partie de la partition. Deux méthodes de quotientement ont été proposées successivement dans la littérature :

- le *groupage* ("*clustering*") qui, dans une première phase, groupe les sommets de G en au moins autant de paquets que de processeurs (et bien moins que $|V(G)|$), puis dans une deuxième phase place le graphe quotient obtenu sur H. Cette méthode est conceptuellement simple, mais peut induire des surcoûts de communication lorsque le quotientement est effectué sans tenir compte de la topologie de l'architecture. De plus, elle augmente la granularité du placement, et induit donc un déséquilibre de charge qui est d'autant plus important que le nombre de parties quotientées est petit ;
- le quotientement *multi-niveaux* ("*multi-level*"). Cette méthode, qui dérive des algorithmes multi-grilles utilisés en algorithmique numérique, construit une

famille de graphes de plus en plus petits au moyen de couplages (les parties des partitions servant aux quotientements sont à chaque fois constituées des extrémités des arêtes des couplages), calcule un placement valide sur le plus petit graphe construit, et projette le résultat de graphe en graphe jusqu'au graphe initial [3, 28, 31, 46].

Cette approche se distingue du groupage par sa souplesse ; en effet, après que le placement obtenu sur un graphe quotienté a été propagé au graphe immédiatement supérieur, il est possible d'appliquer un algorithme d'optimisation locale (le plus souvent de type Fiduccia–Mattheyses, voir section 21.2.3.b.3) pour affiner le placement résultat, ce qui permet d'équilibrer finement la charge du placement final.

Le quotientement multi-niveaux permet de gagner non seulement en vitesse d'exécution, les calculs de placement les plus coûteux n'étant effectués que sur le plus petit graphe, mais aussi en qualité : en contractant les graphes, il met à la portée des optimiseurs locaux la structure globale du graphe initiale, qui serait sinon d'un trop haut niveau pour être prise en compte dans le processus d'optimisation locale.

21.2.3.b.3 Méthodes de décomposition Elles consistent à séparer le graphe à placer en plusieurs parties (habituellement deux, plus rarement quatre ou huit), qui sont affectées à un nombre équivalent de parties de la machine cible. Ce processus est récursivement appliqué aux sous-graphes et sous-machines ainsi obtenus, jusqu'à ce que les sous-machines soient réduites à un unique processeur, sur lequel sont placés les processus du sous-graphe correspondant. Cette approche, employée initialement par Ercal *et al.* pour placer sur l'hypercube, [16], a été étendue par Pellegrini aux graphes quelconques [44, 47] sous le nom de "*Bipartition Récursive Conjointe*".

Remarquons que c'est une approche gloutonne, puisque les choix effectués ne pourront être remis en cause. C'est cependant une "bonne" approche gloutonne, parce que les choix initiaux ne sont pas très informants, et que leurs effets peuvent éventuellement être atténués par des choix ultérieurs. Par exemple, deux processus séparés sur deux sous-machines différentes au début de la récursion peuvent cependant être maintenus à petite distance l'un de l'autre si les séparations ultérieures les affectent à des sous-machines toujours proches les unes des autres.

L'obtention de bons placements nécessite que les bipartitionnements successifs concourent à la minimisation de la fonction de coût choisie. Il s'agit donc de bipartitionnements avec contraintes, sur lesquels se reporte la complexité du problème de placement. Par exemple, l'un des objectifs communs à toutes les méthodes de bipartitionnement est de favoriser la minimisation de la coupe entre les parties qu'elles génèrent, car cela revient à privilégier la localité des communications sur la machine parallèle. En effet, plus la communication entre deux processus est intensive, plus ceux-ci auront tendance à se trouver placés à proximité l'un de l'autre, car les séparer reviendrait à augmenter de beaucoup la coupe correspondante. Dans le cadre du placement, il convient également de minimiser la communication entre les processus du sous-graphe à bipartitionner et les autres processus déjà placés, afin d'éviter de faire localement des choix préjudiciables à la minimisation globale de la fonction de coût [29, 30, 44]. À cet objectif de minimisation de la communication s'ajoute toujours une contrainte d'équilibrage des cardinaux des sous-graphes, afin que le coût de calcul soit équitablement réparti entre toutes les sous-machines, et donc tous les processeurs.

Ces seules contraintes de minimisation de la communication et d'équilibrage de la charge de calcul suffisent à rendre le problème de bipartitionnement NP-complet. En effet, si l'on ne s'attache qu'à minimiser strictement la coupe entre les parties tout en assurant l'égalité à un près de leurs cardinaux, on se ramène au problème du calcul de la bissection arête d'un graphe, qui est NP-complet dans le cas général [21]. Dans le cadre de l'approche "diviser pour résoudre", de nombreux auteurs se sont donc consacrés à l'étude d'heuristiques de bipartitionnement efficaces [3, 8, 11, 17, 27, 34, 61].

L'une des méthodes les plus populaires jusqu'à présent est la bissection spectrale. Celle-ci, issue des travaux de Donath et Hoffman [15], se base sur les propriétés de la matrice de Laplace du graphe à bissecter.

L'autre classe de méthodes de bipartitionnement la plus utilisée est constituée des heuristiques de graphes de type Kernighan–Lin et Fiduccia–Mattheyses [17, 33]. Ces algorithmes partent d'une solution réalisable de coût quelconque, qu'ils améliorent progressivement en construisant des séquences d'échange ou de déplacement de sommets conduisant à une nouvelle solution réalisable de plus faible coût. Afin de ne pas être piégés dans des minima locaux de la fonction de coût, ils tolèrent dans ces séquences des mouvements de sommets conduisant à des solutions intermédiaires de coût plus élevé que celui de la solution de départ, si cela permet des gains ultérieurs plus importants.

D'autres heuristiques de graphes ont également été proposées, telles les méthodes de type GRASP (pour "*Greedy Randomized Adaptative Search Procedure*") [34]. Celles-ci consistent à itérer un grand nombre de fois une routine gloutonne de calcul d'une solution réalisable, en conservant à chaque fois la meilleure solution trouvée. Le calcul de la solution courante s'effectue en construisant une solution initiale au moyen d'un algorithme glouton, puis en lui appliquant un second algorithme glouton d'optimisation locale. Afin d'explorer au fil des itérations le plus de configuration initiales et d'états finaux possibles, les algorithmes gloutons de construction et d'optimisation utilisent explicitement des variables aléatoires pour influencer leur comportement. Ces méthodes se démarquent des méthodes de recherche purement probabilistes en ce que les algorithmes gloutons cherchent toujours à minimiser la fonction de coût, et diffèrent des heuristiques de type Kernighan-Lin en ce que les algorithmes d'optimisation sont beaucoup plus simples, les fonctionnalités de sortie des minima locaux étant ici remplacées par la multiplicité des essais effectués.

Les algorithmes de calcul de flots maximaux ont également été employés pour déterminer la bissection arête de graphes. L'algorithme de Bui, Chaudhuri, Leighton, et Sipser [11] construit, pour toute paire de sommets du graphe considéré, un graphe contracté sur lequel est appliqué l'algorithme de flot maximal. Les fréquences d'apparition des arêtes du graphe initial dans les coupes minimales ainsi obtenues peuvent permettre de construire une bissection optimale prouvée du graphe, mais l'algorithme peut également échouer. Cet algorithme donne d'excellents résultats pour des graphes réguliers de petits degrés et de petite bissection, mais est largement dépassé par les heuristiques de type Kernighan-Lin dès que le degré des graphes dépasse 4.

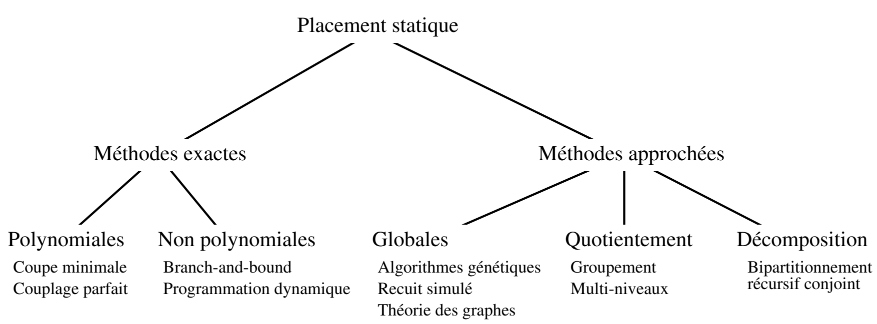


Figure 21.1 : Classification des méthodes de placement statique.

21.2.4 État de l'art

À l'heure actuelle, les meilleurs algorithmes de placement connus réalisent la synthèse des différentes approches décrites ci-dessus : ce sont des algorithmes multiniveaux, qui utilisent un algorithme de décomposition pour calculer le placement sur le plus petit graphe contracté, et un optimiseur local pour raffiner les placements projetés lors de la phase de remontée. Les heuristiques de bipartitionnement employées par l'algorithme de décomposition, ainsi que l'optimiseur local utilisé pour la remontée, sont basés sur des heuristiques de type Fiduccia–Mattheyses.

Parmi les outils de placement implémentant ces techniques, on peut citer MeTiS [32] (qui ne calcule que des partitionnements de graphes, c'est-à-dire des placements sur le graphe complet), et Scotch [45].

21.2.5 Parallélisation

Une des principales applications du placement statique est le découpage des graphes de maillage utilisés en calcul parallèle. L'augmentation de la taille des problèmes fait que les maillages qui seront utilisés dans les prochaines années seront trop gros pour être stockés dans la mémoire d'une machine séquentielle, ce qui rend nécessaire la réalisation d'outils de placement statiques eux-mêmes parallèles.

L'algorithme de Fiduccia–Mattheyses étant par nature séquentiel, aucune formulation parallèle efficace de cet algorithme n'a pu être proposée à ce jour, en dépit des efforts engagés (voir par exemple [23]). En revanche, une voie prometteuse de parallélisation a été récemment ouverte par Schloegel, Karypis et Kumar [52], qui ont proposé une formulation parallèle de leur algorithme multi-niveaux. Dans leur approche, le graphe à partitionner, qui est stocké de manière distribuée sur les processeurs de la machine parallèle servant aux calculs, est contracté en parallèle jusqu'à obtenir un graphe distribué de très petite taille, qui peut alors être centralisé sur un unique processeur. Ce processeur exécutera alors l'algorithme de décomposition séquentiel qui calculera le placement initial, avant de redistribuer ce résultat sur les autres processeurs, qui effectueront en parallèle la remontée du placement jusqu'au graphe initial.

21.2.6 Conclusion

La nécessité de pouvoir calculer rapidement des placement statiques efficaces a conduit au développement de nombreux algorithmes de placement, parmi lesquels les heuristiques multi-niveaux de graphes se montrent à ce jour les plus efficaces.

L'augmentation de la taille des graphes manipulés amène maintenant à se pencher sur la parallélisation de ces algorithmes. dont des formulations parallèles commencent à voir le jour.

21.3 Émulation et algorithmes parallèles

Une propriété importante de la topologie d'un réseau d'interconnexion de machines parallèles ou d'un réseau local de stations de travail est la possibilité d'y implanter facilement des algorithmes parallèles efficaces. Une façon d'évaluer cette propriété est de déterminer si ce réseau peut simuler efficacement l'exécution, sur d'autres réseaux, d'algorithmes parallèles visant la même application. On dira que le réseau cible *émule* ces autres réseaux.

En effet, certains réseaux, comme la grille ou l'arbre binaire complet, sont utilisés pour la mise au point d'algorithmes de tri ou de calcul matriciel, par exemple [18, 35]. Il s'agit donc, dans notre réseau cible, non pas de mettre au point de nouveaux algorithmes pour ces mêmes applications mais d'utiliser ceux existant dans la grille ou l'arbre. Considérons un algorithme de somme de 8 entiers dans un arbre binaire

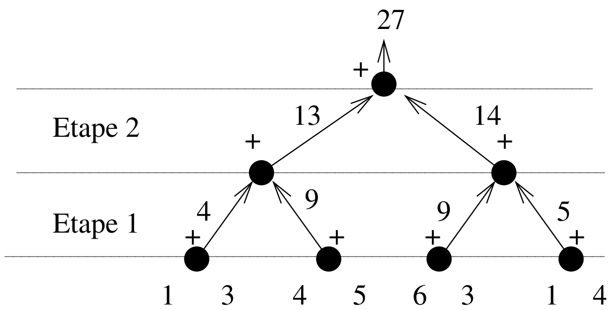


Figure 21.2 : Somme de 8 entiers sur un arbre binaire en deux étapes de communication.

complet de hauteur 2 (voir figure 21.2). Cet algorithme procède en 3 étapes de calcul et 2 étapes de communication (voir paragraphe 21.3.1.a ci-après pour les définitions). On désire simuler cet algorithme dans un réseau classique, l'hypercube [35],

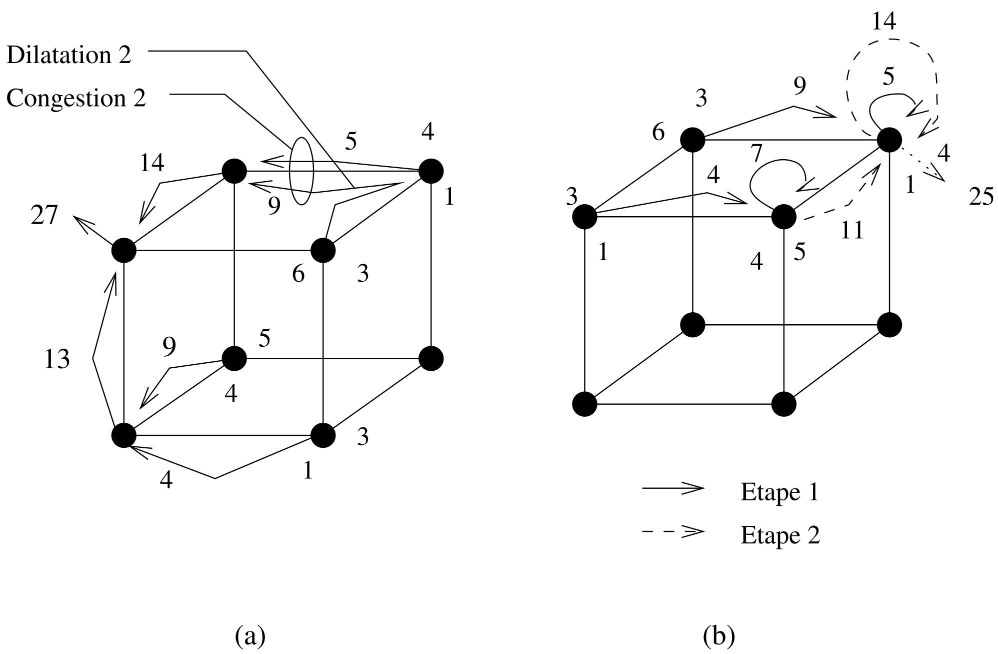


Figure 21.3 : Émulation par l'hypercube de l'algorithme de somme par un plongement (a) et un plongement dynamique (b).

à 8 sommets dans notre exemple (voir figure 21.3). L'émulation représentée par la figure 21.3(a) est obtenue par un *plongement* de l'arbre binaire dans cet hypercube :

on cherche à retrouver dans l'hypercube une structure proche de l'arbre. La qualité de cette émulation dépend de phénomènes de congestion des liens de l'hypercube et de dilatation des arêtes de l'arbre (voir section 21.3.1.b). L'émulation représentée par la figure 21.3(b) est un plongement par morceaux. On utilise ici le fait qu'à chaque étape, seuls certains nœuds et arêtes de l'arbre sont utilisés ; on parle alors de *plongement dynamique* de l'arbre binaire (voir section 21.3.2.a). Ce deuxième type de plongement est plus efficace que le premier en terme de nombre d'étapes de communication et de nombre de nœuds utilisés, mais il est spécifique à un certain type d'algorithmes (fonction des nœuds simultanément utilisés) alors que le plongement initial est plus général.

Les outils théoriques classiquement utilisés pour étudier l'émulation sont le *plongement de graphes* et le *routage de permutations*. De nouveaux modèles, tenant compte de propriétés des algorithmes à simuler, sont aussi proposés.

21.3.1 Approches classiques

21.3.1.a Un modèle algorithmique simple

Nous utilisons ici un modèle simple d'exécution d'un algorithme parallèle dans un réseau ; il s'agit d'un modèle synchrone classique, sans recouvrement calcul-communications (voir [4] et références). Un algorithme est constitué de la succession d'étapes de calcul et d'étapes de communication. Une étape de communication consiste, pour des nœuds voisins, à échanger des messages. Une étape de calcul voit l'exécution d'un traitement de données par un certain nombre de nœuds. Par la suite, une étape de communication sera considérée comme un ensemble d'arêtes (sur lesquelles transitent des données) et une étape de calcul comme un ensemble de nœuds (traitant des données).

Le modèle de communication est Δ-*port* (chaque nœud peut communiquer avec tous ses voisins durant une étape) et *bidirectionnel* (deux messages peuvent se croiser sur un lien) [19]. Le but de ce modèle d'algorithmes parallèles n'est pas de simuler exactement le fonctionnement d'une machine parallèle, mais d'étudier l'efficacité d'un réseau d'interconnexion pour l'implantation d'algorithmes.

21.3.1.b Plongements de graphes

L'outil le plus utilisé pour étudier l'émulation est un cas particulier de placement d'un graphe G dans un graphe H, appelé *plongement*, où l'application p est injective (voir section 21.1.1), ce qui implique $|V(G)| \leq |V(H)|$. Ainsi, chaque nœud de H simule au plus un nœud de G. Afin d'évaluer un tel plongement, trois paramètres, dont nous verrons l'utilisation, sont classiquement définis. Soit (p, R) un plongement de G dans H,

- l'*expansion* est définie par $e(p, R) = \frac{|V(G)|}{|V(H)|}$;

- la *dilatation* est définie par $d(p, R) = max\lim_{[x,y] \in E(G)}(|R(x, y)|)$, où $|R(x, y)|$ est la longueur de la chaîne $R(x, y)$;

- la *congestion d'une arête* $[u, v]$ de H est $c([u, v]) = |\{[x, y] \in E(G) \ / \ [u, v] \in R(x, y)\}|$. La congestion $c(p, R)$ du plongement est le maximum des congestions des arêtes de H.

L'expansion sert à mesurer le nombre de nœuds du réseau cible réellement utilisés lors de l'émulation. Le résultat suivant sur l'émulation met en jeu ces deux derniers paramètres.

Théorème 21.3.1 *Si un graphe G peut être plongé dans un graphe H avec une dilatation d et une congestion c, alors tout algorithme s'exécutant en t étapes de communication dans G sera simulé dans H*
- en au plus $d \times c$ étapes dans H de façon directe ("online") [14],
- en $O(d + c)$ étapes [37].

Le résultat de [37] est un résultat d'existence d'une telle émulation. Récemment, les mêmes auteurs ont montré que nombre de telles émulations peuvent être obtenues de façon précalculée ("*offline*") en temps polynomial [36].

Le but de nombreuses études est de déterminer le meilleur plongement, en terme de dilatation et/ou congestion, d'un réseau dans un autre. Nous sommes donc amenés à considérer les problèmes suivants :

Problèmes
Données : *Un graphe G, un graphe H, et un entier k.*
Problème 1 : *G peut-il être plongé dans H avec une dilatation $d < k$?*
Problème 2 : *G peut-il être plongé dans H avec une congestion $c < k$?*

Le Problème 1 est connu comme étant NP-complet quand H appartient à des classes de graphes particulières (chaînes, arbres, grilles, ...) (voir [14] et références). De même, le Problème 2 est déjà NP-complet quand H n'est qu'une chaîne [22]. Déterminer les meilleures dilatation et congestion d'un plongement de G dans H est donc un problème difficile, surtout étudié au cas par cas (voir [14, 40] et références). La plupart des travaux dans ce domaine cherchent à optimiser la dilatation, la congestion semblant plus difficile à optimiser. Notons que dans la plupart des cas, les plongements optimisant la dilatation et ceux optimisant la congestion sont différents. Le paramètre devant alors être étudié est la somme *dilatation* + *congestion*.

21.3.1.c Routage de permutations

Il s'agit d'un autre cadre d'étude de l'émulation. Considérons un graphe H, et π une permutation de $V(H)$ (voir [24] pour les définitions). Le *routage de π dans H*

consiste en une succession d'étapes de communication réalisant l'envoi par chaque sommet u d'un message à destination de son image $\pi(u)$.

Le lien avec l'émulation est établi ainsi : soit G un graphe où s'exécute un algorithme parallèle sous un modèle **1-port** (c.-à.-d. que chaque sommet communique avec **au plus un** voisin à chaque étape), et p une application injective de $V(G)$ dans $V(H)$. Chaque étape de communication dans G est un ensemble $\mathcal{E}$ d'arêtes indépendantes (c.-à.-d. sans sommets communs) dans G. Considérons l'application μ suivante :

$$\begin{array}{rccl} \mu : & V(H) & \rightarrow & V(H) \\ & v & & \mu(v) = \begin{cases} w & \text{si } [p^{-1}(v), p^{-1}(w)] \in \mathcal{E} \\ v & \text{sinon} \end{cases} \end{array}$$

Par définition de $\mathcal{E}$, l'application μ, bijective, est donc une permutation de $V(H)$ (et plus exactement un ensemble de transpositions [24]). Émuler cette étape consiste dés lors à router μ dans H.

Si on considère un modèle **Δ-port** dans G, une étape de communication $\mathcal{E}'$ dans G peut aisément être divisée en plusieurs ensembles $\mathcal{E}_1, \ldots,$ $\mathcal{E}_k$ d'arêtes indépendantes (c.-à.-d. sans sommets communs) en utilisant une *coloration-arêtes* de G [7]. Pour une telle coloration, deux arêtes indépendantes ont des couleurs différentes ; dans un graphe de degré maximal Δ, on sait déterminer une telle coloration avec $\Delta+1$ couleurs en temps polynomial [13] (sachant qu'il faut au moins $\Delta(G)$ couleurs). Chaque sous-ensemble $\mathcal{E}_i$ contient les arêtes de même couleur dans $\mathcal{E}'$, et détermine ainsi une permutation de $V(H)$. Dés lors,

Lemme 21.3.1 *Émuler dans un graphe H une étape de communication dans un graphe G revient à router simultanément au plus $\Delta + 1$ permutations dans H.*

C'est en utilisant cette idée que, par exemple, le résultat suivant concernant un réseau très étudié, le Butterfly [14], a été obtenu.

Proposition 21.3.1 *[35] Le réseau Butterfly d'ordre N peut émuler toute étape de communication dans un graphe G d'ordre N et de degré maximum δ en au plus $3\Delta log_2 N$ étapes.*

La preuve de cette proposition utilise le fait que chaque permutation du graphe Butterfly d'ordre N peut être routée en au plus $3log_2 N$ étapes [35].

Le routage de permutations dans un réseau a fait l'objet de nombreux travaux de façon générale [25, 50] et dans le cadre particulier de l'émulation [49, 53]. Dans le cas du routage simultané de plusieurs permutations, des résultats spécifiques sont établis dans [56], et un état de l'art pour le cas de la grille bidimensionnelle est proposé dans [55].

21.3.2 Nouveaux modèles et paramètres

Le modèle classique de plongement traite en fait la pire des situations possibles pour l'émulation d'un réseau G par un réseau H, c'est à dire où

1. Chaque étape de communication dans G utilise toutes les arêtes ;
2. Chaque arête de H de congestion c voit effectivement un conflit de c messages lors d'une étape de communication de l'émulation.

Notons par ailleurs que d'autres types de réseaux d'interconnexion comme les *réseaux à bus* doivent être étudiés sous l'angle de l'émulation [6].

Les nouveaux modèles et paramètres de plongements présentés ici ont pour objectif la prise en compte de propriétés des algorithmes à simuler et des modèles de communication pour affiner la qualité d'une émulation.

21.3.2.a Plongements dynamiques

Dans cette partie, nous considérons uniquement des *graphes orientés symétriques* [7] : chaque arête est remplacée par deux arcs "aller-retour". Dans un placement (p, R) de G dans H, R associe ici à chaque arc (x, y) de G un chemin orienté de px à $p(y)$ dans H.

Nous considérons toujours ici un algorithme $\mathcal{A}$, s'exécutant dans un réseau G, comme une suite $t_1, c_1, t_2, c_2, \ldots, t_k, c_k$ d'étapes de calcul et de communication, avec $t_i \subset V(G)$, $c_i \subset E(G)$ et $1 \leq i \leq k$.

Définition 21.3.1 *Soit $\mathcal{E}$ un sous-ensemble d'arcs d'un graphe G. On notera $G[\mathcal{E}]$ le sous-graphe de G induit par $\mathcal{E}$.*

Dans la suite de ce paragraphe, nous considérons un plongement (p, R) d'un graphe G dans un graphe H, et $\mathcal{A} = t_1, c_1, t_2, c_2, \ldots, t_k, c_k$ un algorithme dans G. Nous définissons trois nouveaux paramètres :

- la *congestion dynamique* est définie par

  $$c_d(p, R) = max \lim_{(u,v)\in E(H)} \left(max \lim_{1\leq i\leq d(p,R)} |\{(x, y) \in E(G) \;/\; (u, v) = R_i(x, y)\}| \right)$$

  où $R_i(x, y)$ est le i^{eme} arc, s'il existe, du chemin $R(x, y)$ de $p(x)$ à $p(y)$ dans G. Ce paramètre, inférieur ou égal à la congestion du plongement, compte le nombre maximal de chemins utilisant un arc de H *en même position* [2, 60]. Si la congestion dynamique est 1 par exemple, il n'y aura pas de conflits de messages lors de l'émulation, quelle que soit la valeur de la congestion ;

- le *facteur de ralentissement* $f(p, R)$ est le plus petit nombre d'étapes d'un algorithme dans H réalisant l'envoi par chaque sommet u de G d'un message à chaque sommet de $p(\Gamma_G(p^{-1}(u)))$, où $\Gamma_G(v)$ est l'ensemble des voisins d'un sommet v de G. Ce paramètre, borné supérieurement par $O(c+d)$ (voir Théorème 21.3.1) et inférieurement par la congestion dynamique $c_d(p, R)$ [60], correspond à la performance du meilleur algorithme d'émulation pour un plongement donné ;

- la *congestion algorithmique* $c_a(\mathcal{A}, p, R)$ d'un plongement relatif à $\mathcal{A}$ est la plus grande congestion d'un plongement $(p, R)_{/c_i}$, avec $1 \leq i \leq k$, c'est à dire le plongement de $G[c_i]$ dans H induit par (p, R). De même, la dilatation algorithmique est la plus grande dilatation d'un plongement $(p, R)/_{c_i}$. Ces paramètres permettent d'affiner la qualité de l'émulation d'un algorithme particulier (ou d'une classe d'algorithmes) par H.

L'étude de la congestion dynamique et, de fait, du facteur de ralentissement fait suite à l'obtention d'un plongement, de façon plus fine que la simple utilisation du Théorème 21.3.1. Peu d'exemples ont été étudiés. Nous en évoquons un.

Proposition 21.3.2 *[60] Pour tout plongement (p, R) dans la chaîne $P(N)$ d'un graphe G d'ordre N et Δ-régulier, on a*
(1) $c_d(p, R) = \Delta$,
(2) $f(p, R) = c_d(p, R)$.

En utilisant des propriétés propres aux sommes cartésiennes de graphes (voir [7]), on obtient le résultat suivant dans la grille à d_1 lignes et d_2 colonnes.

Corollaire 21.3.1 *[60] Il existe un plongment de l'hypercube $H(n)$ dans une grille à deux dimension $G(d_1, d_2)$ dont le facteur de ralentissement est $max(d_1, d_2)$, ce qui est optimal.*

Les dilatation et congestion algorithmiques ont été utilisées pour obtenir des algorithmes d'approximation réalisant la *diffusion* d'un message dans un réseau (i.e. communication du type un-vers-plusieurs) [19]. L'arbre binomial à N sommets [14] est, sous un modèle 1-port, l'arbre de diffusion idéal (i.e. que la diffusion nécessite $log_2 N$ étapes). Tout graphe G ne possède malheureusement pas cet arbre comme sous-graphe. L'idée est alors de plonger cet arbre dans une chaîne de même ordre ; la congestion est $O(log_2 N)$, mais la congestion algorithmique propre à cet algorithme de diffusion est elle égale à 1. Ainsi, en utilisant ensuite un plongement particulier de la chaine dans un graphe quelconque G, le résultat suivant est obtenu.

