N° d'ordre : 130
N° bibliothèque : 99 ENSL 0130

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de
Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de la formation doctorale de : Informatique de Lyon

Allocation de graphes de tâches paramétrés et génération de code

par Emmanuel JEANNOT

Date de soutenance : 8 octobre 1999

Après avis de : Paul Feautrier
Jean-Claude König

Devant la commission d'examen formée de :

Michel Cosnard
Paul Feautrier
Apostolos Gerasoulis
Jean-Claude König
Yves Robert (président)

À Sylvaine

Remerciements

Je tiens à remercier chaleureusement les membres du jury :

- Yves Robert qui m'a fait l'honneur de présider ce jury ;
- Paul Feautrier qui a accepté d'être rapporteur de cette thèse. Ses remarques ont été un encouragement et m'ont permis d'améliorer le manuscrit ;
- Jean-Claude König pour également avoir accepté d'être rapporteur de cette thèse. Ses remarques m'ont permis de renforcer certaines parties de cette thèse ;
- Apostolos Gerasoulis pour avoir examiné mes travaux et m'avoir accueilli dans son équipe ;
- Michel Cosnard pour m'avoir guidé depuis mon DEA. Malgré des charges administratives importantes, il a toujours su être disponible. Ses conseils et sa confiance m'ont permis de mener à bien ces travaux : je lui dois beaucoup.

Je tiens aussi à remercier les thésards du LIP avec qui j'ai partagé ces années : Christian Perez, Ahmed Mostefaoui, Georges Silber, Nicolas Schabannel, André Elisseef, Olivier Bournez, Frédéric Prost et tous les autres que j'oublie.

Je remercie aussi Tao Yang avec qui j'ai eu grand plaisir de collaborer sur l'allocation symbolique. Ses conseils et ses idées m'ont été d'une grande aide.

Un remerciement spécial aux concepteurs de PM2 pour avoir assurer le support du logiciel : Jean-François Méhaut (maintenant supporteur de l'OL) et Raymond Namyst (photographe émérite).

Je remercie les autres membres du LIP avec qui j'ai eu grand plaisir de travailler : Luc, Sylvie, Marie, Jean-Christophe, Reynald et bien d'autres.

Je remercie enfin mes parents et amis pour leur soutien et en particulier Sylvaine et Yannick pour la relecture du manuscrit.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Contexte : le parallélisme

L'arrivée, ces dernières années, des réseaux rapides atteignant ou dépassant le débit de 100 Mo/s (Myrinet, Fast-Ethernet, ATM), pour des coûts très abordables, a permis de faire émerger un nouveau type de machines parallèles à mémoire distribuée : les piles. Essentiellement constituées d'un ensemble de cartes mères du marché reliées entre elles par un réseau rapide, les piles offrent de nombreux avantages. On peut installer sur chaque nœud un système standard de style LINUX, l'intégration matérielle est moins compliquée que pour la fabrication des machines parallèles à mémoire distribuée du type *constructeur*, l'utilisation de composants standards assure un moindre coût de fabrication et de maintenance, enfin l'utilisation de réseaux et de processeurs rapides assure de très bonnes performances. Les piles sont donc d'un coût moins élevé, pour des performances correctes, que les machines à mémoire distribuées fabriquées par des constructeurs. Cela permet de rendre accessible des plateformes parallèles à un plus grand nombre d'utilisateurs.

Si la démocratisation des machines parallèles s'accroît de jours en jours, la programmation de celles-ci réclame cependant de nombreux efforts. Pour chaque application, il faut déterminer les calculs qui peuvent être menés concurremment, les répartir sur les processeurs, organiser les communications éventuelles. Une fois menées à bien, ces opérations permettent un gain aussi bien en terme de rapidité d'exécution, qu'en terme de taille des problèmes traités. Paralléliser une application est une tâche longue et difficile qui requiert de l'expertise et une analyse poussée du programme.

De nombreux travaux ont été menés pour aider la conception d'application parallèles. Parmi ceux-ci citons les langages à parallélisme de données(HPF [52]), les langages à parallélisme de tâches (tels que celui qui est utilisé dans Pyrros [74] ou PlusPyr [57]), les bibliothèques d'algèbre linéaire (ScaLAPACK [10]),les bibliothèques de communications (PVM [43], MPI [49], etc ...), ou enfin les compilateurs paralléliseurs (par exemple SUIF [4]).

Dans cette thèse, nous étudions et proposons des algorithmes et des outils d'aide à la parallélisation de noyaux de calculs intensifs réguliers que l'on trouve dans les applications scientifiques. Les aspects de notre étude portent plus particulièrement sur les thèmes décrits ci-après.

1.1.1 L'ordonnancement

Le graphe de tâche est un modèle de calcul où un programme est représenté sous forme d'un graphe acyclique. Les sommets représentent un ensemble d'instructions à exécuter en séquentiel appelé tâche. Les arcs représentent les dépendances entre tâches, le plus souvent dues aux communications. Ce modèle est très utilisé pour la prédiction de performance et l'optimisation d'applications parallèles. Un des problèmes fondamentaux du parallélisme consiste à ordonnancer les graphes de tâches. Il s'agit de déterminer pour chaque tâche un processeur et une date de début d'exécution de manière à optimiser certains critères (temps total d'exécution, équilibrage de charge, etc ...). Il s'agit d'une phase critique si l'on souhaite obtenir de bonnes performances.

1.1.2 La parallélisation (semi-)automatique

Pour décharger le programmeur de la réalisation d'un programme parallèle, une solution consiste à transformer automatiquement un programme séquentiel en un programme parallèle. Un compilateur essaye de prendre en charge toutes les étapes nécessaires à la parallélisation. En particulier, il doit extraire le parallélisme et placer les données et les calculs. Les techniques actuelles fonctionnent très bien lorsqu'il s'agit de programmes réguliers à base de nids de boucles ou pour des machines à mémoire partagée.

1.1.3 La génération de code

La génération de code parallèle pour l'architecture cible est la dernière phase d'un compilateur paralléliseur. Elle peut aussi être accomplie à partir d'informations sur le programme extraites manuellement ou semi-automatiquement. En particulier, la génération de code est grandement facilitée par la présence de directives ou de mots-clés au sein du programme source. En général, le code généré fait appelle à des routines de communications issues de bibliothèques standards ou dédiées à la machine cible.

1.2 Motivations

1.2.1 Les problèmes liés aux techniques d'ordonnancement statiques

Un outil d'ordonnancement statique fonctionne de la manière suivante. Il prend en entrée le graphe de tâche qui correspond à l'instance du problème à analyser. Il l'ordonnance et fournit une table qui, pour chaque tâche décrit le processeur sur lequel elle doit s'exécuter et à quelle date. Cette méthode comporte deux désavantages importants. On est confronté à un problème de mémoire lorsque l'on veut ordonnancer de grands graphes et à un problème d'adaptativité lorsque l'on change de machine cible ou de taille des données.

Le coût mémoire

La taille du graphe de tâches est fonction de la taille du problème qu'il modélise. En général, pour un même programme, plus les données en entrée sont importantes, plus le

graphe de tâches aura de sommets et d'arcs. Par exemple, pour le programme de l'élimination de Gauss décrit dans l'annexe, le graphe de tâches a environ 500 000 tâches et 1 000 000 d'arcs pour une matrice de taille 1000 et 2 000 000 de tâches et 4 000 000 d'arcs pour une matrice de taille 2000.

Il existe donc une taille problème limite au delà de laquelle, sur une machine donnée, par manque de mémoire, un programme d'ordonnancement statique ne peut pas fonctionner.

L'adaptativité

Chaque fois que l'on change de machine ou que l'on change la taille du problème le graphe de tâches est modifié. Il est alors nécessaire de recommencer l'ordonnancement. En effet, ordonnancer un graphe de tâches est un problème NP-complet dans le cas général. Une bonne solution pour un graphe donné n'est en général d'aucune utilité si on change un tant soit peu ce graphe. Ce problème devient critique lorsque pour des graphes importants le temps d'ordonnancement est long.

1.2.2 Les techniques de parallélisation automatiques

Le grain

Les techniques de parallélisation automatiques développées jusqu'à aujourd'hui ont comme caractéristiques de mener une analyse à grain fin. Travailler au niveau des instructions présente l'avantage d'avoir un coût fixe pour les calculs et les communications. Des techniques de restructuration de code, de tuillage, peuvent ensuite être mise en place pour augmenter le grain. Il n'existe pas de travaux qui partent directement d'une représentation en tâches du programme et qui l'analysent pour placer ces tâches sur les processeurs.

La génération de code

Il existe quelques outils qui permettent de générer du code dans le modèle graphe de tâches. Le code généré est basé sur l'ordonnancement du graphe. Le programme n'est alors pas générique puisque chaque fois que la taille des données change, il faut recalculer un ordonnancement et donc regénérer le code.

1.3 Organisation de la thèse

Notre travail est basé sur un modèle symbolique des graphes de tâches modélisant certains noyaux de calculs intensifs : le graphe de tâches paramétré (GTP). Il s'agit d'une représentation compacte indépendante de la taille du problème de certain graphes de tâches. Il utilise des paramètres qui une fois instanciés permettent de construire entièrement le graphe de tâches. Le graphe de tâches paramétré peut être construit automatiquement pour certains programmes annotés par PlusPyr, un prototype développé par Loi durant sa thèse. Le graphe de tâches paramétré ainsi que les autres modèles et outils utilisés sont décrits dans le chapitre 2.

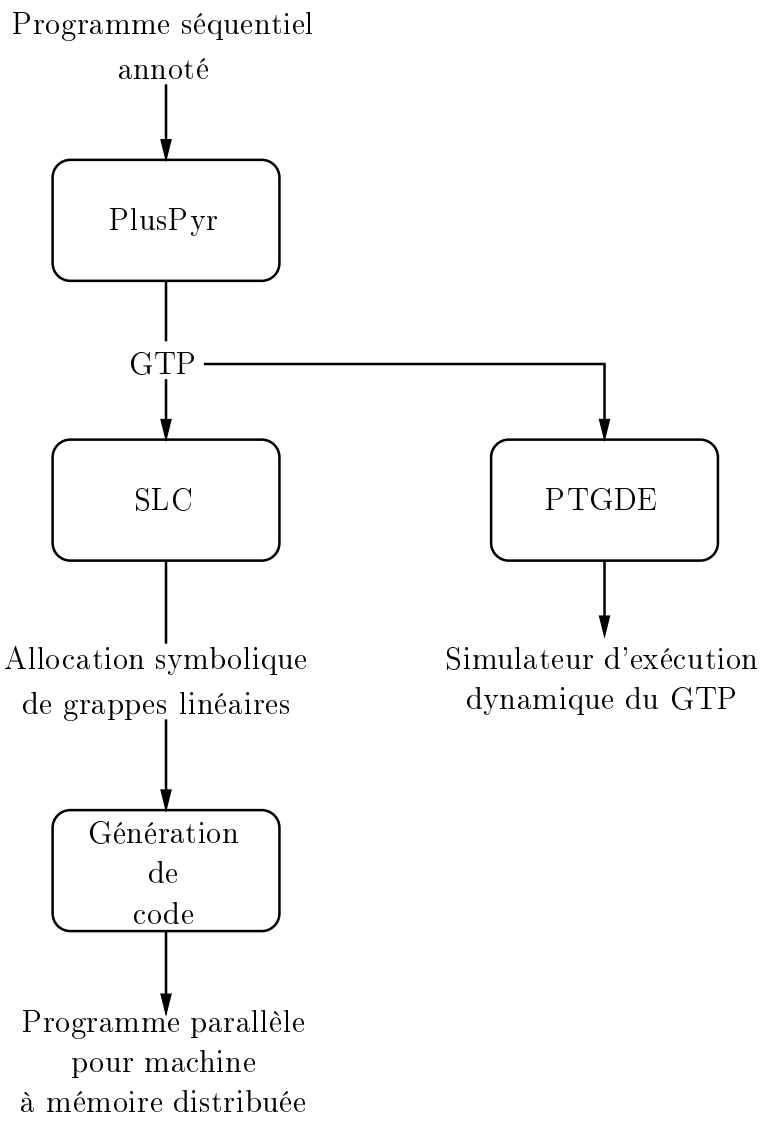


FIG. 1.1: *Articulation des différents travaux de cette thèse*

Dans le chapitre 3 nous présentons et étudions un algorithme d'ordonnancement basé sur le GTP. Le principe est que le graphe de tâches est construit et ordonnancé localement grâce au GTP. Ainsi seule une petite partie du graphe est en mémoire à un instant donné. Cela permet d'ordonnancer des problèmes de grande taille. Nous avons intégré cet algorithme dans un système dynamique appelé PTGDE (Parameterized Task Graph Dynamic Execution). Dans un tel système, la valeur des paramètres est donnée à l'exécution ce qui permet de construire un programme générique qui fonctionne pour toutes les tailles de problèmes.

Dans le chapitre 4 nous présentons SLC (Symbolic Linear Clustering), un algorithme qui utilise uniquement le GTP pour allouer symboliquement les tâches. Le temps d'allocation et le coût mémoire sont alors indépendants de la taille du problème. Cet algorithme calcule une fonction qui donne pour chaque tâche le processeur sur lequel elle doit être exécutée. Cette

méthode garantie que les tâches sont regroupées en grappes linéaires, technique connue pour son efficacité avec de telles applications.

Dans le chapitre 5 nous présentons un prototype de générateur de code. Basé sur l'allocation symbolique, le code, tel qu'il est produit, fonctionne pour toutes les valeurs des paramètres et l'allocation de chaque tâche est calculée en temps constant. Nous décrivons les optimisations nécessaires pour obtenir de bonnes performances et nous évaluons les codes générés sur des machines parallèles à mémoire distribuée.

La figure 1.1 montre comment les travaux décrits dans ce manuscrit s'articulent.

Les conclusions et les perspectives de ces travaux sont données chapitre 6, tandis que les noyaux de calculs intensifs utilisés comme exemples sont décrits dans l'annexe.

Chapitre 2

Modèles et outils

Ce chapitre est composé de deux parties. Dans la première partie nous décrivons et étudions les différents modèles qui seront utilisés tout au long de ce manuscrit. Ces modèles sont :

- les machines parallèles. Pour pouvoir ordonnancer un graphe de tâches il est nécessaire de modéliser les machines sur lesquelles le programme va être exécuté. La modélisation doit être suffisamment fine pour pouvoir obtenir un résultat cohérent avec la réalité. Nous étudierons le modèle des machines à mémoire distribuée,
- le graphe de tâches (GdT). Un graphe de tâches permet de modéliser un programme en décrivant les dépendances entre groupes d'instructions. Ainsi, il est possible de déterminer quelles instructions peuvent être exécutées en parallèle et quelles instructions doivent être séquentialisées,
- le graphe de tâches paramétré (GTP). Le graphe de tâches paramétré est une représentation symbolique du graphe de tâches. Il utilise des paramètres qui, une fois instanciés, permettent de construire le graphe de tâches correspondant. La composante principale du graphe de tâches paramétré est un ensemble de règles de communication dont le nombre ne dépend que du programme analysé et qui décrivent un ensemble d'arcs dans le graphe de tâches.

La deuxième partie de ce chapitre est consacrée aux outils sur lesquels nos travaux s'appuient. Il s'agit de :

- *PlusPyr* qui permet de construire le graphe de tâches paramétré,
- *BIP* qui est un protocole de communication pour le réseau Myrinet,
- *PM2* qui fournit un environnement de processus légers communicants au dessus du langage C,
- *Athapascan-1* qui est un langage de programmation pour machine à mémoire partagée,
- Le *Calculateur Omega*, qui permet de faire des calculs dans l'arithmétique de Presburger et qui possède de nombreux opérateurs sur les relations et les polyèdres,
- Un outil qui permet, à partir d'un polyèdre paramétré, de construire le polynôme de Ehrhart associé et de l'évaluer,
- *Enum* qui permet de construire, à partir d'un polyèdre, un nid de boucle parfait qui parcourt ce dernier.

2.1 Modéliser les machines à mémoire distribuée

Utiliser un modèle de machine parallèle à la fois simple et proche de la réalité est indispensable si l'on veut pouvoir prévoir l'exécution de programme parallèle ou concevoir des algorithmes pour ces machines. De nombreux modèles ont été proposés dans la littérature partant de *Parallel Random Access Machine* (PRAM) jusqu'à *Bulk Synchronous Processing* (BSP) [54] ou *LogP* [26]. Nous adoptons ici un modèle différent de ces derniers. En effet le modèle PRAM modélise une machine à mémoire partagée : il ne prend pas en compte les communications inter-processeurs. BSP est un modèle où les communications sont synchronisées ce qui ne correspond pas, comme nous le verrons dans la section 2.2.3, au modèle d'exécution macro dataflow des graphes de tâches. Enfin, le modèle LogP, la durée de l'envoi d'un message ne dépend pas de sa taille. Dans ce qui suit nous modéliserons une machine à mémoire distribuée par les quatre paramètres suivants :

- p est le nombre de processeurs (ils sont tous identiques),
- ω est le temps pris pour l'exécution d'une instruction élémentaire,
- α est le taux de transmission (l'inverse de la bande passante),
- β est la latence du réseau d'interconnexion.

En ce qui concerne l'exécution des programmes sur une telle machine nous ferons l'hypothèse que les processeurs sont reliés suivant une clique et donc, que la durée des communications ne dépend que de la taille des messages. Cette hypothèse est justifiée par le fait qu'aujourd'hui, dans le mode de routage *wormhole*, la durée des communications n'est pas affectée par la distance inter-processeur, sauf si la contention du réseau est élevée [17]. De plus, lors de nos expériences, nous avons exécuté nos programmes sur une machine dont le réseau est constitué de *cross-bars*. Nos expériences montrent qu'il n'y a pas de différences suivant qu'une communication a lieu entre une paire de processeurs ou une autre.

2.2 Le graphe de tâches

Le graphe de tâches est classiquement considéré comme un modèle de programme devant s'exécuter sur des machines à mémoire distribuée [68]. Il est utilisé pour la prédiction de performance ou l'optimisation d'applications parallèles [2, 18, 28, 38, 44].

