Ordonnancement de programmes parallèles sur grappes Illustrations en algèbre linéaire

Jean-Louis Roch

Laboratoire ID-IMAG, ENSIMAG ZIRST Montbonnot,

51 avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France

Jean-Louis.Roch@imag.fr

Résumé

Pour obtenir la portabilité des programmes sur grappes, on fait appel à un mécanisme d'ordonnancement qui contrôle la distribution des données et des calculs sur les ressources mémoires et processeurs. Un tel ordonnancement entraînant un surcoût lors de l'exécution, cet article présente deux techniques permettant de l'optimiser : dégénérescence séquentielle d'une part et compilation à la volée des communications d'autre part. Un algorithme original couplant ces deux techniques est proposé pour améliorer l'exploitation des ressources d'une grappe de multiprocesseurs.

1 Introduction: grappes et ordonnancement

Du fait de leur très large disponibilité et de leur rapport performance/prix, les grappes de processeurs sont les architectures parallèles les plus courantes. A la différence des architectures parallèles classiques constituées de processeurs identiques connectés par un réseau à haute performance, chaque grappe constitue une architecture spécifique. En effet, bien que construite à partir de composants standards (aussi bien processeurs que réseau – cartes, cables, hubs et switchs –), une grappe possède souvent ses propres caractéristiques : nombre de noeuds (souvent non identiques, mono ou/et multi-processeurs), topologie réseau et conventions d'utilisation.

La portabilité d'un algorithme parallèle sur une telle grappe repose alors sur l'ordonnancement. Il s'agit de contrôler l'exécution de l'algorithme, qui possède généralement un très grand degré de parallélisme, sur une machine comportant un nombre fini de processeurs, petit comparativement à ce degré. Il s'agit donc de limiter le parallélisme, autrement dit de sérialiser le programme en étant le plus efficace possible.

Les programmes mais aussi les machines (notamment en contexte multi-applications) ayant très généralement un comportement dynamique, une technique d'interprétation est utilisée. Cette interprétation permet de connaître en cours d'exécution et de manière distribuée, les différentes unités de calculs à réaliser (tâches) et les conditions de synchronisation associées (entre l'écriture d'une donnée et les lectures postérieures) De manière générale, l'exécution d'un programme est modélisée par un graphe de flot de données [KLadH96, GRCD98] qui évolue dynamiquement au cours de l'interprétation. Ce graphe est orienté et sans circuit; il décrit les synchronisations liées aux dépendances écriture-lecture. Aussi, est-il au moins implicite lors de l'interprétation puisque celle-ci doit respecter ces synchronisations. Au niveau de cette interprétation, on distingue traditionnellement [KR97, WLMR93] : i) le calcul de l'ordonnancement (ou politique de décision); ; ii) la réalisation de l'ordonnancement calculé (ou politique de contrôle).

Généralement, l'analyse d'un algorithme d'ordonnancement est centrée sur la politique de décision et son ratio de perfomance, i.e. le rapport entre le temps de l'exécution du programme avec l'ordonnancement calculé et le temps fourni par l'ordonnancement optimal [CCLL95]. Pourtant, la réalisation d'un ordonnancement entraîne un surcoût qui ne peut pas être négligé en théorie et encore moins en pratique; pris globalement le surcoût dû au contrôle de l'ordonnancement est souvent prohibitif[KR97]. Par ailleurs, le coût du calcul d'un ordonnancement ne se réduit pas au temps; d'autres critères, en particulier de mémoire, s'avèrent essentiels lors de l'interprétation d'un programme parallèle.

Cet article introduit, via des exemples en algèbre linéaire, les principales techniques utilisées pour minimiser le surcoût lié à l'interprétation d'un programme parallèle d'une part sur une architecture SMP (dégénérescence séquentielle) et d'autre part sur une architecture distribuée avec modèle délai (dégénérescence distribuée). Ces techniques sont utilisées de manière complémentaire sur une grappe homogène constituée de nœuds multiprocesseurs symétriques.

Les techniques classiques de calcul par blocs et de partitionnement bidimensionnel sur architecture distribuée homogène sont tout d'abord rappelées (§2). Puis, deux techniques permettant de contrôler efficacement l'ordonnancement sont détaillées : dégénérescence séquentielle pour des ordonnancements gloutons (§3) et dégénérescence distribuée (§4). Des optimisations permettent de limiter les prises de verrou et la place mémoire nécessaire. En conclusion (§5), le couplage de ces techniques sur grappes de multiprocesseurs est expérimenté pour la résolution de systèmes linéaires.

2 Algèbre linéaire : blocs et partitionnement bidimensionnel cyclique

En algèbre linéaire aussi bien dense que creuse, la réorganisation d'une matrice sous forme de blocs bi-dimensionnels sur lesquels des opérations de niveau BLAS 3 sont effectués est très largement utilisée, en séquentiel (par exemple avec LAPACK [ABB+95] ou encore ATLAS) mais également en parallèle [GPS90, Dor99].

Ainsi une matrice de taille $n \times n$ est partitionnée dans les deux dimensions en $N \times N$ blocs de dimensions $k \times k$. Pour des raisons de simplicité, on supposera dans la suite que k est un diviseur de n. Cette réorganisation permet en effet d'augmenter la localité et donc de mieux exploiter la hiérarchie mémoire des architectures actuelles. Sur une opération BLAS de niveau 3 utilisant n^2 mots mémoire, il y a $O(n^3)$ calculs à réaliser. En choisissant la taille des blocs de manière à ce que les n^2 éléments logent dans le cache mémoire de niveau le plus élevé, les $O(n^3)$ calculs peuvent être réalisés en exploitant pleinement la puissance du processeur [Dor99].

En réorganisant les boucles des algorithmes d'algèbre linéaire, on peut alors réécrire ceux-ci non plus en terme d'opérations sur des scalaires mais en opérations sur les blocs de la matrice partitionnée. Le parallélisme est alors exprimé au niveau des opérations sur les blocs, et est aisément identifiable par analyse de dépendances, soit à la compilation à partir du code source [DRV00], soit à l'exécution par interprétation du code au niveau macroscopique des blocs [Dor99].

De manière à limiter le nombre de communications, on utilise souvent un placement par blocs cyclique bidimensionnel qui consiste à distribuer les blocs des matrices de la manière suivante. En supposant que $q^2 = p$ processeurs sont disponibles, le blocs d'indice (i, j) est placé sur le processeur d'indice $(i \bmod q)q + (j \bmod q)$. La figure 1 montre un exemple de placement cyclique bidimmensionel des blocs sur 4 processeurs. Les opérations sur les blocs de matrice, sont alors réalisées sur le site où est localisé le bloc modifié par l'opération. On montre alors que le volume total de communication de la plupart des algorithmes d'algèbre linéaire avec ce placement des données est en $O(n^2\sqrt{p})$ [KGGK93]; en particulier, produit de matrices, factorisation $(LU, LUP, LL^t, ...)$, résolution de système. Cependant, bien que plus performant qu'un placement mono-dimensionnel, il est à noter que ce placement n'est pas montré optimal et n'est que partiellement justifié sur des modèles quantitatifs élémentaires pour les communications (délai par exemple).