Proposition 21.3.3 *[5] Il existe un algorithme d'approximation réalisant la diffusion d'un message dans n'importe quel graphe G d'ordre N, de diamètre D et de degré Δ, en $O(D+\Delta)log_2 N)$ étapes.*

Dans [35], la notion de dilatation et congestion algorithmiques est considérée de façon un peu différente. L'auteur étudie l'émulation par le graphe Butterfly d'algorithmes par *dimensions contiguës* dans l'hypercube $H(N)$; dans un tel algorithme $t_1, c_1, \ldots t_k, c_k$, chaque t_i est égal à $V(H(n))$ et chaque c_i est un sous-ensemble particulier d'arêtes. L'originalité est la définition d'un plongement particulier (p_i, R_i) pour chaque couple (t_i, c_i). Les congestion et dilatation algorithmiques sont ainsi égales à 1, mais s'ajoute le coût de l'envoi à la fin de chaque étape i des données contenues par $p_i(v)$ à $p_{i+1}(v)$ pour chaque sommet v de $H(n)$. Ainsi,

Proposition 21.3.4 *[35] Le réseau Butterfly peut simuler en $O(T)$ étapes tout algorithme procédant par dimensions contiguës en T étapes dans l'hypercube de même ordre.*

Les modèles de plongements et les paramètres présentés dans ce paragraphe permettent donc une évaluation plus fine d'une émulation.

21.3.3 Conclusion

L'émulation définit en quelque sorte le degré d'universalité d'un réseau d'interconnexion : un réseau universel est un réseau pouvant simuler efficacement de nombreux autres réseaux. On peut utiliser cette propriété comme un outil de mise au point d'algorithmes parallèles efficaces.

Notons enfin que les notions de plongement et d'émulation ont été étendues aux réseaux d'interconnexion par bus de communication [6], réseaux modélisés par des hypergraphes (voir [7] pour définition).

Bibliographie

[1] André (F.) et Pazat (J.-L.). – Le placement de tâches sur des architectures parallèles. *Technique et Science Informatiques*, Avr. 1988, pp. 385–401.

[2] Andreae (T.), Nolle (M.), Rempel (C.) et Schreiber (G.). – *On embedding* 2-*dimensional toroidal grids into de Bruijn graphs with clockwise congestion one.* – Rapport technique, Universitat Hamburg-Harburg, 1996.

[3] Barnard (S. T.) et Simon (H. D.). – A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, vol. 6, n° 2, 1994, pp. 101–117.

[4] Barth (D.). – Parallel matrix product algorithm in the de Bruijn network using emulation of meshes of trees. *Parallel Computing*, vol. 22, 1997, pp. 1563–1578.

[5] Barth (D.) et Fraigniaud (P.). – Emulation tools for approximated solutions of structured communication problems. *In : SPAA'97*.

[6] Barth (D.), Heydemann (M.-C.), Germa (A.) et Sotteau (D.). – Emulating networks by bus networks; application to trees and hypermeshes. *In : SIROCO'96*.

[7] Berge (C.). – *Graphes et Hypergraphes*. – Gauthier-Villard, 1971.

[8] Berger (M. J.) et Bokhari (S. H.). – A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, vol. 36, n° 5, Mai 1987, pp. 570–580.

[9] Bokhari (S. H.). – On the mapping problem. *IEEE Transactions on Computing*, vol. C-30, n° 3, 1981, pp. 207–214.

[10] Bollinger (S. W.) et Midkiff (S. F.). – Processor and link assignment in multicomputers using simulated annealing. *In : Proceedings of the 11th Int. Conf. on Parallel Processing*. pp. 1–7. – The Penn. State Univ. Press.

[11] Bui (T. N.), Chaudhuri (S.), Leighton (F. T.) et Sipser (M.). – Graph bisection algorithms with good average case behavior. *Combinatorica*, vol. 7, n° 2, 1987, pp. 171–191.

[12] Chockalingam (T.) et Arunkumar (S.). – A randomized heuristics for the mapping problem: The genetic approach. *Parallel Computing*, vol. 18, 1992, pp. 1157–1165.

[13] Cormen (T.), Leiserson (C. E.) et Rivest (R.). – *Introduction to algorithms*. – MIT Press, 1990.

[14] de Rumeur (J.). – *Communications dans les réseaux d'interconnexion.* – Masson, 1994.

[15] Donath (W.) et Hoffman (A.). – Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, vol. 15, 1972, pp. 938–944.

[16] Ercal (F.), Ramanujam (J.) et Sadayappan (P.). – Task allocation onto a hypercube by recursive mincut bipartitionning. *Journal of Parallel and Distributed Computing*, vol. 10, 1990, pp. 35–44.

[17] Fiduccia (C. M.) et Mattheyses (R. M.). – A linear-time heuristic for improving network partitions. *In : Proceedings of the 19th Design Automation Conference.* pp. 175–181. – IEEE.

[18] Fox (G.), Johnson (M.), Lyzenga (G.), Otto (S.), Salmon (J.) et Walker (D.). – *Solving problems on concurrent processors.* – Prentice Hall, 1988.

[19] Fraigniaud (P.) et Lazard (E.). – Methods and problems of communication in usual networks. *Discr. Appl. Math.*, vol. 53, 1994, pp. 79–133.

[20] Garey (M. R.) et Johnson (D. S.). – *Computers and Intractablility: A Guide to the Theory of NP-completeness.* – San Francisco, W. H. Freeman, 1979.

[21] Garey (M. R.), Johnson (D. S.) et Stockmeyer (L.). – Some simplified NP-complete graph problems. *Theor. Comp. Sci.*, vol. 1, 1976, pp. 237–267.

[22] Gavril (F.). – Some NP-complete problems on graphs. *In : Proc.* 11^{th} *Conf. on Inform. Science and Systems.* John Hopkins Univ., pp. 91–95.

[23] Gilbert (J. R.) et Zmijewski (E.). – *A parallel graph partitioning algorithm for a message-passing multiprocessor.* – Technical Report n° TR 87-803, Cornell University, Jan. 1987.

[24] Graham (R.), Knuth (D.) et Patashnik (O.). – *Concrete Mathematics: A Foundation for Computer Science.* – Addison-Wesley, 1994.

[25] Grammatikakis (M. D.), Hsu (D. F.) et Kraetzl (M.). – Multicomputer routing. – Internal Report, 1994. À paraître.

[26] Hansen (P.) et Giauque (W. C.). – Task allocation in distributed processing systems. *Operations Research Letters*, vol. 5, n° 3, Août. 1986, pp. 137–143.

[27] Hendrickson (B.) et Leland (R.). – *Multidimensional spectral load balancing.* – Rapport technique n° SAND93–0074, Sandia Nat. Lab., Jan. 1993.

[28] Hendrickson (B.) et Leland (R.). – *The* CHACO *user's guide – Version 2.0.* – Rapport technique n° SAND94–2692, Sandia National Laboratories, 1994.

[29] Hendrickson (B.), Leland (R.) et van Driessche (R.). – *Enhancing data locality by using terminal propagation.* – Research report, Sandia National Laboratories, 1996.

[30] Hendrickson (B.), Leland (R.) et Van Driessche (R.). – Skewed graph partitioning. *In : Proceedings of the 8^{th} SIAM Conference on Parallel Processing for Scientific Computing.* IEEE.

[31] Karypis (G.) et Kumar (V.). – *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* – TR n° 95-035, University of Minnesota, Juin 1995.

[32] Karypis (G.) et Kumar (V.). – METIS – *Unstructured Graph Partitioning and Sparse Matrix Ordering System – Version 2.0.* – University of Minnesota, Juin 1995.

[33] Kernighan (B. W.) et Lin (S.). – An efficient heuristic procedure for partitionning graphs. *BELL System Technical Journal*, Fév. 1970, pp. 291–307.

[34] Laguna (M.), Feo (T. A.) et Elrod (H. C.). – A greedy randomized adaptative search procedure for the two-partition problem. *Operations Research*, Juil. 1994, pp. 677–687.

[35] Leighton (F.). – *Introduction to Parallel Algorithms and Architectures: arrays, trees, hypercubes.* – Morgan Kaufman Publisher, 1992.

[36] Leighton (F.) et Maggs (B.). – Fast algorithms for finding $o(c+d)$ packet routing schedules. *Combinatorica*, 1996.

[37] Leighton (F.), Maggs (B.) et Rao (S.). – Packet routing and job-shop scheduling in $O(congestion + dilation)$ steps. *Combinatorica*, vol. 14, n° 2, 1994, pp. 167–186.

[38] Lo (V. M.). – Heuristic algorithms for task assignment in distributed systems. *In : Int. Conf. on Distr. Comp. Sys.* IEEE, pp. 30–39.

[39] Ma (P. R.), Lee (E. Y. S.) et Tsuchiya (M.). – A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, vol. C-31, n° 1, Jan. 1982, pp. 41–47.

[40] Monien (B.) et Sudborough (H.). – Embedding one interconnection network in another. *Computing suppl.*, vol. 7, 1990, pp. 257–282.

[41] Muntean (T.) et Talbi (E.-G.). – Méthodes de placement statique des processus sur architectures parallèles. *T.S.I., Technique et Science Informatiques*, 1991.

[42] Muntean (T.) et Talbi (E.-G.). – A parallel genetic algorithm for process-processors mapping. *High performance computing*, vol. 2, 1991, pp. 71–82.

[43] Pazat (J.-L.). – *Outils pour la programmation d'un multiprocesseur à mémoire distribuée*. – Thèse de Doctorat, LaBRI, Université Bordeaux I, 1989.

[44] Pellegrini (F.). – Static mapping by dual recursive bipartitioning of process and architecture graphs. *In : Proc. SHPCC'94, Knoxville*. IEEE, pp. 486–493.

[45] Pellegrini (F.). – SCOTCH Static Mapping Package v3.1. – Sept. 1996. `http:// www.labri. u-bordeaux. fr/ ~pelegrin/ scotch/`.

[46] Pellegrini (F.). – Graph partitioning based methods and tools for scientific computing. *Parallel Computing, numéro spécial ETPSC'3*, vol. 23, 1997, pp. 153–164.

[47] Pellegrini (F.) et Roman (J.). – *Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping*. – Rapport de Recherche n° 1138-96, LaBRI, Université Bordeaux I, Sept. 1996. `http:// www.labri. u-bordeaux. fr/ ~pelegrin/ papers/ scotch_expanalysis. ps.gz`.

[48] Plateau (B.), Raynal (P.), Trystram (D.) et Vincent (F.). – *Placement de tâches: un tour d'horizon des techniques efficaces*. – Rapport technique, 46 avenue F. Viallet, 38031 Grenoble CEDEX, IMAG, Juin 1991.

[49] Preparata (F.) et Vuillemin (J.). – The cube-connected cycles: a versatile network for parallel computation. *Comm. ACM*, vol. 24, n° 5, 1981, pp. 300–309.

[50] Raghavendra (C.) et Boppana (R.). – On methods for fast efficient parallel memory access. *In : International Conf. on Parallel Processing*, pp. 76–83.

[51] Roucairol (C.) et Hansen (P.). – Cut cost minimization in graph partitioning. *Numerical and Applied Mathematics*, 1989, pp. 585–587.

[52] Schloegel (K.), Karypis (G.) et Kumar (V.). – *Parallel Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes*. – TR n° 97-014, University of Minnesota, 1997.

[53] Schwabe (E.). – On the computational equivalence of hypercube-derived networks. *In : ACM Symp. on Parallel Algorithms and Architectures*, pp. 388–397.

[54] Shen (C.-C.) et Tsai (W.-H.). – A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, vol. C-34, n° 3, Mars 1985, pp. 197–203.

[55] Sibeyn (J.). – *Algorithms for routing on meshes*. – Thèse de PhD, Université d'Utrecht, 1982.

[56] Sibeyn (J.). – Edge coloring bipartite graphs. *In: Computer Science in the Netherlands '92*.

[57] Sinclair (J. B.). – Efficient computation of optimal assignments for distributed tasks. *Journal of Parallel and Distributed Computing*, vol. 4, 1987, pp. 342–362.

[58] Stone (H. S.). – Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, vol. SE 3, n° 2, Jan. 1977, pp. 85–93.

[59] Talbi (E.-G.) et Muntean (T.). – *Évaluation et étude comparative d'algorithmes d'optimisation combinatoire: application au problème de placement de processus*. – Rapport de Recherche n° RR 886-I, 46 avenue F. Viallet, 38031 Grenoble CEDEX, LGI-IMAG, Avr. 1992.

[60] Valencia (M.). – Emulation dynamique de réseaux d'interconnexion. *In: RENPAR'9*.

[61] van Driessche (R.) et Roose (D.). – *A graph contraction algorithm for the calculation of eigenvectors of the laplacian matrix of a graph with a multilevel method*. – Rapport technique n° TW 209, Katholieke Univ. Leuven, Mai 1994.

Chapitre 22

Evolution des mécanismes de communication et du contexte d'utilisation des architectures parallèles : impact sur la performance et la portabilité des applications.

Franck Cappello (LRI)

22.1 Introduction

La conception d'applications parallèles, l'étude de modèles de performance des applications parallèles et la conception de mécanismes d'optimisation supposent une connaissance approfondie des mécanismes de communication et de l'environnement d'exploitation des architectures parallèles.

Nous commencons par présenter l'évolution des mécanismes de communication des architectures parallèles et de leur environnement d'exploitation, les modèles de performances qui en resultent et l'influence des performances des communications sur les performances globales.

Ensuite nous examinons les évolutions récentes en architectures parallèles et celles qui sont proposées pour étudier leur impact potentiel sur la conception d'applications parallèles, de modèles de performance et d'optimisations.

22.2 Evolution des mécanismes de communication dans les architectures parallèles

Les mécanismes de communication actuels (très performants) sont le fruit d'une évolution du contexte d'exploitation des architectures parallèles, d'une étude approfondie des sources de perte de performance dans les mécanismes de communication standards et de la définition de principes d'optimisation.

22.2.1 Evolution de la problématique des communications dans les architectures parallèles

Le cadre des travaux sur les communications dans les architectures parallèles a évolué très rapidement depuis quelques années, influencée par la recherche de performances mais aussi par l'objectif de la minimisation du rapport coût/performance. La recherche de la performance était motivée par la compétition avec les architectures vectorielles et l'apparition d'applications nécessitant des performances et des tailles mémoire jugées inaccessibles par les calculateurs vectoriels. Le rapport coût/performance était aussi utilisé au départ comme un argument contre les architectures vectorielles. Il s'est ensuite transformé en un critère discriminant entre les architectures parallèles. Il motive aujourd'hui les réseaux de stations contre les architectures parallèles classiques.
Les mécanismes de communication jouent un rôle central pour les deux critères. D'abord, les performances de communication ont une influence significative sur les performances globales. Ensuite, le réseau de communication et son intégration avec le processeur peuvent avoir un coût non négligeable face au reste de l'architecture. Les architectes de calculateurs parallèles ont d'abord proposé l'utilisation de ressources spécifiques pour répondre au problème de la performance. Puis la contrainte de la minimisation du rapport coût/performance a impliqué l'utilisation de ressources non spécifiques. Cette évolution s'est faite graduellement en trois étapes :
- le début des années 90 avec la recherche de la performance maximale à travers l'utilisation de processeur et de mécanismes de communication spécifiques,
- le milieu des années 90 avec la recherche d'un meilleur rapport coût/performance qui s'est concrétisé par l'utilisation exclusive de processeurs non spécifiques disponibles sur le marché des stations de travail et de mécanismes de communication spécifiques et
- la fin des années 90 qui voit apparaître l'utilisation, en plus des processeurs non spécifiques, de mécanismes de communications non spécifiques sous l'influence du même critère : minimisation du rapport coût/performance.
Ce critère a aussi eu une influence sur le contexte d'utilisation des calculateurs parallèles. Dans la première étape, l'objectif de performance implique la définition

d'une architecture mono-utilisateur ou à traitement par lots. Le plus souvent une seule application fonctionne à la fois sur un calculateur parallèle isolé et stable. Par exemple, a cette époque, le T3D est branché comme "co-processeur" d'un CRAY YMP qui gère les aspects système.
Au milieu des années 90, il devient clair que la course à la performance n'est plus le seul moteur pour la définition de nouvelles architectures parallèles. Pour continuer à progresser, ces architectures doivent faciliter leur utilisation et élargir leur domaine d'application. La machine ne reoit pas seulement des charges de travail parallèles de nature différentes (traitement numérique, base de données, etc.) mais aussi des activités séquentielles comme la compilation, l'exécution d'applications séquentielles, etc. Le SP2 [1] est un exemple d'architectures de ce type. La difficulté principale consiste à accorder l'utilisation de ressources spécifiques avec un contexte d'utilisation multi-utilisateurs. Le système gère le partage de l'accès aux ressources matérielles du réseau et protège les utilisateurs contre des accès erronés.
A la fin des années 90 des réseaux de communication aux performances voisines de celles des réseaux internes aux architectures parallèles deviennent disponibles. Myrinet [7], ATM [15] ou SCI [21] permettent de construire une architecture parallèle à très bas coût à partir de stations de travail. Une autre approche consiste à utiliser de tels réseaux pour interconnecter des stations ou des serveurs déjà existants dans un site informatique. Les architectures parallèles entrent dans les sites informatiques et deviennent des ressources partagées.

22.2.2 Fonctionnement des mécanismes standards dans un réseau de stations classiques (PVM-UNIX-TCP-IP-Ethernet)

Nous considérons un réseau de stations homogènes fonctionnant avec PVM [20], le système UNIX, la pile de protocoles TCP/IP et un réseau Ethernet. Les fonctions de communication de PVM utilisent les sockets pour communiquer des messages entre processus utilisateur. PVM permet à des tâches utilisateurs de communiquer en organisant une structure de processeurs virtuels complètement connectés par dessus un réseau d'ordinateurs qui peuvent être hétérogènes. PVM utilise TCP et UDP [20]. Les communications entre tâches distantes sont réalisées, soit par l'intermédiaire de démons, soit directement entre tâches.

Le programme utilisateur appel les fonctions d'émission et de réception pvm_send et pvm_recv qui sont les points d'interaction entre les processus utilisateur d'une part, et entre les processus utilisateur et les mécanismes de contrôle de la machine virtuelle d'autre part. Les auteurs de [4] rapportent que PVM représente une part minoritaire du temps de communication lorsqu'il est utilisé par dessus la plateforme que nous considérons. Par contre, par dessus des sockets optimisées, sans intervention du système, et par dessus un matériel performant, la part de PVM dans le temps

de communication peut dépasser 80%. Dans un système UNIX classique, lorsqu'un message doit être envoyé depuis un processus utilisateur, il doit d'abord franchir la barrière du système d'exploitation. Le changement d'espace (utilisateur-système) se traduit généralement par une copie des données entre l'espace utilisateur et l'espace système. TCP offre un service de transmission fiable des données. Un processus utilisant TCP n'a donc pas à déterminer la bonne réception d'un message et n'a pas non plus à gérer la retransmission en cas de mauvaise réception. En contre partie, TCP effectue des opérations coûteuses comme copier les données à transmettre dans un tampon pour les retransmettre en cas de besoin et calculer un checksum sur le message avant et après la transmission. Une description très précise d'une implémentation de TCP/IP peut être trouvée dans [11].
Il ressort de [24] qu'au delà de 2ko, le couple Checksum et mouvement de données (copies) représente plus de 50% du temps passé. Ce coût progresse avec la taille du message alors que le coût des autres composantes reste constant. En dessous de 2ko ce sont les spécificités du protocole (contrôle de flot, etc.) et la gestion des structures des tampons Mbuf (allocation, etc.) qui dominent dans le temps passé. Avant d'être émis dans le matériel du réseau, un message provenant de TCP/IP est à nouveau transformé en paquets manipulables par ce matériel. Ethernet encapsule les datagrammes IP dans des paquets comprenant son propre en-tête et un mot de queue correspondant au checksum de l'en-tête. La taille des données d'un paquet étant comprise entre 46 et 1500 octets, il faut aussi remplir le paquet dans le cas où moins de 46 octets doivent être transmis. Ethernet est souvent considéré comme la source principale de perte de performance. En comparant les performances d'une pile PVM/UNIX/TCP/IP par dessus Ethernet ou un réseau rapide (HPS d'IBM), on observe que pour un message de 1 octet, les temps d'aller-retour dans les deux configurations sont très voisins. Au delà, le débit du HPS (320Mb/s contre 10Mb/s) permet d'obtenir des temps de communication plus réduits. Toutefois le débit atteint pour 16ko est à peine supérieur à 40 Mb/s. Plus généralement, on constate que l'introduction de réseaux rapides ne suffit pas à réduire la latence de communication et le débit atteint reste très inférieur aux possibilités du matériel.

22.2.3 Les principaux mécanismes pour améliorer les performances des réseaux de communication

Un premier niveau d'amélioration consiste à réduire autant que possible le chemin critique logiciel que constitue la traversée de la pile de protocoles qui sépare l'appel à une fonction de communication par le processus utilisateur de la prise en compte du message par le matériel du réseau. Les rôles habituels remplis par la pile de protocole sont la fiabilité de transmission, la protection des utilisateurs, le partage des ressources matériel (virtualisation) et la gestion de l'hétérogénéité.
Ces fonctionnalités reposent sur des mécanismes comme la copie du message dans un

tampon, le formatage du message, la transformation des données, le calcul d'un code de détection d'erreur, le contrôle de flot, un mécanisme multiplexage/démultiplexage de paquet, des appels système, etc. Cinq techniques permettent d'apporter un premier degré d'amélioration : la réduction du nombre de copies des données, les communications en espace utilisateur, la séparation temporelle des transferts de données et de leur protection, la réduction de la complexité des protocoles et l'utilisation d'un matériel rapide et sûre.

La copie des informations à communiquer peut être nécessaire pour des raisons de protection des processus, de retransmission de données, de déséquilibre de performance entre les mécanismes mis en jeux dans une communication ou de sémantique de communication (envoie asynchrone). Les copies dûes au système peuvent être évitée en évitant de faire intervenir le système dans le chemin critique comme c'est le cas dans [6], [18], [4], [33]. Pour éviter la copie nécessaire pour le retransmission, la retransmission peut être assurée par le matériel ou reportée sur l'utilisateur comme dans [42].

Les mécanismes de communications sont des ressources habituellement accessibles à travers le système d'exploitation car c'est lui qui gère les accès concurrents aux ressources partageables (multiplexage) et la protection des processus utilisateurs. Une caractéristique commune aux approches qui abordent ce problème est de déplacer en partie ou totalement la gestion des communications depuis le système vers l'application (processus utilisateur), d'où le terme commun utilisé par ces travaux de "communications en espace utilisateur". Supprimer l'intermédiaire du système peut impliquer de remettre en cause des principes importants du système d'exploitation. Dans le cas des Fast Messages sur le réseau Myrinet, une seule application peut accéder au réseau de communication. Le partage de la ressource réseau n'est donc plus assurée.

Une solution dans le cas général est de dissocier la garantie des protections et le transfert des communications. Plusieurs projets dont SHRIMP [5] et MPC [35] proposent de séparer le traitement des protections et le transfert de données lors des communications. Le principe est d'ouvrir un canal entre un émetteur et un récepteur, de vérifier la protection lors de l'ouverture du canal, puis d'effectuer le transfert des données sans vérification de protection. En pratique, le canal est ouvert par un appel système qui verrouille des pages de la mémoire virtuelle en mémoire physique et attribue ces pages au processus utilisateur. Dissocier l'ouverture d'un canal et le transfert des données à un intérêt seulement si le canal est utilisé par plusieurs transferts qui chacun auraient nécessité une vérification des protections par le système d'exploitation. C'est le cas pour de nombreuses applications [5][8] [13].

[10] identifie les différentes fonctionnalités assurées par la pile de protocoles. Le formatage, la détection d'erreurs, le réordonnancement, le contrôle de flot et l'acquittement sont des parties coûteuses du protocole. Le formatage peut être réduit à une seule étape si le protocole ne manipule que des paquets directement utilisables par le

matériel ou si c'est le matériel qui formate les paquets comme par exemple dans SHRIMP où les paquets sont formatés à la volée. La contrainte est la spécialisation du protocole ou de la pile de protocoles. La détection d'erreur logicielle peut être évitée si elle est assurée par le matériel comme dans [18] [1] [35] [4]. La gestion du contrôle de flot peut être complexe et donc l'origine du coût important des communications. De nombreux protocoles de communication rapides récents font l'hypothèse que le contrôle de flot est assuré par l'application (l'utilisateur).
Dans les messages actifs [41], la communication se fait sans vérification préalable. Le message est recu dans un tampon préalablement alloué et sa réception déclenche une interruption qui provoque l'exécution du gestionnaire de messages spécifié par le programmeur et identifié dans l'en-tête du message. La gestion du tampon est laissée à l'utilisateur qui peut utiliser des mécanismes permettant de réaliser un contrôle de flot. Les fast messages [33] utilisent un mécanisme similaire à celui des messages actifs. Un message est envoyé de manière optimiste. Si le message est accepté, un acquittement est envoyé et la communication est terminée. Dans le cas contraire, le message est renvoyé à l'émetteur. La communication est réessayée ultérieurement.
La fiabilité des mécanismes matériels du réseau a une influence significative sur le délai de communication. Si le support matériel n'est pas fiable, des protocoles logiciels doivent être utilisés par le système ou l'utilisateur pour détecter les erreurs et retransmettre le cas échéant. Les réseaux de communication pour station comme Myrinet ou les réseaux des architectures parallèles possèdent des taux d'erreur très réduits (une erreur tous les 10^{15} bits).
Lorsque les limitations précédentes sont résolues, d'autres sources de réduction de performances apparaissent. Au fur et à mesure que la complexité logicielle diminue, les performances du matériel et l'intégration des ressources de communication (l'interface réseau) avec les autres ressources (processeur et mémoire) deviennent les sources de ralentissement.
On mesure généralement la part du logiciel dans une communication en comptant le nombre d'instructions exécutées pour effectuer cette communication. Ce nombre peut être très important (de l'ordre de plusieurs centaines pour TCP et IP [9] [30]) et varie suivant le type de processeur utilisé (RISC ou CISC) [30]. Vouloir le réduire n'est pas une problématique nouvelle. La J-Machine [14] du MIT a été concue dans ce sens. Ici, l'envoi d'un message nécessite 3 instructions. Le même objectif a été visé dans SHRIMP II ou il faut 15 instructions pour envoyer une page [5] avec cependant une sémantique différente de la sémantique traditionnelle puisque notamment les protections sont supposées avoir été vérifiées préalablement.
Libérer le processeur d'une partie de la gestion des communications implique de traiter cette partie avec un autre dispositif qui, si possible, ne perturbe par le fonctionnement du processeur. La solution utilisée dans les architectures parallèles consiste à associer un processeur de communication au processeur de calcul. Les interfaces du SP2 et de Myrinet utilisent des ressources similaires : un processeur dédié

aux communications, des DMA, un bus interne, une mémoire locale, etc.
Classiquement, l'interface de communication est connectée sur le bus d'entrées sorties de la station. Cette approche à plusieurs inconvénients. D'abord, l'interface réseau doit partager la bande passante du bus E/S avec les autres E/S. Ensuite, les performances des bus E/S sont très variables et toujours inférieures au bus mémoire. Un autre inconvénient est que l'ensemble de composants qui connectent le bus système au bus PCI forme un écran qui, certes, isole le bus système du bus E/S, mais rend aussi plus difficiles les échanges entre la mémoire principale et les E/S dont fait partie l'interface réseau. SHRIMP [5], MINI [29], et TNET [22] proposent de connecter l'interface réseau directement sur le bus système. Une étude très précise de l'implantation des messages actifs sur le Paragon [27] montre qu'une telle organisation peut réduire les performances de communication et ralentir le fonctionnement du processeur de calcul.
Les performances du processeur et de la hiérarchie mémoire des noeuds connectés sur le réseau de communication sont aussi des paramètres importants. Par exemple, dans [19] les auteurs remarquent que le débit mesuré pour une copie de mémoire à mémoire est inférieur au débit des liens du réseau de communication. Une étude comparant différentes configuration de processeurs et de hiérarchie mémoire à été faite par IBM sur le SP2. Les résultats indiquent notamment que la performance du processeur a une influence sur la latence.
La hiérarchie mémoire peut aussi influencer la latence par une très mauvaise localité du code d'un protocole [3]. Différentes approches sont proposées pour palier à ce problème [3] [30]. Enfin notons que dans certain cas, ce sont les performances de la mémoire principale et du bus système qui limitent le débit. C'est le cas dans SHRIMP où le réseau possède un débit maximum de 175 Mo/s par lien alors que les bus Xpress et EISA du PC ne permettent respectivement que 73 Mo/s et 33 Mo/s [18].

22.2.4 Les performances de communication

Les mécanismes précédents permettent d'atteindre des performances de l'ordre de 10 us en latence et 1 Gb/s de débit pour une communication point à point entre deux processus utilisateurs communiquant sur un matériel comme Myrinet en mode mono-utilisateur et sans faire intervenir le système.