2.2.1 Le modèle

Un graphe de tâches G est un graphe acyclique orienté et annoté défini par le quadruplet suivant $G = (V, E, T, C)$ où :

- V est l'ensemble des nœuds, chaque nœud représentant une tâche. On note $v = |V|$ le nombre de tâches du graphe,
- E est l'ensemble des arcs. Il y a un arc de la tâche T_i à la tâche T_j s'il y a une dépendance entre T_i et T_j. Cela signifie que la tâche T_j doit être exécutée après la fin de la tâche T_i. $e = |E|$ est le nombre d'arcs du graphe,

- T est l'ensemble des coûts des tâches (ou le poids des nœuds). $o_i \in T$ représente le nombre d'opérations calculées par la tâche T_i,
- C est l'ensemble des volumes de communications (ou le poids des arcs). $l_{i,j}$ représente la quantité de données transmises entre la tâche T_i et la tâche T_j (s'il n'y pas d'arcs entre T_i et T_j alors $l_{i,j} = 0$).

Un exemple de graphe de tâches est donné dans la figure 2.1.

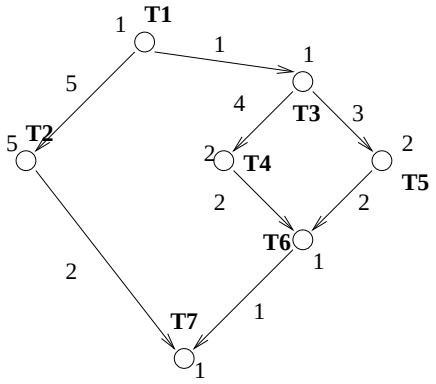


FIG. 2.1: *Exemple de graphe de tâches*

2.2.2 Définitions associées au modèle

Dans cette thèse nous utiliserons les définitions suivantes :

- la durée τ_i de la tâche T_i est calculée en fonction du modèle de machine : $\tau_i = \omega o_i$,
- la durée de la communication $c_{i,j}$ entre la tâche T_i et la tâche T_j dépend aussi du modèle de machine. Elle est nulle si les deux tâches sont exécutées sur le même processeur sinon on a $c_{i,j} = \beta + \alpha \times l_{i,j}$.
- on dit que T_1 est un *père* (resp. *fils*) de T_2 si dans le graphe de tâches il existe une *arête* de T_1 à T_2 (resp. de T_2 à T_1).On note père(T) l'ensemble des pères de T et fils(T) l'ensemble des fils de T.
- on dit que T_1 est un *prédécesseur* (resp. *successeur*) de T_2 si dans le graphe de tâches il existe un chemin de T_1 à T_2 (resp. de T_2 à T_1),
- A l'instar de Gerasoulis et Yang dans [47], on appellera *grain* d'un graphe de tâches G pour une machine donnée la quantité :

$$g = \min_{T_i=1:v} \left(\min \left(\frac{\tau_i}{\max_{T_j \in \text{père}(T_i)} c_{j,i}}, \frac{\tau_i}{\max_{T_j \in \text{fils}(T_i)} c_{i,j}} \right) \right).$$

On voit que cette quantité dépend de la machine sur laquelle est exécuté le graphe de tâches (plus le réseau d'interconnexion est rapide par rapport aux processeurs plus le grain est élevé). On dira qu'un graphe de tâches est à gros grain si le grain est plus grand

que 1. Dans ce cas, les communications entre deux tâches sont toujours plus courtes que la durée des deux tâches en question. Comme le montre la figure 2.2, lorsqu'un graphe est à gros grain ($c_{1,3} \leq \tau_3$) la parallélisation de T_2 avec T_3 est toujours plus efficace que la séquentialisation des trois tâches. En effet, si $c_{1,2} \leq c_{1,3}$ alors, $\tau_1 + \tau_3 \leq \tau_1 + \tau_2 + \tau 3$ et $\tau_1 + c_{1,2} + \tau_2 \leq \tau_1 + \tau_2 + \tau_3$

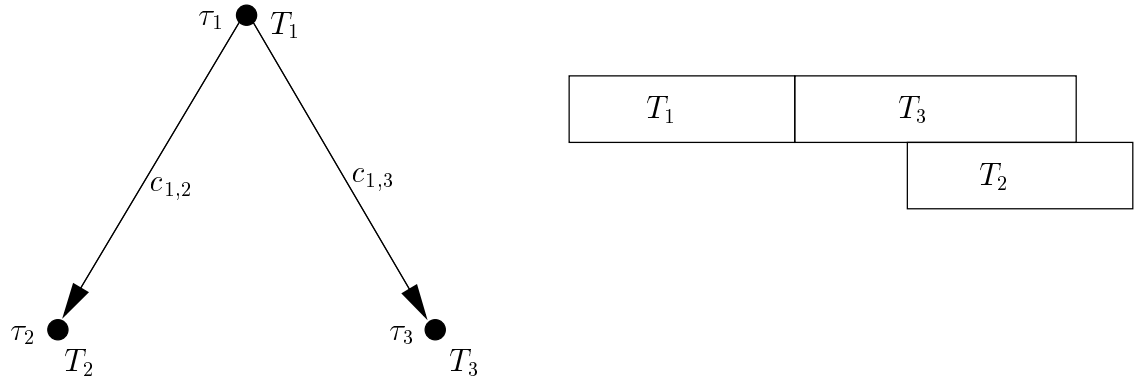


FIG. 2.2: *Exemple d'un graphe où lorsqu'il est à gros grain il est toujours plus intéressant de paralléliser T_2 avec T_3 que de séquentialiser les trois tâches. Remarque : si $c_{1,3} \leq c_{1,2}$ il faut exécuter T_2 à la suite de T_1*

– On appelle *grappe* (cluster) un ensemble de tâches qui doivent toutes être exécutées sur le même processeur. Si pour toutes taches T_1 et T_2 d'une grappe donnée il existe, au sein de celle-ci, un chemin de T_1 à T_2 ou de T_2 à T_1 alors la grappe est dite *linéaire* [45] (la grappe est constituée d'un chemin du graphe). On a à faire à un *regroupement* (clustering) si chaque tâche du graphe est dans une et une seule grappe. La technique du regroupement est souvent utilisée pour ordonnancer un graphe sur un nombre non borné de processeurs. Chaque grappe correspond alors à un processeur virtuel [46]. On a un *regroupement linéaire* (linear clustering) si toutes les grappes ainsi constituées sont linéaires. On notera que mettre chaque tâche toute seule dans une grappe constitue un regroupement linéaire.

2.2.3 Exécution du graphe de tâches

Le modèle de base d'exécution pour les graphes de tâches est appelé *macro dataflow*. Il s'agit d'un modèle d'exécution dirigé par les données où chaque tâche suit le protocole suivant :

– avant de s'exécuter sur son processeur la tâche reçoit toutes les données nécessaires à ses calculs,

– la tâche s'exécute sans interruption, séquentiellement,

– elle transmet ensuite les données aux tâches *filles*.

2.2.4 Ordonnancement de graphe de tâches

L'ordonnancement de graphes de tâches a été très étudié [19, 31, 39, 53, 56, 68, 76]. Ordonnancer un GdT consiste à assigner pour chaque tâche une date de début d'exécution et un processeur. Pour être valide, cette assignation doit vérifier les deux contraintes suivantes :

- les *contraintes de ressources*. A un instant donné un processeur ne peut exécuter qu'une seule tâche,
- les *contraintes de dépendances*. Soit deux tâches T_1 et T_2 telles que $(T_1, T_2) \in E$, alors T_2 ne peut commencer son exécution qu'après la fin de T_1 et des communications entre les deux tâches.

Un exemple d'ordonnancement pour la machine $p = 3$, $\omega = 2$, $\alpha = 0.5$ et $\beta = 1$ du graphe figure 2.1 est donné figure 2.3.

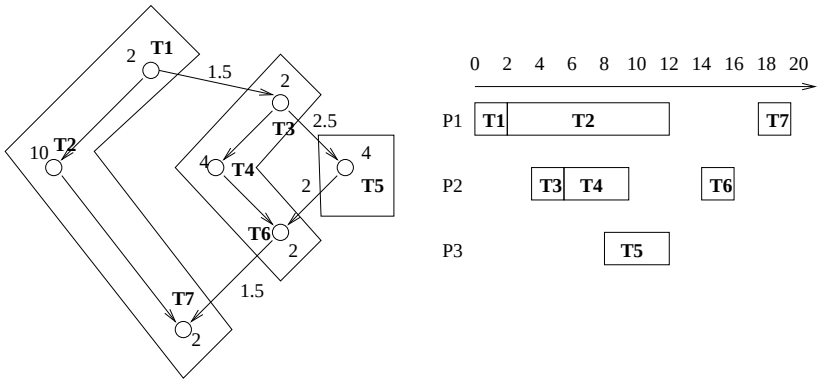


FIG. 2.3: *Ordonnancement valide pour* $p = 3$, $\omega = 2$, $\alpha = 0.5$ *et* $\beta = 1$

Il existe de nombreuses variantes du problème de l'ordonnancement mais nous nous limiterons aux cas suivants :

- la duplication des tâches n'est pas autorisée,
- la durée des tâches et des communications est arbitraire,
- l'objectif de l'ordonnancement est de minimiser le *temps parallèle* à savoir la date de fin d'exécution de la dernière tâche.

Le problème d'ordonnancement tel qu'énoncé plus haut est NP-complet [19]. Cependant de nombreuses heuristiques ont été proposées.

Yang et Gerasoulis ont proposé et étudié l'heuristique *Dominant Sequence Clustering* (DSC) [76] pour ordonnancer un graphe de tâches sur un nombre non borné de processeurs. Cette heuristique à l'avantage de combiner une faible complexité temporelle ($O((e+v)\log v)$) et une bonne efficacité surtout pour les graphes à gros grain. DSC a été intégré dans *Pyrros* [73], où la fusion des grappes pour obtenir un nombre de processeurs égal à celui de la machine cible, est effectué par une heuristique de liste.

Récemment Liou et Palis ont proposé un nouvel algorithme pour un nombre non borné de processeurs appelé CASS-II [56]. La complexité temporelle de CASS-II est meilleure que

celle de DSC ($O(e + v \log v)$) et les expériences prouvent que la qualité de la solution est parfois meilleure que celle de DSC.

2.3 Le graphe de tâches paramétré

Le graphe de tâches paramétré a été introduit par Cosnard et Loi [24, 57] comme une solution pour construire automatiquement un graphe de tâches. Le GTP est une représentation compacte et indépendante de la taille des données de certains graphes de tâches rencontrés dans des applications scientifiques. Il utilise des paramètres qui doivent être instanciés pour construire le graphe de tâches. Une fois construit, le graphe de tâches peut être ordonnancé suivant le schéma décrit plus haut puis être exécuté sur une machine à mémoire distribuée.

2.3.1 Définitions

Un graphe de tâches paramétré est défini par le triplet $GTP = (\mathcal{T}, \mathcal{R}, \mathcal{C})$ où :

- $\mathcal{T}$ est l'ensemble des tâches génériques. On appelle tâche générique un ensemble d'instructions devant être exécutées séquentiellement. Lorsqu'elle est issue d'un programme, une tâche générique est définie par son vecteur d'itération (i.e. les indices des boucles englobantes prises dans l'ordre lexicographique), sa première instruction et sa dernière instruction,
- $\mathcal{R}$ est l'ensemble des règles de communication. Chaque règle de communication décrit de manière symbolique les dépendances entre les tâches génériques,
- $\mathcal{C}$ est la fonction de coût de chaque tâche, $\mathcal{C}$ prend en entrée les paramètres du programme, la tâche et son vecteur d'itération et calcule le nombre d'opérations effectuées par celle-ci.

2.3.2 Les règles de communication

Une règle de communication peut prendre deux formes : une règle de réception ou une règle d'émission. Ces deux formes sont duales l'une de l'autre et l'on peut passer d'une forme à l'autre de manière automatique. Les règles de réception décrivent les arcs entrants (en général les données reçues par les tâches). Les règles d'émission décrivent les arcs sortants des tâches (en général les données envoyées par les tâches).

Définition standard

Comme nous le montrerons dans la section 2.4, il est possible de générer automatiquement le graphe de tâches paramétré à partir d'un programme si celui-ci respecte un certain nombre d'hypothèses. Nous revoyons le lecteur à cette section pour le détail de ces hypothèses. Nous reprenons ici les définitions données par Michel Loi dans sa thèse [57]. Soient :

- R et S deux noms de tâches,
- f_S et f_D des fonctions affines à valeur entière,
- $\mathbf{A}_R$, $\mathbf{C}_R$, $\mathbf{A}_S$, $\mathbf{C}_S$, $\mathbf{C'}_S$, $\mathbf{A}_D$, $\mathbf{C}_D$, $\mathbf{C'}_D$, $\mathbf{C''}_D$ des matrices à coefficients entiers,

- $\vec{b}_R$, $\vec{b}_S$, $\vec{b}_D$ des vecteurs entiers,
- $\vec{z}$ le vecteur des paramètres du programme,
- M un nom de variable.

Les règles de réception ont la syntaxe suivante :

$$\{R(\vec{r})|\vec{r} \in \mathcal{P}_R\} \leftarrow \{S(f_S(\vec{r},\vec{s}))|\vec{s} \in \mathcal{P}_S(r)\} : \{M(f_d(\vec{r},\vec{s},\vec{d}))|\vec{d} \in \mathcal{P}_D(\vec{r},\vec{s})\}$$

où

- $\mathcal{P}_R$ est le domaine paramétré

$$\{\vec{r}|\vec{r} \in \mathbb{N}^m, \mathbf{A}_R\vec{r} + \mathbf{C}_R\vec{z} + \vec{b}_R \geq \vec{0}\},$$

- pour $\vec{r} \in \mathcal{P}_R$, $\mathcal{P}_S(\vec{r})$ est le domaine paramétré

$$\{\vec{s}|\vec{s} \in \mathbb{N}^n, \mathbf{A}_S\vec{s} + \mathbf{C}_S\vec{z} + \mathbf{C'}_S\vec{r} + \vec{b}_S \geq \vec{0}\},$$

- pour $\vec{s} \in \mathcal{P}_S(\vec{r})$, $\mathcal{P}_D(\vec{r},\vec{s})$ est le domaine paramétré

$$\{\vec{d}|\vec{d} \in \mathbb{N}^p \mathbf{A}_D\vec{d} + \mathbf{C}_D\vec{z} + \mathbf{C'}_D\vec{r} + \mathbf{C''}_D\vec{s} + \vec{b}_D \geq \vec{0}\}.$$

Cette règle de réception doit être lue de la manière suivante : «chaque instance de la tâche $R(\vec{r})$ pour $\vec{r} \in \mathcal{P}_R$, reçoit de toutes les instances de la tâche $S(f_S(\vec{r},\vec{s}))$ pour $\vec{s} \in \mathcal{P}_S(r)$ l'ensemble de données $M(f_d(\vec{r},\vec{s},\vec{d}))$ pour $\vec{d} \in \mathcal{P}_D(\vec{r},\vec{s})$».

Notons que les domaines $\mathcal{P}_R$, $\mathcal{P}_S$, $\mathcal{P}_D$ et les fonctions affines f_S et f_D sont aussi paramétrés par le vecteur des paramètres de l'application $\vec{z}$. Pour simplifier les notations ces paramètres sont implicites. La figure 2.4 donne les règles de réception pour l'élimination de Gauss.

1. $\{T_1(k)|2 \leq k \leq n-1\} \leftarrow \{T_2(k-1,k)\} : \{A(k,k)\}$
2. $\{T_1(k)|2 \leq k \leq n-1\} \leftarrow \{T_2(k-1,k)\} : \{A(l,k)|k+1 \leq l \leq n\}$
3. $\{T_2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T_2(k-1,j)\} : \{A(k,j)\}$
4. $\{T_2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T_2(k-1,j)\} : \{A(i,j)|k+1 \leq i \leq n\}$
5. $\{T_2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T_1(k)\} : \{A(i,k)|k+1 \leq i \leq n\}$

FIG. 2.4: *Règles de réception pour l'élimination de Gauss*

Définition simplifiée

La définition précédente, quoique très rigoureuse, est lourde à manipuler. Cette thèse traitant très souvent de règles de communication, nous présentons ici une version simplifiée. Soient :

- T_a et T_b deux tâches de vecteurs d'itération $\vec{u}$ et $\vec{v}$,

– D une variable et $\vec{y}$ un vecteur de même dimension que D,

– $\mathcal{P}$ un polyèdre paramétré.

La version simplifiée des règles de réception a la forme suivante :

$$T_a(\vec{u}) \leftarrow T_b(\vec{v}) : D(\vec{y})|\mathcal{P}$$

Cette règle se lit : «pour chaque $\vec{u}$, $\vec{v}$ et $\vec{y}$ vérifiant le prédicat décrit par $\mathcal{P}$ la tâche $T_a(\vec{u})$, reçoit des tâches $T_b(\vec{v})$ la donnée $D(\vec{y})$».

Il est clair que l'on peut toujours passer de la version standard à la version simplifiée. Il suffit de faire l'union de $\mathcal{P}_S$, $\mathcal{P}_R$ et $\mathcal{P}_D$ pour obtenir $\mathcal{P}$. $\vec{u}$, $\vec{v}$ et $\vec{y}$ sont obtenus à partir de $\vec{r}$, $\vec{s}$ et $\vec{d}$. Dans l'autre sens, il n'est pas toujours possible d'obtenir une règle standard à partir d'une règle simplifiée. On supposera ici, qu'une règle simplifiée sera toujours obtenue à partir d'une règle standard. Cela permettra, en particulier, d'ignorer les données envoyées pour nous consacrer uniquement aux dépendances entre tâches sans pour autant créer de trou dans le polyèdre $\mathcal{P}'$ (Où $\mathcal{P}'$ est obtenu à partir de $\mathcal{P}$ en enlevant les références aux variables de $\vec{y}$.