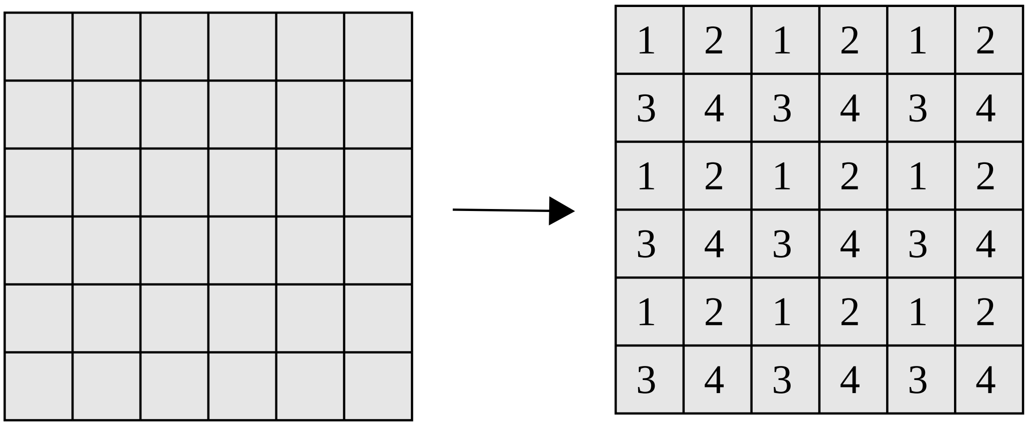


Figure 1: Placement cyclique bidimmensionel sur $4 = 2 \times 2$ processeurs d'une matrice partitionnée en 6×6 blocs.

Grâce à sa caractérisation sous forme d'une fonction close (avec modulo), un tel placement est explicite lors de la compilation. Ainsi, lors de l'exécution, l'indice du bloc suffit pour déterminer le processeur qui le possède par une fonction arithmétique peu coûteuse; toute lecture ou écriture d'un bloc est donc directement traduite en communication unidirectionnelle. Le surcoût de réalisation de l'ordonnancement est donc minimal. Ainsi, le benchmark LINPACK est ainsi basé sur une telle découpe : le nombre de processerurs et le réseau étant fixés, le programme prend en entrée les dimensions de la grille et l'algorithme de broadcast, qui peuvent être adaptés pour exploiter au mieux l'architecture.

Cependant, lorsque les blocs sont creux ou que la découpe en bloc est dynamique (algorithme récursifs par blocs par exemple), l'ordonnancement est calculé de manière dynamique. La réalisation d'un tel ordonnancement dynamique a un surcoût que les sections suivantes visent à limiter.

3 Dégénérescence séquentielle

3.1 Ordonnancement théorique sous-jacent

Sur machine SMP, les ordonnancements classiquement utilisés sont de type liste : ils consistent à affecter à un processeur inactif une tâche prête à être exécutée si il en existe. En effet, en supposant $T_\infty \ll T_1$ et si on néglige le coût de l'interprétation, un tel algorithme fournit un temps d'exécution asymptotiquement optimal comme le montre le résultat suivant. Ce théorème est important; preuve constructive du théorème de Brent, il justifie le fait de s'intéresser à des algorithmes parallèles de temps critique très faible.

Théorème 3.1 *[Gra69] Si l'on ne prend pas en compte le coût de calcul et de réalisation de l'ordonnancement, tout ordonnancement de type liste appliqué à un programme parallèle de coût T_1 et T_∞ conduit à un temps d'exécution T_p sur p processeurs majoré par :*

$$T_p \leq \frac{T_1}{p} + T_\infty$$

Preuve Nous rappelons la preuve donnée dans [Gra69, KR97] car son schéma est à la base des différents résultats présentés dans cette section. Soit t_1 une tâche qui s'est terminée à la date T_p, et soit d_1 la date du début de cette tâche. Avant d_1, deux situations peuvent être distinguées : i) soit aucun processeur n'a été inactif avant d_1; ii) soit au contraire il existe une date $d < d_1$ à laquelle au moins un processeur était inactif.
Dans ce cas ii), soit d' la plus grande de ces dates. L'ordonnancement étant de liste, si à la date d' la tâche t_1 avait été prête alors elle aurait débuté son exécution. Il existe donc une tâche t_2 en cours d'exécution à la date d' telle que $t_2 \prec_G t_1$. Soit d_2 la date de début d'exécution de cette tâche.

L'application récursive de ce schéma permet de construire une séquence $t_k \prec_G \cdots \prec_G t_2 \prec_G t_1$ de tâches telles qu'à tout instant de l'exécution, soit tous les processeurs sont actifs, soit un des processeurs est en train d'exécuter une des tâches de cette séquence. La durée de la première situation est majorée par $\frac{T_1}{p}$ et celle de la seconde par la durée d'un chemin critique du graphe, c'est-à-dire par T_∞. Λ

Cependant, le coût de la réalisation de cet ordonnancement doit aussi être pris en compte [KR97]. Il est a priori borné par le nombre n de tâches; comme $n > \frac{T_1}{T_\infty}$, le surcoût peut être considérable pour un programme de grain fin. Toutefois, la preuve montre que le nombre de tops d'inactivité est majoré par T_∞ sur chaque processeur; ainsi, si les processeurs sont capables de trouver facilement des tâches prêtes à exécuter lorsqu'il en existe, le surcoût d'ordonnancement, majoré par $O(pT_\infty)$, sera négligeable pour des programmes possédant un grand degré de parallélisme.

3.2 Réalisation, contention et verrous

Pour minimiser le surcoût lié à la réalisation de l'ordonnancement, on peut chercher à paralléliser la gestion de la liste des tâches, i.e. les opérations d'ajout et d'extraction (éventuellement avec priorité) dans un ensemble.

D'un point de vue algorithmique "PRAM", cette gestion peut être facilement parallélisée avec un coût $T_\infty = \Theta(\log n)$ et $T_1 = \Theta(n)$ [J92, BGMN97], toutefois au prix d'une augmentation conséquente du nombre d'opérations par rapport au temps d'une gestion séquentielle.

D'un point de vue algorithmique distribuée, la contention sur l'accès à la liste des tâches peut être diminuée en la distribuant entre les processeurs : chaque processeur gère alors localement sa propre liste de tâches. Lorsque cela est nécessaire (synchronisation entre tâches sur différents processeurs ou inactivité d'un processeur), un processeur peut accéder en exclusion mutuelle à la liste gérée par un autre. Cette synchronisation étant rare (la plupart des accès sont locaux comme nous le verrons plus tard), un verrou arithmétique (*spin-off* à partir d'une instruction *test&set* atomique) peut être utilisé pour implémenter cette exclusion mutuelle; la synchronisation peut même être implémentée essentiellement à partir de compteurs très peu coûteux si les deux processus accèdent à des parties distinctes de la liste, typiquement tête et queue [Dij65, FLR98].