22.3 Impact des performances de communication sur les performances globales

Dans le cas courant, une seule application s'exécute à la fois sur une architecture parallèles soit parce que l'architecture est mono-utilisateur soit parce que l'architecture est partitionnée physiquement (space sharing) et correspond à une architecture mono-utilisateur pour la partition soit encore parce qu'un système d'ordonnancement sophistiqué est utilisé (gang scheduling, co-scheduling, etc.) qui permet de partager temporellement (time sharing) l'architecture entre toutes les applications exécutées et de considérer pour l'intervalle de temps (time slice) attribué à une application qu'elle est sur une architecture mono-utilisateur.
Cette partie s'intéresse à la modélisation des communications, à l'estimation de leur impact sur les performances globale et aux techniques pour réduire cet impact dans ce cas de figure.

22.3.1 Modélisation des performances de communication

Différentes modélisations ont été proposées pour rendre compte de l'impact des performances de communication sur les performances globales d'un traitement parallèle. Les modèles les plus connus sont BSP [40] et LogP [12]. BSP comme LogP peuvent être définis comme une machine virtuelle comprenant n processeurs fonctionnant à la même vitesse (même fréquence d'horloge) et possédant chacun une mémoire privée. L'architecture mémoire est donc distribuée et le modèle de communication est à passage de messages. Un réseau de communication relie les processeurs entre eux. Dans BSP, le réseau de communication peut aussi réaliser des barrières de synchronisation.
BSP a été définit pour rendre compte d'une exécution fondamentalement synchrone d'un traitement parallèle. Une machine BSP exécute un programme en effectuant une séquence de $Superstep$. Chaque $Superstep$ consiste en trois phases consécutives : une phase de calcul, une phase de communication globale et une barrière de synchronisation.
La phase de calcul peut utiliser des données provenant de la phase de communication du $Superstep$ précédent et produit des données pouvant être communiquées lors de la phase de communication global du $Superstep$ courant. La barrière de synchronisation garantie à tous les processeurs que tous les calculs sont terminés et que toutes les communications sont arrivées à destination avant le passage au $Superstep$ suivant.
Le temps d'exécution d'un $Superstep$ est exprimé à partir des paramètres :
- w : le maximum du temps de calcul des processeurs,
- g : le débit du réseau pour chaque processeur,

- h : le maximum du nombre de messages émis ou recus par les processeurs et
- l : la durée de la barrière de synchronisation.

$Tsuperstep = w + gh + l$

Le temps de calcul total est la somme des temps d'exécution des $Superstep$ du calcul.
LogP a été définit comme une représentation suffisamment détaillée d'un calculateur parallèle pour rendre compte de l'impact des différentes composantes de la performance de communication tout en restant suffisamment générale pour ne pas capturer des caractéristiques trop spécifiques d'une architecture.
Pour LogP, un multiprocesseur à mémoire distribué peut être caractérisé par quatre paramètres :
- L : la latence de communication d'un petit message (quelques octets) de la mémoire du processeur source jusqu'à sa destination,
- o : l'overhead (le surcoût) définit comme la durée pendant laquelle le processeur est engagé dans l'émission ou la réception d'un message. Durée pendant laquelle il ne peut rien faire d'autre.
- g : le gap (intervalle de temps) est le minimum de temps entre deux émissions consécutives de messages ou deux réceptions consécutives de messages sur un module (processeur et mémoire). C'est l'inverse de la bande passante par processeur du système de communication.
- P : le nombre de modules processeur/mémoire.
Les ressources de stockage du réseau sont supposées finies (en taille et en nombre). Ainsi, seulement L/g messages peuvent être en transit simultanément. Envoyer un paquet entre deux machines prend : $L + 2o$. Une émission bloquante d'un petit message (attente de l'acquittement de réception avant libération du processeur) requière $2L + 4o$.
Voici les valeurs des paramètres de LogP pour quelques architectures parallèles :

	L	o	g
Intel Paragon	6.5 us	1.8 us	7.6 us
Meiko CS2	7.5 us	1.7 us	13.6 us
Berkeley NOW	5 us	2.9 us	5.8 us
CM5 AM	6 us	2 us	4 us

22.3.2 Impact des differentes composantes du temps de communication sur les performances globales

Les modélisations présentées ci-dessus tentent de décomposer le temps de communication en plusieurs composantes. Quelle est l'impact de chacune de ces composantes sur les performances des applications ? Quel gain peut on encore espérer en réduisant davantage le temps de communication ? L'article [28] apporte des éléments de réponse à ces deux questions.
Le point de départ de l'étude est de construire et d'utiliser un environnement permettant de mesurer avec précision l'impact des différents paramètres LogP d'une architecture sur les performances de plusieurs Benchmarks parallèles. En faisant varier indépendamment les paramètres L, o, et g, on détermine la contribution de chacun sur la performance globale.
Les benchmarks étudiés sont un tri parallèle (Radix) sur 16 million de clés, une application de modélisation de propagation d'onde électromagnétique (EM3D),un tri probabiliste (Sample) sur 32 million de clés, une implémentation de problème N-corps (Barnes) avec 1 million de corps, une application de lancé de rayons (P-Ray) avec 16390 objets et une image de 1 million de pixels, une implémentation parallèle d'un outil de vérification de protocole (Mur@), un algorithme recherchant les composantes connexes d'un graphe avec 4 millions de noeuds, ainsi que d'autres applications de tri.
Le support de référence utilisé pour l'expérimentation est la machine NOW de Berkeley à 32 processeurs. Ses caractéristiques LogP sont précisées ci-dessus. L'expérimentation consiste à prendre comme base les paramètres de cette machine et à augmenter progressivement (8 valeurs différentes) chacun d'eux indépendamment des autres. L varie de 5 à 105 us, o varie de 3 à 103 us et g varie de 6 à 105 us. Pour chaque expérience, le temps d'exécution des benchmarks est mesuré. En divisant le temps obtenu par le temps de référence (paramètres initiaux de NOW), on obtient le facteur de ralentissement.
Lorsque o atteint 100 us, avec L et g conservés aux valeurs initiales, le facteur de ralentissement est de 2 à 50 suivant les applications. Toutes les applications (sauf une et pour une raison indirecte) présente un ralentissement linéaire suivant o. Ceci indique que les efforts pour réduire l'overhead (le surcoût) de communication sont essentiels et doivent être poursuivis.
Lorsque g atteint 100 us, toujours avec L et o conservant leur valeurs initiales, le facteur de ralentissement varie entre 1 et 16. Les applications ayant les plus grands ratio communications / opérations (1 message toutes les 6 us pour Radix, 1 toutes les 8 us pour EM3D et un toutes les 13 us pour Sample) sont celles qui subissent les ralentissement les plus sévères. Les autres sont au plus ralenti d'un facteur 4 pour g = 100us.
Lorsque L atteint 100us, avec g et o conservant leur valeurs initiales, le ralentisse-

ment est moins prononcé et est inférieur à 4 pour toutes les applications sauf une : une version de EM3D (ralentissement de 8). Cette version de EM3D effectue beaucoup de lecture bloquante et n'utilise aucune techniques pour recouvrir la latence. Les applications qui n'effectuent pas de lecture distante (implémentation du get) mais sont construites autour d'écritures (send) sont relativement insensibles à la latence.
En conclusion, cette étude montre que les performances des mécanismes de communication affectent sensiblement les performances d'exécution des applications parallèles ; les ralentissements pouvant être très sévères pour certaines applications. Le paramètre le plus important est l'overhead. C'est celui qui a recu le plus d'attention dernièrement. Les surcoûts logiciels étaient encore très élevés sur la génération précédente de machines parallèles (45 us pour la CM5 sans messages actifs). Les progrès actuels (messages actifs, fast messages) permettent de réduire la valeur de ce paramètre à environ 3 us.

22.3.3 Des approches générales pour reduire l'impact des communications

A partir de ces observations, un certain nombre de techniques ont été définit pour tenter de réduire l'impact des performances de communication sur les performances globales. Ces méthodes sont utilisées lors de la conception d'applications parallèles (notamment celles écrites pour le passage de messages) ou pour la génération de code des compilateurs (par exemple : dataparallèles).
L'objectif initial est évidemment d'éviter ou de réduire le plus possible les communications. Le placement des données et la granularité des taches parallèles sont deux paramètres utilisés pour augmenter la localité des références aux données et donc minimiser les communications.
Lorsque des communications sont inévitables, il s'agit de limiter leur nombre. Un autre principe dérive directement de l'impact des différentes composantes du temps de communication : la bande passante du réseau à beaucoup moins d'influence que le couple latence+overhead. Ainsi, il est préférable de regrouper les informations à communiquer pour former moins de messages même si chaque message comporte plus de données. Ce principe s'appelle la vectorisation des communications dans le contexte de la compilation des langages dataparallèles.
Le déplacement des communications et le recouvrement des communications par du calcul sont des techniques visant à réduire encore l'impact des communications inévitables (alors qu'elles ont été vectorisées par exemple). Exécuter les communications le plus tôt possible par rapport aux opérations qui utilisent leurs résultats dépend des relations de dépendance (données, contrôle, ressource) du programme. Recouvrir les communications par du calcul impose que les processeurs émetteurs et récepteur soient impliqués le moins possible dans la communication. Les commu-

nications asynchrones (ou non bloquantes) correspondent à ce principe.

22.4 Influence du support et de l'environnement d'execution

Les techniques d'amélioration de performance, les modèles et les approches de réduction de l'impact des performances de communication sur les performances globales ont été définit par rapport au contexte des architectures existantes et de leur mode d'exploitation (fondamentalement mono-utilisateur).
Le rapport coût-performance reste la motivation principale de l'amélioration des architectures. Ceci a au moins deux conséquences. D'abord l'architecture des machines parallèles continue d'évoluer. Ensuite, l'exploitation des architectures parallèles tend vers la généralisation des types d'application exécutées (base de données, traitement graphique, traitement numérique) et vers un accroissement de leur partage.
Dans cette partie nous présentons trois points qui peuvent influencer la conception des applications parallèles, les modèles de performance et les approches pour réduire le coût des communications.

22.4.1 Evolution des architectures parallèles généralistes

Trois facteurs sont à l'origine de la direction actuelle des architectures parallèles vers les réseaux de multiprocesseurs.
D'abord, l'architecture multiprocesseurs commence à se généraliser et des multiprocesseurs bas-coût deviennent disponibles. C'est le cas par exemple des PC biprocesseurs réalisés à partir de cartes mères spécifiques (biprocesseur) mais reprenant presque intégralement l'environnement PC standard. Cette direction est renforcée par la fabrication en grand volume par Intel de carte mères quadriprocesseurs (SHVS). La plupart des grands fabriquants de stations et de serveurs (UNIX ou Windows NT) proposent des serveurs multiprocesseurs. Pour IBM comme pour DEC ces serveurs peuvent être interconnectés par un réseau rapide. Par exemple, il est possible dans un IBM SP de remplacer des noeuds monoprocesseurs (Power2 SC) par des noeuds multiprocesseurs pouvant comporter jusqu'à 8 powerPC 604 chacun.
Un deuxième argument en faveur des réseaux de multiprocesseur est leur rapport coût-performance théoriquement favorable par rapport à un réseau de monoprocesseurs (à nombre de processeurs égal). En effet, le coût des réseaux rapides comme Myrinet ou SCI évolue approximativement linéairement par rapport au nombre de noeuds connectés. Ceci s'explique par le coût important de l'interface réseau et le coût et le nombre plus réduit de commutateurs (4 ou 8 interfaces par commutateur 4x4 ou 8x8).
Comparativement, le coût d'un multiprocesseur progresse aussi linéairement par rapport au nombre de processeurs mais généralement le coût d'un processeur est beau-

coup plus faible que celui d'une connexion au réseau (3000 Frs pour un PentiumPro 200, 8000 Frs pour une interface Myrinet). Cependant, il faut impérativement que le coût de la carte mère multiprocesseur ne compense pas cet écart.

Le troisième argument est la gradation en performance d'échange d'information pour les réseaux de multiprocesseurs qui n'existe pas pour les réseaux de monoprocesseurs. Dans un réseau de monoprocesseurs, les performances de communication sont généralement les mêmes pour tous les processeurs (environ 10us de latence et 1Gbit/s de débit).

Dans un réseau de multiprocesseurs, les performances d'échange d'information sont meilleures à l'intérieur du multiprocesseur (mémoire partagée) qu'entre multiprocesseurs (traversée du réseau). Un échange d'information (un mot) à l'intérieur d'un multiprocesseur prend moins de 1us. Généralement, le débit du bus interne est de l'ordre d'1 Goctet/s. Les réseaux de multiprocesseurs possèdent donc un avantage potentiel en performance de communication sur les réseaux de monoprocesseurs.

Les réseaux de multiprocesseurs posent plusieurs problèmes nouveaux par rapport aux architectures classiques.

D'abord, il s'agit généralement d'architectures à modèle mémoire hybride : mémoire partagée à l'intérieur de chaque multiprocesseur et passage de messages entre multiprocesseurs. Si comme c'est le cas habituellement, le modèle de programmation bas niveau reflète l'architecture mémoire de la machine, le modèle de programmation des réseaux de multiprocesseurs nécessite de mixer dans une même application le passage de messages et la mémoire partagée. La portabilité est rendue plus difficile puisque l'on ne connaît pas à priori la répartition des processeurs (le nombre de processeurs par multiprocesseur).

Pour éviter ce problème, les modèles de programmation homogènes (passage de messages et mémoire partagée) sont portés sur les réseaux de multiprocesseurs. Par exemple, IBM a implémenter sa version rapide de PVM sur le multiprocesseur pouvant servir de noeud à la machine parallèle SP [2]. D'autres approches visent à installé un espace d'adressage unique virtuel par dessus une architecture à passage de messages (TreadMark [25] par exemple).

Le deuxième problème important, compte tenu de la gradation en performance de communication, est de pouvoir distribuer les données de l'application de sorte à minimiser les échanges entre multiprocesseurs. Ici encore la portabilité est rendu plus difficile.

Les modèles de performance de communication actuels ne sont pas adaptés pour rendre compte de la gradation en performance d'échange d'information présente dans les réseaux de multiprocesseurs.

22.4.2 Environnement d'exécution partagé

Le coût des architectures parallèles et celui des réseaux rapides diminuant, le parallélisme devient possible au sein d'un réseau de stations en activité. L'architecture parallèle constituée de ressources centralisées ou le réseau de stations de travail interconnecté par un réseau rapide devient accessible à de nombreux utilisateurs.
Dans un tel site, les activités de différentes natures (séquentielles, parallèles, interactives, par lots) sont réparties (souvent manuellement) sur les différentes ressources. Par exemple, sur l'IBM SP de l'Université Paris-sud, les noeuds supportent en même temps la compilation et l'exécution d'applications parallèles. Toutefois, généralement, les machines sont partitionnées avec une partition pour le développement et l'expérimentation et une partition pour la production. L'article [23] présente un éventail des systèmes de gestion de ressources partagées disponibles actuellement et comparent leur caractéristiques.
Idéalement, le partitionnement en deux partitions (une pour les applications parallèle et une pour le reste) devrait s'adapter dynamiquement aux besoins des utilisateurs et aux ressources disponibles. Pour pouvoir décider d'un partitionnement (de la taille des deux partitions), il faut connaître précisément l'activité qu'engendre les utilisateurs.
Cette question s'est déjà posée lorsqu'il a s'agit de récupérer les ressources non utilisées par un utilisateur de station de travail pour en faire profiter les autres. Les études présentées dans [32] et [36] ont cherché à caractériser l'activité macroscopique de l'utilisateur. A cette échelle, une station de travail est disponible plus de 50% du temps. Ces travaux ont abouti à la définition de mécanismes de répartition de charge considérant qu'une station est disponible si son utilisateur ne s'est pas manifesté depuis 15 minutes [26] (1 minutes pour GLUNIX [39].
Si le système de répartition de charge doit réagir plus vite, il faut caractériser plus finement le signal d'activité engendré par les utilisateurs. Une étude [37] réalisée sur un réseau de stations de travail en activité à montré que le signal d'activité d'un serveur présente une structure similaire sur les différentes échelles temporelles utilisées (us, seconde, minute) pour son observation. Ce signal correspond à une processus stochastique auto-similaire. En pratique cela signifie que quelque soit l'échelle à laquelle on se place, il est difficile de prévoir l'activité à l'instant suivant.
L'activité interactive des utilisateurs (édition, compilation, gestion de fichiers, etc.) est donc particulièrement complexe à prendre en compte par des mécanismes de gestion des ressources. Deux directions sont possibles pour le mixage des activités interactives et des applications parallèles : l'approche par partitionnement que nous avons déjà évoquée ou le mixage sans précautions [36]. Comme une adaptation dynamique rapide est très complexe voir impossible, l'approche par partitionnement conduit à une gestion sous-optimale des ressources.
L'approche par mixage sans précaution consiste à charger les ressources sans vérifier

leur disponibilité. En plus des activités interactives, les stations recoivent des charges parallèles. Cette approche à deux effets pervers. D'abord elle ralenti l'activité interactive par exemple à cause du swap des pages de l'utilisateur interactif. Ensuite, elle ralenti l'exécution des applications parallèles non seulement à cause du partage du processeur et de la mémoire (vidange du cache lors du changement de contexte parallèle/interactif) mais aussi à cause du rallongement des communications et de la synchronisation.
Des expériences menées sur l'IBM SP de L'Université Paris-sud montrent une très grande variance du temps de communication (plusieurs ordres de grandeur de variation). Ceci s'explique par le fait que l'application parallèle a été déschédulée pendant la communication. Pour limiter les interactions, les auteurs de [36] proposent d'établir un contrat social : un utilisateur ne poura pas être retardé plus de x fois dans une journée à cause d'une application parallèle s'exécutant sur sa station.
Le partage du support d'exécution entre les activités interactives et les applications parallèles posent donc de nombreux problèmes pour la modélisation des performances de communication et l'optimisation des mécanismes de communication. Pour la programmation parallèle, de tels environnements impliquent de gérer soit dans le programme, soit pendant la compilation, soit encore par un exécutif la variation dynamique du nombre de ressources attribuées pour l'exécution parallèle.

22.4.3 Multiple échelles de parallélisme et parallélisme multi-échelle

Un autre facteur risque de perturber la conception des applications parallèles et les modèles de performances. Jusqu'à récemment, les applications parallèles comme celles du traitement numérique étaient exécutées sur les architectures parallèles le plus souvent réalisées à partir d'éléments de stations de travail (microprocesseurs, mémoire) connectés par un réseau rapide.
Plusieurs directions actuelles de la recherche en système distribué à grande échelle et de la recherche en architecture tendent vers la notion d'échelles multiples de parallélisme.
Internet peut être utiliser comme support de communication entre plusieurs sites pour exécuter une application parallèle. Le Metacomputing [38] ou le Globalcomputing sont des cadres d'étude visant à porter le parallélisme sur Internet. A cette échelle, les temps de communication sont très importants, les architectures sont hétérogènes et leur disponibilité évolue dynamiquement dans le temps.
La gradation en performance de communication entre deux sites communiquant à travers Internet et entre deux processeurs d'un même site est encore accrue par rapport au cas des réseaux de multiprocesseurs. L'hétérogénéité des sites interconnectés implique des modèles d'exécution basé sur l'appel de services distant ou sur l'exportation de code. L'évolution dynamique de la disponibilité des sites intercon-

nectés nécessite des mécanismes de recrutement de sites, de répartition de charge, de migration de traitement, etc.
Les applications exécutées à cette échelle seront probablement différentes de celles qui sont exécutées sur les architectures parallèles classiques. Si une application parallèle, concu pour l'échelle des architectures parallèles, devait être exécutée à l'échelle d'Internet, elle nécessiterai probablement de nombreuses modifications.
Une des directions de recherche en architecture des machines vise à intégrer de la logique dans les mémoires DRAM. La capacité de stockage des DRAM évolue avec la technologie d'intégration. La capacité d'intégration d'un boîtier DRAM croit de manière exponentielle dans le temps. En 2001, un boîtier DRAM aura une capacité de 1 Gigabits (128 Mo). Cette valeur est supérieure à la capacité totale de la mémoire des stations actuelles. Si plusieurs boîtiers de ce type sont utilisés, comme c'est le cas actuellement, la mémoire d'une station atteindra plusieurs Gigaoctets.
Si cette capacité mémoire peut s'avérer nécessaire compte tenu de l'évolution des systèmes d'exploitation et des applications, il est envisageable de réserver une petite partie (10%) de la capacité d'intégration d'un boîtier DRAM pour y intégrer de la logique. C'est le principe des Processeurs Intégrés avec la Mémoire (PIM).
Plusieurs utilisations de cette logique sont envisageables comme par exemple faciliter le préchargement des données pour un microprocesseur externe ou intégrer le microprocesseur dans la DRAM. Mitsubishi propose déjà le M32R [17] qui intègre dans un même boîtier un RISC 32 bits, 1Mo de DRAM, 4ko de cache et un bus 128 bits. Cette PIM peut exécuter du Bytecode JAVA (un interprète JAVA a été implémenté).
D'autres projets visent à réaliser des architectures dédiées au traitement d'images, aux bases de données ou au traitement numérique. Dans la dernière approche, le projet IRAM de Berkeley [34] vise à intégrer un processeur vectoriel dans un boîtier DRAM. D'autres approches sont examinées comme des architectures SIMD [16] ou des multiprocesseurs dans un boîtier DRAM (projet PPRAM [31]).
Les études actuelles comprennent l'intégration dans le boîtier DRAM d'un mécanisme de communication rapide. La norme SCI est un candidat pour ces mécanismes. Dans cette perspective, avec des communications à 1 Gigabits/s et moins de 1 us de latence, un réseau de PIM constituera une architecture parallèle. Le parallélisme inter-PIM (réseau de PIM) ou intra-PIM (multiprocesseur dans une PIM) constituera une troisième échelle de ressource pour le parallélisme. Toutefois, les connaissances sur les PIM sont encore trop limitées pour estimer les caractéristiques spécifiques de cette échelle et identifier leurs implications dans la conception des applications parallèles.
De ces multiples échelles de parallélisme, on dérive (trop vite ?) la notion de parallélisme multi-échelle. Les problèmes posés sont alors la programmation, le modèle d'exécution, les modèles de communication, les modèles de gestion des ressources, etc. qui permettent de percevoir une architecture parallèle multi-échelle (réseaux de

stations connectés sur Internet comportant des PIM dans chaque station) comme un tout.

22.5 Conclusion

Après une période d'optimisation, les architectures parallèles connaissent une période de diffusion massive. Les modèles de programmation, les modèles de performance et les techniques d'optimisation ont été définis dans le cas d'une exploitation particulière de ces machines : mono-application et mono-utilisateur et échelle unique (celle des ressources de station). Un nouvel environnement d'exploitation se dessine avec un partage des ressources par des applications de nature variées (interactives, parallèles, etc.) et de multiples échelles de ressources.
De nouveaux modèles de programmation, d'exécution et de performance ou une adaptation des modèles existants sont nécessaire si l'on souhaite exploiter ce nouvel environnement.

Bibliographie

[1] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[2] M. Bernaschi. Efficient message passing on shared memory multiprocessors. *Lecture Notes in Computer Science*, 1156:221–??, 1996.

[3] T. Blackwell. Speeding up protocols for small messages. In *ACM SIGCOMM'96*, pages 85–95, 1996.

[4] J. M. BLUM, T. M. WARSCHKO, and W. F. TICHY. PSPVM: implementing PVM on a high-speed interconnect for workstation clusters. Technical Report iratr-1996-17, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, 1996.

[5] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-memory-mapped network interfaces. *IEEE Micro*, 15(1):21–28, Feb. 1995.

[6] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 142–153, Apr. 1994.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[8] F. Cappello. *Etude d'une architecture parallèle à ressources équilibrées et communications compilées*. 1994.

[9] D. D. Clark, V. Jacobson, J. Romkey, and H. Slawen. An analysis of tcp processeing overhead. pages 23–29, June 1989.

[10] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM'90*, pages 200–208, 1990.

[11] D. Comer. *Internetworking with TCP/IP Vol. 1: Principles, Protocols and Architecture*. Prentice Hall, London, 2 edition, 1991.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and v. Thorsten. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, May 1993.

[13] R. Cypher, H. Ho, S. Konstantinidou, and P. Messina. Architectural caracteristics od scientific applications. In *20th ISCA*, pages 2–13, 1993.

[14] W. J. Dally, A. Chien, R. Davidson, S. Fiske, S. Furman, G. Fyler, D. B. Gaunce, W. Horwat, S. Kaneshiro, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, E. Spertus, B. Totty, D. Wallach, and S. Wills. The J-machine: a fine-grain parallel computer. *Computing Systems in Engineering*, 3(1-4):7–15, Dec. 1992.

[15] T. V. Eicken, V. Avula, A. Basu, and V. Buch. Low-latency communication over ATM networks using active messages. Technical Report TR94-1456, Cornell University, Computer Science Department, Mar. 31, 1995.

[16] D. G. Elliott, W. M. Snelgrove, and M. Stumm. Computational ram: A memory-simd hybrid and its application to dsp. In *Custom Integrated Circuits Conference*, pages 30.6.1–30.6.4, May 1992.

[17] T. S. et al. A multimedia 32b risc microprocessor with 16mb dram. In *1996 ISSCC Digest of Technical Papers*, pages 216–217, Feb. 1996.

[18] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early experience with message-passing on the SHRIMP multicomputer. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 296–307, New York, May22–24 1006. ACM Press.

[19] Y. A. Fouquet, R. D. Schneeman, D. E. Cypher, and A. Mink. Atm performance measurement: throughput bottlenecks and technology barriers. 1995.

[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[21] D. S. Gustavson. Scalable Coherent Interface. In *Proc. of the 34th IEEE Int'l Computer Conf. (COMPCON Spring'89)*, pages 536–544, Feb. 1989.

[22] R. W. Horst. Tnet: a reliable system area network. pages 37–45, Feb. 1995.

[23] J. P. Jones and C. Brickell. Second evaluation of job queuing/scheduling software: Phase 1 report. Technical Report NAS-97-013, NASA Ames Research Center, June 1997.

[24] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in tcp/ip. In *Proc. ACM SIGCOMM'93*, pages 259–268, 1993.

[25] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Tread marks: Distributed shared memory on standard workstations and operating system. In USENIX Association, editor, *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 115–132, Berkeley, CA, USA, Winter 1994. USENIX.

[26] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.

[27] L. T. Liu and D. Culler. Evaluation of the intel paragon on active message communication. In *Intel Supercomputer users group conference*, June 1995.

[28] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *International Symposium on Computer Architecture*, Denver, CO., USA, 1997.

[29] R. Minnich, D. Burns, and F. Hady. The memory-integrated network interface. *IEEE Micro*, 15(1):11–20, Feb. 1995.

[30] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26,4 of *ACM SIGCOMM Computer Communication Review*, pages 73–84, New York, Aug.26–30 1996. ACM Press.

[31] K. Murakami, K. Inoue, and H. Miyajima. Ppram (parallel processing ram): A merged-dram/logic system-lsi architecture. In *International Conference on Solid State Devices and Materials*, Sept. 1997.

[32] M. W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12:269–284, 1991.

[33] Pakin, M. lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *In Supercomputing '95*. ACM, 1995.

[34] D. Patterson, T. Anderson, and K. Yelick. A case for intelligent dram: Iram. In *Hot Chips VIII*, pages 75–93, Aug. 1996.

[35] F. Potter. *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour les machines multiprocesseurs*. 1996.

[36] D. A. P. Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, New York, NY, USA, May 1995. ACM Press.

[37] O. Richard. Disponibilité des ressources dans un réseau de stations de travail : une nouvelle lecture. In *RenPar'9*, May 1997.

[38] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6), June 1992.

[39] A. M. Vahdat, D. P. Ghormley, and T. E. Anderson. Efficient, portable, and robust extension of operating system functionality. Technical Report CSD-94-842, University of California, Berkeley, Dec. 1994.

[40] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, Aug. 1990.

[41] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In D. Abramson and J.-L. Gaudiot, editors, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Australia, May 1992. ACM Press.

[42] M. Welsh, A. Basu, and T. von Eicken. Low-latency communication over fast ethernet. In *Proc. EUROPAR '96*, Aug. 1996.

Chapitre 23

Modélisation Stochastique et Problèmes de Décision

Olivier Brun, Jean-Marie Garcia, Thierry Monteil (LAAS-CNRS)

23.1 Introduction

En tant qu'outil mathématique, les modéles stochastiques sont généralement plus réalistes que les modéles déterministes dans la mesure où ils sont capables d'exprimer le comportement irrégulier et imprévisible des demandes des applications ou des utilisateurs. Au sein d'un système informatique, ces demandes se traduisent généralement par des files d'attente lorsque les ressources requises ne sont pas disponibles. Puisque pour l'utilisateur les performances du système ne sont visibles qu'à travers le prisme du temps de réponse, qui est directement lié à la gestion des files d'attente, les concepteurs de systèmes sont souvent confrontés à des problèmes de décision optimale.

Nous rappelons tout d'abord quelques uns des principaux résultats de la théorie des files d'attente et des processus de décision markovien, puis nous illustrons ces techniques par des exemples d'applications.