2.3.3 Les règles d'émission

Les règles d'émission ont la même syntaxe que les règles de réception, seul le sens de la flèche change :

$$\{S(\vec{s})|\vec{s} \in \mathcal{P}_S\} \rightarrow \{R(f_R(\vec{s},\vec{r}))|\vec{r} \in \mathcal{P}_R(\vec{s})\} : \{M(f_D(\vec{s},\vec{r},\vec{d}))|\vec{d} \in \mathcal{P}_D(\vec{s},\vec{r})\}.$$

Elle se lit : « Soit $\vec{s}$ un élément de $\mathcal{P}_S$. L'instance de la tâche $S(\vec{s})$, envoie les données $M(f_d(\vec{s},\vec{r},\vec{d}))|\vec{d} \in \mathcal{P}_D(\vec{s},\vec{r})$ à toutes les instances des tâches $R(f_R(\vec{s},\vec{r}))|\vec{r} \in \mathcal{P}_R(\vec{s})$».

Nous avons aussi une version simplifiée des règles d'émission :

$$T_a(\vec{u}) \rightarrow T_b(\vec{v}) : D(\vec{y})|\mathcal{P}$$

2.3.4 Les tâches d'entrée et de sortie

Pour pouvoir générer de manière sûr le graphe de tâches à partir du graphe de tâches paramétré et des valeurs des paramètres, il est nécessaire de rajouter deux tâches *artificielles* : les tâches d'entrée et de sortie.

La tâche d'entrée est une tâche qui écrit toutes les données de l'application et qui doit être exécutée en premier. Ainsi, dans le graphe de tâches elle précède topologiquement à toutes les autres tâches.

La tâche de sortie est une tâche qui écrit toutes les données de l'application et qui doit être exécutée en dernier. Ainsi, dans le graphe de tâches elle succède topologiquement toutes les autres tâches.

2.3.5 Du graphe de tâches paramétré au graphe de tâches

Il est possible de construire le graphe de tâches à partir d'un graphe de tâches paramétré et d'une instance des paramètres. Si l'on ne possède que les règles de réception, il faut partir de la tâche de sortie et appliquer l'algorithme de la figure 2.5. Énumérer les pères d'une tâche à partir d'une représentation polyédrique peut-être fait à l'aide d'un outil comme *Enum* décrit section 2.5.6. La fonction paramétrée `nb_points` qui évalue le nombre de points d'une instance d'un domaine peut être construit à l'aide de techniques à base de polynômes de Ehrhart comme décrit section 2.5.5.

Entrée : GTP, $\vec{z}$ une instance des paramètres.
Sortie : le graphe de tâches $G = (V, E, T, C)$.
Premier appel : `Construit_graphe_de_taches`$(T_{\text{sortie}}, \vec{z})$
`Construit_graphe_de_taches`$(R(\vec{r}), \vec{z})$
  pour chaque règle de réception i **faire**
    si $\vec{r} \in \mathcal{P}_R^i$ **alors**
      pour chaque $\vec{s^i} \in \mathcal{P}_S^i(\vec{r})$ **faire**
        si $(S^i(f_S^i(\vec{r}, \vec{s^i})), R(\vec{r})) \notin E$ **alors**
          $E \mathrel{+}= (S^i(f_S^i(\vec{r}, \vec{s^i})), R(\vec{r}))$;
          $l_{S^i(f_S^i(\vec{r},\vec{s^i})),R(\vec{r})} =$ `nb_points`$(\mathcal{P}_D^i(\vec{r}, \vec{s^i}))$;
        sinon
          $l_{S^i(f_S^i(\vec{r},\vec{s^i})),R(\vec{r})} \mathrel{+}=$ `nb_points`$(\mathcal{P}_D^i(\vec{r}, \vec{s^i}))$;
        si $S^i(f_S^i(\vec{r}, \vec{s^i})) \notin V$ **alors**
          $V \mathrel{+}= S^i(f_S^i(\vec{r}, \vec{s^i}))$;
          $o_{S^i(f_S^i(\vec{r},\vec{s^i}))} = \mathcal{C}(S^i, f_S^i(\vec{r}, \vec{s^i}), \vec{z})$;
          `Construit_graphe_de_taches`$((S^i(f_S^i(\vec{r}, \vec{s^i})), \vec{z})$

FIG. 2.5: *Algorithme de construction d'un graphe de tâches à partir d'un GTP*

2.3.6 Supprimer les multi-arcs

Il s'avère parfois que deux règles représentent exactement le même ensemble d'arcs. Il se peut alors que ces deux règles envoient chacune des données qui sont contigües dans la mémoire. C'est le cas des règles 1 et 2 de la figure 2.4, où une règle envoie les éléments $k+1$ à n de la colonne k de la matrice A et l'autre règle envoie l'élément $A(k, k)$. Dans un tel cas, il est utile de fusionner les deux règles en une seule. Ainsi, l'analyse du GTP s'en trouve accéléré et le code généré est simplifié.

Dans le cas des règles 1 et 2 de la figure 2.4 on obtient la règle :

$$\{T_1(k) | 2 \leq k \leq n-1\} \leftarrow \{T_2(k-1, k)\} : \{A(l, k) | k \leq l \leq n\}$$

Entrée : deux règles de communication R_1 et R_2
Sortie : Une règle R, fusion de R_1 et R_2 si possible
`Fusion`(R_1,R_2)
  si `meme_arcs`(R_1,R_2) **alors**
    si la variable transmise (M) est la même pour R_1 et R_2 **alors**
      S_1 = domaine de M qui est transmis par R_1 ;
      S_2 = domaine de M qui est transmis par R_2 ;
      $S = S_1 \cup S_2$;
      E = `Enveloppe_convexe`(S) ;
      si $E = S$ **alors**
        $R = R_1$ dans lequel S_1 est remplacé par E ;

FIG. 2.6: *fusion des règles R1 et R2 décrivant le même ensemble d'arcs*

L'algorithme de la figure 2.6 Permet de simplifier certaines règles. Nous utilisons le Calculateur Omega (Cf. section 2.5.4) pour effectuer les opérations d'union, de calcul de l'enveloppe convexe et d'égalité.

La fonction `meme_arcs`, qui détermine si deux règles décrivent exactement le même ensemble d'arcs, est elle aussi réalisée à l'aide du Calculateur Omega. Pour se faire on représente les règles sous forme de relations puis on calcule leurs différences ($D_1 = R_1 - R_2$ et $D_2 = R_2 - R_1$). Si les ensembles des points D_1 et de D_2 sont tous les deux vides cela signifie que les deux règles décrivent le même ensemble d'arcs.

On calcule l'enveloppe convexe de l'union des deux domaines de la variable M car, avec le Calculateur Omega, dans certains cas l'opérateur $\cup$ ne fournit pas toujours un résultat simple.

2.4 PlusPyr

PlusPyr est un outil réalisé par Loi [57] qui permet, à partir d'un programme séquentiel augmenté de délimiteurs de tâches, de construire automatiquement le graphe de tâches paramétré et de donner une valeur aux paramètres pour construire le graphe de tâches.

2.4.1 Le langage d'entrée

Les techniques aujourd'hui disponibles pour calculer de manière exacte, avant l'exécution du programme, quelles sont les dépendances entre des instructions, sous quelles conditions un branchement va être exécuté, quelles sont les itérations d'une boucle qui vont être effectuées ou quelles parties d'un tableau sont accédées et à quel moment, ne permettent pas de traiter tous les programmes. Ainsi, pour générer automatiquement un graphe de tâches paramétré à partir d'un programme, un certain nombre de contraintes sont imposées. Il doit s'agir d'un programme à *contrôle statique* [33]. Plus précisément,

– le programme doit être écrit en *Tiny* qui est un langage impératif de type FORTRAN,

- les seules structures de contrôle admises sont la boucle `for` et `if-then-else`,
- les bornes des indices de boucles doivent être des fonctions affines des indices de boucles englobantes et des paramètres du programme. En effet pour connaître le coût d'une tâche il est nécessaire de calculer automatiquement le nombre d'itérations que vont effectuer chaque boucle. On ne sait résoudre ce problème correctement que si le domaine d'itération de la boucle (ou du nid de boucles) est décrit par un paramétrage affine [24] indépendant de la valeur des données. Le pas de toutes les boucles doit être constant et égal à 1,
- les conditions sur les branchements `if-then-else` doivent aussi être constituées de comparaison entre fonctions affines des boucles englobantes et des paramètres du programme. En effet, il est nécessaire de connaître avant l'exécution quel branchement sera pris en fonction des indices de boucles et des valeurs des paramètres. Sinon il est impossible de calculer la durée d'un `if`, ni de savoir quelles seront les variables lues et/ou écrites. C'est impossible à faire si le test dépend de la valeur des données ou si les fonctions ne sont pas affines,
- les fonctions d'accès aux tableaux doivent être des fonctions affines à valeur entière des indices de boucles englobantes et des paramètres du programme. Ceci est nécessaire pour pouvoir savoir statiquement quels éléments du tableau vont être lus et/ou écrits. C'est impossible si l'on a une indirection. Les techniques d'analyse de dépendances ne fonctionnent que pour des fonctions affines d'accès aux éléments du tableau [33]. On supposera que tous les accès aux tableaux se font dans l'intervalle de définition de ceux-ci et qu'il n'y a pas d'*aliasing*,
- le langage est augmenté de deux délimiteurs syntaxiques `task` et `endtask` qui servent à l'utilisateur pour regrouper des instructions en bloc d'exécution atomique (séquentiel) : les tâches,
- les tâches ne peuvent être imbriquées,
- toute instruction d'affectation doit être lexicalement inclue dans une tâche,
- les indices de boucles et les paramètres du programme ne peuvent pas être utilisés comme membre gauche d'une affectation.

La figure 7.1 de l'annexe montre le programme de l'élimination de Gauss. Ce programme vérifie les contraintes énoncées plus haut.

Il est clair que la classe des programmes que PlusPyr peut analyser est relativement restreinte. Il s'avère cependant que la plupart des noyaux de calcul intensif sont à contrôle statique [33] et peuvent donc être analysés par cet outil.

2.4.2 Génération automatique de graphe de tâches paramétré

Pour générer automatiquement un graphe de tâches paramétré, la méthode proposée par Cosnard et Loi dans [24] se décompose en quatre étapes :

1. Analyse de dépendance du programme. Il s'agit d'extraire les dépendances de flot, les dépendances de sortie et les anti-dépendances du programme. Les techniques utilisées

sont celles décrites dans [7, 33, 58, 59]. Les règles de dépendances peuvent être notées :

$$\{R(\vec{r})|\vec{r} \in \mathcal{P}_R\} \leftarrow \{S(f_S(\vec{r}))\} : \{M(f_D(\vec{r}))\}$$

pour les dépendances de flot et :

$$\{R(\vec{r})|\vec{r} \in \mathcal{P}_R\} \leftarrow \{S(f_S(\vec{r}))\}$$

pour les anti-dépendances ou les dépendances de sortie.
Chaque dépendance de flot va donner lieu à une règle de réception. Alors que chaque anti-dépendance ou chaque dépendance de sortie va donner lieu à ce qui s'appelle une règle de synchronisation, dont nous n'avons pas encore parlé. Les règles de synchronisation servent à séquentialiser des tâches indépendantes. Elles ne sont pas nécessaires pour une exécution correcte du programme mais elles permettent, dans certain cas, d'éviter des recopies mémoires (cf [57] page 107).

2. Construction des règles de réception. Il s'agit de regrouper les dépendances impliquant les instructions d'une même tâche. En d'autres mots, nous passons d'une représentation à grain fin à une représentation à grain grossier. Notons $\text{itv}(R)$ le vecteur d'itération de l'instruction R, c'est à dire les indices des boucles englobant l'instruction R. L'analyse de dépendance spécifie, pour chaque dépendance, un entier positif δ, la profondeur de la dépendance. Si $\text{itv}(R) \cap \text{itv}(S) = \emptyset$ alors $\delta = 0$. Sinon la profondeur est telle que $\forall \vec{r} \in \mathcal{P}_R$, $\vec{r}_{1...\delta} = f_S(\vec{r})_{1...\delta}$. De plus, si $|\text{itv}(R) \cap \text{itv}(S)| > \delta$ alors $\vec{r}_{\delta+1} > f_S(\vec{r})_{\delta+1}$.
Les dépendances sont classées en deux catégories : les dépendances intra-tâches et les dépendances inter-tâches. Les dépendances intra-tâches sont telles que $\forall \vec{r} \in \mathcal{P}_R$, $\text{tâche}(R(\vec{r})) = \text{tâche}(S(F_S(\vec{r})))$. Les dépendances inter-tâches sont telles que $\forall \vec{r} \in \mathcal{P}_R$, $\text{tâche}(R(\vec{r})) \neq \text{tâche}(S(F_S(\vec{r})))$. Les dépendances inter-tâches impliquent des opérations de transfert de données entre les instances des tâches, ce n'est pas le cas pour les dépendances intra-tâches. Si $\text{tâche}(R) \neq \text{tâche}(S)$ alors $\forall \vec{r} \in \mathcal{P}_R$, $\text{tâche}(R(\vec{r})) \neq \text{tâche}(S(F_S(\vec{r})))$ (inter-tâches). Sinon $\text{tâche}(R) = \text{tâche}(S) = T$. Soit δ la profondeur de la dépendance en question et $m = |\text{itv}(T)|$. Alors soit :
   - $m = 0$ et il n'y a qu'une seule instance pour la tâche T. Ainsi $\text{tâche}(R(\vec{r})) = \text{tâche}(S(F_S(\vec{r})))$ (intra-tâches),
   - ou $\delta < m$ et $\forall \vec{r} \in \mathcal{P}_R$, $\vec{r}_{1...m} \neq f_S(\vec{r})_{1...m}$. Ainsi, $\text{tâche}(R(\vec{r})) \neq \text{tâche}(S(F_S(\vec{r})))$ (inter-tâches),
   - ou $n \leq m$, et $\forall \vec{r} \in \mathcal{P}_R$, $\vec{r}_{1...m} = f_S(\vec{r})_{1...m}$. Ainsi, $\text{tâche}(R(\vec{r})) = \text{tâche}(S(F_S(\vec{r})))$ (intra-tâches).

   Les dépendances intra-tâches peuvent donc être ignorées lorsque l'on construit les règles de communication. Transformer une dépendance $\{R(\vec{r})|\vec{r} \in \mathcal{P}_R\} \leftarrow \{S(f_S(\vec{r}))\} : \{M(f_D(\vec{r}))\}$ en règle de communication $\{R'(\vec{\rho})|\vec{\rho} \in \mathcal{P}'_R\} \leftarrow \{S'(f'_S(\vec{\rho}, \vec{\sigma}))|\vec{\sigma} \in \mathcal{P}'_S(\rho)\} : \{M(f'_d(\vec{\rho}, \vec{\sigma}))\}$ avec $\vec{\rho} = \text{itv}(R')$, $R' = \text{tâche}(R)$ et $S' = \text{tâche}(S)$, se fait comme suit. Soit $m = |\text{itv}(R')|$ et $n = |\text{itv}(S')|$. Alors, $\mathcal{P}'_R = \text{proj}(\mathcal{P}_R, m)$ (la projection des m premières variables de $\mathcal{P}_R$) et $\mathcal{P}'_S(\vec{\rho}) = \text{comp}(\mathcal{P}_R, \vec{\rho})$ (le complément de $\mathcal{P}_R$ par rapport à $\vec{\rho}$). Si $f_S(\vec{r}) = C\vec{r} + \vec{d}$ et $f'_S(\vec{\rho}, \vec{\sigma}) = C'(\vec{\rho} \star \vec{\sigma}) + \vec{d}'$ ($\star$ est l'opérateur de concaténation) alors C' est égal aux n premières lignes de C et $\vec{d}'$ est égal au n premières lignes de $\vec{d}$. Enfin, $f'_D = f_D$.

3. La règle ainsi générée n'a pas de domaine paramétré concernant la partie des données qui sont transmises. Ceci peut conduire à ce que, lors de l'instanciation de la règle, plusieurs arcs soient générés entre deux instances de tâches (on obtient un multi-graphe). Dans PlusPyr, des techniques sont utilisées pour supprimer les transferts de données superflus ainsi que les arcs de communication superflus. Après cette étape les règles ont la forme standard décrite section 2.3.2. L'instance d'une règle ne décrit qu'une et une seule arête entre deux instances de tâches (nous avons montré comment on peut supprimer certains multi-arcs qui subsistent à cause de plusieurs règles dans la section 2.3.6).

4. La dernière étape de la génération automatique du graphe de tâches est le calcul de la fonction de coût des tâches. La forme restreinte du langage source assure que chaque affectation S a un coût de calcul constant. Comme les bornes des indices de boucles sont des fonctions affines des indices de boucles englobantes et des paramètres, il est possible de savoir, pour chaque instance de la tâche, combien de fois une instruction sera exécutée. La fonction de coût de la tâche est alors la somme du coût de chaque instruction multipliée par le nombre de fois qu'est exécutée celle-ci.

2.4.3 Connexion PlusPyr - Pyrros

Une fois le graphe de tâches paramétré construit, il est possible de donner une valeur aux paramètres pour construire le graphe de tâches. PlusPyr appelle alors Pyrros pour ordonnancer le graphe de tâches et l'utilisateur obtient une allocation en temps et en espace du graphe de tâches.

2.5 Autres outils en interaction avec ces travaux

2.5.1 BIP

BIP (*Basic Interface for Parallelism*) est un protocole de communication pour réseaux Myrinet. Il a été réalisé et mis au point par Prylli et Tourancheau [65]

BIP fournit une couche réseau de bas niveau, dont les services sont orientés vers le calcul parallèle, en tirant au mieux parti du matériel. Un programme basé sur BIP utilise un ensemble fixe de n processus répartis sur les processeurs d'une machine parallèle à mémoire distribuée. La transmission de messages peut être aussi bien bloquante que non bloquante. Deux types de sémantiques ont été implantés pour distinguer les messages courts des messages longs. Des transferts DMA via le bus mémoire sont effectués pour éviter les recopies mémoires.

BIP a été testé sur la plateforme POPC qui est une machine à mémoire distribuée constituée de 12 Pentiums Pro à 200 MHz reliés par Myrinet. Les performances crêtes atteintes par BIP sont un débit de 1Gbit/s (126 Mo/s) et une latence de 4,4 μsec.