3.3 Algorithmes de dégénérescence séquentielle en appel de fonction

Même avec l'élimination des verrous, le coût total de gestion reste proportionnel au nombre de tâches. Or, il est possible d'éviter simplement la gestion de certaines tâches prêtes en les exécutant directement sous forme d'appels de fonction. Cette technique, traditionnellement utilisée pour la compilation des langages fonctionnels [MKRHH91], est appelée "work-first principle" dans [FLR98] : elle consiste à mettre la plus grande partie du surcoût de réalisation de l'ordonnancement sur le chemin critique, i.e. lorsqu'un processeur devenu inactif doit voler une tâche à un autre processeur qui possède des tâches prêtes. Par rapport â la preuve du théorème 3.1, le surcoût est mis sur le terme en T_∞ plutôt que sur celui en T_1/p. Ainsi, pour des programmes très parallèles où T_∞ est négligeable devant T_1, le surcoût dû à l'ordonnancement est moindre.

La solution est donc de n'effectuer la gestion locale de tâches que lorsque cela est nécessaire, à savoir lors d'un vol. Pour cela, la création d'une tâche est compilée en appel de fonction locale. Cela nécessite une hypothèse fondamentale : toute tâche créée doit pouvoir être directement exécutée localement, donc être prête. Cette hypothèse est facilement vérifiée pour les langages offrant un parallélisme de type série-parallèle, comme les langages fonctionnels qui ont été les premiers à l'exploiter. Elle n'est cependant pas restrictive à ce type de parallélisme.

Le programme est alors *dégénéré en exécution séquentielle profondeur d'abord.* Le point critique est que cette dégénérescence ne doit pas entraîner de perte de parallélisme au niveau de l'algorithme.

```
void prog() {
   <i1> ;
  spawn f1( args1 ) ;
  <suite> ; // autres spawn...
  sync ; // attente des spawn précédents
  <fin>
}
```

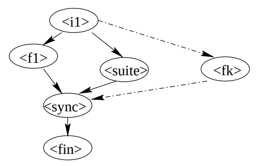


Figure 2: Exemple de programme série-parallèle strict.

Deux algorithmes implémentant une telle dégénérescence séquentielle pour un langage série-parallèle sont présentées. Elles sont illustrées sur le programme pseudo-Cilk de la figure 2, supposé de coût temporel théorique T_1, T_∞.

3.3.1 Dégénérescence séquentielle active

Interpétation `spawn/sync`

```
bool DegenerationSeq = true;
<i1> ;
// spawn f1(args1)    est    compilé    en    :
if  ExisteRequeteDeVol()  {
  DegenerationSeq = false ;
  t1 = CreationTache( f1, args1, s) ;
  Exporter(t1, voleur) ;
}
else {  // dégénérescence séquentielle
  f1 ( args1 ) ;
  <suite> //... idem pour autres spawn
}
// sync est compilé en :
if (DegenerationSeq) { <fin> }
else {
    tfin = CreationTache( <fin>, s ) ;
}
```

Algorithme lorsqu'un processeur devient inactif

```
(t,s) = FaireRequeteDeVol() ;
// t décrit une tâche et la
// synchronisation s correspondante
Executer( t ) ;
SignalerFinTache ( t, s ) ;
```

La synchronisation `sync` est implicite lors d'une dégénérescence séquentielle.
En cas de vol de tâche, elle peut être implémentée en associant deux compteurs à chaque point de synchronisation `s` : nombre de `spawn` exportés d'une part et nombre de fin de tâches exportées d'autre part.
La création de la tâche associée à la continuation `<fin>` peut être réalisée par clonage (`setjump/longjump`).

Figure 3: Dégénérescence séquentielle active.

Lors de la création potentielle d'une tâche, on teste si il existe un processeur inactif[1]. Si oui, on crée une tâche pour réaliser l'appel de fonction (généralement exporté vers le voleur) et une tâche (généralement conservée localement) pour la suite de façon à prendre en compte les synchronisations qui seront à faire avec la tâche exportée. Le code est explicité dans la figure 3.

L'inconvénient de cette solution est de ne détecter l'existence d'un processeur inactif que lors de la création d'une tâche (instruction `spawn`), donc avec retard par rapport à la date à laquelle le processeur voleur est devenu inactif. Ce délai étant majoré par T_∞, l'ordonnancement reste cependant asymptotiquement optimal si $T_\infty \ll T_1$ (cf théorème 3.2).

[1]La détection d'un processeur inactif et de son identité n'est pas détaillée; elle s'effectue à partir d'une liste des processeurs inactifs modifiée en exclusion mutuelle mais dont la taille est lue en concurrence.

Interpétation spawn/sync côté processeur actif

```
void prog_degenere () {
  frame* etat = alloca(..) ;
  etat.fn = &prog_reprise ;
  int co = 0 ; // compteur ordinal
  <i1> ;

  // spawn f1(args1) est compilé en:
  push( etat, ++co ) ;
     // sauvegarde minimale
  f1( args1 ) ;

  etat=pop() ;
  if ( etat.EstExportee() ) {
    // ...le calcul est donc terminé !
    SignalerFinTache ( etat ) ;
    return ;
  }
  // Ici l'exécution continue :
  // dégénérescence séquentielle.
  <suite> ;
    // sync est implicite !
  <fin>     }
```

Reprise côté processeur inactif

```
FaireRequeteDeVol(&etat, &co) ;
  // etat permet la duplication de
  // la frame et la reprise !
etat.fn( etat, co ) ;
```

Interpétation spawn/sync côté processeur inactif

```
void prog_reprise (frame* f, int co) {
  DupliquerFrame( f ) ;
  switch (co) {
  0: <i1> ;
     // code associé à spawn f1( args1 ) ;

  1: // reprise si spawn f1 a été volé
     <suite> ;
     // chaque spawn est interprété comme
     // du côté processeur actif.

     // sync implique la création d'une tâche :
     if (AttendreFinTache(f) != READY) {
       tfin = CreationTache(f,++co);
       return ;
     }
     <fin>
  } }
```

Figure 4: Dégénérescence séquentielle passive

3.3.2 Dégénérescence séquentielle passive

Cette stratégie permet de réduire le délai de détection d'un processeur inactif. Le principe est de laisser le processeur inactif créer lui-même la tâche qui sera exportée par le processeur actif. Pour cela, chaque instruction `spawn` construit les informations minimales pour indiquer un vol possible. Lorsqu'un processeur voleur vient chercher du travail, il fabrique à partir de ces informations minimales une tâche.