23.2 Théorie des files d'attente

Les principales composantes d'un système de files d'attente sont:

- un serveur,
- une file d'attente dans laquelle on place les clients qui ne peuvent pas être immédiatement servis,

- une source, qui est une collection finie ou non de clients.

La modélisation probabiliste d'un système de files d'attente suppose la spécification de la distribution de probabilités décrivant les durées entre arrivées successives de clients, ainsi que la loi de service et le mode de gestion (ou discipline) de la file. On notera dans la suite $\widetilde{t}$ et $\widetilde{x}$ les variables aléatoires représentant respectivement la durée entre deux arrivées successives et la durée de service. Les fonctions de répartition associées sont:

$$A(t) = P[\tilde{t} \leq t] \text{ et } B(x) = P[\tilde{x} \leq x] \tag{23.1}$$

Enfin on notera $\overline{t}$ et $\overline{x}$ (resp. $\overline{t^k}$ et $\overline{x^k}$) les moyennes (resp. les moments d'ordre k) de ces variables aléatoires. On définit souvent les taux inter-arrivées λ et μ par $\overline{t} = \frac{1}{\lambda}$ et $\overline{x} = \frac{1}{m\mu}$, où m est le nombre de serveurs.

$\widetilde{t}$ et $\widetilde{x}$ sont les variables aléatoires qui gouvernent le système. Les autres variables aléatoires qui vont nous intéresser sont celles qui permettent d'évaluer ses performances. Ce sera souvent le nombre $N(t)$ de clients dans le système à l'instant t, $\widetilde{w}$ le temps d'attente dans la file de chaque client et $\widetilde{s}$ le temps de réponse du système.

23.2.1 Résultats généraux

Un des paramètres les plus importants d'un système de files d'attente est son facteur d'utilisation ρ:

$$\rho = \frac{\lambda \cdot \overline{x}}{m} \tag{23.2}$$

Cette quantité correspond en fait à la fraction moyenne des serveurs occupés dans le cas où tous les serveurs ont la même loi de service. Ainsi le paramètre ρ caractérise la stabilité du système. Pour $0 \leq \rho < 1$, le système sera dit stable. La durée moyenne passée dans le système par un client, $T = \overline{s}$, s'exprime simplement comme la somme du temps moyen d'attente W et du temps moyen de service:

$$T = \widetilde{x} + W \tag{23.3}$$

De plus, la formule de Little ([10]) permet de relier le temps de réponse moyen du système au nombre moyen de clients s'y trouvant:

$$\overline{N} = \lambda \cdot T \quad \text{et} \quad \overline{N}_q = \lambda \cdot W = \overline{N} - m \cdot \rho \tag{23.4}$$

où $\overline{N}_q$ désigne le nombre moyen de clients présents dans la file d'attente.

23.2.2 La file $M/M/1$

La file $M/M/1$ est le système de files d'attente le plus simple que nous présentons. Pour autant, le comportement de cette file est comparable en bien des points à celui de systèmes plus complexes.

La probabilité d'avoir k clients dans le système est([9]):

$$p_k = (1-\rho) \cdot \rho^k \tag{23.5}$$

où $\rho = \frac{\lambda}{\mu}$ représente le facteur d'utilisation. On en déduit le nombre moyen de clients:

$$\overline{N} = \frac{\rho}{1-\rho} \quad \text{et} \quad \sigma_N^2 = \frac{\rho}{(1-\rho)^2} \tag{23.6}$$

Grâce à la formule de Little, on a alors:

$$W = \frac{\rho/\mu}{1-\rho} \quad \text{et} \quad T = \frac{1/\mu}{1-\rho} \tag{23.7}$$

On constate que les termes $\overline{N}$, W et T présentent tous le même comportement vis à vis du facteur d'utilisation ρ: quand $\rho \to 1^-$, la taille de la file et les délais d'attente et de service augmentent sans borne.

23.2.3 La file $M/G/1$

Le système $M/G/1$ est donc caractérisé par un processus d'arrivée poissonnien de taux λ et par un processus de service dont la distribution $B(x)$ est générale. Les équations de base décrivant ce système sont:

$$q_{n+1} = \begin{cases} q_n - 1 + v_{n+1} \text{ , si } q_n > 0 \\ v_{n+1} \text{ , si } q_n = 0 \end{cases} \tag{23.8}$$

où q_n est le nombre de clients dans le système au départ du client C_n, tandis que v_n est le nombre de clients rentré dans le système au cours du service de C_n.

Bien que le système $M/G/1$ ne soit pas Markovien, le processus observé aux instants des départs des clients C_n est lui Markovien. En s'appuyant sur cette propriété et sur les équations qui décrivent tout le comportement transitoire du système, on peut démontrer la formule de Pollaczek-Khinchin ([9]):

$$W = \frac{W_0}{1-\rho} \quad \text{où} \quad W_0 = \frac{\lambda \cdot \overline{x^2}}{2} \tag{23.9}$$

W_0 correspond en fait à la durée résiduelle moyenne de service du client en cours de traitement lors de l'arrivée d'un nouveau client.

De là, on peut déduire par la formule de Little:

$$T = \overline{x} + W = \overline{x} + \frac{\lambda \cdot \overline{x^2}/2}{1-\rho} \quad \text{et} \quad N = \rho + \lambda \cdot W = \rho + \frac{\lambda^2 \cdot \overline{x^2}/2}{1-\rho} \tag{23.10}$$

Pour de plus amples développements sur la théorie des files d'attente, le lecteur intéressé par ce sujet pourra notamment consulter [9] et [16].

23.2.4 Le cas d'un réseau

Dans un réseau réel, les files d'attente vont se coupler entre elles de façon complexe d'une part à cause de la topologie d'interconnexion, et d'autre part du fait du routage qui va introduire des lois d'arrivée et de départ des files d'attente ne permettant pas d'isoler chacune des files dans un cas bien connu de type $M/M/1$ ou $M/G/1$. Il existe un cas où la solution globale du réseau est donnée par une forme analytique appelée "forme produit". Ce cas se limite cependant à des routages ne dépendant pas de l'état du système (typiquement "load-sharing" stationnaire).
Pour étudier des politiques optimales de gestion des ressources dans des réseaux (routage, allocation, etc...), on sera amené à développer un modéle fin de représentation du processus et des interactions par les paramètres de commande (choix d'une route, choix d'une affectation, etc...). Ceci peut être fait de manière rigoureuse par la théorie des processus de décision Markovien dont nous faisons un bref rappel ci-après.
Dans le cas de système de grande taille (combinatoire des états possibles), cette approche atteindra malheureusement vite ses limites, et on sera amené à étudier des classes paramètrées de politiques de commande, soit au moyen de simulations événementielles, soit au moyen d'approximations sur les flux dans le réseau (approximation au premier ou au second ordre).

23.3 Processus de décision Markovien

23.3.1 Problème du coût total à horizon infini

Considérons le système dynamique à temps discret suivant:

$$x_{k+1} = f(x_k, u_k, w_k), \quad k \in N \tag{23.11}$$

sous les contraintes:

$$\begin{cases} x_k \in S \\ u_k \in U(x_k) \subset C \\ w_k \in D \end{cases} \tag{23.12}$$

x_k est l'état du système à l'instant k qui doit être dans S, l'espace d'état du système, et u_k est un paramètre de commande. L'ensemble des commandes admissibles dans l'état x_k est noté $U(x_k)$. Enfin w_k correspond à un bruit.

Une politique de commande $\pi = \{\mu_0, \mu_1, \cdots\}$ est une séquence de fonctions,

$$\begin{aligned} \mu_k : \ & S \to C \\ & x_k \mapsto \mu_k(x_k) \in U(x_k) \end{aligned} \tag{23.13}$$

qui à chaque état accessible à l'étape k associe une commande admissible dans cet état.

Etant donné un état initial x_0, le problème du coût total à horizon infini ([2]) est de trouver une politique $\pi = \{\mu_0, \mu_1, \cdots\}$ qui minimise:

$$J_\pi(x_0) = \lim_{N\to\infty} E_{w_k}\{\sum_{k=0}^{N-1} \alpha^k \cdot g(x_k, \mu_k(x_k), w_k)\} \tag{23.14}$$

Dans ce problème, la fonction coût par étape g est connue. Dans la suite nous la supposerons positive et bornée. Le paramètre $\alpha \in]0,1[$, lui aussi connu, est appelé taux d'actualisation du coût. $J_\pi(x_0)$ représente alors l'espérance du coût de fonctionnement du système sous la politique π. Ce qui va nous intéresser, c'est de déterminer, dans l'ensemble Π des politiques admissibles, la politique optimale $\pi^\star$ permettant de faire fonctionner le système avec un coût minimal $J^\star$:

$$J^\star(x_0) = J_{\pi^\star}(x_0) = \min J_\pi(x_0) \tag{23.15}$$

Une classe de politiques admissibles d'un interêt particulier est celle des politiques stationnaires $\pi = \{\mu, \mu, \cdots\}$ telles que la commande appliquée dans un état ne dépende que de cet état et non de l'instant à laquelle elle doit être appliquée. On notera J_μ le coût d'une telle politique. Dans les systèmes d'un interêt pratique, il est fréquent que la politique optimale soit stationnaire. Ceci est d'un grand interêt pour une mise en oeuvre simple.

23.3.2 Coût optimal sur k étapes

Pour toutes fonctions $J : S \to R$, bornée, et $\mu : S \to C$, on définit:

$$\begin{aligned} T(J)(x) &= \min_{u\in U(x)} E_w\{g(x,u,w) + \alpha \cdot J[f(x,u,w)]\} \\ T_\mu(J)(x) &= E_w\{g(x,\mu(x),w) + \alpha \cdot J[f(x,\mu(x),w)]\} \end{aligned} \tag{23.16}$$

T est donc une correspondance associant à une fonction J définie sur S une autre fonction $T(J)$. Il est alors clair que le calcul de $T(J)(x)$ correspond à une itération de l'algorithme de programmation dynamique. Ainsi, $T(J)(x)$ peut être vu comme le coût optimal sur une étape, sachant que l'état initial est x et que le coût terminal est $\alpha \cdot J(x_1)$. Posons alors:

$$T^k = T \circ T^{k-1}, \quad T^0(J) = J \quad \text{et} \quad T_\mu^k = T_\mu \circ T_\mu^{k-1}, \quad T_\mu^0(J) = J \qquad (23.17)$$

$T^k(J)(x)$ correspond alors au coût optimal sur k étapes lorsque l'état initial est x et que le coût terminal est $\alpha^k \cdot J(x_k)$.

23.3.3 Convergence de l'algorithme des approximations successives

Pour toute fonction $J : S \to R$, bornée, on a:

$$\forall x \in S, \qquad J^\star(x) = \lim_{k\to\infty} T^k(J)(x) \qquad (23.18)$$

De plus,

$$\forall k \in N, \qquad max_{x\in S} \left|T^k(J^\star)(x) - T^k(J)(x)\right| \leq \alpha^k \cdot max_{x\in S} \left|J^\star(x) - J(x)\right| \quad (23.19)$$

La méthode des approximations successives permet donc, à partir d'une fonction de coût arbitraire, d'obtenir la fonction coût optimal, et ce avec une convergence au moins aussi rapide qu'une convergence géométrique.

23.3.4 Equation d'optimalité de Bellman

La fonction coût optimal $J^\star$ est la seule solution bornée de l'équation:

$$\forall x \in S, \qquad J(x) = T(J)(x) \qquad (23.20)$$

De plus, une politique stationnaire $\left\{\mu^\star, \mu^\star, \cdots\right\}$ est optimale si et seulement si:

$$\forall x \in S, \qquad T(J^\star)(x) = T_{\mu^\star}(J^\star)(x) \qquad (23.21)$$

Pour conclure ce bref rappel sur les processus de décision Markovien, nous signalons quelques extensions du problème considéré ci-dessus.
Les résultats obtenus pour les processus de décision Markovien à temps discret s'étendent sans difficulté aux processus Markovien à temps continu par la technique d'uniformisation ([2]). Toujours dans [2], on considère également le critère coût moyen à horizon infini. On montre d'autre part comment modéliser un problème de décision lorsque l'information disponible est incomplète ou incertaine. Dans [13], on traite en détail des processus de décision semi-markovien à temps discret et continu. Une approche algorithmique de la modélisation stochastique et des problèmes de décision est disponible dans [16].

23.4 Applications

Dans cette partie, nous illustrons la mise en oeuvre des techniques des théories des files d'attentes et du contôle stochastique sur des applications concrètes. Le premier exemple que nous donnons concerne l'ordonnancement de la file d'accès à un processeur. Puis nous considérons le problème du routage optimal dans un système constitué de deux serveurs. Ensuite, nous développons l'exemple de l'acheminement optimal dans un système à commutation de circuits. Enfin, nous concluons ce chapitre par l'étude d'un problème moins classique dont le sujet est la gestion optimale du trafic d'un algorithme parallèle synchrone.

23.4.1 Disciplines de la file d'accès à un processeur

23.4.1.a Disciplines de gestion Round-Robin et Processor Sharing

La discipline Round-Robin (RR) a été une des premières employées dans les systèmes à temps partagé, essentiellement du fait de sa simplicité et parcequ'elle a la propriété d'associer les temps d'attente les plus courts aux tâches les plus courtes ([15], [12]).

23.4.1.a.1 Principe A chaque fois que le processeur est libre, il est alloué à la tâche en tête de la file d'attente pour un quantum Q de temps. Si la tâche se termine avant l'expiration de son quantum, alors elle quitte simplement le système. Sinon, elle est remise en queue de la file d'accès au processeur. De cette façon, une tâche fait autant de passe dans la file que son exécution requiert de quantum.

A leur arrivée, les nouvelles tâches sont simplement mises en queue de la file d'attente.

23.4.1.a.2 Modèle Nous supposerons que la loi d'arrivée est poissonnienne de taux λ. En ce qui concerne la loi de service, nous faisons l'hypothèse d'une distribution géométrique de paramètre σ:

$$g_i = \sigma^{i-1} \cdot (1 - \sigma) \tag{23.22}$$

Ainsi g_i représente la probabilité que l'exécution d'une tâche requière i quanta, c'est à dire $i \cdot Q$ unités de temps.

23.4.1.a.3 Expression du temps de réponse du système Considérons l'arrivée dans le système à l'équilibre d'une tâche P dont l'exécution requiert k quanta. Nous notons T_k le temps total passé dans le système par P. On montre que ([12]):

$$T_k = T_1 + \frac{(k-1) \cdot Q}{1 - \rho} + Q \cdot \{\lambda \cdot T_1 + \sigma \cdot \overline{n} - \frac{\rho}{1 - \rho}\} \cdot \frac{1 - \alpha^{k-1}}{1 - \alpha} \tag{23.23}$$

où:

- $\alpha = \lambda \cdot Q + \sigma$.
- $\rho = \frac{\lambda \cdot Q}{1-\alpha}$ est le facteur d'utilisation du système RR.
- $\overline{n} = \rho + \frac{(1+\sigma)\cdot\rho^2}{2\cdot(1-\rho)}$ représente le nombre moyen de tâches dans le système.
- $T_1 = \frac{1-\rho}{2} \cdot Q + \overline{n} \cdot Q$ est le temps passé dans le système RR par une tâche dont l'exécution ne requiert qu'un seul quantum de temps.

Le mode de gestion Processor Sharing (PS) correspond au cas limite de la discipline RR pour $Q \to 0$. En posant $t = k \cdot Q$, et en prenant la limite de T_k dans (23.23) quand $Q \to 0$ mais que $\rho = \frac{\lambda \cdot Q}{1-\alpha}$ reste constant, on a:

$$T(t) = \lim_{Q\to 0} T_{(t/Q)} = \frac{t}{1-\rho} \tag{23.24}$$

On peut montrer que ce résultat est vrai même pour une loi de service s quelconque ([14]):

$$T = \frac{E(s)}{1-\rho} \tag{23.25}$$

Ainsi on voit qu'avec la discipline Processor Sharing, la durée totale de service d'une tâche est directement proportionnelle au temps d'exécution de cette tâche. En particulier, T est indépendant de la variance de la loi de service. Intuitivement, il est clair que cela vient du fait que les tâches longues ne bloquent pas les tâches courtes comme c'est le cas avec la discipline FIFO.

23.4.1.b Comparaison entre les disciplines FIFO et PS

Il est important de réaliser que la loi du service a un effet important sur les performances des différentes diciplines de gestion de file. On peut illustrer ceci dans le cas des disciplines PS et FIFO. On a:

$$T_{PS} = \frac{E(s)}{1-\rho} \quad \text{et} \quad T_{FIFO} = E(s) + \rho \cdot \frac{[1 + C^2(s)]}{2\cdot(1-\rho)} \cdot E(s) \tag{23.26}$$

Donc:

$$T_{FIFO} - T_{PS} = \rho \cdot \frac{[C^2(s) - 1]}{2\cdot(1-\rho)} \cdot E(s) \tag{23.27}$$

Si pour la distribution exponentielle $T_{FIFO} = T_{PS}$, par contre pour une autre distribution nous avons:

$$\begin{array}{ll} T_{FIFO} > T_{PS} & \text{, si } C(s) > 1 \\ T_{PS} > T_{FIFO} & \text{, si } C(s) < 1 \end{array} \tag{23.28}$$

En ce qui concerne le mode de gestion de la file d'accès à un processeur, de nombreux autres résultats sont disponibles. Dans [3], on s'intéresse aux disciplines basées sur la gestion de priorité. Dans [5], on trouvera des comparaisons entre les différentes politiques. Enfin [4] est un ouvrage qui traite des politiques de gestion des ressources dans un système d'exploitation.

23.4.2 Routage optimal de messages dans un système à deux serveurs

Considérons le système constitué de deux serveurs, dont les temps de service sont indépendants et exponentiellement distribués de paramètre μ_1 et μ_2, représenté sur la figure 23.1. Chaque serveur a sa propre file d'attente. Un mécanisme permet de router les clients, qui arrivent suivant un flot poissonnien de taux λ, vers l'une des deux files.

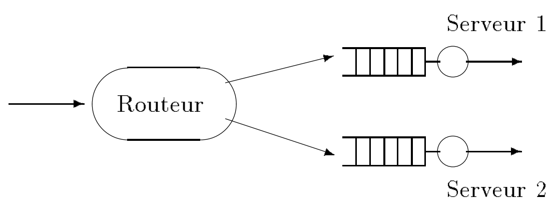


Figure 23.1 : Système considéré. Le problème est de router les messages vers une des deux files de façon à minimiser l'attente dans les files.

On veut déterminer quelle est la politique optimale de routage. On peut exprimer le coût de la façon suivante:

$$E\{\int_0^\infty e^{-\beta \cdot t} \cdot [c_1 \cdot x_1(t) + c_2 \cdot x_2(t)]\, dt\} \tag{23.29}$$

,où β, c_1 et c_2 sont des réels donnés, et $x_1(t)$ et $x_2(t)$ représente le nombre de clients respectivement dans les files 1 et 2 à l'instant t.

On peut se ramener à un problème de coût total à horizon infini en temps discret en construisant la version uniforme de ce problème ([2]).

Nous prenons comme espace d'états du système l'ensemble des paires (i, j) décrivant le nombre de clients dans les files 1 et 2. L'équation de Bellman prend alors la forme suivante:

$$\begin{aligned} J(i,j) = \; & \tfrac{1}{\beta+\nu} \cdot \{c_1 \cdot i + c_2 \cdot j + \mu_1 \cdot J[(i-1)^+, j] + \mu_2 \cdot J[i, (j-1)^+]\} \\ & + \tfrac{\lambda}{\beta+\nu} \cdot \min\{J(i+1,j), J(i,j+1)\} \end{aligned} \tag{23.30}$$

où pour tout x nous notons: $(x)^+ = max[0, x]$.

Le premier terme de $J(i, j)$ correspond au coût de l'état présent (i, j) et le second au coût de la décision optimale. On voit donc que la politique optimale est de router un client qui arrive vers la file 1 si et seulement si l'état (i, j) du système appartient à l'ensemble

$$S_1 = \{(i, j) \,/\, J(i + 1, j) \leq J(i, j + 1)\} \tag{23.31}$$

Intuitivement, il est clair qu'il est optimal de router vers la file qui réalise le meilleur compromis entre durée de service et nombre de clients dans la file. Ainsi, si la décision optimale est de router vers la file 1 quand le système est dans l'état (i, j), alors la décision optimale doit être la même quand le système est dans l'état $(i - 1, j)$. Autrement dit, l'ensemble S_1 des états du système pour lesquels il est optimal de router vers la première file peut être caractérisé par une fonction de seuil monotone et non-décroissante F ([2]):

$$S_1 = \{(i, j) \,/\, i = F(j)\} \tag{23.32}$$

En raison de ses nombreuses applications en télécommunication et en système de production, ce type de problème a suscité de nombreux travaux. Dans [6] on considère le cas d'un processus d'arrivée général et de deux serveurs identiques ($\mu_1 = \mu_2$). On montre alors que si le nombre de clients dans les deux files est observé la décision optimale est de router vers la file la plus courte, tandis que si cette information n'est pas connue mais que l'on sait qu'à l'instant initial les deux files sont de même longueur, il est optimal d'alterner entre les deux files. Dans le cas de serveurs hétérogènes, [8] démontre l'existence d'une politique de seuil aussi bien pour le critère coût total que pour le critère coût moyen. Enfin, [1] considère le cas intéressant dans lequel l'information sur l'état du système n'est disponible qu'après un certain délais.

23.4.3 Acheminement optimal dans un système à commutation de circuit

Considérons un noeud de commutation et de routage R qui reçoit des requêtes pour acheminer des messages par commutation de circuits. Ces messages peuvent être acheminés sur un des n circuits disponibles au départ du centre R. Un circuit, une fois affecté à une requête, reste occupé pendant le temps de traitement de la requête (temps de communication).
La requête une fois envoyée sur un des circuits peut être bloquée en aval dans le réseau de commutation. Pour représenter ce phénomène (dû au réseau), nous affectons à chaque circuit une probabilité fixe de blocage du chemin en aval du circuit. Soit $(1 - e_i)$ cette probabilité pour le circuit i.

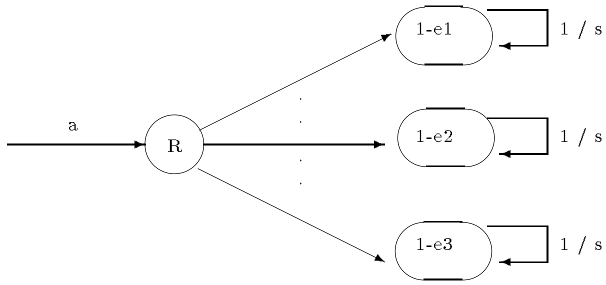


Figure 23.2 : Le noeud de commutation A et les circuits qui lui sont reliés

Soit $X_t = \{x_t^1, \ldots, x_t^i, \ldots, x_t^n\}$ l'état des circuits à l'instant t, avec $x_t^i = 0$ si le circuit est libre et $x_t^i = 1$ sinon. Les requêtes arrivent avec un taux a suivant une loi d'arrivée quelconque. La durée d'occupation d'un circuit obéit à une loi exponentielle de durée $\frac{1}{s}$ quelque soit le circuit occupé.
Le problème de décision qui se pose est alors de trouver la politique d'acheminement optimale permettant de traiter le maximum de requêtes dans un intervalle de temps donné, c'est à dire la politique qui maximise l'utilisation des circuits :

$$maxE\{\int_0^{\infty} |X_t|\, dt\} \tag{23.33}$$

On montre par résolution analytique des équations de Bellman que la politique MOP (Myopically Optimal Policy) qui consiste à affecter la requête sur le circuit i tel que:

$$i = arg \quad max\{e_i / x_t^i = 0\} \tag{23.34}$$

est la politique optimale du système.
Une généralisation de ce résultat permet de montrer que dans un réseau à commutation de circuit il faut faire du débordement des routes les plus efficaces vers les routes les moins efficaces [7].

23.4.4 Allocation dynamique de messages dans un algorithme synchrone itératif

23.4.4.a Introduction

Un algorithme parallèle est un système de tâches qui collaborent à un but commun en passant par des phases de calcul et de communication. Les phases de communication sont structurées, i.e. elles peuvent être décrites à l'aide d'un graphe appelé graphe de communication. Les noeuds de ce graphe représentent les tâches et ses arcs correspondent aux flots d'information échangés entre les tâches.

Lorsque la communication est asynchrone, les informations échangées par les tâches doivent être stockées en un ou plusieurs points intermédiaires. Dans une architecture faiblement couplée, le ou les points intermédiaires, qui sont souvent implémentés de manière logicielle, sont appelés serveurs de données.

Nous appelerons fonction d'allocation des messages une fonction associant un serveur de données à chaque flot d'information du graphe de communication. Le problème que nous considérons ici est, étant donné une application et un état de la machine parallèle (charge des processeurs), de déterminer quelle est la fonction d'allocation qui minimise la durée de l'application.

Nous illustrons dans la suite comment on peut résoudre ce problème dans un cas particulier.

23.4.4.b Etude d'un cas particulier

Nous considérons ici un algorithme itératif synchrone constitué de deux tâches indépendantes, T_1 et T_2 communiquant entre elles à chaque itération. Cet algorithme est exécuté sur une architecture faiblement couplée constituée de trois processeurs réliés par un bus d'interconnexion. Sur chacun de ces processeurs est exécuté un serveur de données, S_1, S_2 et S_3, qui jouent le rôle de buffer pour les messages. Ce système est représenté sur la figure 23.3.

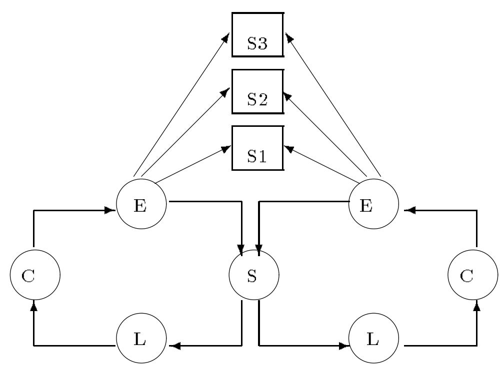


Figure 23.3 : Un algorithme itératif synchrone. A chaque fois qu'une tâche écrit un message, une décision d'allocation doit être prise.

Nous supposerons que les durées des phases de calcul et des phases de communication sont indépendantes, exponentiellement distribuées, et que leurs moyennes sont identiques pour les deux tâches. Cette hypothèse correspond en fait à celle d'un algorithme très irrégulier. On peut néanmoins se ramener à d'autres distributions de ces durées par la méthode des états fictifs ([9]).

Nous avons deux flots de communication et par conséquent 9 fonctions d'allocations.

Les 9 fonctions d'allocation correspondent à l'ensemble des couples (R_1, R_2), avec R_1 et R_2 appartenant à $\{1, 2, 3\}$.

23.4.4.b.1 Modèle Nous prenons comme espace d'état du système l'ensemble S des quadruplets $x = (e_1, e_2, n_1, n_2)$, où e_1 et e_2 représentent respectivement l'activité des tâches T_1 et T_2, et où n_1 et n_2 indiquent respectivement le serveur de données sur lequel sont écrits les messages à destination de T_1 et T_2 (cf. Tableau 23.1).

Numérotation des états				
Etat	Calcul	Ecriture sur le serveur R_i	Synchronisation après l'état R_i	Lecture
e_i	0	R_i	$R_i + 3$	7
n_i	0	0	0	$R_{(i+1)}$

Tableau 23.1 : Numérotation des états (e_i, n_i) pour la tâche $T_i \in \{T_1, T_2\}$

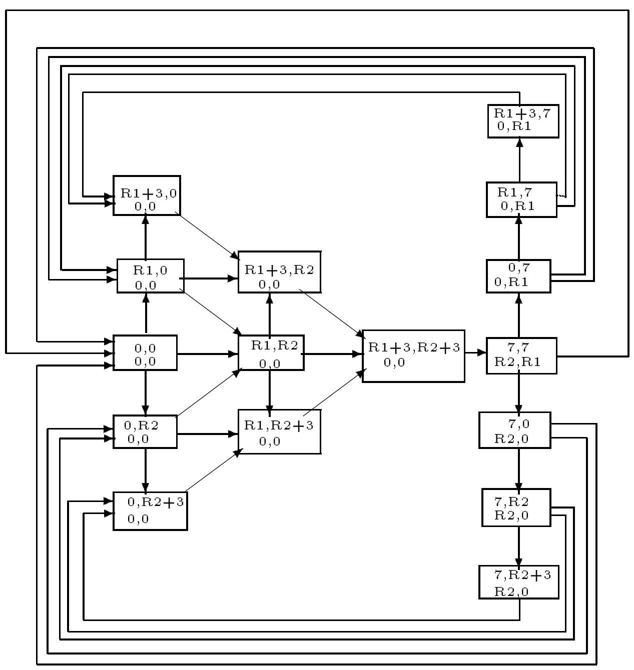


Figure 23.4 : Graphe de transition associé à l'allocation (R_1, R_2)

Nous avons représenté sur la figure 23.4 le graphe de transition associé à une allocation donnée (R_1, R_2). Les taux de transition doivent prendre en compte la charge des processeurs et l'abscence de recouvrement entre calcul et communication.