2.5.2 PM2

PM2 (*Parallel Multithreaded Machine*) est un environnement de programmation distribué à base de processus légers. Il a été développé conjointement au LIP et au LIFL [62, 61].

PM2 est le langage cible du générateur de code décrit dans le chapitre 5.

Un processus léger (*thread*) peut être considéré comme une file d'exécution au sein d'un processus. Il est *léger* dans le sens où, contrairement à un processus système, peu d'informations sont nécessaires à sa gestion. Seuls une pile et un pointeur de programme sont absolument indispensables. Ainsi, la table des fichiers ouverts, la table des signaux, etc ... peuvent être gérés par le processus lui-même.

PM2 peut gérer plusieurs centaines de processus légers au sein de chaque processus système. Dans un tel environnement coexistent alors deux niveaux de parallélisme. En effet, chaque processus léger s'exécute concurremment au sein d'un même processus (premier niveau). Un processus peut être répliqué sur les différents nœuds d'une architecture parallèle (deuxième niveau). Le modèle de programmation est appelé *SPMD* (un Seul Programme de Multiples Données). Les processus communiquent entre eux à l'aide d'appel de procédures à distance léger (*LRPC : Light Remote Procedure Call*).

Ce mécanisme d'appel de procédures à distance est très utilisé en système (UNIX) et dans les systèmes à objet. Il est aussi utilisé en calcul scientifique, c'est le cas de la plateforme de méta-computing Globus [36] et plus particulièrement de Nexus [37], sa couche de communication.

Le principe est le suivant. Lorsque l'on veut transmettre des données on appelle une procédure (un «*service*» dans la terminologie PM2), dont les paramètres sont le noeud où doit être exécuté le service un pointeur sur les données en entrées de ce service, un pointeur sur les données produites (éventuellement) par ce service. Une fois appelée, cette fonction s'exécute sur le nœud en question avec les données transmises. Elle peut éventuellement renvoyer des données qui seront récupérés par l'appelant.

PM2 propose plusieurs fonctionnalités qui nous seront utiles lors de la réalisation de programmes parallèles à base de tâches. Parmi celles-ci on notera : une gestion de priorité des processus légers, la possibilité de communications asynchrones entre processus légers et la possibilité de communications globales. PM2 propose aussi des mécanismes de migration des processus légers, mais ceci ne nous a pas été utile.

Le noyau de processus léger s'appelle *Marcel*. Marcel n'implante pas toutes les fonctionnalités de la norme POSIX car elles ne sont pas toujours nécessaires en parallélisme. Ainsi, Marcel se montre très efficace comme le montre la table 2.1.

	Marcel	Muller pthreads	Provenzano pthreads	Solaris pthreads
changement de contexte	8	220	34	48
création	37	45	26	86
création destruction	85	532	109	305

TAB. 2.1: *Comparaison de Marcel avec d'autres noyaux de processus légers. Les temps sont en microsecondes sur une SPARC5 85 MHz (source : R. Namyst)*

Dans PM2 l'interface de communication s'appelle *Madeleine*. Madeleine est disponible au dessus d'un certain nombre de protocoles de communication comme : VIA, BIP, SBP, SCI, MPI, PVM ou TCP. La figure 2.7 montre la vitesse de transmission de messages avec PM2. La plateforme utilisée est la machine POPC.

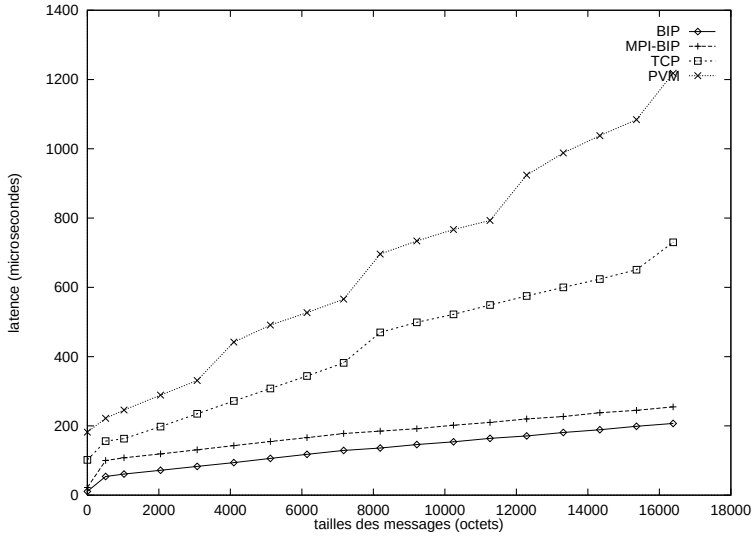


FIG. 2.7: *Temps de transfert de message sous PM2 pour différentes couches de communications (source : R. Namyst)*

2.5.3 Athapascan-1

Athapascan-1 est une interface de programmation basée sur le noyau de processus légers Athapascan-0. Il a été développé au LMC-IMAG [42]. Le parallélisme est exprimé à l'aide d'appels asynchrones de fonctions par la primitive `fork`. Le modèle sous-jacent est une machine à mémoire partagée où les accès aux variables partagées (en lecture, en écriture ou en lecture/écriture) sont décrits dans le prototype des fonctions. La sémantique séquentielle du programme est assurée par ces modes d'accès.

Les dépendances entre les différentes fonctions sont calculées en cours d'exécution, par le système. Le grain de calcul est explicite mais le système peut décider de la dégénérescence séquentielle d'une tâche. Le graphe de tâches ainsi obtenu peut être ordonnancé à la volée. Les travaux théoriques développés autour de cet aspect prouvent que le surcoût induit par cette démarche peut-être borné. Il est aussi possible d'ordonnancer statiquement le graphe de tâches si celui-ci est calculé suffisamment à l'avance.

Au cours de notre thèse nous avons intégré Pyrros comme ordonnanceur statique dans Athapascan-1. Nous avons aussi montré qu'il est possible de générer du code Athapascan-1 à partir du graphe de tâches paramétré. Ceci sera détaillé dans la section 5.4.

2.5.4 Le Calculateur Omega

Le Calculateur Omega est un puissant outil qui permet de décrire des relations sur des n-uplets ou des ensembles. Il a été développé à l'université du Maryland [66]. Les relations et les ensembles sont décrits par des formules de Presburger, une classe de formules logiques qui peuvent être construites par des combinaisons affines sur des variables à valeur entière à l'aide des connecteurs logiques $\neg$, $\vee$ et $\wedge$ et à l'aide des quantificateurs $\exists$ et $\forall$. Il est possible aussi d'utiliser des variables libres. Cela permet de décrire des relations ou des ensembles paramétrés.

La relation $\{[i,j] \rightarrow [i] : 1 \leq i,j \leq 2\}$ décrit l'ensemble des relations simples suivantes : $\{[1,1] \rightarrow [1]\}$, $\{[1,2] \rightarrow [1]\}$, $\{[2,1] \rightarrow [2]\}$ et $\{[2,2] \rightarrow [2]\}$.

La meilleure borne supérieure connue d'un algorithme vérifiant les formules de Presburger à n variables est $2^{2^{2^n}}$ [64].

De nouvelles relations peuvent être construites à partir de relations existantes à l'aide d'opérateurs. Les opérateurs disponibles sont par exemple l'intersection, la composition, l'inverse, la réunion, l'enveloppe convexe. Si une nouvelle relation contient tous les points la formule associée se note `true`, si elle ne contient aucun point la formule associée se note `false`.

Nous aurons besoin de transformer une règle de communication en relation utilisable par le Calculateur Omega. Cela permettra entre autre de déterminer si deux règles décrivent des arcs communs. Soit R la règle d'émission suivante :

$$\{T_1(i,j) | 1 \leq i \leq n, i \leq j \leq n\} \rightarrow \{T_2(i+1,k) | i \leq k \leq j\} : \{A(i,l) | 1 \leq l \leq n\}$$

avec $n \geq 3$ comme équation sur le paramètre n. Transformer R en une relation pour le Calculateur Omega se fait comme suit :

1. Comme on ne s'intéresse qu'aux arcs et pas à leur poids, on retire de R la partie concernant les données transmises.
2. On fait un changement de variables dans les vecteurs d'itération de la tâche émettrice et de la tâche réceptrice. Cela permet de nous abstraire du nom des variables et des fonctions appliquées sur celles-ci. On obtient le début de relation :

$$\{[o_1,o_2] \rightarrow [o_3,o_4] :\}$$

3. Il nous reste à écrire la formule de Presburger de manière à ce que la relation obtenue décrive exactement l'ensemble des arcs de R. Le changement de variables se décrit par $\exists i,j,k : o_1 = i, o_2 = j, o_3 = i+1, o_4 = k$. les autres équations restent les mêmes et sont accolées à la suite. Ainsi, la relation finale est :

$$\begin{aligned}\{[o_1,o_2] \rightarrow [o_3,o_4] : (\exists i,j,k \quad : \quad & o_1 = i \wedge o_2 = j \wedge o_3 = i+1 \wedge o_4 = k \\ & \wedge 1 \leq i \leq n \wedge i \leq j \leq n \wedge i \leq k \leq j \wedge n \geq 3)\}\end{aligned}$$

Le Calculateur Omega nous permet aussi de calculer la restriction d'une règle d'émission R suivant un ensemble I (noté $R\backslash I$). Nous représentons la règle sous forme d'une relation r. Alors $r\backslash I$ la relation correspondant à $R\backslash I$ est telle que $r\backslash I = x \rightarrow y$ si $x \rightarrow y \in r$ et $x \in I$.

Cela signifie que la nouvelle règle est obtenue en restreignant à I le domaine du vecteur d'itération de la tâche émettrice seulement. Par exemple soit R la règle suivante

$$R : T_1(i,j) \rightarrow T_2(i+1,k) | 1 \leq i,j \leq n, 1 \leq k \leq i$$

et

$$I = \{(i,j) | 1 \leq i \leq j \leq n\}$$

alors $R \backslash I$ est la règle R' suivante :

$$R' : T_1(i,j) \rightarrow T_2(i+1,k) | 1 \leq i \leq j \leq n, 1 \leq k \leq i$$

2.5.5 Compter les points d'un polyèdre

Le problème consistant à compter le nombre de points d'un polyèdre apparaît lorsque, par exemple, on veut évaluer le coût d'une tâche, estimer la durée de communication d'une tâche, compter le nombre de fils ou le nombre de pères d'une tâche. Il est important de noter que dans tous ces cas, le polyèdre décrivant l'ensemble des points entiers solution du problème est décrit par un système d'équations de la forme :

$$A\vec{x} + \vec{b} \geq \vec{0}$$

L'ensemble des $\vec{x}$ qui vérifie cette équation décrit un polyèdre convexe. Un polyèdre convexe borné est appelé un polytope. La dimension de $\vec{x}$ est aussi la dimension du polyèdre. Un polyèdre de dimension k est appelé un k-polyèdre. Comme nous voulons énumérer les points d'un polyèdre, seul les polytopes seront considérés dans ce qui suit.

Travaux connexes

Plusieurs méthodes ont été proposées pour résoudre ce problème :

- M. Haghighat et C. Polychronopoulos ont présenté dans [32] une méthode de calcul de volume. La forme de leur réponse implique la présence de min et de max.
- Pugh dans [67] présente des techniques visant à simplifier les contraintes. Ces techniques sont coûteuses si l'on a affaire à des ensembles compliqués.
- Tawbi dans [69] utilise des techniques de décompositions des domaines pour les évaluer.
- Cosnard et Loi [25] ont introduit le descripteur de domaine hiérarchique pour évaluer des polytopes à l'aide d'une technique proche de celle de Tawbi.

Les approches présentées plus haut peuvent toutes s'appliquer à notre problème. Nous avons néanmoins choisi une technique différente car elle présentait l'avantage d'être complètement implantée et donc directement utilisable. Il s'agit des travaux réalisés par Clauss et Loechner [20] sur les polynômes de Ehrhart.

Les polynômes de Ehrhart

Définition 2.1 *Un nombre périodique unidimensionnel $u(n) = [u_1, u_2, \ldots, u_p]_n$ est égal à l'élément dont le rang vaut $n \bmod p$, p est appelé la période de $u(n)$.*

$$u(n) = \begin{cases} u_1 & si\ n = 1 \pmod p \\ u_2 & si\ n = 2 \pmod p \\ \ldots & \\ u_p & si\ n = 0 \pmod p \end{cases}$$

Soit $N = (n_j)$ un vecteur de dimension q, $j \in [1 \ldots q]$. Un nombre q-périodique $U(N)$ est défini par un tableau de dimension q $(u_{i_1,i_2,\ldots,i_q})$ de taille $p_1 \times p_2 \times \ldots \times p_q$ (pour chaque dimension j, $1 \leq i_j \leq p_j$) tel que :

$$U(N) = u_{i_1,i_2,\ldots,i_q}\ si\ \forall j \in [1 \ldots q], n_j = i_j \bmod p_j$$

Le vecteur $p = (p_j)$ est appelé la multi-période de $U(N)$

Exemples :

- $(-1)^n = [-1, 1]_n$
- $(-1)^{n-m} = U\begin{pmatrix} m \\ n \end{pmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}_{(n,m)}$. En effet $U\begin{pmatrix} m \\ n \end{pmatrix} = u_{1,1} = -1$ si $n = 1 \bmod 2$ et $m = 1 \bmod 2$; $U\begin{pmatrix} m \\ n \end{pmatrix} = u_{2,1} = 1$ si $n = 2 \bmod 2$ et $m = 1 \bmod 2$; $U\begin{pmatrix} m \\ n \end{pmatrix} = u_{1,2} = 1$ si $n = 1 \bmod 2$ et $m = 2 \bmod 2$; $U\begin{pmatrix} m \\ n \end{pmatrix} = u_{2,2} = -1$ si $n = 2 \bmod 2$ et $m = 2 \bmod 2$.

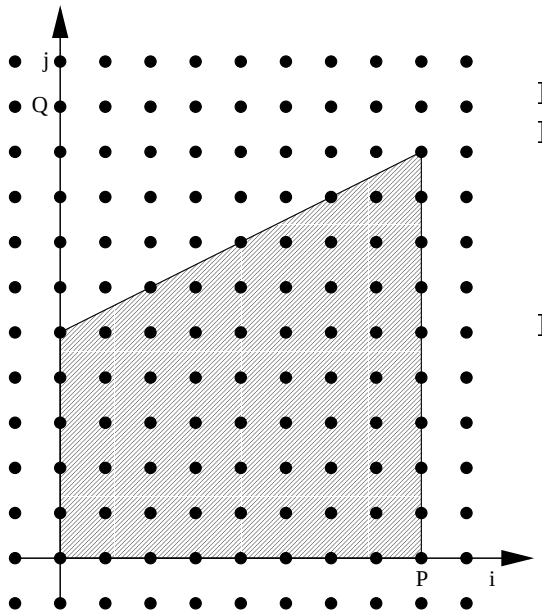
Définition 2.2 *Un pseudo-polynôme ou polynôme de Ehrhart est un polynôme où certains coefficients sont des nombres périodiques. Le plus petit multiple commun des périodes des coefficients périodiques est appelé la pseudo-période.*

Dans [20], Clauss et Loechner ont étendu les travaux de Ehrhart pour prouver le théorème suivant :

Théorème 1 *soit $N = (n_1, n_2, \ldots, n_p)$ un ensemble de p paramètres. L'énumérateur j_N d'un k-polytope paramétré quelconque P_N s'exprime, pour différents domaines des paramètres, par différents pseudo-polynômes à plusieurs variables en N de degré k^p. Ces pseudo-polynômes ont la forme :*

$$j_N = \sum_{i_1=0}^{k} \sum_{i_2=0}^{k} \ldots \sum_{i_p=0}^{k} c_{i_1,i_2,\ldots,i_p} n_1^{i_1} n_2^{i_2} \ldots n_q^{i_q}$$

où les $c_{i_1,i_2,\ldots,i_p}$ sont des nombres périodiques de dimension au plus p.



Paramètres : P, Q
Equations : $0 \leq i \leq P$
$0 \leq 2j \leq i + Q$
$P \geq 0$
$Q \geq 0$

Pseudo-polynôme :

$$\frac{1}{4}P^2 + (\frac{1}{2}Q + 1)P + [(\frac{1}{2}Q + [1, \frac{1}{2}]_Q), (\frac{1}{2}Q + \frac{3}{4})]_P$$

FIG. 2.8: *Exemple de domaine paramétré et de son évaluateur sous forme de pseudo-polynôme*

Exemple

Le figure 2.8 donne l'exemple d'un domaine paramétré et du pseudo-polynôme qui énumère le nombre de points de ce domaine.

Clauss, Loechner et Wilde ont réalisé un programme calculant le pseudo-polynôme énumérant le nombre de point d'un polytope [21]. Ce programme est une extension de la *Polylib* de Wilde [70]. Nous avons modifié ce programme pour qu'il génère automatiquement une fonction qui évalue le pseudo-polynôme suivant la valeur des paramètres. Ces nouvelles fonctionnalités ont été intégrées dans la distribution officielle de la *Polylib*.

2.5.6 Enum

Lorsque, par exemple, on veut parcourir l'ensemble des fils d'une tâche, une solution est de générer un programme qui parcourt l'ensemble des points du polyèdre correspondant.

La littérature regorge de solutions à ce problème [5, 13, 16, 22, 25] mais peu ont été implantées.

Enum est un outil développé par M. Le Fur durant sa thèse [40]. Il permet, à partir d'un polytope paramétré de générer un nid de boucle parfait qui parcourt l'ensemble des points de ce polytope. Cette fonctionnalité est similaire à `codegen` du Calculateur Omega à la différence que, dans Enum, le nom de chaque variable est conservé.

Comme le nid de boucles est parfait, nous avons pu modifier ce programme afin que le code généré puisse intégrer, au cœur du nid de boucles, les instructions que nous souhaitons.