Aussi, dans cette solution, l'appel de fonction correspondant au `spawn` est réalisé localement tandis que la continuation (la suite des instructions effectuées après le `spawn`) est exportée. Ainsi, le code `prog` possède deux implémentations (figure 4) : l'une implémentant la dégénérescence séquentielle, l'autre permettant à un processeur inactif de reprendre le calcul à partir de la continuation.

Analyse des deux stratégies Le théorème suivant montre que ces deux stratégies sont asymptotiquement optimales.

Théorème 3.2 *Sur une architecture à mémoire partagée, en incluant le surcoût de contrôle de l'ordonnancement, le temps d'exécution T_p sur p processeurs d'un programme ordonnancé par dégénérescence séquentielle (active ou passive) est borné par :*

$$T_p \leq \frac{T_1}{p} + T_\infty + O(n).$$

Si de plus le programme est série-parallèle et le processeur volé tiré au hasard uniformément, alors T_p est majoré avec une bonne probabilité par :

$$T_p \leq \frac{T_1}{p} + O(p.T_\infty).$$

Preuve La preuve suit le même schéma que 3.1 en considérant que la liste des n tâches à exécuter est manipulée en exclusion mutuelle (cf [KR97]).
Dans le cas d'un programme série-parallèle, l'ordonnancement par dégénérescence séquentielle garantit qu'avec une bonne probabilité $O(T_\infty)$ vols sont effectués par processeur [BL98] (cf la preuve de 3.1). Avec une bonne

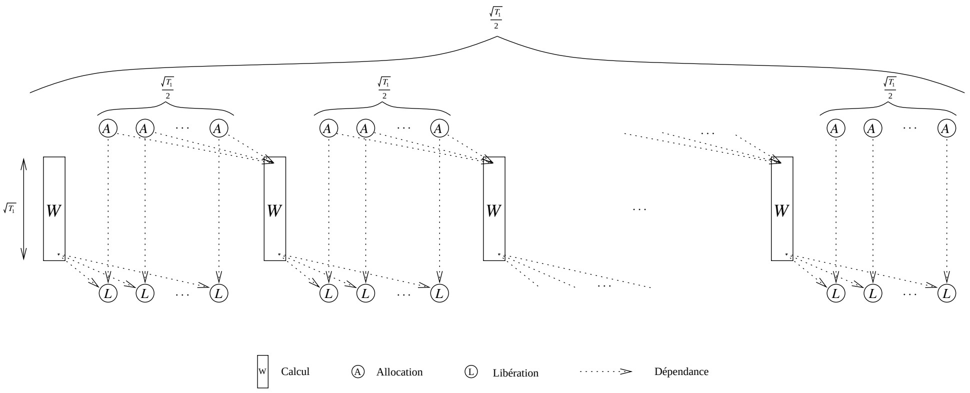


Figure 5: Programme dont l'exécution parallèle requiert de la mémoire [Gal99].
Ce programme, tiré de [Gal99] et inspiré de [BL98], comporte : $\frac{\sqrt{T_1}}{2}$ tâches W de durée $\sqrt{T_1}$; $\frac{T_1}{4}$ tâches d'allocation A chacune de durée 1 et allouant un espace mémoire S_1; $\frac{T_1}{4}$ tâches L de libération chacune associée à une tâche A. Le temps et l'espace mémoire séquentiels sont T_1 et S_1. Bien que $T_\infty = \sqrt{T_1}$, les contraintes de précédence sont telles que toute exécution parallèle vérifiant $T_p < \frac{T_1}{2}$ implique une consommation mémoire $S_p \geq \frac{\sqrt{T_1}}{2} S_1$.

probabilité, le nombre total de vols réalisés par les p processeurs est alors majoré par $O(p.T_\infty)$, d'où la deuxième majoration. Λ

Considérons la modélisation dynamique de l'exécution sous forme du graphe des appels de procédure. La dégénérescence active consiste à voler la tâche la plus profonde dans ce graphe à un instant donné; alors que la dégénérescence passive consiste à voler la plus haute qui est *a priori* de plus gros grain. En pratique le coût de l'exportation est relativement important; exporter les tâches de plus grosse granularité permet de masquer ce surcoût.

Par exemple, le langage Cilk [FLR98] implémente cette dégénérescence séquentielle passive. Dans le cas de ce langage, les frames sont chaînées entre elles sous forme d'une pile en cactus, structure classique que l'on retrouve classiquement dans des implémentations parallèles de Prolog [Kow88]. Dans le cas de Cilk, les synchronisations sont gérées par des compteurs arithmétiques.

3.4 Minimisation des prises de verrou

Les stratégies actives et passives visent à minimiser le surcoût de contrôle de l'ordonnancement en le restreignant au cas, rare si le chemin critique est négligeable devant le nombre d'opérations, où un processeur devient inactif. Dans ce cas, le processeur devenu inactif retire une tâche parmi celles prêtes générées par un processeur actif. Cette situation étant de type producteur-consommateur, l'utilisation coûteuse d'un verrou peut être souvent évitée pour protéger les accès conflictuels à la liste des tâches prêtes.

Ainsi, dans le cas du langage Cilk, un protocole appelé "THE" [FLR98], basé sur un verrou arithmétique, est proposé. Le processeur actif ajoute et retire les tâches prêtes dans sa liste selon une stratégie LIFO. En revanche, le processeur inactif vole toujours la tâche la plus ancienne (FO, first out). Il n'y a alors conflit d'accès à la liste que lorsque celle-ci est réduite à une seule tâche, seul cas où l'on a recours à l'utilisation d'un verrou. Dans le cas de Cilk où seuls des programmes série-parallèle sont considérés, cette situation s'avère rare sur les exemples évalués.

Dans la bibliothèque Hood [BP98], ce protocole est étendu au cas de programmes quelconques. L'exploitation de l'atomicité matérielle de certaines opérations sur une architecture donnée permet en effet de réaliser la synchronisation liée au vol de tâches de manière arithmétique, sans recours à un verrou système. Dans le cas de Hood, une implantation est proposée sur Sparc/SOLARIS.

3.5 Minimisation du surcoût mémoire

La diminution du temps d'exécution nécessite que plusieurs processeurs exécutent simultanément des instructions du programme donc requièrent ensemble potentiellement un plus grand espace mémoire qu'un unique processeur séquentiel. La figure 5 en donne un exemple extrême.

Cependant, l'ordonnancement du programme peut lui aussi influencer conséquemment sur la consommation mémoire. Considérons par exemple un programme comportant n tâches a_i allouant 1 mot chacune et n tâches b_i effectuant les libérations correspondantes; les seules contraintes de précédence soient du type $a_i \prec l_i$. Considérons les deux ordonnancements séquentiels suivants :

$$a_1 < b_1 < \ldots < a_i < b_i < \ldots < a_n < b_n \qquad (1)$$
$$a_1 < \ldots < a_i < \ldots < a_n < b_1 < \ldots < b_i < \ldots < b_n \qquad (2)$$

Le volume mémoire requis pour l'exécution correspondant à l'ordre (1) est de 1 mot, tandis que celui correspondant à l'ordre (2) est de n mots. Le volume mémoire requis dépendant fortement de l'ordre, le principe pour contrôler cette consommation lors d'une exécution parallèle est de suivre l'ordre séquentiel qui a été utilisé pour définir l'espace mémoire séquentiel S_1 qui sert de référence [BGMN97, NB97]. Il est alors possible de comparer S_p à S_1.