23.4.4.b.2 Fonction de coût On peut voir le problème d'allocation de messages comme un problème de contrôle stochastique. On se limitera ici aux politiques stationnaires $\pi = (\mu, \mu, \cdots)$, où $\mu(x) = (R_1, R_2)$ est une fonction d'allocation. Si on note P_μ la matrice de transition associée à la commande μ, le coût moyen $J_\mu(x_0)$ d'une telle politique peut alors s'écrire:

$$J_\mu(x_0) = \lim_{N \to \infty} \frac{1}{N} \sum_{k=0}^{N-1} P_\mu^k \cdot g_\mu \tag{23.35}$$

où x_0 est l'état initial du système et $g_\mu = (g[i, \mu(i)])$ est le vecteur coût instantané associé à la politique μ. Dans notre cas, puisque nous désirons minimiser la durée de l'application, il est naturel que $J_\mu(x_0)$ représente la durée moyenne d'une itération ou, ce qui est équivalent, le nombre moyen d'itérations par unité de temps. Il s'agit alors de déterminer le coût g correspondant.

En introduisant la matrice,

$$P_\mu^\star = \lim_{N \to \infty} \frac{1}{N} \sum_{k=0}^{N-1} P_\mu^k \tag{23.36}$$

on a alors l'expression suivante de $J_\mu(x_0)$:

$$J_\mu(x_0) = P_\mu^\star \cdot g_\mu \tag{23.37}$$

Cette équation a une interprétation naturelle. En effet à partir de (23.36), on voit que l'élément (i, j) de $P_\mu^\star$ correspond à la fraction du temps sur le long terme où le système visite l'état j en partant de l'état initial i. Dès lors, il est clair que $J_\mu(i)$ est la somme de tous les coûts $g[j, \mu(j)]$ ajouté quand le système est dans l'état j, pondérée par la fraction du temps que le système passe dans l'état j lorsque son état initial est i. Posons alors:

$$g[j, \mu(j)] = \delta_j^s \tag{23.38}$$

où δ est la fonction de Kronecker et s est l'état dans lequel les deux tâches sont en synchronisation. On a:

$$J_\mu(i) = p_{i,s}^\star = \lim_{N \to \infty} \frac{1}{N} \sum_{k=0}^{N-1} p_{i,s}^{(k)} \tag{23.39}$$

On sait d'autre part que toute politique stationnaire engendre une chaîne de Markov ayant une seule classe ergodique. Donc l'équation d'optimalité admet une solution unique et le coût λ_μ de cette solution est indépendant de l'état initial. En prenant un état i_0 fixé, on a donc :

$$\lambda_\mu = p^\star_{i_0,s} \tag{23.40}$$

Le coût moyen λ_μ associé à la politique $\pi = \{\mu, \mu, \cdots\}$ apparaît alors comme la fraction du temps sur le long terme que le système passe dans l'état s. Puisque le temps de séjour de cet état est constant et unitaire, λ_μ correspond bien au nombre moyen d'itération de l'algorithme par unité de temps sous la politique $\pi = \{\mu, \mu, \cdots\}$. En utilisant la fonction de coût g définie par (23.38), on peut donc déterminer, dans une situation de charge donnée, la fonction d'allocation qui minimise la durée de l'application.
La politique optimale est adaptative. En effet, lorsque les charges sont équilibrées, la politique optimale est l'allocation à la source pour les deux tâches. Par contre, dans le cas contaire, le message de la tâche sur la machine la plus chargée est là encore alloué à la source, mais le message de l'autre tâche est alloué au troisième serveur de données.

23.5 Conclusion

Nous avons abordé dans cet article quelques exemples de problèmes de gestion de ressources d'une part permettant de montrer comment effectuer une modélisation dynamique et stochastique des phénomènes (chaînes de Markov) et d'autre part de montrer les techniques de résolution des problèmes de commande optimale qui résultent de cette modélisation (équation de Bellman, processus de décision Markoviens).

Le premier exemple développé concernait une simplification de la politique d'utilisation d'un processeur par un système d'exploitation multi-tâches. Ce modèle simple peut permettre de prédire le temps d'exécution de processus donnés sur une machine dont le comportement aléatoire est estimé et prédit. Des algorithmes de placement de processus utilisent alors cette prédiction du temps d'exécution pour bien utiliser un parc de machines [11].

Les deux exemples suivants concernent le routage optimal dans un système à deux serveurs et l'acheminement optimal dans un système à commutation de circuit. Ce sont des problèmes de base dans les réseaux de communication. Ces deux exemples sont vraiment élémentaires et ils montrent très bien la complexité des phénomènes étudiés. Ils sont cependant riches d'enseignements sur les politiques de routage et sur les "intuitions" que l'on pourrait avoir en ce domaine. En particulier, des politiques plus complexes, pour de vrais réseaux (de grande taille), peuvent être extrapolées à partir de ces résultats.

Dans le dernier exemple, nous avons étudié la modélisation de l'allocation dynamique de messages dans une machine distribuée. Ce problème permet de déterminer des politiques de stockage en fonction de l'état des machines. Avec l'utilisation des réseaux haut débit, il devient intéressant de stocker des données sur la mémoire d'une machine distante à moindre coût. Les résultats de cette étude permettent d'obtenir des politiques optimales de stockage des messages (et d'utilisation de la mémoire) pour des applications parallèles à grain fin.

Il est évident que la recherche d'une performance très pointue des systèmes distribués (machines parallèles, réseaux de communication, etc) passe par une modélisation fine des phénomènes étudiés (systèmes à événements discrets, systèmes stochastiques, etc). Le point critique reste le passage à des systèmes de très grande taille. Une extension de ces résultats est alors nécessaire et le recours à des techniques de simulations événementielles quasiment indispensable.

Bibliographie

[1] D. Artiges. Optimal routing into two heterogeneous service stations with delayed information. *IEEE Tran. Aut. Control*, 40, 1995.

[2] D. P. Bertsekas. Dynamic programming, deterministic and stochastic models. *Prentice-Hall*, 1987.

[3] A. Cobham. Priority assignment in waiting-line problem. *Opns. Res.*, 2, 1954.

[4] E. G. Coffman and P. J. Denning. Operating systems theory. *Prentice-Hall series in automatic computation*, 1973.

[5] R. Conway and W. M. L. Miller. Theory of scheduling. *Addison-Wesley, Reading, Mass.*, 1967.

[6] A. Ephremides, P. Varaiya, and J. Walrand. A simple dynamic routing problem. *IEEE Trans. Aut. Control*, 25, 1980.

[7] J. M. Garcia, B. Gopinath, and P. Varaiya. Routing traffic in telephone networks. *20th IEEE Conference on Decision and Control, San Diego (USA)*, décembre 1981.

[8] B. Hajek. Optimal control of two interacting service stations. *IEEE Trans. Aut. Control*, 29, 1984.

[9] L. Kleinrock. Queueing systems, volume 1 : Theory. *Wiley-Interscience*, 1975.

[10] J. D. Little. A proof for the queueing formula $l = \lambda \cdot w$. *Opns. Res. 9*, 3, 1961.

[11] T. Monteil and J. Garcia. Task allocation strategies on workstations using processor load prediction. *PDPTA'97 International conference, Las Vegas*, 1997.

[12] R. R. Muntz. Wainting time distribution for round-robin queueing systems. *Proc. Symp. Comp. Commun., Networks, Teletraffic, Microwave Research Institute, Polytechnic Institute of Brooklyn*, 1972.

[13] S. M. Ross. Applied probability models with optimization applications. *Holden-Day*, 1970.

[14] M. Sakata, S. Noguchi, and J. Oizumi. Analysis of a processor-shared queueing model for time-sharing systems. *Proc. 2^{nd} Hawaii Int'l Conf. Sys. Sci., Univ. of Hawaii, Honolulu*, 1969.

[15] L. E. Schrage. Some queueing models for a time-shared facility. *PhD. thesis, Cornell University*, 1966.

[16] H. C. Tijms. Stochastic models, an algorithmic approach. *Wiley series in probability and mathematical statistics*, 1994.

Chapitre 24

Optimisation des communications et régulation de charge pour la résolution par méthode directe de grands systèmes linéaires creux

P. AMESTOY (ENSEEIHT-IRIT, Toulouse),
F. DESPREZ (LIP INRIA Rhône-Alpes, Lyon),
P. RAMET, J. ROMAN (LaBRI, Bordeaux)

La résolution de systèmes linéaires creux est une brique de base pour traiter numériquement de nombreux problèmes de calcul scientifique. Ces systèmes, qui apparaissent en particulier dans le cadre de la discrétisation d'équations aux dérivées partielles par éléments ou volumes finis, sont de très grande taille pour les applications en vraie grandeur. Le parallélisme est alors une technique incontournable pour résoudre ces très grands systèmes. Dans ce contexte, les méthodes directes jouent un rôle très important car elles sont générales et surtout robustes numériquement ; il existe en effet de nombreux cas où une factorisation numérique s'avère être le seul recours pour une résolution raisonnable du système.

Pour obtenir de bonnes performances globalement, il est obligatoire d'utiliser systématiquement des noyaux de calcul performants implémentés au plus bas niveau pour exploiter au mieux les effets de caches. C'est le cas des primitives de calcul BLAS avec un maximum d'efficacité pour celles de niveau 3 dans le cadre de calculs matriciels denses par bloc ; il faut noter cependant qu'il est important de bien gérer la taille des blocs pour atteindre des vitesses de calcul maximales.

Cet état de fait a un impact considérable sur l'algorithmique des solveurs, sur le prétraitement des données à réaliser et enfin sur l'étude des problèmes liés à la gestion du creux des matrices car tout doit être pensé en termes de blocs : les structures de données seront par bloc et les calculs se feront par bloc.

Un des points essentiels propre aux méthodes directes sur les matrices creuses est le problème du *remplissage*. En effet, la matrice factorisée de Cholesky L est plus pleine que la matrice initiale A de part la création de nouveaux termes non nuls au cours du processus de factorisation. Un outil fondamental pour étudier ce remplissage est le modèle de graphe associé à l'élimination de Gauss. Celui-ci étant directement lié à l'ordre d'élimination des inconnues, il s'agit donc de trouver une renumérotation de ces inconnues, c'est à dire une renumérotation des sommets du graphe associé à la matrice initiale qui minimise le remplissage et en même temps le nombre d'opérations à faire effectivement pour factoriser.

L'arbre d'élimination associé à la numérotation joue un rôle fondamental pour la parallélisation des solveurs creux directs car il indique les dépendances dans les calculs : il ne peut y avoir de dépendances (et donc une séquentialité dans les calculs) entre deux éliminations d'inconnues que si les sommets associés sont sur une même branche de l'arbre d'élimination. Dans un cadre parallèle, le but est donc de trouver une renumérotation des sommets qui *minimise le remplissage* et qui *maximise l'indépendance dans les calculs*, c'est à dire conduisant à des arbres d'élimination *large et de faible hauteur*. Les renumérotations performantes sont celles basées sur les techniques de "degré minimum" et de "dissections emboîtées".

Une autre étape fondamentale est celle dite de *factorisation symbolique* qui va permettre de calculer algorithmiquement la structure creuse de la matrice L à partir de celle de A. La résolution générique d'un système linéaire creux sera donc constituée de 4 étapes:

1. renumérotation des sommets du graphe initial,
2. factorisation symbolique pour construire la structure creuse de la matrice factorisée,
3. factorisation numérique sur la structure précédente, les seuls termes non nuls étant ceux de la matrice initiale,
4. résolution des systèmes triangulaires creux par descente-remontée.

Comme il a été dit plus haut, il est indispensable d'utiliser un algorithme de calcul par bloc pour profiter des performances des primitives de type BLAS. Cela veut dire que la structure de donnée pour la matrice L doit être par bloc, et donc que l'algorithme de factorisation symbolique se fera aussi au sens des blocs. Cette notion de bloc dense de coefficients dans la matrice creuse correspond à la notion de *supernode* ; le but est de rechercher dans la matrice factorisée des ensembles de colonnes ayant la même structure de creux et qui pourront donc être "amalgamées" en une "super variable". Ceci se fera de manière couplée avec la factorisation symbolique.

Le cadre parallèle étant celui des machines MIMD à mémoire distribuée, une autre phase de prétraitement des données va précéder l'exécution parallèle du solveur numérique. Avec un modèle de programmation de type SPMD et l'utilisation de bibliothèque de communication de type MPI, ce prétraitement a pour rôle de calculer algorithmiquement un bon partitionnement et une bonne distribution des données ; ceci se fera à partir de la factorisation symbolique et par bloc. Le prétraitement va donc, en plus de la phase de renumérotation, calculer de manière statique une régulation équilibrée pour le solveur (critère d'équilibrage de charge et prise en compte des contraintes de précédence) ; on utilisera ici la structure de l'arbre d'élimination. Le but sera d'exploiter les différents niveaux de parallélisme dans les calculs, à savoir *un premier niveau induit par le creux* présent dans les calculs entre blocs indépendants, et *un deuxième niveau induit par les calculs dans les blocs pleins*. À noter que l'on peut aussi optimiser la localité des communications pour éviter des contentions dans le réseau du calculateur.

Les solveurs numériques eux-mêmes sont basés sur l'approche supernodale (stratégies fan-out, fan-in, fan-both) ou sur l'approche multifrontale. La gestion de l'irrégularité des calculs nécessite la mise en œuvre dans la programmation de techniques assez fines : gestion de l'indéterminisme d'exécution (threads ou régulation algorithmique intégrée dans le code SPMD), utilisation de communications asynchrones avec gestion du recouvrement calcul/communication, ...

Le but des sections qui suivent est de faire le point sur les aspects liés aux phases de renumérotation et de distribution des données, puis d'aborder d'autres aspects de nature différente ; il s'agira tout d'abord de la mise en place d'une nécessaire régulation dynamique lorsque l'on fait du pivotage par exemple dans le cas non-symétrique, puis de la gestion du grain pour optimiser les recouvrements calcul/communication.

24.1 Prétraitement statique pour obtenir une bonne régulation

24.1.1 L'algorithme du degré minimum

L'algorithme du degré minimum est un des algorithmes locaux de réordonnancement de matrices creuses symétriques les plus classiquement utilisés. Cet algorithme [39, 50] trouve son nom dans son interprétation en tant que manipulation du graphe associé à la matrice creuse; en effet le degré minimum est un algorithme local qui à chaque étape choisit comme pivot le nœud de degré minimum dans le graphe associé à la matrice réduite. Cet algorithme pose en fait de nombreux problèmes d'implémentation et a fait l'objet de nombreux travaux répertoriés dans l'article [1]. Les principales étapes de l'algorithme de degré minimum sont décrites ci-après.

Pour $k = 1$ **au** nombre de nœuds **faire**
1/ Choisir un pivot p non encore sélectionné et de degré minimum
2/ Construire la liste d'adjacence de p
et mettre à jour la liste d'adjacence des $i \in Adjacence(p)$
3/ Calculer le degré modifié des $i \in Adjacence(p)$
Fin Pour

Sur de nombreux problèmes, l'étape 3 de l'algorithme est l'étape la plus coûteuse. Amestoy, Davis et Duff [1] ont proposé de remplacer le calcul du degré par une approximation, pouvant être calculée durant l'étape 2 sans sur-coût significatif. La précision de l'approximation conduit à un algorithme (*AMD: Approximate Minimum Degree*) qui s'avère de qualité comparable aux algorithmes calculant un degré exact. Pour illustrer l'influence sur le temps de calcul, AMD a été comparé au Multiple Minimum Degree (MMD, voir [38]) qui fait référence en la matière. Les matrices de test sont issues de [18] et du jeu de matrices de Davis (Université de Floride).

matrice	ordre	non-zéros	AMD	MMD
BCSSTK32	44609	985046	3.3	6.4
WANG3	26064	75552	3.6	5.3
FINAN512	74752	261120	5.6	292.6
ORANI678	2529	85426	8.2	168.6
BBMAT	38744	1274141	16.5	111.7
PSMIGR_1	3140	410781	17.0	270.9
LHR71	70304	1199704	21.8	130.2

Tableau 24.1 : Comparaison de AMD et MMD (SPARC 10, temps en secondes).

D'autre part, dans le cas de matrices possédant un bloc de lignes pleines ou quasiment pleines, l'étape 2 de l'algorithme du degré minimum peut induire une mise à jour aussi coûteuse qu'inutile de la liste d'adjacence de variables associées au ligne quasiment pleines (voir par exemple [27]).

Matrice	Ordre	Non-zéros	AMD		AMDD	
			non-zéros dans $L+L^T$ $(\times 10^6)$	temps sec.	Non-zéros dans $L+L^T$ $(\times 10^6)$	temps sec.
Gupta 1	31802	2164210	4.080	483	4.066	9
Gupta 2	62064	4248286	11.717	1414	11.911	8
Gupta 3	16783	9323427	11.414	121	11.869	28

Tableau 24.2 : Etude comparative de AMD et AMDD. Temps obtenus sur un processeur HP (PA-RISC 150 MHertz).

Amestoy, Davis et Duff ont exploré deux voies pour traiter ce problème. Une première solution consiste à pré-traiter la matrice pour supprimer les lignes quasiment denses. AMD est appliqué sur la matrice réduite, et des heuristiques pour réordonnancer les lignes quasiment denses peuvent alors être appliquées. Une autre solution (l'algorithme *AMDD*) modifie l'algorithme de degré minimum afin de détecter les lignes quasiment pleines dans la matrice réduite à chaque étape de l'algorithme. A l'étape 2 de l'algorithme du degré minimum, la liste d'adjacence des lignes sélectionnées est partiellement mise à jour afin de permettre un redémarrage récursif de AMD sur les lignes sélectionnées. Un gain significatif en temps peut être obtenu avec les deux approches. Ceci est illustré par les performances obtenues de l'algorithme AMDD (voir Table 24.2) sur des matrices provenant de [27].

24.1.2 Méthodes de Dissections Emboîtées

La méthode de dissections emboîtées [23] consiste à déterminer un ensemble de sommets S qui sépare le graphe en deux parties A and B non voisines, numéroter S avec les plus grands numéros disponibles, et recommencer récursivement sur les sous-graphes A et B jusqu'à ce que leurs tailles deviennent inférieures à une valeur limite. Cette numérotation garantit qu'aucun terme non-nul ne peut apparaître lors de la factorisation entre des inconnues de A et des inconnues de B.

À la différence des méthodes de degré minimum, il est possible de prédire, au moins en ordre de grandeur, la qualité des renumérotations obtenues par dissections emboîtées pour les graphes usuels. On utilise pour cela les théorèmes de séparation, définis par Lipton, Rose, et Tarjan, qui donnent pour les familles de graphes les plus courantes (graphes planaires, tridimensionnels, graphes à genre borné, ...) des bornes sur la taille des séparateurs sommets et l'équilibrage des sous-parties obtenues [24, 36, 25] ; on a ainsi pu montrer analytiquement que la méthode de dissection emboîtées est proche de l'optimum pour les graphes à degré borné. Malheureusement, souvent ces théorèmes n'utilisent pas de preuves constructives, ni ne proposent d'algorithmes permettant d'atteindre les bornes prouvées.

De nombreuses techniques ont été proposées pour trouver des séparateurs sommets de petite taille qui équilibrent les sous-graphes autant que possible. Une première approche consiste à construire les séparateurs sommets à partir de séparateurs arêtes [34]. Pour trouver le plus petit séparateur sommet constitué d'extrémités d'arêtes de la coupe, on utilise classiquement la décomposition sous forme triangulaire par blocs ("*block triangular form*") du graphe biparti associé à la coupe, qui nécessite au préalable le calcul d'un couplage parfait de ce graphe. D'autres heuristiques de calcul de séparateurs sommets ont été proposées, basées sur la géométrie ou les méthodes spectrales [26], éventuellement dans un contexte multi-niveaux [32].

Jusqu'à récemment, les comparaisons effectuées entre les méthodes de degré minimum et de dissections emboîtées penchaient en faveur des premières. Cependant, le développement d'algorithmes efficaces de partition de graphes a conduit à inverser cette tendance. On a obtenu sur des problèmes de mécanique des structures, pour les méthodes de dissection emboîtées, une réduction moyenne du nombre d'opérations de 15 pour cent pour la factorisation de Cholesky par rapport aux méthodes de degré minimum [9, 47] ; de plus, ces méthodes résistent aux cas pathologiques mis en évidence pour les méthodes de degré minimum, notamment dans le domaine de la modélisation financière [10]. Enfin, le degré minimum donne généralement des arbres d'élimination de bien moins bonne qualité que ceux produits par la dissection emboîtée. En fait, la bonne approche consiste à mixer les deux méthodes pour profiter des avantages des deux numérotations : ainsi, dans la plupart des distributions logicielles proposées, seuls les premiers niveaux sont calculés par dissections emboîtées, les sous-graphes obtenus étant numérotés par degré minimum en dessous d'une certaine taille [32, 35, 45]. Les logiciels de référence dans ce cadre sont METIS [35] et SCOTCH [45, 46, 41]. Le tableau 24.3 compare ces deux logiciels avec MMD; NNZ est le nombre de termes non nuls dans L et OPC est le nombre d'opérations à faire pour factoriser. Le seuil à partir duquel la dissection est arrêtée est de 30 sommets. Les figures 24.1 et 24.2 illustrent la variation de NNZ, OPC et de la hauteur moyenne de l'arbre d'élimination pour les graphes tests 598A et BCSSTK32 quand on augmente la valeur en nombre de sommets du seuil à partir duquel on bascule de la dissection emboîtée au degré minimum.

Graphe	Scotch 3.2			O-MeTiS 2.0		
	NNZ	OPC	Temps	NNZ	OPC	Temps
144	**5.04302e+07**	**6.30637e+10**	54.18	5.14774e+07	6.63160e+10	32.07
3ELT	9.34520e+04	2.91077e+06	0.44	9.78490e+04	3.43192e+06	0.27
598A	2.80753e+07	2.15109e+10	32.85	**2.80390e+07**	**2.10550e+10**	22.69
AUTO	**2.42459e+08**	**5.45092e+11**	673.78	–	–	–
BCSSTK29	1.95601e+06	4.98316e+08	5.05	2.00891e+06	5.22292e+08	2.88
BCSSTK30	4.70394e+06	1.35818e+09	16.79	4.91682e+06	1.48445e+09	10.26
BCSSTK31	**5.22795e+06**	1.84196e+09	12.64	5.27408e+06	**1.72527e+09**	7.73
BCSSTK32	6.31493e+06	1.69557e+09	20.12	6.93198e+06	2.23416e+09	12.20
BODY	1.21614e+06	1.08353e+08	8.45	1.10336e+06	**7.92548e+07**	4.44
BRACK2	7.38112e+06	**2.54711e+09**	15.07	**7.20380e+06**	2.56784e+09	9.19
BUMP	**2.67353e+05**	**1.48797e+07**	1.02	2.73569e+05	1.55700e+07	0.63
M14B	**6.77465e+07**	**7.08264e+10**	102.05	6.89223e+07	7.18790e+10	52.67
PWT	**1.43140e+06**	**1.22120e+08**	6.05	1.53126e+06	1.40441e+08	3.76
ROTOR	**1.76431e+07**	**1.11101e+10**	28.88	1.81358e+07	1.16120e+10	18.36
TOOTH	**1.31559e+07**	**9.02613e+09**	19.48	1.32536e+07	9.44619e+09	12.34

Graphe	MMD		
	NNZ	OPC	Temps
144	9.81030e+07	2.62934e+11	28.30
3ELT	**8.98200e+04**	**2.84402e+06**	0.06
598A	4.63319e+07	6.43375e+10	19.96
AUTO	5.16945e+08	2.41920e+12	139.14
BCSSTK29	**1.69480e+06**	**3.930592+08**	0.76
BCSSTK30	**3.84344e+06**	**9.28353e+08**	1.58
BCSSTK31	5.30825e+06	2.55099e+09	1.96
BCSSTK32	**5.24635e+06**	**1.10873e+09**	2.14
BODY	**1.00908e+06**	8.93073e+07	1.25
BRACK2	7.30755e+06	2.99122e+09	3.90
BUMP	2.81482e+05	1.76288e+07	0.12
M14B	1.22359e+08	2.56226e+11	44.07
PWT	1.59060e+06	1.75944e+08	0.70
ROTOR	2.60982e+07	2.86029e+10	11.27
TOOTH	1.56226e+07	1.65121e+10	5.06

Tableau 24.3 : Comparaisons entre Scotch 3.2, O-MeTiS 2.0 (arrêt de la dissection en dessous de 30 sommets) et MMD; les meilleures valeurs sont en gras et les temps sont mesurés pour un processeur R10000 190 MHertz.

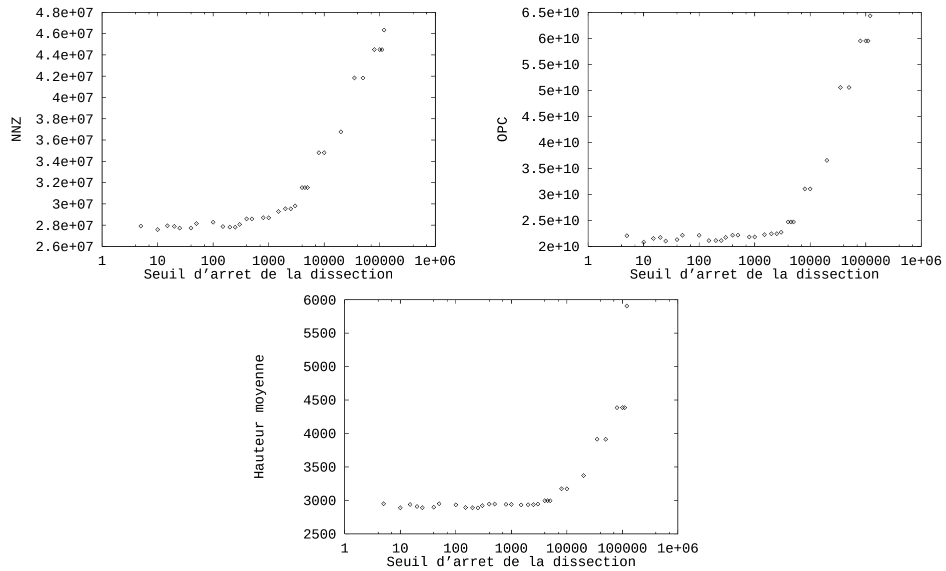


Figure 24.1 : 598A, $|V| = 110971$, $|E| = 741934$

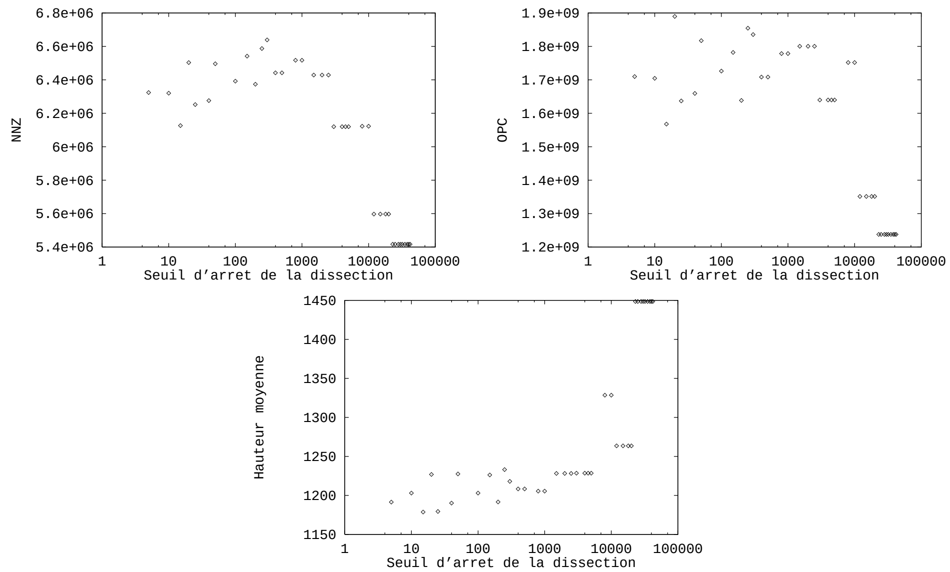


Figure 24.2 : BCSSTK32, $|V| = 44609$, $|E| = 985046$

Un couplage optimisé entre la dissection emboîtée produite par SCOTCH et le degré minimum produit par AMD fait l'objet d'une collaboration entre l'ENSEEIHT et le LaBRI.

24.1.3 Distribution des données et solveurs parallèles

Les solveurs séquentiels ou parallèles utilisés aujourd'hui sont nécessairement par bloc pour exploiter la puissance locale des processeurs via les primitives BLAS3.