Chapitre 3

Ordonnancement dynamique de graphes de tâches paramétrés

3.1 Introduction

Dans ce chapitre, nous présentons PTGDS un algorithme d'ordonnancement qui utilise uniquement le GTP. Nous montrons comment cet algorithme peut être utilisé au cours d'une exécution dynamique. Il s'agit d'un système où l'ordonnancement est réalisé en cours d'exécution dans un schéma maître-esclave. Le maître est exécuté sur un processeur dédié et calcule l'ordonnancement. Les esclaves sont exécutés sur les nœuds de la machine parallèle et reçoivent les ordres du maître. Dans un tel système, pour chaque GTP, nous avons un programme générique qui prend en entrée les paramètres du programme et les caractéristiques de la machine cible avant de s'exécuter. La présentation détaillée de la méthode est faite section 3.2. La section 3.3 présente une étude théorique de l'algorithme d'ordonnancement ainsi qu'une condition pour qu'un tel schéma soit efficace. Dans la section 3.4 nous comparerons les mérites d'une approche centralisée avec une approche décentralisée. Les résultats expérimentaux sont présentés section 3.5.

3.1.1 Les différents systèmes existants

L'ordonnancement dynamique est une solution communément utilisée pour répondre à deux problèmes :

1. traiter des programmes non déterministes,
2. faire de l'équilibrage de charge et de la migration de processus.

Dans certaines applications, le comportement du programme dépend fortement de la valeur des données. Il est alors impossible de déterminer, à la compilation, quelles tâches seront exécutées ni quelles seront leurs durées. Ordonnancer en cours d'exécution de telles tâches est une solution qui permet de s'affranchir de ces contraintes.

C'est l'approche employée par *Cilk* [11]. L'équilibrage de charge est alors effectué par une politique de *vol de travail*. Dans *Cilk* des problèmes de grandes tailles peuvent être traités mais les coûts de communications ne sont pas pris en compte. Ce système donne de très

bon résultats pour des calculs dont le graphe est un arbre (recherche min-max, *branch and bound*, etc ...), mais il n'a pas été conçu pour des applications scientifiques à base de nid de boucles.

Dans Athapascan-1 [42] l'ordonnancement des tâches est réalisé *à la volée*. Dans ce système l'accent a surtout été mis sur les fonctionnalités et la facilité d'utilisation, plus que sur les performances.

Dans [41] Blelloch, Gibbons et Matias proposent un algorithme d'ordonnancement dynamique pour des programmes à grain fin à base de nid de boucles. Ils minimisent à la fois le temps parallèle et la mémoire requise par chaque nœud. Le graphe est déplié à l'exécution. Cependant le modèle est à grain fin et ils ne s'intéressent pas à minimiser le coût induit par la taille du graphe.

Dans [12] Bougé et al. ont mis au point un système qui permet de faire de la migration de processus au dessus de HPF. Ceci permet de réaliser un équilibrage de charge dynamique des tâches. Ces travaux ont été intégrés dans le compilateur *Adaptor* [15, 14].

3.1.2 Une réponse à notre problématique

En ce qui nous concerne, l'ordonnancement dynamique de graphes de tâches paramétrés est une réponse aux problèmes qui ont été énoncés dans la chapitre 1. En effet :

- La taille du GTP est petite et indépendante des valeurs des paramètres. Construire un ordonnancement à partir du GTP nous permet de traiter des problèmes de grandes tailles,
- Les valeurs des paramètres peuvent être données juste avant l'exécution. Ainsi, nous pouvons construire un programme générique qui fonctionne pour toutes les valeurs des paramètres du problème et pour n'importe quelles valeurs des paramètres de la machine.

3.2 PTGDS

3.2.1 Présentation générale

Notre méthode présente les caractéristiques suivantes :

- elle prend en entrée un graphe de tâches paramétré, c'est à dire : les règles de communication, le code et la fonction de coût de chaque tâche générique,
- on obtient un programme SPMD composé de deux parties :
  1. un algorithme d'ordonnancement dynamique qui est exécuté par le maître. Cet algorithme s'appelle PTGDS (*Parameterized Task Graph Dynamic Scheduler*). Il explore le graphe en utilisant les règles de communication. Il ordonnance localement chaque tâche. Quand une tâche T est affectée à un processeur P, il donne à P l'ordre d'exécuter T,
  2. une partie composée des processeurs de la machine parallèle. Chaque processeur possède le code des tâches et reçoit les ordres du maître. Il exécute les instances des tâches et gère les communications.

- la valeur des paramètres est donnée à l'exécution,
- seule une petite partie du graphe de tâches est connue à un instant donné.

3.2.2 L'algorithme

PTGDS ordonnance les tâches. Il détermine sur quel processeur et à quelle date une tâche doit être exécutée. Le schéma général de PTGDS est présenté figure 3.1. Il fonctionne comme suit. Étant donné un GTP ($\mathcal{T}$,$\mathcal{R}$,$\mathcal{C}$). Pour chaque tâche T, quand la valeur des paramètres est connue :

- on calcule la durée de T en fonction de $\mathcal{C}(T)$ et des paramètres de la machine parallèle.
- on utilise le GTP pour calculer l'ensemble des fils de T, l'ensemble des pères de T et le volume de communication entre ses pères ou ses fils. L'ensemble des fils ou des pères de T est décrit par un polyèdre. Dans le programme nous générons un nid de boucles qui parcourt l'ensemble des fils ou des pères de T. Nous avons détaillé dans le chapitre 2 comment faire pour générer un tel nid de boucles ou pour générer la fonction qui évalue le nombre de points entiers d'un polyèdre.

Entrée : un GTP, une instance des paramètres du programme et de la machine cible
Sortie : un ordonnancement du graphe de tâches
Premier appel : `PTGDS`(T_{sortie})

```
1 PTGDS(T)
2     pour chaque tâche T' ∈ père(T) faire
3         si pas allouée(T') alors
4             PTGDS(T') ;
5     allouer T sur le processeur qui minimise son début d'exécution ;
6     allouée(T)=vrai ;
7     pour chaque tâche T' ∈ père(T) faire
8         si  T' n'a plus de fils à ordonnancer alors
9             retirer de la mémoire toutes les informations sur T' ;
```

FIG. 3.1: *L'algorithme PTGDS*

- Avant d'ordonnancer T, PTGDS vérifie que tous les prédécesseurs de T ont déjà été affectés à un processeur. Si des prédécesseurs de T doivent être ordonnancés PTGDS s'appelle récursivement pour ces tâches puis alloue T de manière gloutonne au processeur qui minimise la date de début d'exécution. Afin que toutes les tâches soient ordonnancées, le premier appel se fait pour la tâche de sortie car elle succède topologiquement à toutes les autres tâches.

Le fait que l'on retire de la mémoire les informations de T quand tous ses fils sont ordonnancés est justifié comme suit. Chaque fois qu'une tâche T est ordonnancée, on stocke dans la mémoire le processeur qui lui est alloué, sa date de début d'exécution, et le nombre de ses fils qu'il reste à ordonnancer. Quand tous les fils de T seront alloués on n'aura plus jamais

besoin de ces informations. Elles sont donc retirées de la mémoire. Ainsi, cette étape permet une grande réduction du coût mémoire de l'algorithme. En contre partie, l'algorithme tel qu'il est écrit ne peut pas être utilisé comme algorithme d'ordonnancement statique.

3.2.3 Combiner ordonnancement et exécution dynamique

Le code cible est un programme générique de style SPMD qui utilise un protocole maître/esclave. Il y a un esclave par processeur. On suppose que le maître est exécuté par l'hôte de la machine parallèle. Chaque esclave reçoit les messages du maître. Ces messages donnent l'ordre d'exécuter une tâche ou d'envoyer des données à un autre processeur. Par exemple quand la tâche T est affectée au processeur P, le maître ordonne aux processeurs $P' \neq P$ qui ont exécuté des pères de T d'envoyer à P les données nécessaires pour l'exécution de T. Les messages qui arrivent sur un processeur sont gérés de manière indépendante par chaque esclave.

Code des esclaves

L'exécution est dynamique. En effet, le maître envoie des ordres alors qu'il ordonnance le graphe. L'algorithme de la figure 3.2 donne le code du protocole de communication d'un esclave.

```
pour chaque nouveau message faire
   en fonction type_message faire
      exécuter :
         si toutes les données requises sont en mémoire alors
            exécuter la tâche ;
            envoyer les nouvelles données calculées si nécessaire (vérifier liste_envoie) ;
            vérifier à l'aide de liste_exec si les nouvelles données calculées
            permettent d'exécuter d'autres tâches ;
         sinon
            stocker le message dans liste_exec ;
      envoyer :
         si toutes les données nécessaires sont en mémoire alors
            envoyer la donnée ;
         sinon
            stocker le message dans liste_envoie ;
      recevoir :
         stocker les données en mémoire ;
         vérifier à l'aide de liste_exec si les nouvelles données arrivées
         permettent d'exécuter d'autres tâches ;
```

FIG. 3.2: *Protocole de communication d'un esclave*

Chaque esclave gère deux listes. Il se peut que l'ordre d'exécution d'une tâche ne puisse

être accompli car toutes les données ne sont pas encore présentes dans la mémoire locale du processeur. Cet ordre est alors stocké dans la liste `liste_exec`. Cette liste est mise à jour lorsque de nouvelles données arrivent ou que de nouvelles données sont calculées. Certains messages ne peuvent pas être envoyés car les données n'ont pas encore été calculées par le processeur. L'ordre d'envoi est alors stocké dans la liste `liste_envoie`. Cette liste est mise à jour quand de nouvelles données sont calculées.

Coût mémoire des esclaves

Nous montrerons plus loin que, dans beaucoup de cas, le temps d'exécution de PTGDS est négligeable devant le temps d'exécution du programme parallèle. Cela signifie que chaque esclave va finir de recevoir tous les ordres du maître dés le début de l'exécution. Un soin particulier doit donc être porté sur la gestion des messages, en particulier sur le coût mémoire de ceux-ci.

Un esclave reçoit autant de messages qu'il exécute de tâches et qu'il a de données à envoyer à ces fils. La mémoire requise par chaque esclave correspond dans le pire des cas à la mémoire nécessaire pour représenter le sous-graphe qu'il exécute.

En ce qui concerne la représentation d'une tâches, 3 entiers sont nécessaires : un pour le numéro de la tâches générique et autant d'entiers que la dimension du vecteur d'itération (deux en moyenne).

En ce qui concerne la représentation d'un arc de communication 7 entiers sont nécessaires : trois pour la tâche de départ, trois pour la tâche d'arrivée et un pour le numéro de la règle décrivant la communication.

Ainsi, si un esclave doit exécuter 1 million de tâches et d'arêtes et si un entier est codé sur 4 octets, le coût mémoire total pour stocker ces informations est un peu plus de 38 Mo.

Dans un tel schéma le graphe de tâche est distribué sur les processeurs. Si l'on suppose que l'équilibrage de charge est assez bon chaque processeur reçoit une fraction du graphe de tâches total.

3.2.4 Discussion

PTGDS est un algorithme récursif qui ordonnance le GdT en partant du bas. Nous avons préféré cette méthode à celle qui consisterait à ordonnancer le graphe en partant des premières tâches du graphe. Ceci pour deux raisons :

1. une exploration de haut en bas est similaire à une exploration en largeur d'abord. Une exploration du bas vers le haut est similaire à une exploration en profondeur d'abord. Il est plus difficile d'implanter un algorithme d'exploration en largeur d'abord car il nécessite une file comme structure de données (cf [23] pages 461–468), pour une complexité équivalente,
2. une exploration de haut en bas est coûteuse en mémoire s'il y a beaucoup de tâches avec un degré sortant élevé, alors qu'une exploration de bas en haut est coûteuse s'il y a beaucoup de tâches avec un degré entrant élevé. Il n'y a donc pas de méthode générale qui permette de traiter l'ensemble des graphes.

Dans la pratique, il s'avère que PTGDS est une bonne méthode pour les graphes que nous avons testés.

3.3 Résultats théoriques

Nous étudions ici, les performances théoriques de PTGDS. Dans un premier temps nous montrons que l'ordonnancement donné par PTGDS est valide. Puis nous calculons sa complexité mémoire et temporelle. Ensuite, nous montrons que tant que nous avons à faire à des graphes à gros grain, et que nous disposons d'un nombre non borné de processeurs, le temps parallèle trouvé par PTGDS est à deux de l'ordonnancement optimal. Enfin, nous évaluons le surcoût dû à l'ordonnancement dynamique et nous montrons que cette approche est d'autant meilleure que la durée des tâches est grande.

3.3.1 Validité de l'ordonnancement

Lemme 1 *L'ordonnancement calculé par PTGDS est valide.*

Preuve du lemme 1 :

1. Les contraintes de précédence sont respectées. Une tâche est ordonnancée après que tous ses prédécesseurs aient été ordonnancés. Sa date de début d'exécution est calculée de telle manière qu'elle est plus grande que la date de fin d'exécution plus la durée des communications de tous ses pères.
2. Les contraintes de ressource sont respectées. Quand une tâche est affectée à un processeur, sa date de début d'exécution est calculée de telle manière qu'elle est plus grande que la date de fin d'exécution des tâches précédemment ordonnancées sur le même processeur.

∎

3.3.2 Complexité temporelle

Dans la section 3.2.2 nous avons précisé qu'il est nécessaire de stocker des informations sur les tâches qui ont été ordonnancées. Nous utilisons un arbre AVL [50] comme structure de données. Dans un arbre AVL de taille s, l'insertion la recherche et la suppression d'éléments se fait en $O(\log s)$.

Théorème 2 *La complexité temporelle de PTGDS est* $O((e+v)(\log v + p))$

Preuve du théorème 2 : Reprenons l'algorithme de la figure 3.1. Pour évaluer la complexité temporelle de PTGDS nous calculons, pour chaque ligne, la complexité de celle-ci que nous multiplions par le nombre de fois qu'elle est appelée. Nous supposons qu'il y a s éléments dans l'arbre AVL.

La procédure `PTGDS` est appelée une fois pour chaque tâche T, il y a v tâches dans le graphe de tâches donc v appels de `PTGDS`.

Dans les lignes 2 et 7, on utilise le GTP pour construire le nid de boucles qui parcourt l'ensemble des pères de T. Ainsi, trouver le prochain père d'une tâche se fait en temps constant. Durant l'exécution de cet algorithme tous les pères de toutes les tâches sont explorés une et une seule fois. Cela signifie que les arcs sont examinés une et une seule fois, le coût des lignes 2 et 7 est donc $O(e)$.

Le test de la ligne 3 est exécuté e fois (le nombre de pères de chaque tâche du graphe). On utilise l'arbre AVL pour déterminer si `allouee`(T) est vrai. Ceci est fait en $O(\log s)$. Le coût de la ligne 3 est donc $O(e \log s)$.

En ce qui concerne l'allocation des tâches, elle est faite de manière gloutonne. Nous avons besoin d'examiner chaque arc une fois pour allouer toutes les tâches. Chaque fois qu'un arc est examiné il coûte $O(\log s)$ pour trouver dans l'arbre AVL les informations sur le père de T et il coûte $O(p)$ pour mettre à jour les processeurs en fonction des communications induites par l'arc. Pour chaque tâche, trouver le processeur qui minimise la date de début d'exécution prend aussi $O(p)$. On calcule la durée d'une tâche à l'aide du GTP en temps constant. Ainsi le coût total du processus d'allocation (ligne 5) est $O(e \log s + ep + vp)$.

La ligne 6 est exécutée v fois et son coût est $O(\log s)$ (une mise à jour dans l'arbre AVL). Le coût total de la ligne 6 est donc $O(v \log s)$.

Comme le nombre de fils qu'il reste à ordonnancer est maintenu, pour chaque tâche, dans l'arbre AVL, le test de la ligne 8 est exécuté $O(e)$ fois en $O(\log s)$ soit un coût total de $O(e \log s)$.

Enfin, la suppression d'information sur T' dans l'arbre AVL se fait en $O(\log s)$ pour les v tâches. Le coût total de la ligne 9 est donc $O(v \log s)$

Ceci prouve que la complexité temporelle de PTGDS est $O((e + v)(\log s + p))$. Nous prouvons le théorème en majorant s par v, le nombre de tâches. ∎

Remarque : La taille maximale de l'arbre AVL en cours d'exécution dépend de la structure du graphe. Par exemple en ce qui concerne l'élimination de Gauss on a $s = \sqrt{v}$.

3.3.3 Complexité mémoire

Dans l'algorithme PTGDS rien n'est dit en ce qui concerne l'ordre dans lequel les tâches doivent être ordonnancées. Il est cependant facile de voir que le coût mémoire de l'exécution de l'algorithme dépend de cet ordre. Nous discutons ici du pire cas, c'est à dire le cas où l'ordre dans lequel le graphe est ordonnancé conduit à la plus grande utilisation mémoire. Il existe des graphes (par exemple un graphe de type *join*) où le coût mémoire de PTGDS est proportionnel au nombre de nœuds du graphe. Nous donnons ici la complexité mémoire de PTGDS en fonction de la topologie du graphe. Cela permet de caractériser les graphes pour lesquels PTGDS a un bon comportement.

Définition 3.1 *Soit T une tâche.*

On note γ_T^+ le degré sortant de T (le nombre d'arcs issus du nœud représentant T dans le graphe de tâches).

On note γ_T^- le degré entrant de T (le nombre d'arcs arrivant sur le nœud représentant T dans le graphe de tâches).

Δ *est le maximum, pour toutes les chaînes* C *du GdT, des degrés entrants :* $\Delta = \max_C \sum_{T \in C} \gamma_T^-$.

Γ *est le nombre de tâches de degré sortant au moins 2 :* $\Gamma = |\{T : \gamma_T^+ \geq 2\}|$.

Théorème 3 *La complexité mémoire de PTGDS est* $O(\Delta + \Gamma)$.

Preuve du théorème 3 : Une tâche est gardée en mémoire soit parce qu'elle a plusieurs fils, soit parce qu'un de ces fils a plusieurs pères.

Soit T une tâche qui vient d'être ordonnancée.