Théorème 3.3 *[BGMN97] Un ordonnancement de liste qui, parmi les tâches prêtes, affecte à un processeur inactif la plus prioritaire selon l'ordre séquentiel requiert un espace mémoire s majoré par :*

$$s \leq S_1 + pKT_\infty.$$

où S_1 est l'espace mémoire requis par l'exécution séquentielle de référence et K l'espace mémoire maximum alloué par une tâche.

Preuve Les tâches restant à exécuter sont stockées dans une liste triée selon l'ordre d'une exécution séquentielle[2]. Lorsqu'un processeur termine une tâche, il prend la tâche en tête de cette liste pour l'exécuter. Les tâches les plus vieilles selon l'ordre séquentiel sont donc prioritaires. La durée d'une tâche étant majorée par T_∞, à chaque instant au plus T_∞ tâches peuvent être exécutées en avance par rapport à l'ordre séquentiel sur chaque processeur. Chaque tâche allouant au plus un espace mémoire de K, l'espace mémoire total requis est majoré par $S_1 + pKT_\infty$. Λ

Même si elles sont basées sur une dégénérescence séquentielle, les sérialisations vues précédemment ne suivent pas cet ordre. Cependant, en restreignant le cadre d'applications, il est possible de majorer l'espcae mémoire requis. Par exemple, Cilk se restreint au cas d'un langage série-parallèle *strict* où les seules synchronisations sont entre une tâche et les fins des tâches qu'elle a directement créées. Dans ce cas, considérons la sérialisation passive : lorsqu'un processeur crée une tâche, il "stoppe" l'exécution en-cours pour exécuter cette nouvelle tâche par appel de fonction (parcours profondeur d'abord). Comme le programme est série-parallèle strict, à tout instant, il y a au plus p branches dans le graphe en cours de calcul. Or, l'espace mémoire pour chaque branche est majoré par S_1; l'espace mémoire total requis s est donc majoré par $s \leq p.S_1$.

4 Dégénérescence distribuée

Dans cette section nous considérons que le flot de données décrivant l'exécution est connu; après le calcul de son ordonnancement, la réalisation de cet ordonnancement sur une architecture distribuée doit minimiser le surcoût induit par les communications. Il s'agit donc ici d'interpréter une dépendance entre deux tâches adjacentes soit sous forme d'accès mémoire local si les tâches sont sur le même processeur, soit sous forme de communication (si possible uni-directionnelle) si elles sont sur deux processeurs distincts.

4.1 Ordonnancement théorique sous-jacent

Les algorithmes théoriques d'ordonnancement les plus classiquement utilisés sont ETF *Earliest Task First* ou ERT *Earliest Ready First* [HCAL89]. Ces deux ordonnancements sont de type liste. Ils consistent à ordonnancer la tâche qui pourra commencer son exécution au plus tôt (en prenant en compte les délais éventuels de communication) sur l'un des processeurs inactifs. Soit $P = \{P_1, \ldots, P_p\}$ les p processeurs. L'algorithme évalue donc, pour chaque couple $(r_i, p_j) \in R \times P$, la date de démarrage de la tâche r_i sur le processeur p_j, en considérant les délais de communication ainsi que la date de disponibilité de p_j. Soit (r_i, p_j) un couple associé à la plus petite date de démarrage; la tâche r_i est alors ordonnancé sur le processeur p_j.

Le théorème suivant évalue la performance d'un tel ordonnancement sur le modéle délai proposé dans [HCAL89] : le temps de communication de n mots entre deux processeurs est supposé prendre τn unités de temps.

Théorème 4.1 *[Liu90] Si l'on néglige le coût de calcul et de réalisation de l'ordonnancement, tout ordonnancement de type liste assignant une tâche prête sur le processeur qui démarrera son exécution au plus tôt conduit à un temps d'exécution T_p majoré par :*

$$T_p \leq \frac{T_1}{p} + (1 - \frac{1}{p})T_\infty + \tau C_{max}.$$

[2] L'insertion et la suppression dans la liste peuvent être effectuées en un nombre constant d'opérations par chaînage : les tâches filles sont insérées à l'ancienne position de la mère.

Il est à noter que la majoration utilise le fait que le graphe de flots de données est entièrement connu, ainsi que les durées des tâches et les volumes de communication. Cependant, ces algorithmes peuvent aussi être utilisés à la volée; dans ce cas, Doreille établit une majoration similaire : $T_p \leq \frac{T_1}{p} + (1 - \frac{1}{p})T_\infty + \tau C_{max^\star}$ [Dor99]. Par ailleurs, le coût du calcul de l'ordonnancement est $O(pn^2)$ [HCAL89].

Remarque. D'autres techniques sont utilisées en ordonnancement théorique pour réduire les communications. Parmi celles-ci, le *regroupement* de tâches consiste à regrouper sur un même processeur les tâches ayant de nombreuses dépendances. Ainsi, Pyrros [GY94] calcule à partir des précédences un premier ordonnancement sur un nombre non borné de processeurs virtuels pour minimiser le surcoût lié au délai; puis, pour le regroupement, les processeurs virtuels sont placés sur les processeurs physiques.
Une autre technique, la *duplication*, consiste à dupliquer une tâche sur plusieurs processeurs; ainsi les données écrites par cette tâche pourront être accédées localement sans communication sur ces processeurs. Bien que validée théoriquement, en particulier sur un nombre non borné de processeurs [CCLL95], l'utilisation pratique de la duplication au sein d'un langage de programmation et sur un nombre borné de ressources reste un problème ouvert.

4.2 Surcoût de réalisation de l'ordonnancement

Même avec un ordonnancement optimal, l'exécution sur une architecture distribuée nécessite des communications. Pour recouvrir les délais de communication par les calculs, la technique de virtualisation des processeurs (*parallel slackness*), consiste à simuler q processeurs virtuels sur les p processeurs physiques [KR97]. D'un point de vue pratique, les techniques de multiprogrammation légère (*multithreading*) permettent de réaliser une telle virtualisation.

Indépendamment de cette technique, l'interprétation des dépendances, soit en accès local, soit en communication pose un autre problème. Une solution [LFA96] est de placer les objets correspondants à des dépendances dans une mémoire virtuellement partagée. Lorsqu'une donnée est écrite, la page qui contient cette donnée est disponible sur le processeur P_W qui réalise l'écriture. Ainsi, si une autre tâche accède cette donnée en lecture sur ce même processeur, il est probable que la page sera encore en mémoire et donc l'accès possible sans communication.