Il y a essentiellement deux types d'approche algorithmique qui sont l'approche *supernodale*, qui provient directement des algorithmes d'élimination par colonne réécrits par blocs et avec les variantes que l'on appelle *fan-in* ou *fan-out* [5, 6, 7, 8, 31, 42], et l'approche *multifrontale* [19, 3, 14, 31]. En ce qui concerne les méthodes supernodales, l'approche fan-in ou "left-looking" consiste à répercuter sur le bloc colonne courant les modifications des blocs colonnes de gauche concernés, puis à calculer la valeur finale du bloc colonne courant ; de manière symétrique, l'approche fan-out ou "right-looking" consiste à calculer définitivement le bloc colonne courant et à répercuter ses contributions sur les blocs colonnes de droite concernés. En parallèle, l'approche fan-in est plus performante car elle permet de limiter la masse de communication.

A partir de la structure de données par bloc calculée par la factorisation symbolique [12], la phase de prétraitement va donc calculer une bonne distribution des données sur les processeurs. On utilisera l'arbre d'élimination pour distribuer de manière équilibrée les blocs indépendants qui sont les plus bas dans l'arbre, puis on découpera les blocs situés dans les niveaux les plus hauts et qui représentent de manière inhérente la plus grande masse de calcul. Les sous blocs seront alors distribués de manière bloc cyclique pour utiliser le parallélisme contenu dans les calculs en dense. On construit ainsi de manière statique une bonne régulation des calculs [22, 30, 40, 20]. Cette technique a été utilisée selon un schéma de distribution 1D ou l'on distribue de manière élémentaire des blocs colonnes entiers. Schreiber [49] a montré que ce type de schéma limite la scalabilité des solveurs et qu'il faut en fait avoir un schéma de distribution 2D. C'est dans ce dernier cadre que l'on a atteint les meilleures performances connues à ce jour. Avec une approche supernodale, Rotherg et Schreiber [48] ont obtenu 2.8 GFlops sur un Paragon à 196 noeuds pour une matrice de taille 64000 correspondant à une grille 3D d'éléments finis. Par une approche multifrontale, Gupta, Karypis et Kumar [28, 29] ont obtenu entre 4.5 et 15.7 Gflops sur un T3D à 1024 noeuds sur des matrices de tailles variant entre 3136 et 42875. Dans les deux cas, les graphes associés aux matrices ont été renumérotés par dissections emboîtées couplées avec du degré minimum.

24.2 Régulation dynamique pour la factorisation des matrices creuses non-symétriques

La factorisation de matrices creuses peut être modélisée par un arbre d'élimination qui exprime les dépendances entre l'élimination des colonnes de la matrice. Dans le cadre de matrices symétriques définies positives la structure de cet arbre peut être calculée avant l'étape de factorisation numérique. Par contre, lors de la factorisation de matrices creuses non-symétriques, uniquement une estimation de la structure de cet arbre peut être connue à priori. En effet, le pivotage numérique peut modifier de façon significative la structure estimée.

Dans cette section nous décrirons l'impact du pivotage numérique aussi bien sur la structure de l'arbre d'élimination que sur les choix algorithmiques pour la parallélisation sur machines à mémoire distribuée de la factorisation. Ce travail (P. Amestoy, I. Duff et J.Y. L'Excellent) est développé dans le cadre du projet PARASOL EU ESPRIT IV LTR. Il s'appuie d'une part sur l'expérience acquise sur machine à mémoire partagée et virtuellement partagée ([2, 4], code MA41 dans la bibliothèque Harwell), d'autre part sur le développement à l'ENSEEIHT d'une version initiale pour machines à mémoire distribuée [21].

24.2.1 Arbre d'élimination

L'arbre d'élimination (voir par exemple [37]) exprime les connections/dépendances entre les différentes étapes de la factorisation. Il exprime donc le parallélisme induit par la structure creuse de la matrice.

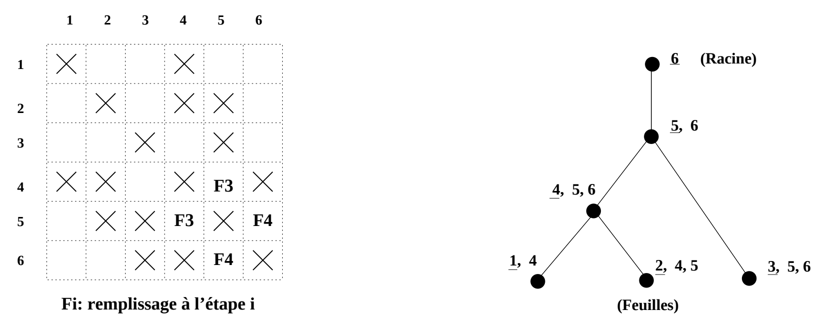


Figure 24.3 : Exemple de construction d'un arbre d'élimination.

L'analyse de la structure de la matrice des facteurs ($\mathbf{A}_F$) permet de construire l'arbre d'élimination. Si l'on suppose que les pivots choisis sur la diagonale (et dans l'ordre) constituent une séquence de pivots numériquement stable, alors il est effectivement possible de construire l'arbre d'élimination avant l'étape de factorisation numérique. Chaque colonne correspond à un nœud du graphe. Chaque nœud k de l'arbre correspond à la factorisation d'une matrice frontale dont la structure est celle de la colonne k de $\mathbf{A}_F$. S'il n'y a pas de non-zéro sur la partie gauche de la diagonale de la ligne j alors le nœud j est une **feuille**. Le nœud j est père de k dans l'arbre si et seulement si j est l'indice de ligne du premier élément non-nul sous diagonale dans la colonne k.

Pour mieux exploiter l'architecture des processeurs actuels (vectoriels ou scalaires) il est fondamental de détecter/exploiter toutes les sous-structures régulières manipulées par l'algorithme. On peut observer sur notre exemple, que le bloc d'indices

$(3, 5, 6)$ constitue une structure pleine de la matrice des facteurs. Extraire les structures pleines de la matrices des facteurs peut aussi être vu comme une amalgamation des nœuds de l'arbre d'élimination. Ce pré-traitement (avant factorisation) du graphe permet effectivement d'accélérer de façon significative l'étape de factorisation. Chaque nœud de l'arbre amalgamé correspond ainsi à l'élimination d'un bloc de colonnes de la matrice initiale (voir nœud $(3, 5, 6)$ de la Figure 24.4). La parallélisation de la factorisation des matrices frontales pleines de l'arbre amalgamé offre un degré de parallélisme supplémentaires qui peut être combiné au parallélisme de l'arbre.

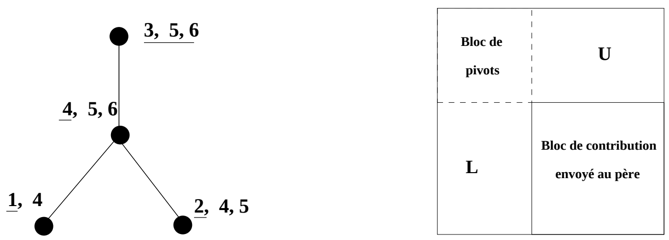


Figure 24.4 : Arbre amalgamé et structure générale d'une matrice frontale.

24.2.2 Arbre d'élimination et pivotage numérique

La structure de l'arbre d'élimination est en fait basée sur une estimation de la structure de la matrice des facteurs obtenue en choisissant les pivots sur la diagonale.

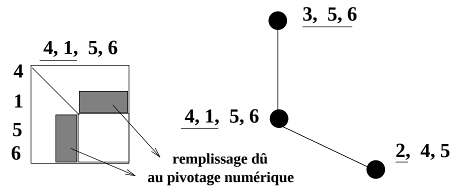


Figure 24.5 : Influence du pivotage numérique sur la structure de l'arbre.

La stabilité de la factorisation d'une matrice creuse non-symétrique est basée sur un contrôle de la taille relative du pivot (pivotage partiel avec critère de seuil, [17]). Si par exemple on suppose que $|a_{1,1}| << |a_{1,4}|$ dans notre matrice de test, alors le pivot $a_{1,1}$ ne peut plus être éliminé lors du calcul associé au nœud 1. Les modifications du graphe initial (voir Figure 24.5) conduisent à une modification de la taille des facteurs (les valeurs (1,5) et (1,6) en plus), de la structure de l'arbre, de la taille et du nombre de pivots des matrices frontales (élimination des pivots 4 et 1 au niveau du nœud 4). Il est à noter que les deux niveaux de parallélisme de la méthode (arbre et traitement d'un nœud) sont ainsi perturbés par le pivotage numérique.

24.2.3 Parallélisation sur machine à mémoire distribuée

Dans la première étape de l'algorithme, l'arbre d'élimination est distribué sur les processeurs de la machine cible. Cette étape conduit au regroupement de nœuds d'un sous-arbre sur le même processeur cible (voir Figure 24.6). Le parallélisme induit par l'arbre sera dit de *niveau 1*.

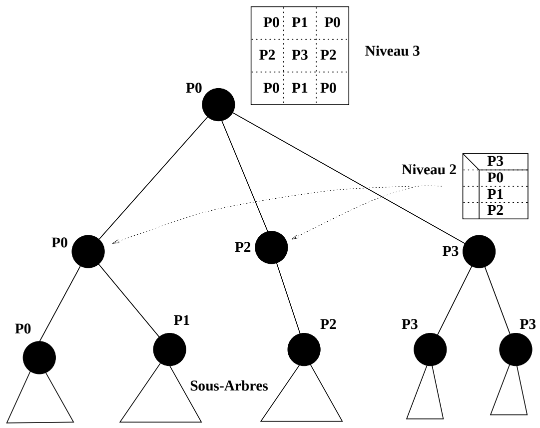


Figure 24.6 : Niveaux de parallélisme exploités durant la factorisation.

Pour pouvoir prendre en compte dynamiquement les problèmes liés au pivotage numérique deux types d'algorithmes de parallélisation des matrices frontales ont été développés:

1. *Niveau 2* : Partitionnement 1D des lignes associées au bloc de contribution de la matrice frontale
2. *Niveau 3* : Partitionnement 2D de la factorisation des matrices frontales.

Lors de l'algorithme de niveau 2, le processus maître associé au traitement du nœud est en charge uniquement de l'élimination du bloc de lignes des pivots. Il distribue alors dynamiquement des tâches par un découpage 1D des lignes associées au bloc de contribution. L'assemblage de la matrice frontale est alors automatiquement distribué sur les processus participant au travail. Le pivotage numérique est effectué localement par le processus maître qui peut factoriser le bloc de pivots indépendamment de la disponibilité des autres processus. Les matrices frontales candidates au niveau 2 sont typiquement celles qui possèdent un bloc de contribution de taille importante par rapport à la taille de la matrice frontale.

Le parallélisme de niveau 3 doit au contraire permettre de traiter les matrices frontales ayant un bloc de contribution de taille relativement faible par rapport à la taille de la matrice frontale (par exemple la racine). Le bloc de pivots estimé à l'étape d'analyse peut être assemblé sur la base d'informations statiques. Une correction (dynamique) sera cependant nécessaire pour prendre en compte les variables non-éliminées envoyées par les nœuds fils.

		Arbre complet			Arbre sans la racine	
Matrice	Ordre	Niv. 1	Niv. 1+2	Niv. 1+2+3	Niv. 1	Niv. 1+2
GOODWIN	7320	1.73	1.88	2.09	1.83	2.01
K15	3948	1.98	4.58	10.0	2.35	11.1
WANG3	26064	1.38	3.08	13.8	1.60	11.9
WANG4	26068	1.67	2.78	20.3	2.45	18.2

Tableau 24.4 : Etude du parallélisme théorique (réordonnancement basé sur AMD).

L'apport théorique de chaque niveau de parallélisme est indiqué dans la Table 24.4. On peut noter le faible degré de parallélisme de l'arbre. Ceci doit pouvoir s'améliorer par l'utilisation d'un algorithme de réordonnancement combinant dissections emboîtées et degré minimum. Cet axe de recherche fait l'objet d'une collaboration en cours avec F. Pellegrini et J. Roman (LaBRI, Bordeaux). Les niveaux 2 et 3 offrent cependant un bon complément au parallélisme de l'arbre même s'il est clair que notre modèle simplifié ne donne qu'une borne supérieure du speed-up réel. Il sera notamment important de contrôler la taille des blocs lors des partitionnements 1D ou 2D des matrices frontales.

La version α du code distribué est développée en FORTRAN90 et s'appuie sur la couche MPI. L'analyse de performance des niveaux 2 et 3 est actuellement en cours.

24.3 Calcul adaptatif du grain dans les recouvrements calcul/communication

24.3.1 Introduction, problématique et travaux connexes

L'utilisation des machines parallèles à mémoire distribuée apporte un gain important en performances et en taille mémoire mais amène en contre partie un surcoût en communications. Afin d'obtenir des programmes performants et extensibles, il convient de masquer ce surcoût. Plusieurs solutions existent. La première consiste en un choix judicieux de la distribution des données qui réduira au maximum le nombre et la taille des communications. De plus, si les dépendances le permettent, on essaiera d'initialiser les communications au plus tôt et de manière asynchrone : pendant l'exécution de ces communications, on effectuera d'autres calculs, sans rapport avec l'échange de données. Des algorithmes ont été proposés pour recouvrir les communications en particulier dans le cas des transformées de Fourier [51]. Dans [43], la taille optimale des paquets est calculée pour des nids de boucles `DOACROSS` généraux. De nombreux travaux autour des compilateurs data-parallèles traitent ce problème [11, 44, 33].

Quand les calculs sont irréguliers, on fait souvent appel à des techniques multi-threads. Des threads de calculs sont lancés en parallèle avec des threads de communication. S'ils résolvent le problème des recouvrements dans le cas de calculs et de communications indépendants, ils ne traitent pas le problème du découpage et du calcul du grain dans le cas où les calculs dépendent des communications (et vice-versa). Ils sont malgré tout une solution intéressante et relativement transparente pour l'utilisation de communications asynchrones.

Notre étude concerne donc le cas où calculs et communications sont dépendants entre eux comme dans le schéma de la Figure 24.7.

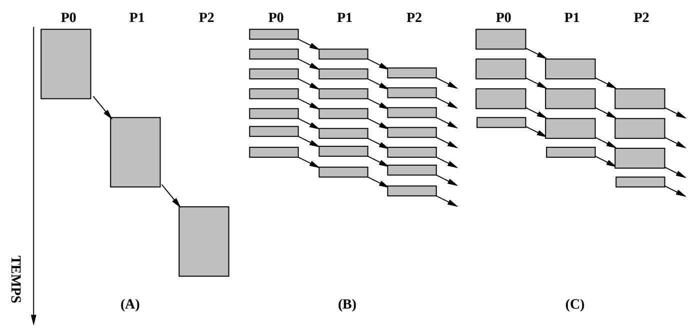


Figure 24.7 : Recouvrement des communications grâce à un pipeline

Il y a une séquentialité dans l'exécution (A). Le processeur P1 doit attendre que le processeur P0 ait fini de calculer pour recevoir les résultats du calcul et commencer à travailler. Une fois qu'il a terminé, il envoie les données résultats au processeur P2 qui peut, à son tour, travailler sur les données reçues. Le temps total est plus important que le temps séquentiel puisque l'on introduit le surcoût des communications. Une première solution consiste à commencer les communications au plus tôt, c'est-à-dire dès qu'un processeur a calculé une donnée. Pour chaque donnée calculée, un envoi est effectué et les processeurs commencent leur travaux dès que possible. On a donc un pipeline à grain fin (B). Cette solution introduit malgré tout un surcoût dû aux temps d'initialisation des communications. Ces temps sont toujours plus importants que le coût d'envoi d'un élément et il en résulte que le temps total peut être encore plus important que le temps séquentiel. Un compromis consiste donc à trouver le grain de calcul (et de communication) qui minimisera le temps total d'exécution. On aura alors un pipeline à grain moyen qui correspond à un schéma de type macro-pipeline (C).

Nous avons développé deux bibliothèques permettant un traitement efficace et portable de ce type de problèmes : la bibliothèque LOCCS[1] [15] qui permet d'effectuer les communications et la gestion de buffers pour les calculs pipelinés et la bibliothèque OPIUM[2] [16] qui permet de calculer le grain de calcul optimal en fonction des paramètres de l'algorithme et les caractéristiques de la machine.

Dans la suite de ce paragraphe, nous allons présenter le calcul de la taille optimale de paquet dans les cas réguliers et "irréguliers".

[1]Low Overhead Communication and Computation Subroutines
[2]Optimal Packet sIze compUtation Methods

24.3.2 Cas régulier

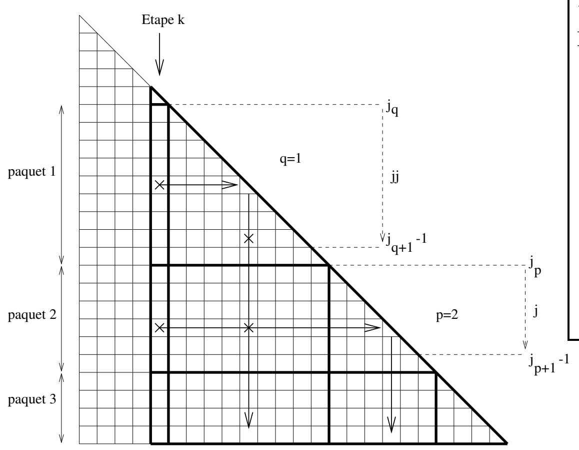
Dans le cas régulier on peut expliciter une formulation du problème qui conduit à un schéma de calcul numérique du grain optimal. Le modèle permet de prendre en compte à la fois la complexité de l'algorithme et les caractéristiques physiques du calculateur. On peut, suivant la complexité de l'algorithme obtenir une solution analytique ou une solution numérique approchée. Notre travail consiste dans une première phase à instrumenter le code par des appels explicites aux primitives OPIUM ; la complexité de l'algorithme est passée comme argument. Il est à noter que, dans le cadre des problèmes d'algèbre linéaire réguliers, les complexités des algorithmes cibles sont généralement connues ou simples à calculer car elles dépendent de la taille des données et non des données elles-mêmes. Pour cela il sera nécessaire d'avoir une analyse du code donnant une estimation fine de la complexité de l'algorithme.

On obtient, sur l'exemple de la factorisation LU, des résultats encourageants [16]. L'efficacité du recouvrement nécéssite cependant une connaissance précise des paramètres de communication.

24.3.3 Calcul adaptatif

Dans le cas où les calculs n'ont pas tous la même taille, nous devons avoir, au sein d'une même étape de pipeline, des paquets de taille variables afin de garantir un équilibre dans les calculs et les communications et ainsi obtenir les meilleures performances. Nous devons donc calculer la suite des tailles de paquets qui conduit au temps d'exécution minimum. D'autres travaux existent sur des sujets proches comme dans [13], où une solution adaptative pour recouvrir les communications est proposée dans le cas du produit matrice-vecteur.

Nous allons maintenant décrire une méthode pour calculer une suite de taille de paquets qui maximise le recouvrement pour l'algorithme de factorisation de Cholesky. Le point de départ est l'algorithme séquentiel colonne classique que l'on modifie pour mettre en évidence le macro-pipeline :



Algorithme modifié par paquet :

Pour $k = 1$ à n faire
  $a_{k,k} = \sqrt{a_{k,k}}$
  Pour $p = 1$ à P_k faire
    Pour $j = j_p$ à $j_{p+1} - 1$ faire
      $a_{j,k} = a_{j,k} / \sqrt{a_{k,k}}$ (BLAS)
    Pour $q = 1$ à $p - 1$ faire
      Pour $jj = j_q$ à $j_{q+1} - 1$ faire
        Pour $j = j_p$ à $j_{p+1} - 1$ faire
          $a_{j,jj} = a_{j,jj} - a_{j,k} * a_{jj,k}$ (BLAS)
    Pour $j = j_p$ à $j_{p+1} - 1$ faire
      Pour $i = j$ à $j_{p+1} - 1$ faire
        $a_{i,j} = a_{i,j} - a_{i,k} * a_{j,k}$ (BLAS)

Figure 24.8 : Factorisation de Cholesky

La redistribution des calculs correspond à une progression “ligne” dans la factorisation. La symétrie conduit à des tailles de paquets variables au cours du pipeline. A chaque étape de la factorisation, on découpe la colonne pivot en paquets de tailles inégales ; c'est cette suite de taille de paquets que l'on cherche à optimiser. Pour l'étape k, il y a P_k paquets de taille ν_p correspondant aux indices de ligne j_p à $j_{p+1}-1$ (Figure 24.8). On calcule les contributions engendrées par le paquet en cours (bloc diagonal) ainsi que le couplage avec les paquets précédents. Il est donc nécessaire de conserver les paquets précédement reçus tant que le pipeline n'est pas terminé.

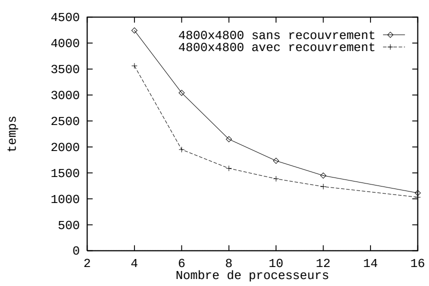


Figure 24.9 : Extensibilité sans BLAS 1

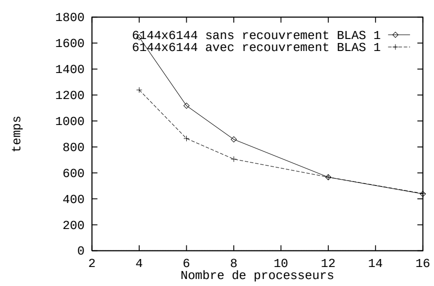


Figure 24.10 : Extensibilité avec BLAS 1

Les figures 24.9 et 24.10 montrent que l'influence du recouvrement devient significative dès que la masse de calcul à traiter par processeur est suffisante (pour de petits macro-pipelines, le nombre de paquets générés est trop faible pour avoir un recouvrement efficace).

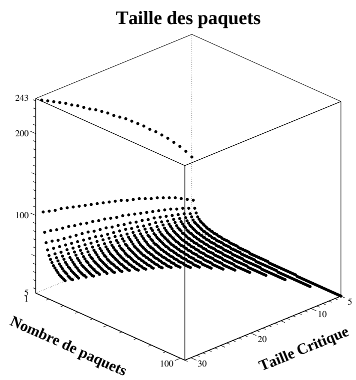


Figure 24.11 : Suite de taille de paquets

On appelle "calcul avant" ($Tcalc_A$) le travail qui doit être effectué sur un paquet avant de communiquer les résultats correspondants. De même, le "calcul après" ($Tcalc_B$) représente le travail qui doit être effectué à l'aide des résultats communiqués. On note $\tau_\times$ le temps élémentaire de mise à jour et $\tau_\div$ le temps élémentaire du calcul de la contribution.

Le processeur émetteur effectue les mises à jour de ses colonnes dans le "calcul avant", de sorte que le "calcul avant" et le "calcul après" soient asymptotiquement équilibrés.

Pour un paquet p de taille ν_p, pour la colonne k, $1 \leq k \leq n$, $1 \leq p \leq P_k$, on a :

$$Tcalc_A(\nu_p) = \frac{\sum_{q=1}^{p-1} \nu_p \nu_q \tau_\times + \frac{\nu_p(\nu_p+1)}{2}\tau_\times}{NProcs} + \nu_p \tau_\div \quad \text{et} \quad Tcalc_B(\nu_p) = \frac{\sum_{q=1}^{p-1} \nu_p \nu_q \tau_\times + \frac{\nu_p(\nu_p+1)}{2}\tau_\times}{NProcs}.$$

En supposant que la taille des paquets est supérieure au nombre de processeurs, et grâce à une *distribution cyclique*, tous les processeurs sont concernés par les contributions.

Quand p croît, la taille du calcul augmente comme $p\nu_p^2$, et donc nécessairement la suite de taille de paquets est décroissante. Pour expliciter le couplage entre les temps de calcul et de communication, qui évoluent de manière opposée, on impose la contrainte : $\forall p,\ Tcalc_A(\nu_p) + Tcomm(\nu_p) = C$, avec C à déterminer (voir proposition 24.3.2). D'autre part, on considère un modèle linéaire de communication : $Tcomm(\nu_p) = \beta + \nu_p \tau$. Sous ces hypothèses, on démontre le résultat suivant :

Proposition 24.3.1 *Si L est la taille du pipeline, la suite de taille de paquets est donnée par :*

$$\left\{ \begin{array}{l} \nu_1(C) = \frac{-b+\sqrt{b^2+4a(C-\beta)}}{2a} \\ \forall p \geq 2, \quad \nu_p(C) = \frac{\sqrt{b^2+4ap(C-\beta)}-\sqrt{b^2+4a(p-1)(C-\beta)}}{2a} \end{array} \right. , \text{ avec } \left\{ \begin{array}{l} a = \frac{\tau_\times}{2NProcs} \\ b = \frac{\tau_\times}{2NProcs} + \tau + \tau_\div \end{array} \right. .$$

De plus le nombre de paquets du pipeline est donné par : $P_k(C) = \frac{L(b+aL)}{C-\beta}$.

L'utilisation du noyau de calcul BLAS impose une taille minimale de paquets Λ. On cherche donc C vérifiant les contraintes : $\begin{cases} \forall p, \nu_p \geq \Lambda, \\ \forall p, Tcalc(\nu_p) \geq Tcomm(\nu_p). \end{cases}$

On démontre le résultat suivant :

Proposition 24.3.2 $C = max(C_1, C_2)$ *avec :*

$$\begin{cases} C_1 = \beta + 2aL\Lambda + b\Lambda - a\Lambda^2 \geq 0 \quad \textit{obtenue avec la } 1^{i\grave{e}re} \textit{ contrainte}, \\ C_2 = \beta + \frac{2a\beta - 2b\tau + 4\tau^2 + 2\tau\sqrt{b^2 - 4b\tau + 4a\beta + 4\tau^2}}{2a} \quad \textit{obtenue avec la } 2^{i\grave{e}me} \textit{ contrainte}. \end{cases}$$

La Figure 24.11 illustre l'évolution des tailles de paquets, pour un macro-pipeline de taille 1024, en fonction de la taille critique imposée Λ.

La prise en compte des optimisations de type BLAS doit être poursuivie pour les BLAS de niveau 2 et 3, l'objectif est ici d'intégrer ces techniques de recouvrement dans des algorithmes par blocs comme ceux proposés dans ScaLAPACK, et de comparer les 2 versions.

L'utilisation de telles optimisations permet d'avoir de bons gains en performances mais pose de nombreux problèmes. Des problèmes théoriques puisqu'il faut être capable de calculer le grain optimal de calcul et communication. Et des problèmes logiciels puisque l'intégration dans une application réelle doit être la plus transparente, la plus portable et, bien entendu, la plus performante possible. L'optimisation d'algorithmes traitant des structures de données irrégulières complique ces calculs.

La suite de ce travail consiste tout d'abord à généraliser l'approche aux algorithmes par blocs et à traiter la factorisation de Cholesky pour des matrices creuses quelconques; dans ce contexte plus général d'irrégularité, l'objectif sera de calculer, à l'aide d'un *schéma adaptatif*, la suite optimale de taille de paquets. On intégrera ensuite ces techniques dans un schéma de compilation automatique pour le langage HPF (plateforme Adaptor). De plus, nous souhaitons par la suite tenir compte des hiérarchies mémoires et de l'utilisation de routines de calcul performantes comme les BLAS. Du point de vue des bibliothèques LOCCS, nous pensons regarder les approches multi-threads et messages actifs. Un équilibrage dynamique (à l'exécution) est également prévu.

Bibliographie

[1] Amestoy (P.), Davis (T.) et Duff (I.). – An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, vol. 17, 1996, pp. 886–905.

[2] Amestoy (P.), Daydé (M.) et Duff (I. S.). – Use of level 3 BLAS in the solution of full and sparse linear equations. *In : Proceedings of the International Symposium*

on High Performance Computing, Montpellier, France, 22–24 March, 1989, éd. par Delhaye (J.-L.) et Gelenbe (E.). pp. 19–31. – Amsterdam, 1989.

[3] Amestoy (P.) et Duff (I. S.). – Memory allocation issues in sparse multiprocessor multifrontal methods. *International Journal of Supercomputer Applications*, vol. 7, 1993, pp. 64–82.

[4] Amestoy (P. R.), Daydé (M. J.), Duff (I. S.) et Morère (P.). – Linear algebra calculations on a virtual shared memory computer. *Int. Journal of High Speed Computing*, vol. 7, 1995, pp. 21–43.

[5] Ashcraft (C.). – *The fan-both family of column-based distributed cholesky factorization algorithms.* – In Alan George, John R. Gilbert, and Joseph W.-H. Liu, editors, *Graph Theory and Sparse Matrix Computations*. Springer-Verlag, New York, NY, 1993.