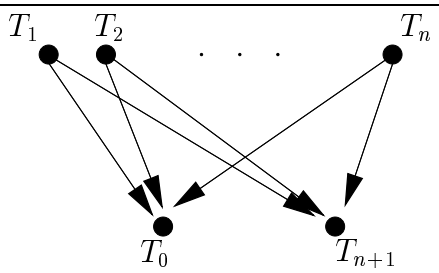


FIG. 3.3: *Cas où, après que* T_0 *ait été ordonnancée, les* $\Gamma = n$ *tâches* T_1 *à* T_n *sont en mémoire avant l'ordonnancement de* T_{n+1}

- Si $\gamma_T^+ \geq 2$ alors T sera gardé en mémoire tant que ces γ_T^+ fils n'auront pas été ordonnancés. Dans le pire cas (cf. figure 3.3), il se peut que toutes les Γ tâches doivent être gardées en mémoire en même temps.
- Soit T' un fils de T. T restera en mémoire tant que T' n'aura pas été ordonnancée. C'est à dire, pas avant que les $\gamma_{T'}^- - 1$ autres pères de T' n'aient été ordonnancés. A ce moment là, on aura en mémoire $\gamma_{T'}^-$ arcs et sommets. PTGDS ordonnance les tâches de proche en proche le long d'une chaîne. Il se peut que tous les pères des tâches d'une chaîne (à l'exception des tâches de cette chaîne), soient déjà en mémoire lorsque PTGDS décide de l'ordonnancer (cf. figure 3.4). Dans le pire cas, le nombre de ces tâches est donc Δ.

Rien ne peut empêcher que les deux cas décrits ci-dessus ne se produisent pas en même temps. Ainsi la complexité mémoire de PTGDS est $O(\Delta + \Gamma)$. ∎

3.3.4 Borne sur l'ordonnancement pour un nombre non borné de processeurs

Nous calculons ici, une borne sur le temps parallèle d'un GdT ordonnancé par PTGDS sur un nombre non borné de processeurs. Un tel ordonnancement est souvent appelé regroupement (*clustering*). Ainsi, nous utiliserons grPTGDS pour rappeler que le nombre de processeurs est supposé non borné (au plus le nombre de tâches du GdT). Nous prouvons que grPTGDS est une heuristique qui est approchée à un facteur deux de l'optimal.

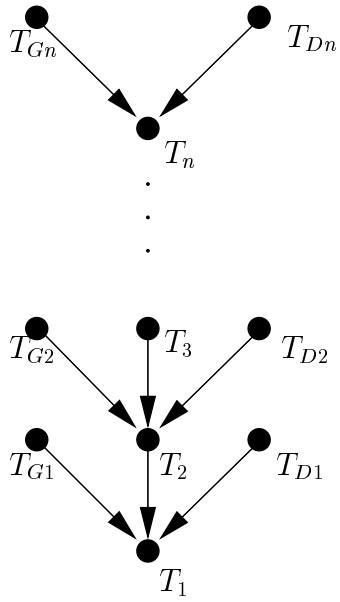


FIG. 3.4: *Si les $\Delta = 2n$ tâches T_{Gi} et T_{Di} $(1 \leq i \leq n)$ ont déjà été ordonnancées par ailleurs, elles sont en mémoire quand PTGDS ordonnance la chaîne T_1 à T_n*

Le théorème suivant donne une borne sur la qualité de grPTGDS par rapport au regroupement optimal.

Théorème 4 *Soit T_P le temps parallèle de grPTGDS pour un GdT donné, T_{opt} le temps parallèle d'un regroupement optimal pour le même GdT et g sa granularité. Alors sur un nombre non borné de processeurs :*

$$T_P \leq (1 + \frac{1}{g})T_{opt}$$

Pour prouver ce théorème, nous avons besoin du lemme suivant.

Lemme 2 *Soit T_{rl} le temps parallèle d'un regroupement linéaire quelconque. Sur un nombre non borné de processeurs on a :*

$$T_{opt} \leq T_{rl} \leq (1 + \frac{1}{g})T_{opt}$$

Preuve du lemme 2 : Le lemme est démontré par Gerasoulis et Yang dans [47]. ∎

D'après le lemme 2, si l'on prouve qu'il existe toujours un regroupement linéaire qui a un temps parallèle plus long que l'ordonnancement trouvé par grPTGDS, le théorème 4 est immédiatement prouvé.

D'après la définition d'un regroupement linéaire donnée dans le chapitre précédent, nous savons qu'un regroupement où chaque grappe est constituée d'une seule tâche est linéaire. Un tel regroupement minimal sera noté *min*. Nous pouvons alors prouver le thèorème suivant :

Théorème 5 *Soit $fin_{min}(t)$ la date de fin d'exécution de la tâche t ordonnancée suivant le regroupement minimal et $fin_P(t)$ la date de fin d'exécution de t ordonnancée suvant grPTGDS alors,*

$$fin_P(t) \leq fin_{min}(t)$$

Preuve du théorème 5 : Nous procédons par induction. Considérons t une tâche qui n'a pas de père. Comme grPTGDS ordonnance t au plus tôt on a $\text{fin}_\text{P}(t) \leq \text{fin}_\text{min}(t)$. Considérons une tâche t telle que quelque soit t' un de ces pères on ait : $\text{fin}_\text{P}(t') \leq \text{fin}_\text{min}(t')$. Si grPTGDS place t seule dans une grappe sa date de fin d'exécution sera plus petite que $\text{fin}_\text{min}(t)$. Comme grPTGDS place t dans la grappe qui minimise sa date de début d'exécution on a toujours $\text{fin}_\text{P}(t) \leq \text{fin}_\text{min}(t)$. ∎

Preuve du théorème 4 : Le théorème 5 prouve que grPTGDS produit un ordonnacement plus court que le regroupement minimal qui est un regroupement linéaire. Le lemme 2 montre que tout regroupement linéaire vérifie les bornes énoncées. ∎

Finalement on a le corollaire suivant

Corollaire 1 *Pour tout GdT à gros grain ($g \geq 1$) :*

$$T_P \leq 2 \times T_{opt}$$

Discussion :

Montrer que PTGDS est à $1 + \frac{1}{g}$ de l'optimal peut ne pas sembler un résultat intéressant dans la mesure où cette borne est obtenue si l'on place chaque tâche dans une grappe différente. Cependant, avec PTGDS le nombre de grappes est toujours très inférieur au nombre de tâches (En général le processeur qui minimise la date de début d'exécution d'une tâche est celui sur lequel est exécuté un des pères de cette tâche).

3.3.5 Minimiser l'impact dû au surcoût de l'ordonnancement

Nous donnons ici une condition suffisante pour que le surcoût de l'ordonnancement soit négligeable par rapport au temps parallèle.

Définition 3.2 *Soit G le graphe de tâches ordonnancé par PTGDS à partir d'un GTP pour une certaine valeur des paramètres.*
Soit T_{seq} le temps d'exécution séquentiel de G sur la machine cible $\mathcal{M}$.
On appelle M_{op} le nombre moyen d'opérations des tâches du graphe ($M_{op} = \frac{T_{seq}}{v\omega}$).
Soit T_{PTGDE} le temps d'exécution du programme dynamique.
Soit T_{PTGDS} le temps d'exécution de G ordonnancé par PTGDS sur $\mathcal{M}$
Soit $T_{maître}$ le temps d'exécution du maître.

Lemme 3 $T_{PTGDS} \leq T_{PTGDE} \leq T_{maître} + T_{PTGDS}$

Preuve du lemme 3 : À cause du surcoût de l'ordonnancement, le temps d'exécution du programme dynamique est plus grand que la seule exécution du graphe, une fois ordonnancé. De plus, comme PTGDS est exécuté par le maître et que le code parallèle est exécuté simultanément sur $\mathcal{M}$, le temps total de l'exécution du programme dynamique est plus petit que la séquentialisation du calcul de l'ordonnancement et de son exécution. ∎

Dans la majorité des cas, le nombre d'arcs d'un graphe de tâches est du même ordre que le nombre de nœuds. La principale limitation de notre approche est que l'ordonnancement dynamique est coûteux lorsque l'on traite des graphes de tâches à grain fin. Le théorème 6 donne une condition suffisante sur le nombre moyen d'opérations des tâches pour que le surcoût de l'ordonnancement soit négligeable en comparaison du temps total d'exécution.

Théorème 6 *Si* $e = O(v)$ *et* $p \max(p, \log v) = o(M_{op})$ *alors*

$$T_{PTGDE} = T_{PTGDS} + \epsilon$$

où ϵ *est un temps négligeable en comparaison de* T_{PTGDS}.

Preuve du théorème 6 : $T_{\text{maître}}$ est le temps d'exécution du maître. On a donc :

$$T_{\text{maître}} = \text{temps de l'ordonnancement} + \text{temps d'envoi des messages}$$

Le maître doit, au plus, communiquer autant de fois qu'il y a de tâches et qu'il y a d'arêtes dans le graphe.

$$T_{\text{maître}} = O((e+v)(p+\log v)) + O(e+v)$$

Posons $m = \max(p, \log v)$. Comme par hypothèse $e = O(v)$ on à :

$$T_{\text{maître}} = O(vm) + O(v)$$

C'est à dire qu'il existe un k tel que :

$$T_{\text{maître}} \leq kvm\omega \tag{3.1}$$

De plus, on ne peut pas avoir d'accélération super linéaire, donc :

$$T_{\text{PTGDS}} \geq \frac{T_{\text{seq}}}{p} \tag{3.2}$$

Par hypothèse $pm = o(M_{\text{op}})$. Ceci implique que $pm = o(\frac{T_{\text{seq}}}{v\omega})$ et donc que finalement $kvm\omega = o(\frac{T_{\text{seq}}}{p})$. En remplaçant dans les équations 3.1 et 3.2 on obtient $T_{\text{maître}} = o(T_{\text{PTGDS}})$. Comme d'après le lemme 3 : $T_{\text{PTGDS}} \leq T_{\text{PTGDE}} \leq T_{\text{maître}} + T_{\text{PTGDS}}$, on a : $T_{\text{PTGDE}} = T_{\text{PTGDS}} + \epsilon$. ∎

3.4 Comparaison entre une approche centralisée et une approche distribuée

La critique principale que l'on peut faire au schéma maître/esclave tel qu'il est présenté ici est qu'il s'agit d'une méthode centralisée. On pourrait lui préférer une approche décentralisée où chaque processeur exécute PTGDS et détermine les tâches qu'il doit exécuter ainsi que les messages qu'il doit envoyer.

Remarquons tout d'abord que, quelque soit l'approche choisie, les résultats théoriques concernant PTGDS restent valides. Comparons les mérites respectifs de ces deux méthodes :

- dans l'approche décentralisée, il n'y a plus de communication de contrôle. Comme le maître envoie autant de messages qu'il y a de nœuds et d'arcs dans le graphe la contention du réseau s'en trouve réduit et l'exécution de l'ordonnancement accélérée. Cette approche est aussi d'avantage extensible, le maître pouvant parfois jouer le rôle de goulot d'étranglement. Cependant dans les cas que nous avons étudiés le nombre de messages de contrôle généré par le maître est très inférieur, en ordre de grandeur, à la quantité de calcule. Il s'en suit que Le temps pris par l'envoie des messages et la contention du réseau ne doit pas être trop élevée.
- Dans le schéma maître/esclave, les processeurs qui exécutent les tâches ne sont pas les mêmes que celui qui calcule l'ordonnancement. Cela permet de recouvrir le calcule de l'ordonnancement avec l'exécution. De plus, bien que PTGDS soit un algorithme peu gourmand en mémoire, il en requiert parfois beaucoup. Centraliser l'ordonnancement permet donc de gagner de la mémoire sur les processeurs qui exécutent les tâches, ce qui est un des objectifs prioritaires de la démarche. Si le réseau est suffisement rapide il permet de recouvrir le calcul de l'ordonnancement avec son exécution.

Il apparaît que chacune de ces approches correspond à un type d'application. On préférera l'approche centralisée si les problèmes de mémoire sont critiques. On préférera l'approche décentralisée si on recherche une exécution extensible.

3.5 Résultats expérimentaux

Dans le but de valider notre approche nous avons étudié les performances de PTGDS pour certains noyaux de calcul décrits dans l'annexe 7. Nous avons tout d'abord effectué des mesures d'accélération pour montrer l'efficacité des ordonnancements trouvés par PTGDS. Nous avons ensuite étudié le coût mémoire de PTGDS : pour chaque GdT nous avons, comparé sa taille (nombre d'arcs et de sommets) avec le nombre maximum d'arcs et de sommets en mémoire à un instant donné. Enfin nous avons comparé le temps de calcul de l'ordonnancement avec le temps parallèle et le temps séquentiel de l'application.

Comme décrit dans le chapitre précédent, notre modèle de machine est formé de quatre paramètres. Pour nos simulations nous avons pris α =0,06μs/double, β =4,4μs et ω =0,02μs, ce qui correspond aux performances crêtes de la POPC sous BIP et $p \in [1, 32]$

Nous avons mené nos tests sur les noyaux de calcul intensifs suivants :

- *Gauss* est l'élimination de Gauss sans pivotage,

- *Givens* est l'algorithme de Givens,
- *Gauss & BS* est l'élimination de Gauss sans pivotage suivit d'une résolution triangulaire,
- *Jordan* est l'algorithme de diagonalisation de Jordan,
- *Mult_ mat* est un algorithme de multiplication de matrice (Gaxpy par ligne),
- *Power* est le calcul d'une puissance $m^{ième}$ d'une matrice. Pour chaque puissance intermédiaire un calcul est effectué sur le résultat.

Toutes les matrices sont d'ordre n. Plus de détails sur ces noyaux peuvent être trouvés dans l'annexe 7

3.5.1 Le simulateur

Pour mener à bien un certain nombre des expériences décrites plus bas nous avons mis au point un simulateur de PTGDS que nous avons intégré à PlusPyr. Il s'agit d'un programme qui exécute PTGDS sur un graphe de tâches paramétré particulier en fonction des valeurs des paramètres du programme et de la machine cible. Il est composé d'une partie indépendante de l'application et d'une partie qui est générée automatiquement à partir du GTP.

La partie indépendante de l'applications concerne l'algorithme PTGDS l'implantation de l'algorithme sur les arbres AVL, la fonction d'évaluation des polynômes de Ehrhart, et l'affichage des résultats.

À partir du GTP nous générons automatiquement les nids de boucles qui parcourent l'ensemble des pères de l'instance d'une tâche donnée, qui calculent la durée, le nombre de fils et le nombre de pères de l'instance d'une tâche donnée. Nous générons aussi les structures qui décrivent les polynômes de Ehrhart nécéssaires.

La figure 3.5 montre les codes générés pour parcourir l'ensemble des pères de T_1 et pour calculer le nombre de pères de T_1 à partir de la règle : $\{T1(k)|2 \leq k \leq n-1\} \leftarrow \{T2(k-1,j)|2 \leq j \leq k\} : \{A(l,k)|k \leq l \leq n\}$

Pour calculer le nombre de pères d'une tâche `t` il se peut qu'il y ait plusieurs règles qui décrivent des arcs arrivant sur `t`. On incrémente donc le nombres de pères courant (`res`) pour chaque règles.

3.5.2 Simulation d'accélération

Nous avons simulé l'exécution de l'ordonnancement en suivant les modèles décrits dans le chapitre précédent.

La figure 3.6 donne les simulations d'accélérations pour une valeur de 1000 et 3000 de la taille des matrices.

Ces accélérations sont toujours supérieures à 29 pour 32 processeurs. Cependant, cela ne signifie pas que l'exécution des programmes suivant l'ordonnancement calculé par PTGDS donnera lieu à une accélération linéaire. Cela signifie juste que PTGDS arrive à allouer de manière satisfaisante les tâches sur les processeurs.

```
void ordonnance_pere(tache *t){
  tache *pere ;
...
  {
    int k,j ;
      if(t->numero==1){
        k=t.param[0] ;
        if((k-2>=0)&&(n-k-1>=0)){
          /*cree une tache T2
            avec 2 paramètres*/
          pere=cree_tache(2,2) ;
          for(j=2 ;j<=k ;j++){
            pere->param[0]=k14-1 ;
            pere->param[1]=j ;
            PTGDS(pere) ;
          }
          libere(pere) ;
        }
      }
    }
...
}
```

```
int nb_peres(tache *t){
  int res=0 ;
  /* le type "Enumeration" decrit un
  polynome de Ehrhart*/
  Enumeration *poly ;
...
  {
    int k ;
    if(t->numero==1){
      k=t->param[0] ;
      if((k-2>=0)&&(n-k-1>=0)){
        /* on affecte poly au polynôme
        qui correspond aux contraintes
        décritent par la règle de
        communication*/
        poly=construit_poly9() ;
        /* n est une variable globale.
        calcule_poly evalue le polynôme
        en fonction de la valeur du
        parametre n*/
        res+=calcule_poly(poly,n) ;
      }
    }
  }
...
  return res ;
}
```

FIG. 3.5: *Codes C pour parcourir et ordonnancer les pères de la tâche T_1 (à gauche) et pour calculer le nombre de pères de T_1 (à droite)*

3.5.3 Coût mémoire

Expériences sur des programmes réels

Les tableaux 3.1, 3.2, 3.3 et 3.4 comparent la taille du GdT ordonnancé par PTGDS pour différentes valeurs des paramètres et le coût mémoire requis par ce dernier.