L'inconvénient de cette solution est losque la tâche qui fait la lecture est exécutée par un processeur P_R différent de P_W. Dans ce cas, au minimum deux communications doivent alors être réalisées : une première de P_R vers P_W pour la demande de la page; l'autre en retour, de P_W vers P_R, pour l'envoi de la page. Cet aller-retour avec synchronisation intermédiaire introduit un délai important.

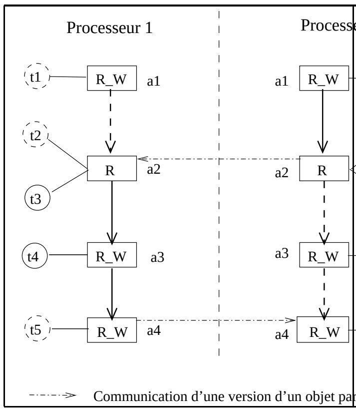
Cependant, supposons que l'ordonnancement a été calculé avant l'exécution des tâches. Lorsque P_W termine l'écriture, il sait donc que la donnée sera lue par P_R : il peut donc directement lui envoyer la donnée correspondante. Similairement, P_R sait (l'ordonnancement a été pré-calculé) qu'il possède une tâche t qui doit lire une donnée écrite par P_W. Il est donc prêt à recevoir une donnée de P_W (par exemple par dépôt asynchrone d'une attente de réception). Ainsi, l'accès à la donnée distante peut être réalisé par une seule communication unidirectionnelle et qui peut, de plus, être anticipée de manière asynchrone.

On obtient ainsi, par cette technique d'interprétation, un nombre de communications identique à celui requis par un programme équivalent écrit avec une biblliothèque de passage de messages comme MPI par exemple. Cependant, l'intérêt de l'interprétation est ici de limiter la taille du programme en entrée. En effet, l'ordonnancement pouvant a priori être quelconque, la taille du programme MPI peut être dans le pire cas égale à la taille du programme déroulé donc à la durée d'exécution du programme. Dans le cas de l'interprétation, le code reste de même longueur que le code source; par contre le codage en mémoire de l'ordonnancement est proportionnel au nombre de dépendances.

Cet algorithme d'interprétation est présenté et analysé dans la section suivante.

4.3 Algorithme de compilation à la volée des communications

L'interprétation précédente est basée sur la connaissance du graphe de flot de données et de son placement avant l'exécution des tâches qui constituent ce graphe. Cet algoritme d'interprétation est donc restreint aux applications régulières pour lesquelles le graphe de flot de données (et donc le parallélisme de l'application) peut être décrit de manière fine à partir de l'exécution d'une petite partie du programme sur les données en entrée. Cependant, elle s'étend directement à un cadre plus dynamique, notamment aux applications dans lesquelles le parallélisme est décrit par phases : l'exécution correspond à une succession de phases de description du parallélisme séparées par des phases d'exécution; la description d'une phase dépend des calculs effectués dans la phase précédente.



Sur le processeur 1, la tâche t_1 est prête mais comme elle a été placée sur le processeur 2, elle est retirée du graphe sans être exécutée. Pour t_3, le processeur 1 attend alors la version a_2 de l'objet modifié par t_1.
Sur le processeur 2, la tâche t_1 est exécutée puis retirée du graphe. Le nœud a_2 devient prêt; le graphe permet de de voir que le processeur 1 attend la version a_2 de l'objet partagé pour exécuter t_3. Le processeur 2 peut alors dès la terminaison de t_1 envoyer a_2 sur le processeur 1.
Le même mécanisme est appliqué lors de la terminaison de t_4. Le processeur 1 peut alors immédiatement envoyer a_4 au processeur 2 pour t_5.

Figure 6: Interprétation d'un programme comportant 5 tâches sur 2 processeurs[Dor99].
Les cercles représentent les tâches (en pointillé si la tâche est placée sur un autre processeur) et les rectangles les données en assignation unique (chacune correspond à une version d'un objet a).

Pour simplifier, une seule phase est considérée ici : on suppose qu'une tâche t_0 génère l'ensemble du flot. La figure 6, tirée de [Dor99], illustre sur un exemple simple cette interprétation.

L'exécution du programme est alors divisée en quatre étapes distinctes :
1. Exécution de la tâche racine t_0 et construction d'un graphe G décrivant le flot de données.
2. Calcul d'un ordonnancement de G (par exemple avec ETF/ERT). Ainsi, dans G, chaque tâche (resp. chaque donnée en assignation unique) est alors annotée par le processeur qui en réalise l'exécution (resp. l'écriture).
3. Diffusion de G annoté sur les processeurs.
4. Interprétation locale de G sur chaque processeur qui exécute les tâches qui lui ont été affectées.

L'étape 4 d'interprétation utilise activement le fait que le graphe est dupliqué sur chaque processeur. Chaque processeur gère ses propres tâches, qu'il ordonnance localement par un algorithme de liste. Pour chaque donnée écrite sur un autre processeur et lue par au moins une tâche localement, une requète de réception asynchrone est postée; en effet, chaque donnée en assignation unique peut être facilement identifiée grâce à un numéro unique dans G (*tag*). Ce mécanisme de réception asynchrone permet d'anticiper les communications.
L'algorithme d'interprétation sur chaque processeur est alors le suivant :

```
while IlResteDesTachesLocalement() {
  EffectuerReceptionsPretes() ;
  R = MettreAjourLaListeDesTachesPretes() ;
  while ! R.empty() {
    t = R.pop() ;
    Executer(t) ;
    EnvoiEcritures() ; // les écritures sont interprétées comme des
                       // envois asynchrones (avec tag) vers les processeurs
                       // qui ont posté des requêtes de lecture.
} }
```

Ce mécanisme d'interprétation conduit à la borne suivante sur le temps d'exécution.

Théorème 4.2 *[Dor99] En incluant le contrôle de l'ordonnancement, le temps d'exécution T_p sur p processeurs d'une machine à mémoire distribuée (modèle délai) d'un programme dans lequel seule la tâche racine crée d'autres tâches est borné par :*

$$T_p \leq \frac{T_1}{p} + T_\infty + \tau C_{max} + O(pn^2) + D(n)$$

où $O(pn^2)$ majore le calcul arithmétique du graphe G et de son ordonnancement (sans communication) et $D(n)$ est le temps de diffusion de G sur chacun des processeurs.