[6] Ashcraft (C.), Eisenstat (S. C.) et Liu (J. W. H.). – A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, vol. 11, 1990, pp. 593–599.

[7] Ashcraft (C.), Eisenstat (S. C.), Liu (J. W. H.), Peyton (B. W.) et Sherman (A. H.). – *A compute-ahead implementation of the fan-in sparse distributed factorization scheme.* – Rapport technique n° ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[8] Ashcraft (C.), Eisenstat (S. C.), Liu (J. W. H.) et Sherman (A. H.). – *A comparison of three column based distributed sparse factorization schemes.* – Rapport technique n° YALEU/DCS/RR-810, Yale University, New Haven, CT, 1990.

[9] Ashcraft (C.) et Liu (J. W. H.). – *Robust ordering of sparse matrices using multisection.* – Rapport technique n° CS 96-01, Departement of Computer Science, York University, Ontario, Canada, 1996.

[10] Berger (A.), Mulvey (J.), Rothberg (E.) et Vanderbei (R.). – *Solving multistage stochastic programs using tree dissection.* – Rapport technique n° SOR-95-07, Princeton University, 1995.

[11] Brandes (T.) et Desprez (F.). – Implementing Pipelined Computation and Communication in an HPF Compiler. *In : Proceedings of Euro-Par'96, LNCS 1123*. pp. 459–462. – Springer Verlag.

[12] Charrier (P.) et Roman (J.). – Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, vol. 55, 1989, pp. 463–476.

[13] Colombet (L.), Michallon (P.) et Trystram (D.). – Parallel matrix-vector product on rings with a minimum of communications. *In : Parallel Computing*, pp. 289–310.

[14] Conroy (J. M.), Kratzer (S. G.) et Lucas (R. F.). – Multifrontal sparse solvers in message passing and data parallel environments - a comparitive study. *In : Proceedings of PARCO 93*.

[15] Desprez (F.). – A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures. *In : IFIP 10.3 Conference on Programming Environments for Massively Parallel Distributed Systems*. pp. 365–371. – Birkhaeuser Verlag AG, Basel, Switzerland.

[16] Desprez (F.), Ramet (P.) et Roman (J.). – Optimal grain size computation for pipelined algorithms. *In : Proceedings of Euro-Par'96, LNCS 1123*. pp. 165–172. – Springer Verlag.

[17] Duff (I. S.), Erisman (A. M.) et Reid (J. K.). – *Direct Methods for Sparse Matrices*. – London: Oxford Univ. Press, 1986.

[18] Duff (I. S.), Grimes (R. G.) et Lewis (J. G.). – Sparse matrix test problems. *ACM Trans. Math. Software*, vol. 15, 1989, pp. 1–14.

[19] Duff (I. S.) et Reid (J. K.). – The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, vol. 9, n° 3, 1983, pp. 302–325.

[20] Dumitrescu (B.), Doreille (M.), Roch (J. L.) et Trystram (D.). – *Two-dimensional Block Partitionings for the Parallel Sparse Cholesky Factorization : the Fan-in Method*. – Rapport technique n° 3156, INRIA Rhône-Alpes, France, Avr. 1997.

[21] Espirat (V.). – *Développement d'une approche multifrontale pour machines a mémoire distribuée et réseau hétérogène de stations de travail*. – Rapport technique n° Rapport de stage 3ieme Année, ENSEEIHT-IRIT, 1996.

[22] Facq (L.) et Roman (J.). – *Algèbre linéaire creuse : distribution par bloc pour une factorisation parallèle de Cholesky*. – In G. Authié, J. M. Garcia, A. Ferreira, J. L. Roch, G. Villard, J. Roman, C. Roucairol, and B. Virot, eds, Parallélisme et applications irrégulières, 135–147, Hermès, 1995.

[23] George (A.) et Liu (J. W. H.). – *Computer Solution of Large Sparse Positive Definite Systems*. – Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[24] Gilbert (J. R.). – A separator theorem for graphs of bounded genus. *Journal of Algorithms*, vol. 5, 1984, pp. 391–407.

[25] Gilbert (J. R.). – Some nested dissection order is nearly optimal. *Information Processing Letters*, vol. 26, 1988, pp. 325–328.

[26] Gilbert (J. R.), Miller (G. L.) et Teng (S. H.). – *Geometric mesh partitioning: Implementation and experiments*. – Rapport technique n° CSL-94-13, Xerox Palo Alto Research Center, 1994.

[27] Gupta (A.). – *Fast and effective algorithms for graph partitioning and sparse matrix ordering*. – Rapport technique n° TR RC-20496, IBM research division, T.J. Watson Research Center, Yorktown Heights, 1996.

[28] Gupta (A.), Karypis (G.) et Kumar (V.). – *Highly scalable parallel algorithms for sparse matrix factorization*. – Rapport technique n° 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994.

[29] Gupta (A.) et Kumar (V.). – *A scalable parallel algorithm for sparse matrix factorization*. – Rapport technique n° 94-19, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994.

[30] Gupta (A.) et Rothberg (E.). – An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Scientific Computing*, vol. 15, 1993, pp. 1413–1439.

[31] Gupta (A.) et Rothberg (E.). – An evaluation of left-looking, right-looking, and multifrontal approaches to sparse cholesky factorization on hierarchical-memory machines. *Int. J. High Speed Comput.*, vol. 5, 1993, pp. 537–593.

[32] Hendrickson (B.) et Rothberg (E.). – *Improving the runtime and quality of nested dissection ordering*. – Rapport technique n° SAND96-0868J, Sandia National Laboratories, Albuquerque, 1996.

[33] Hiranandani (S.), Kennedy (K.) et Tseng (C.). – Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, vol. 21, 1994, pp. 27–45.

[34] Karypis (G.) et Kumar (V.). – *A fast and high quality multilevel scheme for partitioning irregular graphs*. – Rapport technique n° TR 95-035, Department of Computer Science, University of Minnesota, 1995.

[35] Karypis (G.) et Kumar (V.). – *METIS : Unstructured graph partitioning and sparse matrix ordering system*. – Rapport technique, Department of Computer Science, University of Minnesota, 1995.

[36] Lipton (R. J.), Rose (D. J.) et Tarjan (R. E.). – Generalized nested dissection. *SIAM Journal on Numerical Analysis*, vol. 16, 1979, pp. 346–358.

[37] Liu (J. H. W.). – The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, vol. 11, 1990, pp. 134–172.

[38] Liu (J. W. H.). – Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, vol. 11, n° 2, 1985, pp. 141–153.

[39] Markowitz (M. H.). – The elimination form of the inverse and its application to linear programming. *Management Science*, vol. 3, Avr. 1957, pp. 255–269.

[40] Michel (J.), Pellegrini (F.) et Roman (J.). – Techniques de partitionnement, de renumérotation, et de distribution de données pour la factorisation parallèle de Cholesky de matrices creuses. *In : Second séminaire sur les techniques nouvelles de traitement des matrices creuses pour les problèmes industriels, USTL, Lille.*

[41] Michel (J.), Pellegrini (F.) et Roman (J.). – Unstructured graph partitioning for sparse linear system solving. *In : Proceedings of IRREGULAR'97, Paderborn, LNCS 1253*, pp. 273–286.

[42] Ng (E.) et Peyton (B. W.). – Block sparse cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, vol. 14, 1993, pp. 1034–1056.

[43] Ohta (H.), Saito (Y.), Kainaga (M.) et Ono (H.). – Optimal Tile Size Adjustement in Compiling General DOACROSS Loop Nests. *In : International Conference on Supercomputing*, éd. par Press (A.). ACM SIGARCH, pp. 270–279. – Barcelona, Spain, Juil. 1995.

[44] Palermo (D.), Chandy (E. S. A.) et Banerjee (P.). – Communication Optimization used in the Paradigm Compiler for Distributed-Memory Mutlicomputers. *In : International Conference on Parallel Processing*, pp. 1–10. – St. Charles, IL, Août. 1994.

[45] Pellegrini (F.). – SCOTCH 3.1 *User's guide*. – Rapport technique n° 1137-96, LaBRI, Université Bordeaux I, Août. 1996.

[46] Pellegrini (F.) et Roman (J.). – Sparse matrix ordering with SCOTCH. *In : Proceedings of HPCN'97, Vienna, LNCS 1225*, pp. 370–378.

[47] Rothberg (E.). – *Exploring the tradeoff between imbalance and separator size in nested dissection ordering*. – Rapport technique, Silicon Graphics Inc, 1996.

[48] Rothberg (E.) et Schreiber (R.). – Improved load distribution in parallel sparse Cholesky factorization. *In : Supercomputing '94 Proceedings*.

[49] Schreiber (R.). – *Scalability of sparse direct solvers.* – Rapport technique n° RIACS TR 92.13, NASA Ames Research Center, Moffet Field, CA, Mai 1992.

[50] Tinney (W. F.) et Walker (J. W.). – Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. of the IEEE*, vol. 55, 1967, pp. 1801–1809.

[51] Walker (D. W.). – Portable programming within a message-passing model: the fft as an example. *In: The Third Conference On Hypercube Concurrent Computers and Applications*. Geoffrey Fox California Institute of Technology.

Chapitre 25

Système de communication dans les réseaux locaux à haut débits

Bernard Tourancheau (LHPC & INRIA ReMaP, laboratoire LIGIM[1])

Nous étudions les systèmes de communication type échange de messages sur des réseaux locaux à haut débit connectant des ordinateurs de type PCs.

Notre but est de montrer la faisabilité de couches logicielles réseaux performantes pour le transfert de données rapide, avec de faibles latences.

Après l'étude d'un réseau local ATM. Nous présentons plus largement nos travaux pour la création d'un protocole performant sur une plate-forme Myrinet. Les résultats obtenus avec une latence de 4.3 us et une bande passante supérieure au Gb/s montrent que les solutions matérielles de réseaux performants sont déjà présentes "dans les rayons". Notre apport se trouve dans la création de couches logicielles intégrant les besoins des applications et les techniques issues de l'algorithmique du parallélisme pour donner aux applications l'accès aux performances des réseaux locaux à haut débit.

25.1 Une étude avec un réseau local ATM

25.1.1 Le protocole Asynchronous Transfer Mode

Le protocole ATM a été choisi par l'ITU[2] comme standard des télécommunications du futur. De ce fait, beaucoup de matériels à bas prix vont/sont développés.

Ce protocole à été conçu pour offrir une bande passante élevée et “scalable”, une latence faible ainsi que la possibilité d'assurer plusieurs qualités de service (donc plusieurs prix) pour diverses activités de transfert d'information numérique.

[1]bat710, UCB-Lyon 69622 Villeurbanne - France
[2]International Télécommunications Union

L'idée de qualité de service permet théoriquement de proposer à la fois du transfert de données, de la vidéo, du temps réel, ... et des communications téléphoniques numériques. Pour cela, la bande passante totale est partagée par plusieurs canaux virtuels, assurant la qualité de service choisie au départ, sur une même ligne. Les services concernent la bande passante moyenne, la bande passante maximum (par pics ou soutenue), le pourcentage de cellules perdues, diverses stratégies d'utilisation de la bande passante disponible, ... et aussi la latence et la variation des délais d'acheminement (temps réel), ... La “scalabilité” de la bande passante est conçue par tranche de 51.84 Mbits/s. Actuellement, la bande passante de base est 155 Mbits/s, l'étape suivante est 622 Mbits/s (déjà supportée par les commutateurs du commerce) et le 2.2 Gbits/s est déjà dans les installations performantes (entre auto-commutateurs Télécom par exemple).

Au niveau de son fonctionnement, le protocole ATM propose la commutation de circuits (virtuels) et la commutation de paquets de taille fixe ou cellules, de 53 octets (5 d'en-tête + 48 de message) Pour plus détails voir [23, 20, 3, 10, 12, 25].

Du point de vue calcul parallèle sur LAN, peu de nouveautés : des cellules (i.e. flits ou paquets de taille fixe) sont pipelinés sur des canaux (virtuels) avec une bande passante maximum et des paramètres d'acheminement fixés au départ lors de l'établissement de la “connexion” (représenté par le canal virtuel).

25.1.2 Description de notre système

La plate-forme d'expérimentation est constituée de 8 stations (SUN SPARC5 85 MHz) connectées à un commutateur ATM par des liaisons en fils type paires torsadées UTP-5 avec des cartes d'adaptation Sbus. Nous utilisons 8 cartes, un commutateur 16×16 de marque FORE[3] et 2 cartes nous ont été prêtées par SUN pour nos tests. Le commutateur FORE ASX-200 utilisé gère les différentes classes de service à l'aide de plusieurs niveaux de priorités pour chaque flux, les paramètres de trafic sont testés pour assurer que leur somme ne dépasse pas la capacité des liens utilisés. Le coût de l'ensemble est beaucoup plus élevé que celui d'une installation similaire avec Ethernet 10/100.

25.1.3 Performances de communication

Après l'établissement d'un canal virtuel, on lui associe un descripteur de fichiers. Pour envoyer des trames de l'AAL sur ce canal virtuel on fait alors tout simplement des appels d'écriture ou de lecture sur ce descripteurs de fichiers. Du point de vue utilisateur, les trames AAL sont des paquets de données quelconques d'une taille maximale de 64K octets (en fait 1365 cellules de 48 octets moins 8 octets utilisés par le protocole AAL5).

[3]Commutateur ASX200, cartes SBA200e

Nous avons effectué des tests de modélisation pour déterminer les paramètres du modèle linéaire classique pour la communication de messages et nous en présentons ici un résumé. Le temps d'initialisation β, la vitesse de transfert maximum τ (i.e. l'inverse de la bande passante) et le débit (BP) correspondant sont donnés dans la Table 25.1.

	cartes FORE(carte SUN)		
	$\beta(\mu s)$	$\tau(\mu s/Byte)$	BP (Mo/s) [Mbits/s]
AAL5	510	0.104	8.77 [70.2]
IP sur ATM	690	0.128	7.81 [62.5]
PVM sur IP/ATM	1890	0.181	5.52 [44.2]
IP sur Ethernet	695	0.84	1.19 [9.52]
PVM sur IP/Ethernet	1500	0.85	1.17 [9.36]

Figure 25.1 : Performances de communication maximum obtenues avec notre réseau ATM 155 Mbits/s et sur Ethernet.

Les temps Ethernet ont été mesurés en utilisant l'interface Unix-BSD socket au niveau IP. Les tests ATM ont été effectués avec l'interface propriétaire des cartes concernées au niveau AAL et la qualité de service "best-effort" sur le réseau non chargé (le standard IP sur ATM crée une connexion de même type pour effectuer les transferts).

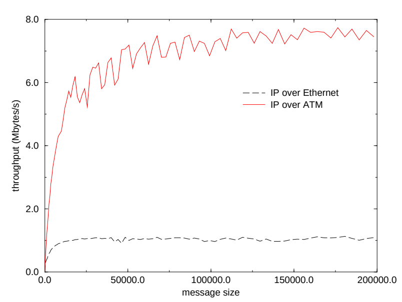


Figure 25.2 : Comparaison de la bande passante entre IP sur Ethernet et IP sur ATM

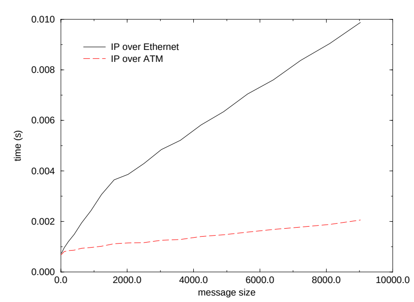


Figure 25.3 : Comparaison du temps de latence entre IP sur Ethernet et IP sur ATM

Avec l'AAL5, la bande passante obtenue est encourageante mais reste faible comparée à la bande passante crête. Nous en donnons l'explication en section 25.1.4.

En complément de la Table 1, les Figures 25.3 et 25.2 montrent, en fonction de la taille de données à transférer, le gain entre ATM et Ethernet sous IP (Figure 25.2).

Avec IP sur ATM, la latence est sensiblement la même qu'avec Ethernet. Elle provient en majeure partie des délais logiciels plus que matériels. Par contre dès les tailles relativement modérés la bande passante disponible sur ATM est 7 fois plus grande. De plus pour des connections multiples entre stations appartenant à des ensembles disjoints, le commutateur ATM ne perdra pas en bande passante alors que le bus Ethernet est partagé.

25.1.3.a Performances d'ATM avec différentes interfaces utilisateurs

Nous avons effectués des tests sur ATM en utilisant différentes interfaces de programmation, ces tests permettent essentiellement de comparer le temps de traitement logiciel imposé par ces différentes interfaces [20, 18]. Les données transférées physiquement ne sont pas forcément complètement identiques (encapsulation, gestion). Avec IP, pour assurer le contrôle de flot, on a régulièrement des acquittements dans le cas de transfert de très grand messages. Dans le cadre de nos tests (ping-pong) on peut toutefois considérer que l'impact de ces mécanismes sur la charge du réseau physique est complètement négligeable. Les différences viennent par conséquent de délais logiciels dans les protocoles et les interfaces et non pas matériels.

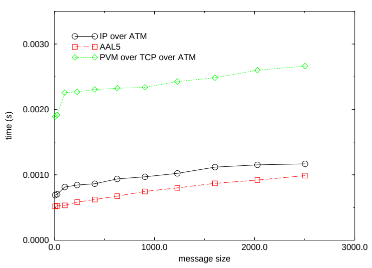


Figure 25.4 : Latence initiale sur ATM en utilisant respectivement le niveau AAL5, IP et PVM

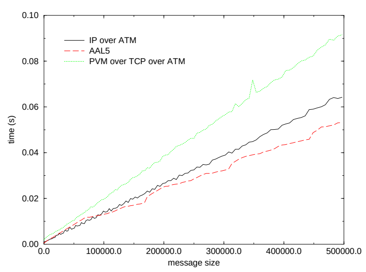


Figure 25.5 : Temps de communications sur ATM en utilisant respectivement le niveau AAL5, IP et PVM, grands messages

Les trois interfaces utilisées sont d'une part l'interface au niveau AAL, d'autre part les BSD socket à un niveau IP, et enfin PVM (qui de manière sous-jacent utilise IP avec les BSD socket). Ces résultats sont présentées sous forme de trois courbes, la Figure 25.4 pour les petits messages permet de constater l'impact sur la latence. La Figure 25.5 pour les grands messages permet de voir asymptotiquement l'impact sur la bande passante des différents niveaux d'utilisation.

25.1.3.b Effet pipeline

La Figure 25.6 permet de constater un phénomène à priori étrange, pour une certaine étendue de tailles de messages IP permet de meilleure performance que le niveau AAL, cela provient du découpage en paquets de plus petite taille (1500 octets) effectuée sous IP contrairement à notre implémentation qui envoie des paquets AAL de taille maximum (64Koctets) en vue de fournir la BP maximale sur de grands messages. L'utilisation de paquets plus petits permet pour une gamme précise de taille de messages de "pipeliner" l'envoi et la réception de paquets, le gain obtenu dépasse les délais dus à l'augmentation du nombre de paquets. Or la taille de ces PDU[4] est paramétrable (9k par défaut) jusqu'à 64Ko. On devrait pouvoir améliorer les performances obtenues avec le niveau AAL en adaptant la taille de paquets utilisés[5]. Ce découpage optimal varie suivant l'interface utilisée car la meilleure taille varie en fonction de la taille des messages échangés.

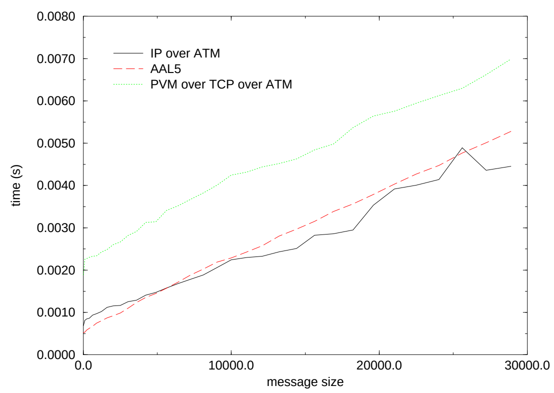


Figure 25.6 : Temps de communications sur ATM en utilisant respectivement le niveau AAL5, IP et PVM

[4]PDU : Protocol Data Unit représente la taille des trames AAL5 utilisées

[5]Nous utilisons cette technique dans la section 25.2

25.1.4 La bande passante du réseau est elle vraiment utilisable ?

On peut penser a priori qu'une station n'a aucun mal a envoyer des données stockées en mémoire sur un fil à une cadence de 155Mbits/s. Examinons le chemin parcouru par des données de leur emplacement initial dans un processus quelconque, jusqu'à leur transmission sur le fil relié au commutateur. Supposons que le processus en question gère directement des trames AAL (donc sans utiliser un protocole TCP/IP ou autre). Les deux pilotes de cartes ATM que nous avons essayé (FORE et SUN) sont de type ¡¡ STREAM ¿¿[6]. Cela signifie en particulier qu'ils prennent en entrée un bloc de donnée qui doit être présent dans l'espace mémoire du noyau. Lors de l'exécution de l'appel système demandant l'envoi d'un message sur ATM, les données utilisateur vont d'abord être recopiées dans l'espace mémoire du noyau, avant d'appeler la procédure suivante du pilote ATM. Dans le code du pilote de la carte ATM, les données sont à transmettre en utilisant un mécanisme d'accès direct à la mémoire (DMA). Cela n'est en général possible que si les données sont dans une zone appropriée pour le DMA (zone contiguë physiquement par exemple)[7]. Le pilote recopie donc les données dans cette zone. Il s'assure que des données ne sont pas restées dans un cache intermédiaire et initialise le transfert DMA. Une fois le transfert DMA achevé ou dès que suffisamment de données ont été transférées, la carte peut commencer à émettre les premières cellules sur le fil.

En résumé, avec la conception actuelle des pilotes, l'échange de message sous Unix demande 3 étapes sans recouvrement avant de pouvoir réaliser le transfert ATM:

- recopie espace d'adressage utilisateur vers l'espace d'adressage du noyau,
- recopie espace d'adressage du noyau vers l'espace d'adressage (contiguë) du pilote,
- lecture DMA.

Soit BP_l et BP_r les bandes passantes de la machine utilisée en lecture mémoire et en recopie mémoire. La borne supérieure de la bande passante de transfert est donc :

$$BP_{max} = \frac{1}{\frac{1}{BP_r} + \frac{1}{BP_r} + \frac{1}{B_l}}.$$

[6]les streams désignent un mécanisme utilisé dans le noyau pour implémenter des piles de protocole avec une interface ¡¡ standardisée¿¿

[7]Pour être "portable" le pilote doit avoir allouée une zone spéciale pour le transfert DMA, mais c'est essentiellement une contrainte de réutilisation de vieux logiciels, les machines modernes sont généralement capable de faire du DMA sur n'importe quelle adresse virtuelle

Examinons maintenant quelques chiffres, pour une SPARC 5, la bande passante continue pour une recopie mémoire est $BP_r = 33$Moctet/s, et pour une lecture en mémoire $BP_l = 66$Moctets/s. En mode continu, le coût des opérations mémoires est donc pour n Moctets de $2\frac{n}{33} + \frac{n}{66}$ secondes, soit une bande passante de $\frac{1}{\frac{2}{33}+\frac{1}{66}} = 13.20 Mbytes/s$. Ainsi avec les pilotes actuels, la bande passante disponible sur une connexion ATM 155Mbits/s de notre plate-forme, soit 17,12[8] Moctet/s de données, n'est pas exploitable.

Notons que le problème être moins critique pour des machines avec des capacités de transfert mémoire différentes.

25.2 Le réseau Myrinet et nos travaux pour l'obtention du Gb/s sur un LAN de PCs

Les applications multimédia, le calcul parallèle, les bases de données, ...ont besoin de réseaux à haut débit. Le niveau des performances requises demande de nouveaux types de protocoles pour exploiter au mieux les nouvelles technologies réseau.

Les évolutions des technologies récentes ont changé le rapport de performance entre les réseaux et les architectures internes des ordinateurs. Jusqu'à présent, le seul goulot d'étranglement était le réseau. Aujourd'hui, des technologies Gigabit/s comme Myrinet reportent le problème sur l'interface entre le processeur et le réseau de manière encore plus nette que dans la partie 25.1.

L'idée de base dans nos recherches sur le protocole BIP[9] a été de rendre disponible à l'application l'intégralité du potentiel du réseau. Nous partons du principe que ce qui compte n'est pas le débit crête du réseau mais ce que l'application peut réellement utiliser.

25.2.1 Présentation du réseau local Myrinet

Myrinet est une technologie de réseau local basé sur les communications et la commutation utilisées dans les machines massivement parallèles [1]. Myrinet est né de deux projets de recherche : Caltech MOSAIC C Multicomputer [24, 11] et USC/SI ATOMIC LAN [5]. Cette technologie est commercialisée par la jeune société californienne Myricom, Inc. (`http://www.myri.com`), fondée en avril 1994 par des membres de ces deux équipes.

Myrinet est une technologie réseau, basée sur des liens de communication point à point reliant entre eux les commutateurs et les noeuds finaux du réseau. Cette

[8]La charge est 48/53 octets par cellule et la gestion du protocole SONET au niveau physique coûte environ 3Mbits/s soit 137Mbits/s disponibles, i.e. 17.12 Mo/s

[9]Basic Interface for Parallelism

technologie est construite actuellement pour des débits de 1,28 Gbits/s sur chaque lien, tout en garantissant des latences très faibles. Les contrôles de flux et d'erreur sur les liens assurent une très bonne fiabilité aux niveau physique. Les commutateurs possèdent une bande passante agrégée suffisante pour supporter simultanément le débit maximal sur tous les liens entrants et sortants. Myrinet est utilisé aussi bien en tant que réseau local (LAN) de petit diamètre, ou réseau *système* (SAN)[10]. Chaque message contient dans son en-tête des octets de routage qui sont consommés par les commutateurs au fur et à mesure de sa progression.

La technologie Myrinet s'appuie sur des composants matériels et logiciels classiques :

- Câble électrique pour LAN
- Commutateur Myrinet 4, 8, 16 ou 32 ports. Il repose sur deux puces VLSI : la première assure la commutation (de type *crossbar*) et la seconde gère le protocole et le contrôle de flot.
- L'interface réseau possède un port et permet la connexion entre l'ordinateur hôte et le réseau. Elle possède un processeur et une mémoire SRAM (de 128 Ko à 1Mo). Cette mémoire sert à la fois de tampon pour la gestion des communications et de support pour les variables et le code du programme de contrôle. L'ensemble formé par le processeur, la gestion du port réseau et les mécanismes de DMA[11] sont implémentés dans une puce VLSI appelé LANai.
- Le programme de contrôle (MCP[12]) est chargé par l'hôte dans la mémoire SRAM de carte d'interface réseau et s'exécute dès que le pilote de la carte en donne l'ordre. Il doit assurer la réception et l'émission des paquets, il peut gérer les protocoles.
- Les logiciels de l'hôte : Myricom fournit à la fois un pilote pour différents systèmes permettant d'utiliser TCP/IP et UDP/IP sur Myrinet, et une API[13] spécifique.

La Figure 25.7 décrit l'architecture d'une station connectée à un commutateur.

25.2.2 Présentation de notre plate-forme expérimentale

La plate-forme d'expérimentation est constituée d'un commutateur 8 ports (4 ports SAN et 4 ports LAN) auxquels sont connectées 4 stations à base de Pentium Pro 200 MHz et 2 stations à base de Pentium 133 MHz. Ces stations utilisent des

[10] SAN, System Area Network
[11] Direct Memory Access
[12] Myrinet Control Program
[13] Application Programming Interface

cartes Myrinet/PCI 256Ko.
Les communications sont assurées par deux réseaux :

- Ethernet10; Il gère le trafic IP[14] à destination des adresses standards lorsque celui ci ne passe pas par Myrinet.
- Myrinet; Utilisé pour les applications et les tests.

Le système d'exploitation est Linux 2.0. Le coût de l'ensemble est de l'ordre de celui d'une installation similaire avec un réseau Ethernet-100 commuté (le prix des cartes réseaux compense le coût du commutateur).

25.2.3 Une vue d'ensemble du protocole BIP

L'objectif premier de BIP est de fournir une couche réseau de bas niveau, dont les services sont orientés vers le calcul parallèle, en tirant au mieux parti du matériel. Les fonctionnalités sont donc limitées: les couches supérieures doivent fournir des services de plus haut niveau en exploitant les performances et les fonctionnalités de BIP. La Figure 25.8 décrit notre point de vue sur l'organisation des différents protocoles réseaux pour conserver de bonnes performances au niveau des applications.