La colonne *# Nœuds* est le nombre de nœuds du GdT et la colonne *Max # nœuds* est le nombre de nœuds maximum en mémoire pendant l'exécution de l'algorithme, c'est à dire la taille maximum de l'arbre AVL. On constate que, mis à part pour *power* le nombre de nœuds croît quadratiquement alors que le coût mémoire est linéaire en fonction de la taille de la matrice. Le cas pathologique de *power* vient du fait qu'une partie du GdT est un pire cas pour la complexité mémoire : lorsque PTGDS ordonnance le GdT de PTGDS il garde en mémoire presque toutes les tâches avant d'ordonnancer une chaîne dont la somme des degrés

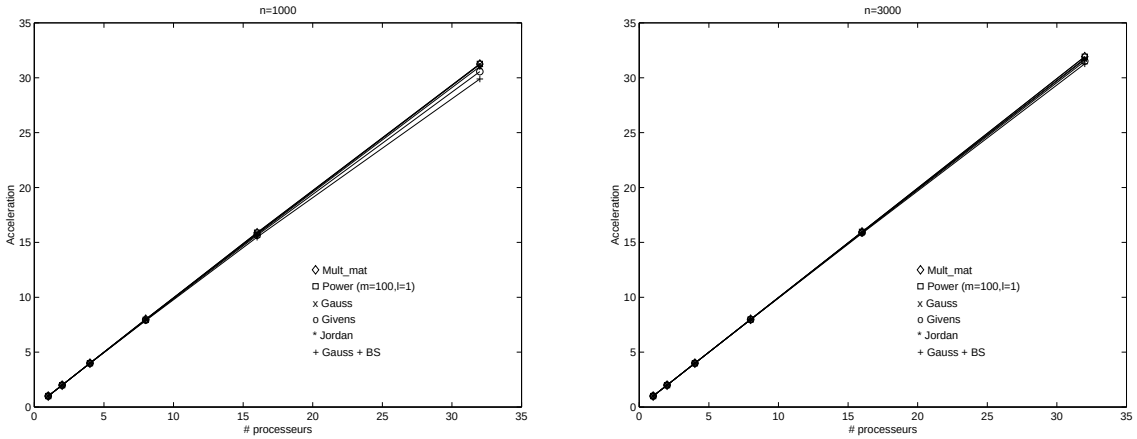


FIG. 3.6: *Simulations d'accélération pour différentes valeur des paramètres*

Programme	Prof. rec.	# Nœuds	Max # nœuds	# Arcs	Max # arcs
Gauss	198	5150	200	10199	200
Givens	197	4952	199	9900	296
Gauss & BS	100	5052	200	10001	200
Jordan	100	5052	200	10001	200
mult_mat	100	10002	201	20000	299
Power (m=100)	101	10002	9805	29801	10000

TAB. 3.1: *Coût mémoire de PTGDS* ($n = 100$)

entrant est de l'ordre de n^2 (voir la preuve de la complexité mémoire de PTGDS section 3).

La colonne *# Arcs* représente le nombre d'arcs du GdT et la colonne *Max # arcs* représente le nombre maximum d'arcs. Ici aussi, on constate que le nombre d'arcs croit quadratiquement alors que le coût mémoire est linéaire en fonction de n. L'exception est ici aussi *power*, pour la même raison qu'énoncée précédemment.

La colonne *Prof. rec.* donne la profondeur maximale des appels récursif de PTGDS. On remarque cette valeur est proportionnelle à n (sauf pour *power* où elle est proportionnelle à m. Cela vient du fait que les graphes que l'on ordonnance sont issus de domaines triangulaires ou rectangulaires de hauteur n (m pour *power*).

Il se peut aussi que le GTP ne soit pas parfaitement optimisé et que l'on observe des multi-arcs dans le GdT. Ceci n'induit pas un surcoût mémoire car ils sont fusionnés par PTGDS lors de l'exploration. En revanche les multi-arcs impliquent un surcoût de contrôle donc augmentent la durée du calcul de l'ordonnancement.

Expérience sur des graphes aléatoires

Dans les tables 3.5, 3.6 et 3.7 nous analysons le coût mémoire de PTGDS pour différents graphes générés aléatoirement.

Programme	Prof. rec.	# Nœuds	Max # nœuds	# Arcs	Max # arcs
Gauss	1998	501500	2000	100199	2000
Givens	1997	499502	1999	999000	2996
Gauss & BS	1001	500502	2001	1499501	2001
Jordan	1000	500502	2000	1000001	2000
mult_mat	1000	1000002	2001	2000000	2999
Power (m=100)	101	99102	98005	297101	99100

TAB. 3.2: *Coût mémoire de PTGDS ($n = 1000$)*

Programme	Prof. rec.	# Nœuds	Max # nœuds	# Arcs	Max # arcs
Gauss	3998	2003000	4000	4003999	4000
Givens	3997	1999002	3999	3998000	5996
Gauss & BS	2001	2001002	4001	5999001	4001
Jordan	2000	2001002	4000	4000001	4000
mult_mat	2000	4000002	4001	8000000	5999
Power (m=100)	101	198102	196005	594101	198100

TAB. 3.3: *Coût mémoire de PTGDS ($n = 2000$)*

Pour chaque graphe nous avons un nombre moyen de niveaux (colonne *niveaux*) et un nombre moyen de tâches par niveaux (colonnes *tâches*). Dans les graphes faiblement (resp. moyennement, fortement) couplés, chaque nœud a un nombre moyen de 2 (resp. 4, 8) fils. Les fils sont choisis parmi les tâches des deux niveaux suivants. Les distributions sont toutes uniformes. Chaque ligne représente un type de graphe. Nous avons construit dix graphes de chaque type. Mous avons calculé les valeurs moyennes de la profondeur maximale de la récursivité, du nombres de nœuds du nombre maximal de nœuds en mémoire, du nombre d'arcs et du nombre maximal d'arcs en mémoire (entre parenthèses est donné l'écart type de cette moyenne). Nous pouvons affirmer, tout d'abord, que nos expériences sont pertinentes car l'écart type est toujours inférieure à 10% de la moyenne.

Nous remarquons que quelque soit le type de graphe considéré, il y a toujours moins

Programme	Prof. rec.	# Nœuds	Max # nœuds	# Arcs	Max # arcs
Gauss	5998	4504500	6000	9005999	6000
Givens	5997	4498502	5999	8997000	8996
Gauss & BS	3001	4501502	6001	13498501	6001
Jordan	3000	4501502	6000	9000001	6000
mult_mat	3000	9000002	6001	18000000	8999
Power (m=100)	101	287102	294005	891101	297100

TAB. 3.4: *Coût mémoire de PTGDS ($n = 13000$)*

niveaux	tâches	Prof. rec.	# nœuds	max nœuds	# arcs	max arcs
100	20	80(7)	1996(131)	158(11)	3906(270)	99(8)
100	40	79(4)	3992(229)	221(10)	7889(469)	100(6)
100	80	78(6)	7932(458)	354(12)	15730(939)	100(7)
200	20	154(10)	3863(224)	249(20)	7586(435)	185(14)
200	40	154(8)	7803(453)	314(15)	15440(901)	186(11)
200	80	153(10)	15741(1011)	441(23)	31292(1970)	187(13)
400	20	315(9)	7896(269)	434(16)	15519(537)	366(12)
400	40	308(21)	15998(976)	508(29)	31702(1942)	365(24)
400	80	307(19)	31393(1786)	630(34)	62475(3549)	361(24)
800	20	635(25)	16245(610)	836(42)	31960(1189)	741(35)
800	40	629(41)	32347(2089)	890(41)	64154(4158)	736(40)
800	80	607(38)	62868(3935)	992(43)	125196(7839)	709(40)

TAB. 3.5: *Expériences sur des graphes faiblement couplés*

d'arcs et de nœuds en mémoire que le nombre d'arcs et de nœuds du graphe.

Dans tout les cas, le coût mémoire de PTGDS est très faible devant la taille du graphe ordonnancé. La profondeur maximale des appels récursifs de PTGDS croit avec le couplage du graphe. Cela est dû à la manière dont sont générés les graphes. Ce sont toujours les arcs du niveaux immédiatement supérieurs qui sont explorés en premier. Ainsi moins il y a d'arcs dans le graphe plus grande est la chance d'avoir comme premier père une tâche deux niveaux au dessus. Une fois que ce père est ordonnancé il n'y aura plus d'appel de PTGDS pour celui-ci à partir de ses autres fils.

Au vu de ces expériences nous pouvons conclure que le coût mémoire de notre algorithme est faible. En effet si l'on effectue un calcul de régression linéaire sur l'ensemble des données, on peut modéliser le nombre maximum de nœuds en mémoire (n^*) par :

$$n^* = 0,94l + 3,02t + 0,63f$$

si l est le nombre de niveaux, t est le nombre de tâches par niveaux et f est le nombre moyen de fils par tâches. Ce modèle linéaire ayant une erreur relative moyenne d'environ 2% ($\pm 1\%$), on en déduit avec une bonne «confiance», que n^* est bien une fonction linéaire des paramètres. Le même type de calcul sur le nombre maximum d'arcs (a^*) en mémoire donne :

$$a^* = 0,94l + 0,02t + 7,15f$$

L'erreur relative moyenne étant de 3% ($\pm$ 3%) on en déduit que le phénomène modélisé ici est lui aussi linéaire.

Un calcul de régression linéaire entre coût mémoire de l'algorithme ($c^* = n^* + a^*$) et la taille du graphe ($c = n + a$) où n est le nombre de nœuds et a le nombre d'arcs montre que :

$$\log c^* = 0,48 \log c + 1,33$$

L'erreur relative moyenne étant de 4% ($\pm$ 3%), on en déduit que :

$$c^* = O(\sqrt{c})$$

niveaux	tâches	Prof. rec.	# nœuds	max nœuds	# arcs	max arcs
100	20	92(6)	1978(105)	152(6)	7490(393)	104(8)
100	40	90(5)	3838(205)	212(7)	14871(842)	104(5)
100	80	92(6)	7965(368)	336(10)	31222(1488)	110(6)
200	20	182(11)	3935(217)	244(11)	14988(842)	196(11)
200	40	177(12)	7777(490)	299(15)	30283(1862)	193(14)
200	80	185(9)	16473(680)	428(15)	64838(2683)	204(9)
400	20	362(12)	7897(270)	429(13)	30149(1023)	382(14)
400	40	352(22)	15471(854)	479(24)	60320(3320)	372(22)
400	80	356(15)	31585(1467)	607(17)	124565(5783)	376(16)
800	20	715(31)	15592(642)	785(33)	59553(2491)	737(34)
800	40	718(41)	31834(1938)	852(41)	124314(7566)	744(42)
800	80	711(27)	63766(2112)	964(31)	251704(8333)	740(28)

TAB. 3.6: *Expériences sur des graphes moyennement couplés*

3.5.4 Temps d'exécution

Nous étudions ici les conditions pour que le temps d'ordonnancement de PTGDS soit petit par rapport au temps d'exécution parallèle.

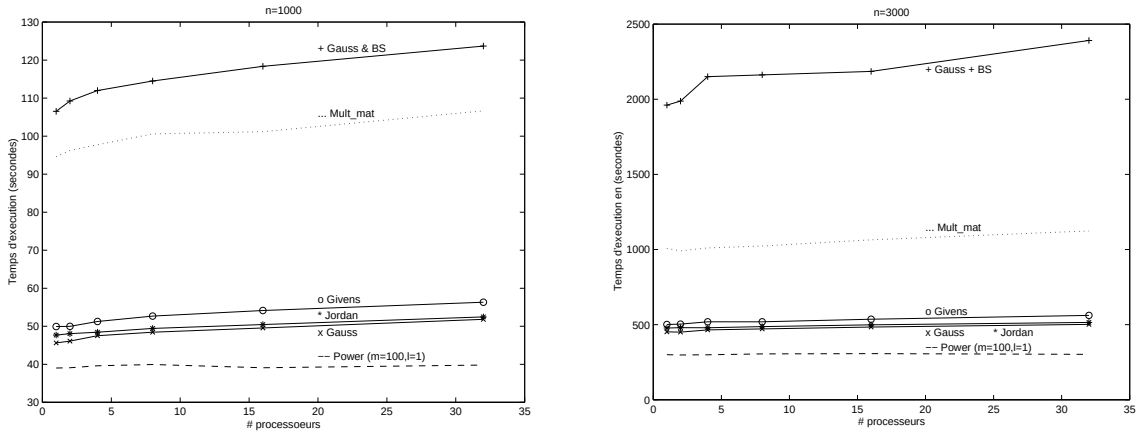


FIG. 3.7: *Temps d'ordonnancement des différents noyaux*

La figure 3.7 montre l'évolution du temps d'exécution de PTGDS pour différentes tailles de matrices ($n = 1000$ et $n = 3000$) en fonction du nombre de processeurs. Il apparaît que, conformément au calcul de la complexité, le temps pour calculer l'ordonnancement croît légèrement avec le nombre de processeurs.

On remarque que, pour des grandes valeurs de n, le temps d'ordonnancement de PTGDS est très compétitif. Cela est particulièrement vrai pour *power* ou la quantité de calcul est très grande.

niveaux	tâches	Prof. rec.	# nœuds	max nœuds	# arcs	max arcs
100	20	101(6)	1982(110)	157(7)	14237(813)	118(6)
100	40	97(7)	3868(280)	214(9)	29186(2135)	118(7)
100	80	99(6)	7947(389)	330(10)	61504(3003)	123(8)
200	20	204(12)	4020(224)	261(13)	29056(1642)	220(13)
200	40	197(12)	7829(437)	311(13)	59426(3319)	216(11)
200	80	194(11)	15705(877)	428(11)	122134(6841)	218(11)
400	20	386(24)	7665(486)	442(23)	55514(3629)	402(23)
400	40	401(27)	16106(1073)	515(26)	122553(8146)	421(27)
400	80	388(20)	31275(1652)	620(18)	243713(13041)	410(22)
800	20	767(23)	15243(459)	824(23)	110518(3243)	783(23)
800	40	782(50)	31399(2001)	898(49)	239168(15294)	802(50)
800	80	786(31)	63439(2566)	1021(28)	494941(20108)	810(31)

TAB. 3.7: *Expériences sur des graphes fortement couplés*

3.5.5 Comparaison entre le surcoût de l'ordonnancement et le temps d'exécution

Nous avons comparé le temps d'ordonnancement avec une simulation de l'exécution de power parallèle sur la plateforme POPC du LHPC.

Cette expérience confirme le résultat théorique du théorème 6. En effet on constate que lorsque $9000 \times p^2 \approx M_{\text{op}}$ on a $T_{\text{PTGDS}} = 17 \times T_{\text{maître}}$.

3.6 Conclusion et perspectives

Dans ce chapitre, nous avons étudié PTGDS, un algorithme d'ordonnancement de graphes de tâches paramétrés.

Du point de vue théorique nous avons calculé la complexité mémoire et temporelle de cet algorithme. Nous avons montré qu'il fournit une solution 2-approchée sur un nombre non borné de processeurs si le graphe est à gros grain.

Les simulations montrent que l'ordonnancement fourni par PTGDS donne une bonne occupation des processeurs et minimise bien les communications. Nos études de coût mémoire montrent que, aussi bien pour des noyaux standard de calcul, que pour des graphes aléatoires PTGDS est en général très peu coûteux en mémoire.

Nous avons présenté comment cet algorithme peut-être intégré dans un schéma maître-esclave d'ordonnancement dynamique. Nous avons prouvé que le surcoût de l'ordonnancement dynamique s'efface dès que la durée des tâches est suffisamment grande.

Cette méthode, telle qu'elle est présentée ici, est centralisée. Cependant, on peut aisément la décentraliser en faisant exécuter PTGDS sur tous les processeurs en même temps. Nous pensons que le surcoût ne serait guère plus élevé. De cette manière l'extensibilité de cette approche serait accrue car alors il n'y aurait plus de messages de contrôle.

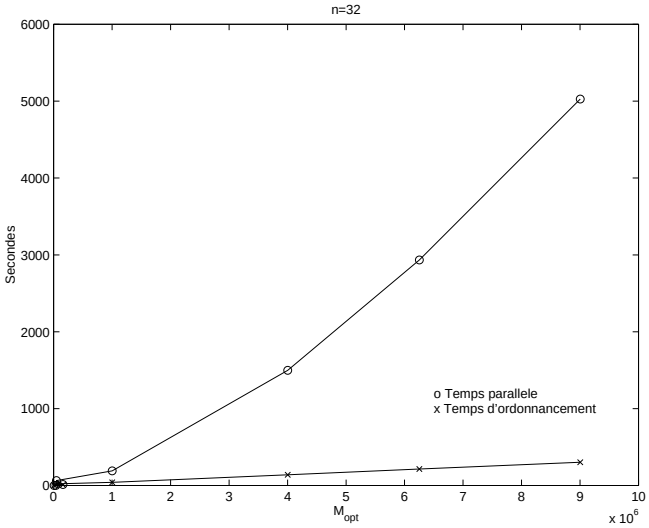


FIG. 3.8: *Comparaison entre le temps de l'ordonnancement et le temps parallèle*

PTGDS peut aussi être utilisé comme un algorithme d'ordonnancement statique. Pour ce faire, il faut, avant de supprimer les informations inutiles sur les tâches déjà ordonnancées, les sauver dans un fichier. Dans ce cas la méthode n'est plus adaptative (il n'y a plus de programme générique), en revanche il est toujours possible de traiter de gros graphes.

Chapitre 4

Allocation symbolique de graphe de tâches paramétrés

4.1 Introduction

Dans ce chapitre, nous présentons SLC (*Symbolic Linear Clustering*), une heuristique d'allocation symbolique du GTP. Il s'agit d'une méthode qui permet d'associer à chaque tâche générique une fonction qui donne, suivant les valeurs de son vecteur d'itération des paramètres et du nombre de processeurs de la machine, le processeur sur lequel doit être exécutée la tâche. Dans la suite de cette introduction, nous montrons pourquoi une telle approche répond aux problèmes posés au début de cette thèse. Puis, nous détaillons les travaux qui sont connexes en relevant que tous sont basés sur une analyse à grain fin des dépendances. Enfin, nous donnons un aperçu général de SLC. La section 4.2 présente en détail notre algorithme ainsi que les techniques utilisées pour calculer la solution. À la fin de cette section on décrit comment on calcule l'ensemble des tâches de démarrage (celles qui doivent être exécutées en premier) ainsi que l'ensemble des tâches de fin (les dernières à être exécutées) et comment on évalue le nombre de grappes de l'application. La section 4.3 montre comment se comporte SLC pour un certain nombre de noyaux de calcul intensif et compare les allocations trouvées avec des algorithmes d'ordonnancement statiques.