Preuve. Dans l'interpétation précédente, les communications sont unidirectionnelles; la borne τC_{max} sur le surcoût de communication est donc valide. Le surcoût lié à l'ordonnancement est majoré par : le calcul de l'ordonnancement $O(pn^2)$; la diffusion du graphe annoté $D(n)$; enfin le calcul local sur chaque processeur des tâches prêtes $O(n)$. D'où la majoration du surcoût $O(pn^2) + D(n)$ qui vient s'ajouter à la longueur de l'ordonnancement (théorème 4.1). Λ

Ainsi, cette interprétation distribuée permet d'exécuter un programme parallèle de grain fin (donc *a priori* portable) de manière aussi efficace que si il avait été ordonnancé explictement dans un code SPMD pour p

processeurs (écrit en MPI par exemple). Aussi, on retrouve le principe de cette dégénérescence distribuée au coeur de la définition et de l'implémentation de nombreuses interfaces de programmation parallèle, en particulier Jade [Rin98], Rapid [YF99] et Athapascan-1 [GRCD98]. Dans le cas d'Athapascan-1, l'apport expérimental de la dégénérescence distribuée est considérable comme en témoignent les mesures réalisées sur un IBM SP1 à 16 processeurs pour une factorisation dense de Cholesky (fig. 7).

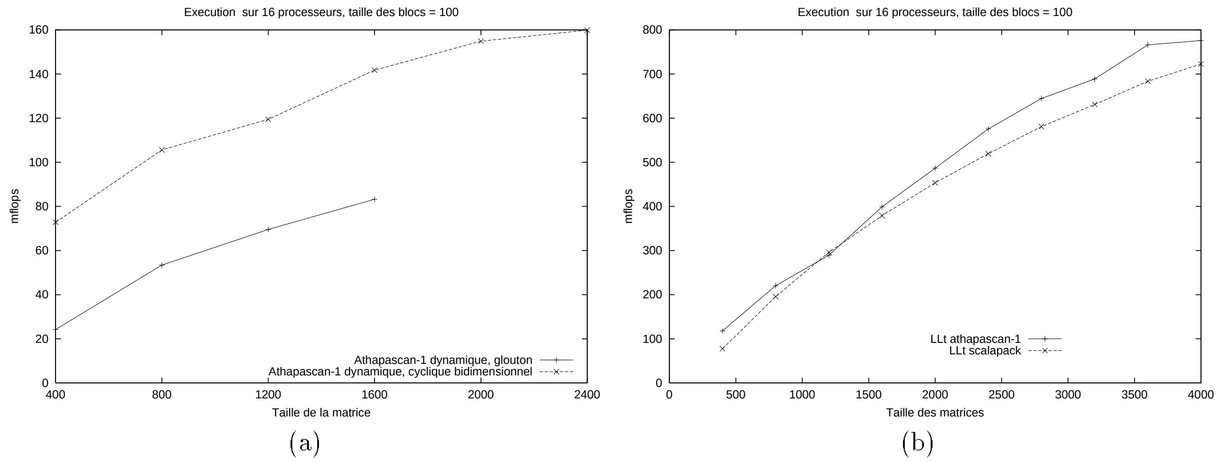


Figure 7: Factorisation de Cholesky sur IBM SP1 avec 16 processeurs d'une matrice dense partitionnée en blocs carrés de dimension 100 : (a) sans dégénérescence distribuée; (b) avec dégénérescence distribuée et comparaison avec ScaLapack) .

5 Ordonnancement sur grappe de SMPs

Dans le cas d'une grappe de SMP, les deux dégénérescences séquentielles et distribuées peuvent être couplées pour replier le parallélisme du programme sur les deux niveaux de parallélisme :

- dégénérescence distribuée : le programme est interprété et ses tâches réparties sur les différents noeuds de l'architecture.
- multi-threading local : sur chaque noeud SMP, les tâches sont ordonnancées sur des threads pour exploiter le parallélisme physique (threads système) et recouvrir les délais de communication par du calcul (threads utilisateur).
- dégénérescence séquentielle : sur chaque noeud SMP, l'interprétation du programme permet de minimiser le surcoût local d'ordonnancement et de limiter l'explosion mémoire dûe au parallélisme.

Grâce à ces techniques, les algorithmes parallèles de grain fin ($T_\infty \ll T_1$) mais qui ne génèrent pas un nombre de tâches trop conséquent ($n \ll T_1$) peuvent être ordonnancés efficacement sur des architectures de type grappes homogènes pour autant que le nombre de processeurs reste petit devant $\frac{T_1}{T_\infty}$ (*work-first principle*) et n (*parallel slackness*).

L'implantation sur grappes du systéme Athapascan est basée sur un tel couplage [CDR98] (Fig. 8). Le graphe de flot de données associé à l'exécution est construit par intepprétation, avec des annotations de coût fournies par le programmeur. Le nombre des noeuds étant connus (on suppose ici ceux-ci identiques pour simplifier, une extension par placement proportionnel pour des processeurs uniformes est proposée dans [Dor99]), un ordonnancement ETF de ce graphe est calculé. Le programme est alors ordonnancé par dégénérescence distribuée entre les noeuds selon l'algorithme §4.3. Ensuite, au niveau de chaque noeud multiprocesseur, un ordonnancement glouton est réalisé localement par dégénérescence séquentielle: les créations de tâches sont interprétées sous forme d'appel de fonction selon une dégénérescence séquentielle active par l'algorithme présenté en §3.3.1.

La figure 9 compare les performances obtenues par couplage sur une grappe comportant 16 noeuds quadri-processeurs à celles obtenues avec un placement bidimensionnel cyclique sur les 64 processeurs.

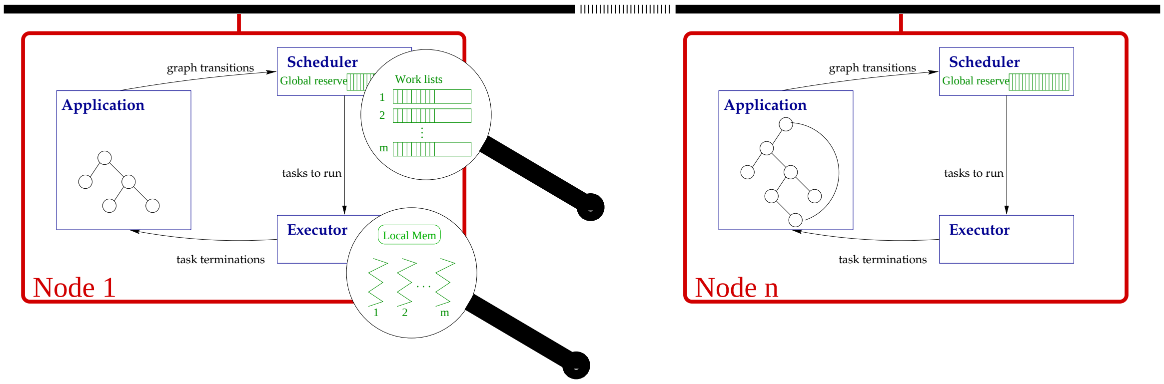


Figure 8: Mise en oeuvre d'un ordonnancement couplé dans le système Athapascan-1. Les tâches sont ordonnancées entre les noeuds par dégénérescence distribuée; sur chaque noeud SMP, les tâches sont ordonnancées par dégénérescence séquentielle.