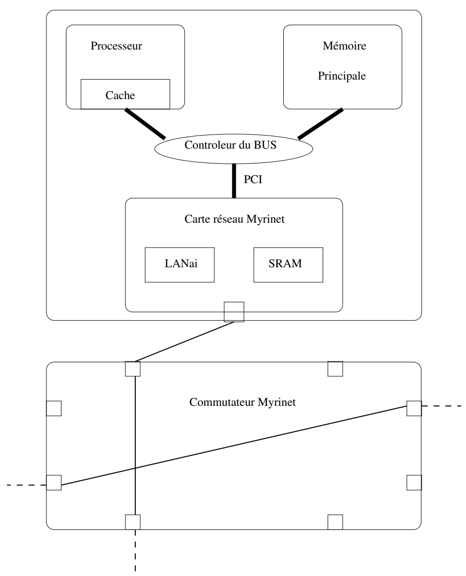


Figure 25.7 : Architecture générale

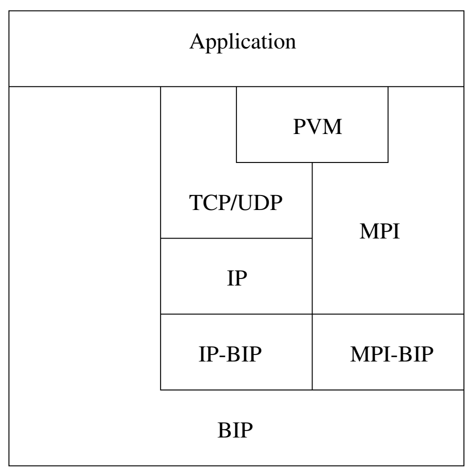


Figure 25.8 : Description de la pile des protocoles utilisables sur notre plateforme

[14] Internet Protocol

La conception de BIP permet d'offrir une interface directe à l'application. Les services primordiaux (envoi/réception simple) des autres interfaces peuvent être redirigés quasiment directement sur BIP, ce qui leur permet de conserver de très bonnes performances.

25.2.4 Les fonctionnalités fournies par l'interface BIP

Une application BIP se compose d'un ensemble fixe de n processus. BIP fournit des envois et réceptions aussi bien bloquants que non-bloquants.

Les communications se font avec une sémantique de rendez-vous pour les messages longs. BIP assure qu'un envoi pourra se terminer (le message a quitté le buffer d'émission) uniquement si un *receive* a été posté par le destinataire. Le réseau possède une capacité limitée de mémorisation: si un message ne "tient" pas dans ses buffers, la machine émettrice sera obligée d'attendre que le récepteur commence la réception, c'est à dire vide les buffers du réseau. Par contre, si le message est suffisamment petit, l'émission pourra se faire qu'il y ait un *receive* de posté ou non grâce à un système de queues. Compte tenu de la fiabilité du réseau Myrinet, BIP assure que les messages arriveront dans l'ordre d'émission sur le récepteur.

Un mécanisme d'étiquettage (*tag*) est fournit: l'application peut choisir de ne recevoir que le prochain message portant un certain *tag*. Il peut ne pas être le prochain message à arriver. Pour chacun des *tags*, une file d'attente statique est gérée pour les petits messages. Aucun contrôle de flot n'étant assuré par BIP, il est possible qu'il y ait saturation de cette file d'attente, dans ce cas, BIP se reposera directement sur le contrôle de flot physique du réseau Myrinet pour stopper l'émetteur[15].

BIP détecte actuellement les erreurs de transmission qui peuvent se produire, mais n'implémente pas d'algorithme de retransmission. Sur détection d'une telle erreur l'application est stoppée avec une erreur ou bien interrompue pour permettre la reprise d'erreur. On se reportera au manuel de l'utilisateur [17] pour une description plus précise des fonctions de BIP.

25.2.5 Implémentation de BIP

Notre but était d'apporter les performances réseau jusqu'au niveau de l'application [19, 21]. Nous avons mis en avant les réflexions suivantes pour préserver les performances réseaux.

25.2.5.a Contrôle du réseau au niveau de l'utilisateur

L'accès aux entrées/sorties et au réseau dans un système UNIX passe par les appels systèmes, mécanismes lourds et coûteux en temps, BIP donne à l'utilisateur un accès direct au réseau, sans passer par le noyau:

[15] Cette situation peut être à éviter car elle est nuisible aux performances du reste du réseau, en effet le dernier message mis en attente monopolise le chemin entre l'émetteur et le récepteur

25.2.5.b Éviter les copies mémoire

La bande passante du réseau étant comparable à celle de la mémoire, le nombre de recopies effectuées (pour l'ajout ou le retrait d'un en-tête par exemple) devient critique. Pour permettre la copie des données directement du buffer d'émission dans la carte d'interface réseau, BIP utilise le DMA via le bus mémoire (PCI dans notre cas), effectue les conversions adresse virtuelle à adresse physique et verrouille la page à transmettre en mémoire centrale. BIP propose un "loadable module" du noyau Linux, qui ajouté au système d'exploitation assure ces fonctionnalités. Son implémentation assure un niveau de sécurité correct.

25.2.5.c Optimisation de la gestion des messages longs

Une étude du chemin d'un message à transmettre montre qu'au minimum les données sont acheminées en 4 étapes sur notre plate-forme:

1. de la mémoire principale à la mémoire de la carte sur la machine émettrice,
2. de la mémoire de la carte au fil sur la machine émettrice,
3. du fil à la mémoire de la carte sur la machine destinataire,
4. de la mémoire de la carte à la mémoire principale sur la machine destinataire.

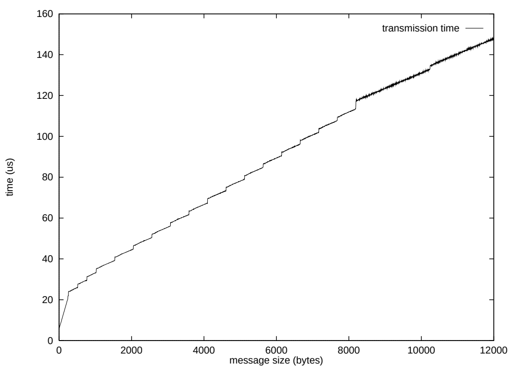


Figure 25.9 : Temps de transmission en fonction de la taille de message, les paliers sont dus aux découpages en paquets

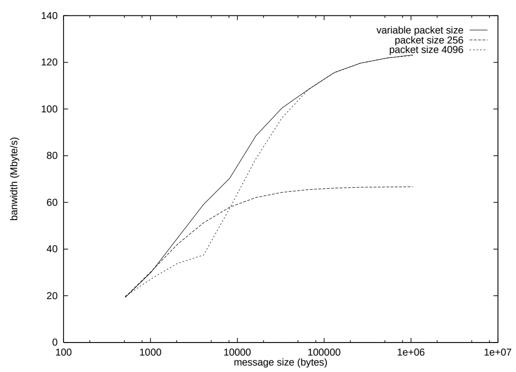


Figure 25.10 : Bande passante avec une stratégie de taille de paquets fixe et avec notre méthode adaptative

Pour optimiser ce transfert dans BIP, les messages sont découpés en paquets et les quatre phases précédentes sont pipelinées. Des tampons assurent le remplissage permanent des pipelines. Le meilleur compromis de taille de paquet dépend de la taille du message transféré. Nous avons tabulé les tailles de paquets quasi optimales (erreur max 4%) en fonction de la longueur du message. Cette stratégie adaptative est mise en évidence sur la Figure 25.9 (présence de plusieurs paliers). Le gain obtenu est important comme le montre la Figure 25.10. La taille optimale des paquets est calculée en fonction d'une modélisation précise de la plate-forme décrite dans [22].

De plus, la majeure partie de la transmission (découpage en paquets, transfert par DMA) est entièrement géré par le MCP sur le LANai, jusqu'à terminaison. Notre implémentation autorise donc un recouvrement calcul/communication important.

25.2.5.d Optimisation de la gestion des petits messages

Pour optimiser la gestion des petits messages, notre étude a montré qu'en émission, recopier directement les données dans la mémoire du LANai est plus rapide que de faire un transfert DMA (de même en réception). De plus, la recopie ultérieure vers le buffer de l'utilisateur permet au passage de charger les données dans le cache du processeur. Cette stratégie permet donc un gain d'accès substantiel, les petits messages étant le plus souvent utilisés immédiatement.

Nous avons utilisé autant que possible les mécanismes du processeur hôte pour la gestion de cohérence de cache (par "snooping"), et l'écriture groupée (mode "burst") afin d'optimiser aux mieux les échanges d'information.

25.2.6 Les performances

La latence obtenue, inférieure à 5 micro-secondes, représente l'équivalent à l'exécution de 1000 instructions assembleur exécutées par le Pentium Pro, ou environ 150 instructions assembleur exécutées par le LANai. Si l'on prend en compte les délais de transmission eux-mêmes, ces nombres sont divisés par au moins deux. Il est clair que chaque instruction compte à ce niveau de performance.

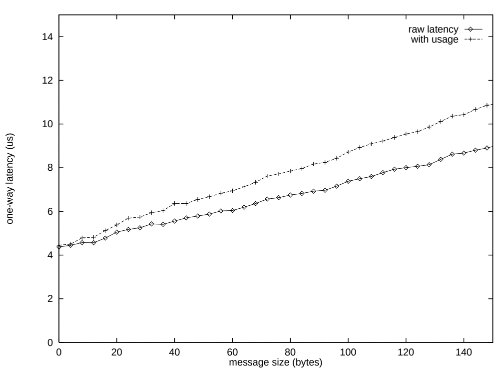


Figure 25.11 : Latence aller simple de BIP sur des messages de petites taille.

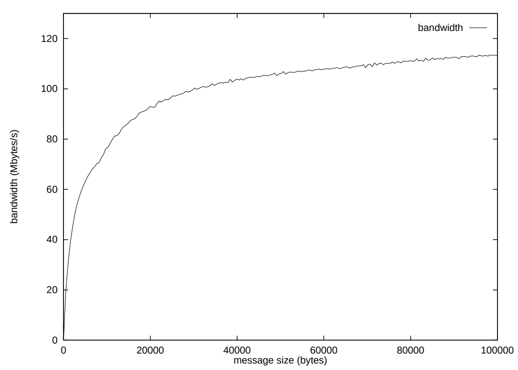


Figure 25.12 : Bande passante de BIP, calculée à partir du volume de données utiles transférées

Notre plate-forme autorise des transferts sur le bus PCI à un peu moins de 132Moctet/s. La bande passante du bus mémoire est légèrement supérieure (180 Moctet/s en lecture). Les cartes réseau Myrinet/PCI sont cadencées à la vitesse du bus PCI, et sont limitées à un débit de 132Moctets/s bien que le réseau supporte 160Moctets/s. Ces valeurs montrent que, bien que nous disposions d'un réseau à 160Moctets/s, deux des étapes sont limitées à 132Mo/s. La Figure 25.12 met en évidence le potentiel de bande passante délivré par BIP à partir de tailles de message relativement modérées. Pour des messages de 4Koctets, la moitié de la bande passante maximale (soit 66Moctet/s) est déjà atteinte. Le maximum de 126Mo/s, représente représente plus de 96% du maximum théorique supporté par le bus PCI, actuel goulot d'étranglement.

25.2.7 Utilisation de BIP pour IP

Le noyau Linux utilisé pour notre recherche permet d'écrire relativement simplement des pilotes de périphériques pour de nouvelles cartes réseaux. Le premier protocole IP-BIP que nous avons mis au point permet d'obtenir de meilleures performances que le pilote réseau fournit par Myricom au vu des premières expériences sur la Figure 25.13. Ce prototype permet d'envisager des recherches futures sur l'optimisation de l'implémentation du protocole TCP de Linux pour le haut débit. D'ores et déjà, nous avons intégré des modifications à la pile IP nécessaires pour assurer la régularité des performances à ces débits. Ces modifications ont aussi été introduites dans la version de TCP/IP servant à la comparaison de la Figure 25.13.

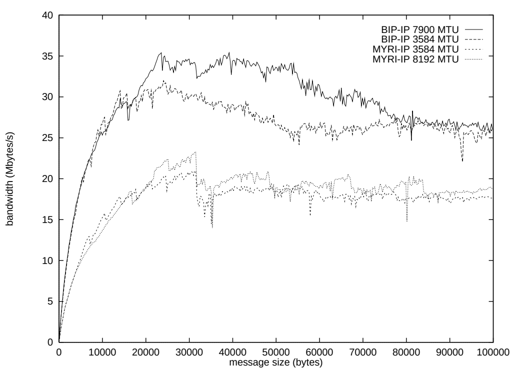


Figure 25.13 : Comparaison de la bande passante mesurée sous TCP/IP avec notre pilote et le pilote fourni par Myricom pour deux tailles de MTU différentes

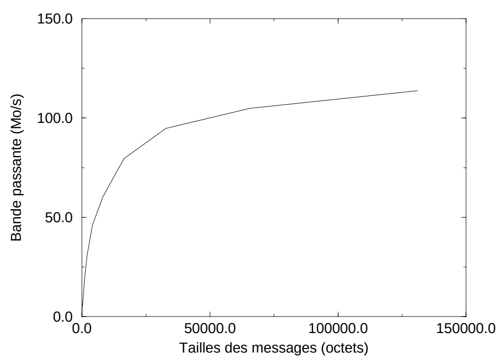


Figure 25.14 : Bande passante sous MPI-BIP

25.2.8 Utilisation de BIP pour implémenter MPI

Notre première version du protocole MPI-BIP[28] interface MPICH [7] en implémentant les fonctions de la Channel Interface [6] et un contrôle de flot basé sur une méthode à crédits qui interagit directement avec les queues BIP.

La Figure 25.14 présente nos premiers résultats. La latence sous MPI-BIP est de $9\mu s$ et la bande passante atteinte dépasse le Gbits/s pour 10Moctets. Cette première version a été validée sur différentes applications dont les NAS Parallel Benchmarks.

25.2.9 Comparaison avec d'autres interfaces

En préalable à nos études, nous avons réalisé une série de tests de performance pour évaluer à la fois la plate-forme réseau Myrinet et l'efficacité des différents logiciels disponibles. Les tests comparatifs présentés sont de 3 types: les interfaces utilisant IP comme protocole de base, les Fast Messages et BIP[16].

Les tests qui utilisent la couche IP (PVM, MPI, les Sockets) ont pour base le *driver* IP fournit en standard par Myricom. Nous présentons sur les Figures 25.15 et 25.16 un résumé des tests effectués pour les différentes API.

Comme on pouvait le supposer, la latence des interfaces des bibliothèques de haut niveau est plus importante que celle les API de bas niveau. Par comparaison, le support Ethernet-10 de notre plate-forme donne des temps très voisins de ceux obtenus avec le driver IP de Myricom, mais tous, à l'avantage d'Ethernet.

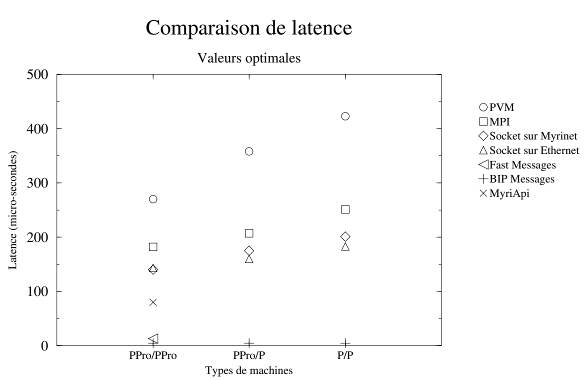


Figure 25.15 : Comparaison des différentes interfaces, mesure de latence

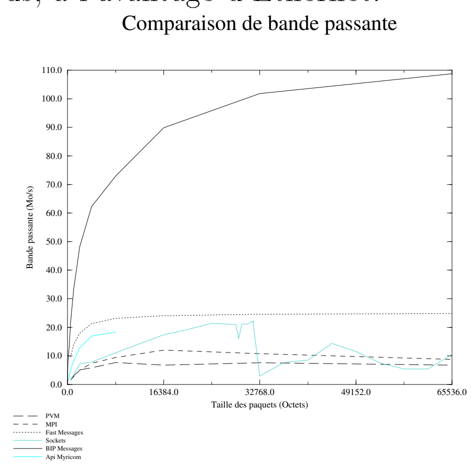


Figure 25.16 : Comparaison des différentes interfaces, mesure de bande passante

Pour ce qui concerne le débit, deux points sont à étudier. Le débit maximum qui donne une indication sur les limitations du matériel (Figure25.16) et la taille

[16]Pour plus de détails se reporter à [13]

de message qui permet d'atteindre la moitié de cette bande passante maximum qui donne une évaluation de la montée en débit du réseau (Table 25.1).

25.2.10 Comparaison avec d'autres architectures

La Table 25.2 compare les performances de communication de différentes architectures parallèles avec notre plate-forme. Ces résultats situent notre plate-forme réseau au niveau de celle du Cray T3D !

25.2.11 Travaux du domaine

Les approches utilisées pour le développement des couches logicielles en interaction directe avec les réseaux ont été profondément modifiées avec l'arrivée des réseaux haut débits. Cette partie présente tout d'abord trois bibliothèques construites dans l'optique du calcul parallèle: les *Actives Messages*, les *Fast Messages* et NX, puis d'autres projets plus orientés systèmes: U-Net, UHN Net* et PM.

Interface	Bande passante maxi (Mbits/s)	Taille pour demi débit (octets pour Mbits/s)
PVM (Myricom/IP)	60	1280 pour 30
PVM (BIP/IP)	128	2176 pour 64
MPI (Myricom/IP)	100	3072 pour 50
MPI (BIP/IP)	205	3072 pour 102.5
Sockets (TCP/Myricom/IP)	190	8192 pour 95
Sockets (TCP/BIP/IP)	272	5120 pour 136
Fast Messages	200	768 pour 100
BIP Messages	1005	4096 pour 502.5

Tableau 25.1 : Comparaison générale

25.2.11.a Les *Active Messages* (AM)

Ce projet[27, 26] a débuté à l'université de Californie à Berkeley où il se poursuit en parallèle avec des travaux à l'université de Cornell. La fonctionnalité des AM est celle d'un RPC[17] de bas-niveau.

25.2.11.b Les *Fast Messages* (FM)

Ce projet[15, 14] a été développé à l'université d'Illinois à Urbana Champaign. Il utilise le même principe que les AM pour ce qui est du format du message et de

[17] Remote Procedure Call

l'utilisation d'un *handler*. Les FM différent par la façon dont un message est intégré dans le calcul sur le noeud destinataire. Dans les AM, le *handler* doit être exécuté à la réception du message. Pour les FM, le déclenchement du *handler* est contrôlé par le programme utilisateur par l'intermédiaire d'une primitive. Les FM gèrent le contrôle de flot de manière interne.

machine	latence (μs)	Débit maxi (Moctets/s)	taille (en octets) pour un $\frac{1}{2}$ débit maxi
ATM155/sparc5 (AAL5) [20]	500	10	10000
iPSC/860 (NX)) [4]	65	3	340
TMC CM-5 (CMMD) [4]	95	9	962
Intel Paragon (NX) [4]	29	154	7236
Intel Paragon (MPI) [7]	40	70	~7000
Meiko CS2) [4]	83	43	3559
IBM SP-2 (MPI) [4]	35	35	3262
T3D (shmem) [4]	3	128	363
T3D (MPI) [7]	21	100	~7000
SGI Power CHallenge (MPI) [7])	47	55	~5000
Myrinet/Ppro200 (BIP)	4.3	126	4096

Tableau 25.2 : Comparaison des performances avec plusieurs architectures.

25.2.11.c L'interface NX (Intel)

Le logiciel NX[16] de la machine développée par Intel contrôle 2 processeurs, un dédié au calcul et un aux communications. Le rôle du deuxième processeur est similaire à celui du LANai. Les problèmes soulevés sont donc les mêmes que pour le réseau Myrinet.

25.2.11.d U-Net

Ce projet[2] est développé à l'université de Cornell. U-Net place l'ensemble de la pile du protocole dans l'espace utilisateur (en ne sollicitant plus le noyau) tout en fournissant un système de sécurité (plusieurs processus peuvent se partager le réseau). U-Net n'est donc pas une librairie de communication mais une interface pour le développement de protocoles de communication dans l'espace utilisateur.

25.2.11.e UNH Net*

Ce projet[8] est développé à l'université du New Hampshire. Leur SBP (Streamline Buffer Protocol) [8] permet un accès efficace au réseau. Cette approche ne cherche pas à éliminer le système d'exploitation (Linux) du chemin des données, le noyau a été modifié pour permettre l'utilisation de mémoire partagée entre le système d'exploitation et l'application de l'utilisateur (files de messages). Aucun appel système n'est nécessaire à l'envoi ou à la réception d'un message.

25.2.11.f PM

Ce projet[9] est développé au Japon à RWCP (Real World Computer Partnership). Les travaux portent essentiellement sur le developpement de systèmes d'exploitation spécifiques au projet qui est base sur une plate-forme Myrinet de stations SUN. La bibliothèque de communication PM a été développée pour donner un accès rapide au réseau.

25.3 Conclusion

Les opérateurs de télécommunications l'ayant choisi comme support futur, l'ATM est en plein développement. Bien qu'il ne soit pas conçu dans cette optique, son intérêt pour les LAN et donc le calcul parallèle/distribué sur réseau de stations peut par effet de bord commercial devenir très important. L'utilisation de IP au dessus de ATM a, par exemple, été standardisée très rapidement.

Nous comparons les performances des APIs: $PVM < IP < AAL5$ dans la partie 25.1. Le gain en débit est immédiat par rapport à Ethernet10 mais reste bien en dessous des performances maximales du matériel. En latence les résultats sont du même ordre qu'Ethernet10 alors que l'on pourrait espérer plus d'une telle technologie. Cela vient du fait que les logiciels d'exploitation ne sont pas optimisés. Nous mettons en évidence ce problème pour de tels transferts à haut débit.

L'architecture logicielle du pilote doit donc être conçue de manière assez performantes pour soutenir les performances réseaux disponibles par rapport aux possibilités du processeur. Nous montrons qu'avec les logiciels de notre plate-forme, les transferts entre stations SPARC5 ne peuvent pas dépasser 12.2 Moctets/s soit 97.6 Mbits/s alors que le réseau supporte 155Mb/s.

Pour se rapprocher des performances crête des réseaux à haut débit, des recherches doivent donc être menées au niveau de l'interface entre les applications et le système de communication (PVM sur ATM/AAL5 par exemple).

Nous avons effectué des recherches pour développer un tel protocole spécifique, nécessaire pour garantir des transferts de données à hauts débits entre applications. Nous avons essayé de faire bénéficier la conception d'un tel protocole de notre expérience des applications et systèmes parallèles:

- Garantir les performances au niveau application.
- Modélisation et optimisation du point de vue algorithmique.

Le protocole BIP issu de ces recherches, décrit dans la partie 25.2, est sans compromis par rapport aux performances et son optique applicative permet de l'utiliser comme couche de base pour des protocoles ou interfaces de plus haut niveau.

Le protocole BIP arrive à maturité avec l'implémentation des couches MPI-BIP et IP-BIP qui permettent une utilisation simple de très hauts débits pour de nombreuses applications existantes. Par exemple, pour validation, nous utilisons exclusivement notre pilote IP-BIP pour nos expériences et le trafic IP classique depuis plusieurs mois.

Notre plate-forme de PC présente de meilleures performances réseau que la plupart des machines parallèles actuelles pour le couple (débit, latence) et ceci pour tous les protocoles/interfaces implémentées.

Les performances obtenues ouvrent de nouvelles perspectives d'utilisation car l'accès à la mémoire distante est peu coûteux, même pour de petites quantités de données garantissant la “scalabilité” (systèmes de fichiers distribués, migration de processus, transfert de vidéo peu ou pas compressée, animations vidéo, temps réel, ...)

Notre recherche future consiste en la stabilisation des trois protocoles existants (BIP, MPI-BIP, IP-BIP), leur portage sur d'autres plate-formes réseaux, l'implémentation de nouvelles fonctions dans BIP issues de la demande des protocoles de niveau supérieur, et la création de nouveaux protocoles au dessus de BIP pour des applications spécifiques.

Les pages web du projet se trouvent à l'URL: http://www-bip.univ-lyon1.fr

Bibliographie

[1] Myrinet link and routing specification. http://www.myri.com/myricom/document.html.

[2] Basu (A.), Buch (V.), Vogels (W.) et vonEicken (T.). – U-Net: A user-level network interface for parallel and distributed computing. *In : Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP).* – Copper Mountain, Colorado, December 1995.

[3] Berger-Sabbatel (G.). – Pvm and atm networks. *In : EuroPVM*, éd. par Dongarra, Gengler, Tourancheau et Vigouroux. – Hermes.

[4] Dongarra (J.) et Dunigan (T.). – *Message-Passing Performance of Various Computers.* – Rapport technique, University of Tennessee, 1995.

[5] Felderman (R.), DeSchon (A.), Cohen (D.) et Finn (G.). – ATOMIC: A high speed local communication architecture. *Journal of High Speed Networks*, 1994. – Ios Press.

[6] Gropp (W.) et Lusk (E.). – *MPICH Working Note: Creating a new MPICH device using the Channel interface.* – Rapport technique, Argonne National Laboratory, 1995.

[7] Gropp (W.), Lusk (E.), Doss (N.) et Skjellum (A.). – *A high-performance, portable implementation of the MPI message passing interface standard.* – Rapport technique, Argonne National Laboratory, 1996.

[8] Hatcher (P. J.), Russell (R. D.), Kumaran (S.) et Quinn (M. J.). – Implementing data-parallel programs on commodity clusters. *In : Proceedings of the Spring School on Data Parallelism.* – Les Menuires, France, March 1996.

[9] Hiroshi Tezuka, Atsushi Hori (Y. I.). – *PM:A High-Performance Communication Library for Multi-user Paralllel Environments.* – Rapport technique n° TR-96015, RWCP, 1996.

[10] Huang (C.) et McKinley (P. K.). – Communication issues in parallel computing across atm networks. *IEEE Parallel & Distributed Technology*, Dec 1994, pp. 73–86.

[11] L. Seitz (C.). – *Mosaic C : An experimental fine-grain multicompute r.* – Rapport technique, Pasadena, CA, California Institute of Technology, 1992.

[12] Lin (M.), Hsieh (J.), Du (D. H.), Thomas (J. P.) et MacDonald (J. A.). – Distributed network computing over local ATM networks. *In : Proceedings of Supercomputing 94.* pp. 154–163. – IEEE.

[13] Naquin (F.). – *Evaluation d'architecture réseaux locaux haut débit: Myrinet.* – Thèse, LHPC et Laboratoire d'informatique de Besançon, 1997.

[14] Pakin (S.), Karamcheti (V.) et Chien (A.). – Fast messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, 1997.

[15] Pakin (S.), Lauria (M.) et Chien (A.). – High performance messaging on workstations: Illinois fast messages (FM) for myrinet. *In : Supercomputing '95.* – San Diego, California, 1995.

[16] Pierce (P.) et Regnier (G.). – The paragon implementation of the NX message passing interface. *In : SHPCC '94.* – Knoxville, Tennessee, 1994.

[17] Prylli (L.). – *BIP User Reference Manual.* – Rapport technique, LIP/ENS-LYON, 1997.

[18] Prylli (L.) et Tourancheau (B.). – Calcul parallele sur reseau ATM de stations de travail. *In : RENPAR'8*, éd. par Roman. – Universite de bordeaux.

[19] Prylli (L.) et Tourancheau (B.). – *New protocol design for high performance networking.* – Rapport technique, 69364 Lyon, France, LIP-ENS Lyon, 1997.

[20] Prylli (L.) et Tourancheau (B.). – Parallel computing on an ATM LAN. *In : ATM97.* – Bradford, UK, 1997.

[21] Prylli (L.) et Tourancheau (B.). – Bip: a new protocol designed for high performance networking on myrinet. *In : Workshop PC-NOW, IPPS/SPDP98.* – Orlando, USA, 1998.

[22] Prylli (L.), Tourancheau (B.) et Westrelin (R.). – Modeling of a high speed network to maximize throughput performance: the experience of bip over myrinet. – Submitted.

[23] Pujolle (G.). – *Les reseaux.* – Eyrolles, 1995.

[24] Seitz (C. L.), Boden (N. J.), Seizovic (J.) et Su (W.-K.). – The design of caltech mosaic c multicomputer. – Proceedings of the University of Washington Symposium on Integrated Systems.

[25] Thekkath (C. A.), Levy (H. M.) et Lazowska (E. D.). – *Efficient Support for Multicomputing on ATM Networks.* – Rapport technique, Department of Computer Science and Engineering, University of Washington, 1993.

[26] vonEicken (T.). – *Active Messages: an Efficient Communication Architecture for Multiprocessors.* – Thèse de PhD, University of California at Berkeley., November 1993.

[27] vonEicken (T.), Culler (D. E.), Goldstein (S. C.) et Schauser (K. E.). – Active messages: a mechanism for integrated communication and computation. *In : Proceedings of the 19th Int'l Symp. on Computer Architecture.* – Gold Coast, Australia, may 1992.

[28] Westrelin (R.). – *Réseaux haut débit et calcul parallèle: étude de Myrinet.* – Thèse, ENS Lyon, Université Lyon 1, INSA Lyon, 1997.