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 exécutées 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 tout au long du chapitre 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.

```
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);
    }
}
```

$$\int_a^b f dx = \int_a^{\frac{(a+b)}{2}} f dx + \int_{\frac{(a+b)}{2}}^b f dx$$

$$\forall |b-a| < h, g(a,b) = \int_a^b f dx$$

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 de toutes les données situées en mémoire partagée, chaque état étant le résultat de l'intervention d'une tâche (synthèse ou modification).

Dans la figure 9.2 sont représentés les graphes de précédence 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-

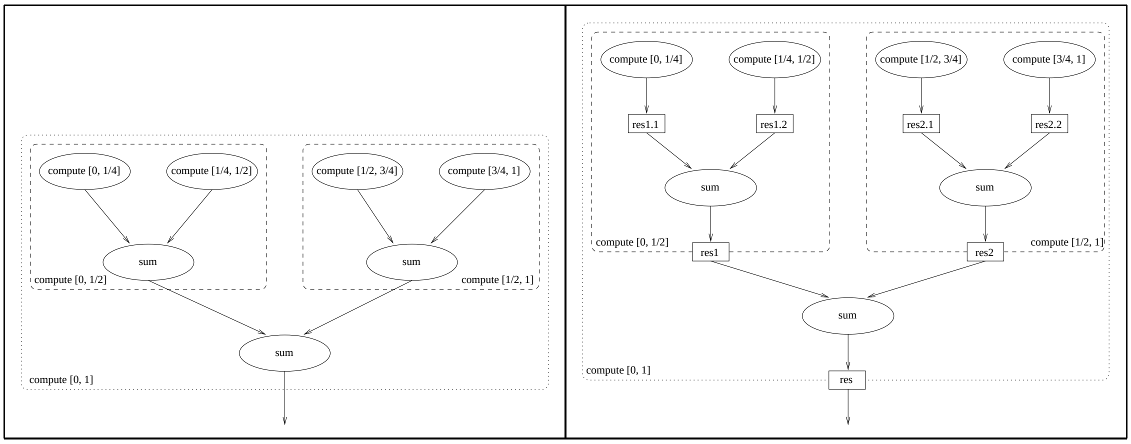

Figure 9.2 : Graphe de précédence et graphe de flot de données. La différence fondamentale se situe au niveau des synchronisations. Dans le cas d'un graphe de précédence une tâche est prête si toutes les tâches qui la précèdent sont terminées, tandis que dans le cas d'un graphe de flot de données une 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].

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 (cela arrive typiquement 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 en Cilk.

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 parallèles 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);
    }
}
```

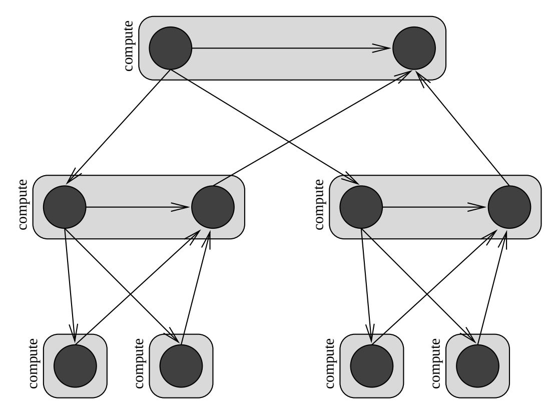

Figure 9.3 : Code Cilk et graphe d'exécution ($a = 0$, $b = 1$ et $h = \frac{1}{4}$) du calcul de l'intégrale par Newton-Cotes. Les threads, représentés par des cercles, sont regroupés en procédures, représentées par des rectangles grisés. Les flèches vers le bas indiquent une création de procédure (conséquence 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.

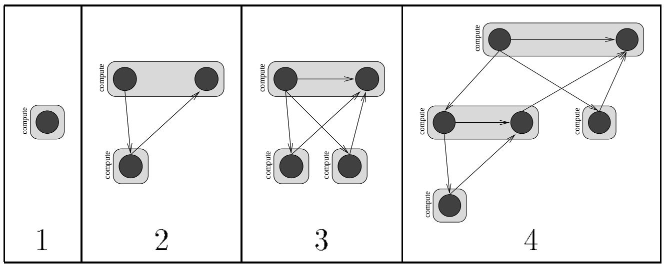

Figure 9.4 : La génération, dans Cilk, du graphe de précédence des threads est dynamique et distribuée. À l'étape 1 la première tâche est créée. Cette tâche crée aux étapes 2 et 3 deux nouveaux threads de calcul (correspondant respectivement aux données `res1` et `res2`) et se synchronise sur l'attente 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 données. 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);
    }
}
```

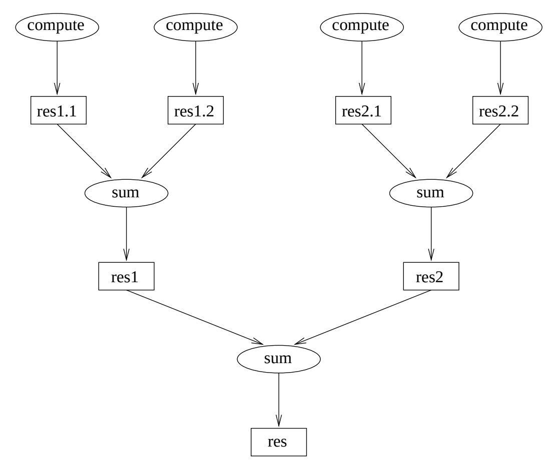

Figure 9.5 : Code ATHAPASCAN-1 et graphe d'exécution ($a = 0$, $b = 1$ et $h = \frac{1}{4}$) du calcul de l'intégrale par Newton-Cotes. Ce sont les accès aux données situées en mémoire partagée qui définissent implicitement les synchronisations entre les tâches.