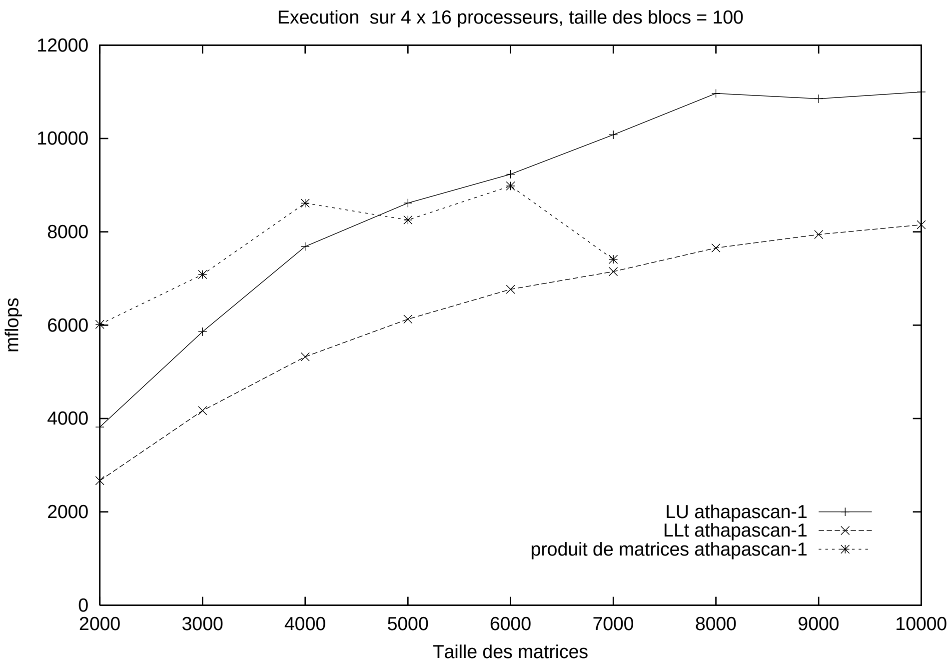


Figure 9: Ordonnancement par couplage des dégénérescences distribuées (ETF) et séquentielles sur un réseau de 16 SUN UltraSparc quadri-processeurs; comparaison avec un ordonnnancement bloc-cyclique bidimensionnel sur les 64 processeurs.

6 Conclusion

Automatiser l'ordonnancement d'un programme permet l'écriture de codes portables, indépendants de l'architecture. Nous montrons deux techniques, dégénérescence séquentielle d'une part et distribuée d'autre part, qui permettent de limiter le surcoût de la réalisation de cet ordonnancement. Un couplage de ces deux techniques est proposé; les expérimentations menées sur des grappes homogènes en contexte mono-utilisateur sur des problèmes d'algèbre linéaire permettent de valider l'approche d'un point de vue pratique.

Ces dégénérescences correspondent au couplage entre deux ordonnancements. Ainsi, la dégénérescence séquentielle est un couplage entre un ordonnancement par vol de travail d'une part et un ordonnancement séquentiel d'autre part. De même, la dégénérescence distribuée correspond à un couplage entre un ordonnancement statique quelconque et un ordonnancement qui peut être décrit par une fonction simple de taille constante (fonction de placement).

De part la complexité intrinsèque du problème de l'ordonnancement, non ϵ-approximable même sur un modèle de machine distribuée élémentaire à 2 processeurs, différentes heuristiques peuvent être utilisées. Le choix d'une heuristique donnée pour une tâche est alors fait en cours d'exécution, en fonction de l'état courant, i.e. les caractéristiques du programme et de la machine qui de plus en plus est une grappe hétérogène partagée entre plusieurs utilisateurs.

Ainsi, au sein d'une même exécution, différents ordonnancements peuvent être potentiellement utilisés. Le couplage générique entre deux ordonnancements quelconques est alors une technique qui permettrait d'optimiser l'exécution d'un programme, en généralisant les techniques présentées ici.

Références

[ABB+95] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM, Philadelphia, PA, 1995.

[BGMN97] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space efficient scheduling of Parallelism with Synchronization Variables. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures.* ACM Press, June 1997.

[BL98] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[BP98] Robert D. Blumofe and Dionisos Papadopoulos. HOOD: A User-level Threads Library for Multiprogrammed Multiprocessors. Technical Report http://www.cs.utexas.edu/users/hood/, The University of Texas at Austin, 1998.

[CCLL95] P. Chretienne, E.G. Jr Coffman, J. K. Lenstra, and Z. Liu. *Scheduling Theory and its Applications.* John Wiley and Sons, England, 1995.

[CDR98] Gerson G. H. Cavalheiro, Yves Denneulin, and Jean-Louis Roch. A general modular specification for distributed schedulers. In *Proc. of EuroPar'98, Southampton.* Springer-Verlag LNCS 980, Sep. 1998.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[Dor99] Mathias Doreille. *Athapascan-1 : Vers un modèle de programmation parallèle adapté au calcul scientifique* . PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, dec 1999.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization.* Birkhäuser, 2000.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, June 1998.

[Gal99] Franççois Galilée. *Athapascan-1 : Interprétation distribuée du flot de données d'un programme parallèle.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, sept 1999.

[GPS90] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.

[Gra69] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–426, 1969.

[GRCD98] François Galilée, Jean-Louis Roch, Gerson Cavalheiro, and Matthias Doreille. Athapascan-1: On-line Building Data Flow Graph in a Parallel Language. In IEEE, editor, *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris, France, October 1998.

[GY94] Apostlos Gerasoulis and Tao Yang. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, Sep 1994.

[HCAL89] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[J9́2] J. Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Massachussets, 1992.

[KGGK93] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings, 1993.

[KLadH96] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. *Algorithmica*, 16:517–542, 1996.

[Kow88] J.S. editor Kowalik. *Parallel Computation and Computers for Artificial Intelligence.* Kluwer Academic Publishers, 1988.

[KR97] Jean-Claude Konig and Jean-Louis Roch. Machines virtuelles et techniques d'ordonnancement. In *Proceedings école d'hiver de PRS ICARE'97*, Auusois, France, dec 1997. http://www-apache.imag.fr/ jlroch/ps/97-sched-aussois.ps.gz.

[LFA96] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing. *IEEE Journal of Parallel and Distributed Computing*, Aug 1996.

[Liu90] Zhen Liu. A note on Graham's bound. *Information Processing Letters*, 36:1–5, 1990.

[MKRHH91] Eric Mohr, David A. Kranz, and Jr Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):263–280, July 1991.

[NB97] Girija J. Narlikar and Guy E. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, June 1997.

[Rin98] M.C. Rinard. The design, implementation and evaluation of Jade : a portable, implicitly parallel programming language. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.

[WLMR93] M.H. Willebeek-Le-Mair and P. Reeves. Strategies for dynamic load-balancing on higly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.

[YF99] Tao Yang and Cong Fu. Space/Time-Efficient Scheduling and Execution of Parallel Irregular Computations. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.