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 pour 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 ressources 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.

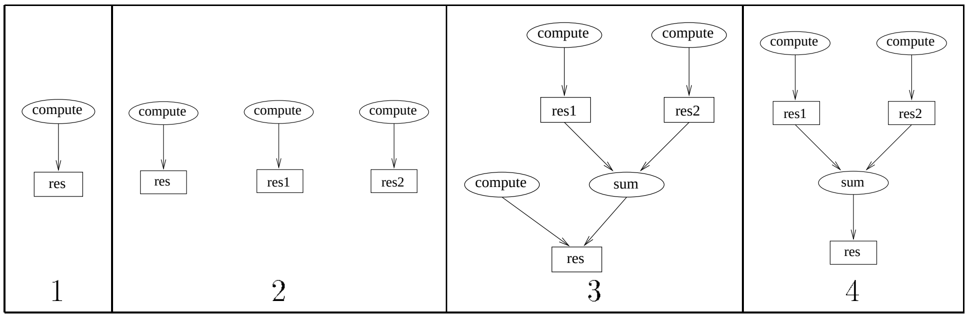

Figure 9.6 : La génération, dans ATHAPASCAN-1, du graphe de flot de données est dynamique et distribuée. À l'étape 1 la première tâche est créée, pointant en écriture sur une donnée en mémoire partagée où sera stoquée la donnée. Cette tâche crée à l'étape 2 deux nouvelles tâches de calcul qui stoqueront 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 à `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. 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 en Jade.

```
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);
    }
}
```

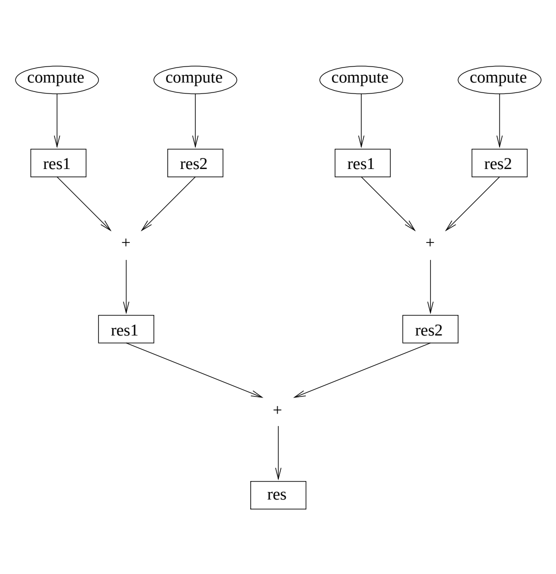

Figure 9.7 : Code Jade et graphe d'exécution ($a = 0$, $b = 1$ et $h = \frac{1}{4}$) du calcul de l'intégrale par Newton-Cotes. Ce sont les accès aux données situées en mémoire partagée qui définissent les synchronisations entre les tâches.

Jade maintient un graphe de type flot de données 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.

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 la sémantique [1] 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.

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.

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 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 (typiquement 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).

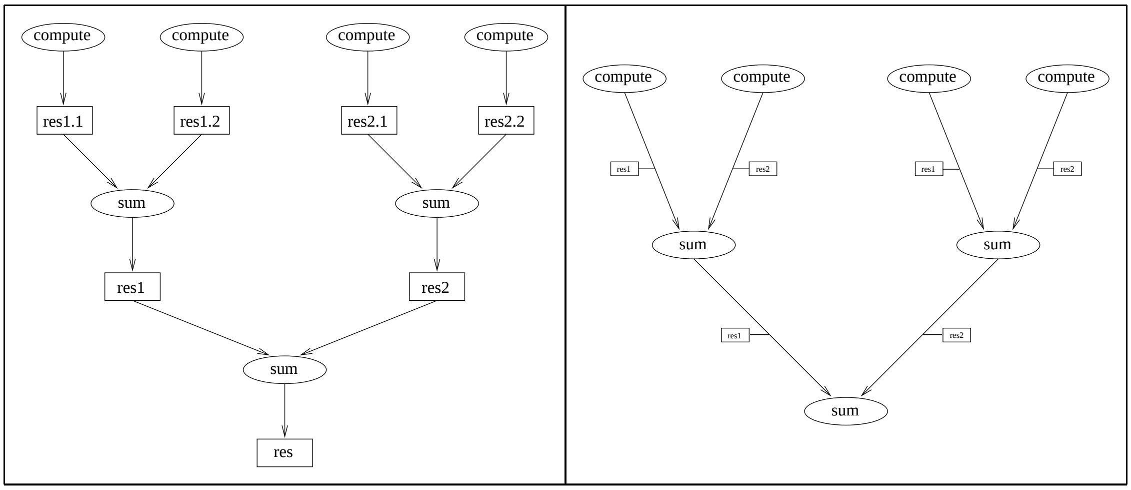

Figure 9.8 : Génération du graphe de précédence adapté à l'ordonnancement à partir du graphe de flot de données en Athapascan-1.

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, cette tâche accède en lecture au moins 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 pris 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 globale 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.