4.1.1 Une réponse à notre problématique

Être capable de trouver une allocation symbolique d'un graphe de tâches paramétré permet de résoudre les problèmes énoncés dans le chapitre 1. En effet :

- L'allocation trouvée par SLC est une fonction des paramètres du programme : elle est virtuellement résolue pour n'importe quelle taille du problème.
- Les fonctions calculées par SLC ne dépendent que du texte du programme source. Le temps pour les calculer est indépendant de la taille du problème.
- Pour évaluer une fonction d'allocation, il est seulement nécessaire de connaître la valeur du vecteur d'itération de la tâche ainsi que les paramètres du programme et de la machine cible. Cela se fait donc de manière décentralisée et en temps constant.

4.1.2 Les techniques existantes

Le parallélisme automatique est un ensemble de techniques et d'algorithmes qui permet d'extraire et d'exploiter automatiquement (à l'aide d'un ordinateur) le parallélisme d'un programme. Le but ultime des recherches menées dans ce domaine est de mettre au point un compilateur qui prend en entrée un programme séquentiel et qui, en sortie, donne un programme parallèle. Aujourd'hui, la classe des programmes qui peuvent être parallélisés automatiquement est très restreinte. En particulier, il est très difficile de prévoir le comportement d'un programme sans l'exécuter. Il est donc difficile de répartir les calculs et d'organiser les communications de telle sorte que le programme parallèle ainsi obtenu, ait un comportement satisfaisant. Cependant, quand les programmes sont écrits suivant des critères précis, il est possible de placer efficacement les calculs et les données sur les processeurs. La plupart des travaux menés dans cette direction ont en commun de traiter le cas où les dépendances sont affines. Certaines techniques d'allocations des données sont basées sur la règle «*owner compute rule*» [71]. Cela signifie que les données sont placées sur le processeur qui les calcule. Alternativement, si on alloue les calculs suivant la règle «*owner compute rule*», cela signifie que chaque calcul est effectué sur le processeur qui possède le membre gauche de l'affectation.

Dans le cadre de programmes à base de nids de boucles, Anderson et Lam [6] proposent un algorithme qui place les calculs et les données de manière à ce que le parallélisme soit maximisé et les communications minimisées. Dans une telle approche, la règle «*owner compute rule*» n'est en général pas vérifié. Le code généré à l'aide de cet algorithme est formé de `doall` ou de `forall`. Lorsque plusieurs nids de boucles sont séquentialisés l'algorithme décide s'il faut redistribuer les données et les calculs. Ce travail a été intégré dans le compilateur SUIF [4].

Dans [27], Darte et Robert on introduit la notion de graphe de communication pour modéliser les données lues ou écrites par chaque instruction dans le cas de nids de boucles affines. Ils prouvent que le problème de l'alignement des calculs avec les données de manière à minimiser les communications est NP-complet dans le cas simple du nid de boucles uniforme et proposent alors une heuristique.

Ces travaux sont étendus par Dion et Robert dans [29]. Ils introduisent la notion de graphe d'accès qui est un graphe de communication où les arêtes sont étiquetées par les matrices d'accès aux tableaux. Ils prouvent que le problème de l'alignement des données avec les calculs en minimisant les communications est NP-complet pour le cas d'un nid de boucles affine et proposent alors plusieurs heuristiques.

Feautrier dans [34] propose un algorithme pour couper les arcs du graphe de flot de données. Il propose un algorithme glouton pour choisir les arcs et applique l'élimination de Gauss pour trouver la fonction de placement des instructions. Les données sont alors placées en suivant la règle «*owner compute rule*».

Dans [35] Feautrier étend cet algorithme pour trouver une distribution automatique des données en même temps que le placement des calculs. Pour cela, il utilise le graphe de communication du programme où chaque composante connexe de ce graphe est associée à un processeur virtuel. Comme dans le cas général le graphe de communication n'a qu'une seule composante connexe, il utilise un algorithme glouton pour sélectionner les contraintes

à prendre en compte.

Dans [60], Mongenet traite le problème de l'allocation des données et de l'ordonnancement des instructions quand le programme est représenté sous forme d'un système d'équations affines récurrentes. Les dépendances sont classées en fonction de la dimension des éléments transmis. C. Mongenet donne des conditions sur les dépendances pour que celle-ci n'impliquent pas de communications et montre comment on peut localiser les communications point à point et les diffusions. Une fois les données placées, elle utilise la règle «*owner compute rule*» pour placer les calculs.

Les travaux que nous venons de citer permettent de placer les instructions sur les processeurs, en cela l'analyse est à grain fin. Cela signifie que chaque opération et chaque communication sont considérées comme unitaire. S'il reste trop de communications, il peut en résulter une grande perte d'efficacité. Il existe des techniques qui une fois le placement effectué permettent de regrouper les calculs entre eux (tuilage, etc...). Le but final étant de n'avoir plus qu'une tâche par processeur. Notre approche a ceci d'original que le regroupement des calculs en tâche se fait avant la phase de placement : le grain est fixé à priori. Ainsi, le fait que le programme à paralléliser soit à gros grain permet de sélectionner plus facilement les communications à supprimer, en contre partie le fait que chaque tâche et chaque communication ait une durée différente complexifie le problème.

En disant cela, notre but, n'est pas de prouver la supériorité d'une approche sur l'autre, mais bien de montrer que les techniques à grain fin ne sont pas utilisables directement, sans précaution, pour notre problème.

4.1.3 Présentation générale de SLC

Une des grandes difficultés du parallélisme est de trouver un moyen terme entre un parallélisme total (toutes les tâches/instructions sont sur un processeur différent) et une suppression complète des communications (toutes les tâches/instructions sont sur le même processeur). La majorité des programmes parallélisés suivant un de ces deux schémas ont un très mauvais comportement.

Dans ce chapitre nous analysons les règles de communications et allouons les tâches en supprimant des communications. Or, si l'on ne prends garde, toutes les communications seront supprimées et le parallélisme détruit. Pour éviter cet écueil nous imposons qu'un certain nombre de communications soient conservées. Pour ce faire, SLC (*Symbolic Linear Clustering*) construit un regroupement linéaire des tâches à partir du GTP. Dans un regroupement linéaire les tâches sont regroupées en grappes (processeur virtuel) qui correspondent à un chemin du graphe. Aucune tâche indépendante ne se trouve dans la même grappe.

Intuitivement, le fait de regrouper ensemble des tâches le long d'un chemin permet de supprimer des communications inutiles tout en préservant du parallélisme.

Les communications que nous conservons sont celles qui empêchent d'obtenir un regroupement linéaire. Elle sont décritent par deux types de règles :

- les règles qui correspondent à une diffusion ou à un regroupement de données,
- les règles qui rentrent en *conflit* avec une autre. Si deux règles sont en conflit, cela signifie, que pour certaines instances de ces deux règles on a un schéma de type «*join*» ou

«*fork*» pour des tâches qu'elles décrivent, ce qui empêche de réaliser un regroupement linéaire.

Nous avons vu dans 3.3.4 qu'un regroupement linéaire est à $1 + 1/g$ de l'optimal, si g est le grain du graphe. De plus dans [47], Yang et Gerasoulis montrent que tout regroupement non linéaire d'un graphe à gros grain peut être transformé en regroupement linéaire qui a un temps d'exécution parallèle inférieur ou égal à celui du regroupement non linéaire. Cela prouve que, pour chaque graphe à gros grain, il existe un regroupement linéaire optimal.

Les différentes étapes de SLC sont décrites ci-après :

1. On supprime toutes les règles qui concernent les tâches d'entrée ou de sortie. Ces tâches sont artificielles et elles ne correspondent pas réellement à un calcul dans le programme. Elles décrivent comment distribuer les données. Comme on s'intéresse à paralléliser des noyaux de calculs ceux-ci seront intégrés pour une vraie application dans un programme parallèle plus vaste.

2. L'étape suivante consiste à modifier le GTP en fusionnant les règles de manière à en réduire le nombre au maximum. En effet, la durée d'exécution de SLC étant fonction du nombre de règles, plus ce nombre est faible plus l'allocation sera calculée rapidement et plus la génération des messages se fera rapidement à l'exécution. Nous avons décrit dans 2.3.6 comment simplifier un GTP.

3. Nous analysons ensuite chacune des règles de communication pour extraire les *règles bijectives*. Il s'agit de règles qui décrivent des communications point à point (ce ne sont ni des diffusions ni des regroupements de données).

4. Les règles bijectives sont alors triées. En effet, comme nous le verrons plus tard, l'ordre dans lequel les règles sont examinées est très important. Tout d'abord, on trie les règles suivant la taille des données qu'elles transmettent, puis, on met à la fin les règles qui décrivent des arcs transitifs dans le GTP.

5. A partir de l'ensemble constitué des règles bijectives on construit un regroupement des tâches à l'aide du processus de *mise à zéro* décrit section 4.2.1. Une règle décrit un ensemble d'arcs et d'arêtes. Mettre à zéro une règle consiste à regrouper sur un même processeur virtuel les deux tâches qui se trouvent aux extrémités de chaque arêtes. En outre, on impose que ce regroupement soit transitif. Ainsi, si une règle est bijective, elle décrit un ensemble de chemins indépendant dans le graphe. Donc, une fois mise à zéro toutes les tâches d'un même chemin seront allouées au même processeur virtuel. Pour chaque règles bijective mises à zéro nous obtenons un regroupement en grappes.

   Nous linéarisons ce regroupement en extrayant,parmi les règles bijectives, un ensemble de règles qui ne sont pas en *conflit*. Deux règles sont en conflit si, une fois mises à zéro, le regroupement ainsi construit n'est pas linéaire.

6. Une fois le regroupement linéaire construit la dernière étape est l'identification des grappes. Il s'agit de trouver une fonction qui, une fois les valeurs des paramètres connus, donne un numéro de grappe à chaque tâche suivant le regroupement linéaire trouvé à l'étape précédente.

4.2 SLC

Nous examinons ici, en détail l'algorithme de regroupement linéaire symbolique que nous avons mis au point.

Soit R une règle d'émission :

$$R : T_a(\vec{u}) \rightarrow T_b(\vec{v})|P$$

Nous cherchons une fonction κ qui, pour l'instance d'une tâche, donne le numéro de la grappe sur laquelle elle va être exécutée : $\kappa(T_a, \vec{u})$ est le numéro de la grappe de la tâche T_a pour la valeur $\vec{u}$ de son vecteur d'itération.

4.2.1 Mise à zéro des règles

Mettre à zéro une règle revient à annuler toutes les communications qu'elle implique. La règle $R : T_a(\vec{u}) \rightarrow T_b(\vec{v})|P$ est mise à zéro si et seulement si pour tout $\vec{u}$ et tout $\vec{v}$ dans P on a :

$$\kappa(T_a, \vec{u}) = \kappa(T_b, \vec{v})$$

Les étapes 2 à 5 de la présentation générale de SLC section 4.1.3 sont résumées dans l'algo-

Entrée : un GTP=$(\mathcal{T}, \mathcal{R}, \mathcal{C})$
Sortie : l'ensemble $\mathcal{Z}$ des règles sélectionnées
1 $\mathcal{Z} = \emptyset$;
2 `fusionner_règles`$(\mathcal{R})$;
3 $\mathcal{B}$= `extrait_règles_bijectives`$(\mathcal{R})$;
4 `trier`$(\mathcal{B})$;
5 **pour chaque** r **dans** $\mathcal{B}$ **faire**
6 **si** `pas_en_conflit`$(r, \mathcal{Z})$ **alors**
7 $\mathcal{Z}$+=r ;
8 **finsi**
9 **finpour**

FIG. 4.1: *L'algorithme de selection des règles*

rithme de la figure 4.1 qui sélectionne l'ensemble des règles qui vont former un regroupement linéaire. Les lignes 3 à 7 de l'algorithme vont maintenant être décrites. Puis une preuve que l'ensemble $\mathcal{Z}$ forme bien un regroupement linéaire sera donnée.

4.2.2 Recherche de règles bijectives

Pour construire un regroupement linéaire, il est nécessaire que toutes les règles qui vont être mises à zéro décrivent des communications point à point. Nous appelons de telles règles des règles *bijectives*. Plus précisement, soit une règle d'émission R de la forme :

$$R : T_a(\vec{u}) \rightarrow T_b(\vec{v})|P$$

R est une règle bijective si pour chaque instance valide de $\vec{u}$ il existe une et une seule instance valide de $\vec{v}$.

Dans ce qui suit, nous décrivons une méthode pour déterminer si R est une règle bijective.

Soit $\vec{\imath}$ le vecteur constitué des paramètres et des variables des vecteurs d'itérations des tâches génériques. Il existe alors deux matrices D_1 et D_2 telles que :

$$\vec{u} = D_1\vec{\imath} + \vec{k_1}$$

et

$$\vec{v} = D_2\vec{\imath} + \vec{k_2}$$

où $\vec{k_1}$ et $\vec{k_2}$ sont deux vecteurs d'entiers constants.

Tout d'abord, nous supprimons les égalités concernant les variables de P en faisant les changements de variables nécessaires. Ceci est toujours possible car P est un polytope (toutes les variables sont encadrées). Ensuite nous construisons D_1 et D_2. Puis nous trouvons E_1 et $\vec{w_1}$ tels que $\vec{\imath} = E_1(\vec{u} - \vec{k_1}) + \vec{w_1}$. Il existe beaucoup de E_1 et $\vec{w_1}$ qui vérifient la précédente équation. Nous imposons que $\vec{w_1}$ soit composé des variables qui n'apparaissent pas dans $\vec{u}$. Formellement, $w_{1_j} = \delta_j i_j$ où les w_{1_j} (resp. i_j) sont les j$^{\text{ème}}$ éléments de $\vec{w_1}$ (resp. $\vec{\imath}$) ; $\delta_j = 0$ si i_j apparaît dans $\vec{u}$ sinon $\delta_j = 1$. Ainsi nous avons $\vec{v} = D_2E_1(\vec{u} - \vec{k_1}) + D_2\vec{w_1} + \vec{k_2}$. De la même manière nous construisons E_2 et $\vec{w_2}$ tels que $\vec{u} = D_1E_2(\vec{v} - \vec{k_2}) + D_1\vec{w_2} + \vec{k_1}$. Nous avons alors le théorème suivant :

Théorème 4.1 *R est une règle bijective si et seulement si $D_2\vec{w_1}$ et $D_1\vec{w_2}$ sont tous deux constants (on considère les paramètres du programme comme des constantes).*

Preuve du théorème 4.1 : Il est clair que si $D_2\vec{w_1}$ et $D_1\vec{w_2}$ sont tous deux constants alors R est une règle bijective : pour chaque instance valide de $\vec{u}$ il y a une et une seule instance de $\vec{v}$ et vice-versa.

Supposons que R soit une règle bijective. Cela signifie que pour chaque instance valide de $\vec{u}$ il y a une et une seule instance valide de $\vec{v}$. Ainsi après les changements de variables dûs aux égalités toutes les variables de $\vec{v}$ apparaissent aussi dans $\vec{u}$. Or, $\vec{w_1}$ est constitué uniquement de variables et de paramètres qui n'apparaissent pas dans $\vec{u}$. De plus, $D_2\vec{w_1}$ est la composante de $\vec{v}$ suivant le sous-espace engendré par $\vec{w_1}$ donc $D_2\vec{w_1}$ est un vecteur dont aucunes des variables ne sont dans $\vec{u}$. Donc $D_2\vec{w_1}$ n'est constitué que de constantes. Un raisonnement similaire montre que si R est une règle bijective alors $D_1\vec{w_2}$ est aussi constant.

■

Exemples : Considérons les deux règles suivantes issues du GTP de l'élimination de Gauss :

$$R_1 : T_1(k) \rightarrow T2(k,j) : A(i,k)|1 \leq k \leq n-1, k+1 \leq j \leq n+1, k+1 \leq i \leq n$$

$$R_2 : T_2(k,j) \rightarrow T_1(k+1) : A(i,k+1)|1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$$

La règle R_1 n'est pas une règle bijective. En effet, $\vec{u} = (k)$ et $\vec{v} = \begin{pmatrix} k \\ j \end{pmatrix}$. Le vecteur des indices de boucles et des paramètres est : $\vec{\imath} = \begin{pmatrix} n \\ k \\ l \\ j \\ i \end{pmatrix}$. Ainsi $D_1 = (0,1,0,0,0)$ et $D_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$. On trouve alors $E_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$ et $\vec{w_1} = \begin{pmatrix} n \\ 0 \\ l \\ j \\ i \end{pmatrix}$ tels que $\vec{\imath} = E_1\vec{u} + \vec{w_1}$. Ainsi $D_2\vec{w_1} = \begin{pmatrix} 0 \\ j \end{pmatrix}$ n'est pas constant.

En revanche, la règle R_2 est bijective. En effet, $\vec{u} = \begin{pmatrix} k \\ j \end{pmatrix}$ et $\vec{v} = (k+1)$. Comme $j = k+1$ on substitue toutes les occurences de j par $k+1$. On obtient : $\vec{u} = \begin{pmatrix} k \\ k+1 \end{pmatrix}$. De plus $D_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$ et $D_2 = (0,1,0,0,0)$. On trouve que : $\vec{w_1} = \vec{w_2} = \begin{pmatrix} n \\ 0 \\ l \\ j \\ i \end{pmatrix}$. Ainsi $D_2\vec{w_1}$ et $D_1\vec{w_2}$ sont tous deux constants.

4.2.3 Tri des règles bijectives

L'ordre dans lequel sont considérées les règles bijectives relève d'une grande importance. A priori les règles bijectives ne seront pas toutes mises à zéro. Dans l'algorithme de la figure 4.1 les premières règles considérées sont celles qui ont le plus de chance d'être mises à zéro. On effectue donc deux tris :

1. Les règles bijectives sont triées suivant la dimension des données qu'elles transmettent. On met en dernier les règles qui décrivent l'échange de scalaires, puis les règles qui décrivent l'échange d'une ligne ou d'une colonne de matrice ou d'un vecteur, puis celles qui décrivent l'envoi d'un bloc de matrice. La dimension des données transmises se détermine facilement en comptant le nombre de variables libres de la partie donnée de la règle. Par exemple la règle :

   $$R : T_2(k,j) \rightarrow T_1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$$

   transmet la colonne $k+1$ de A. La dimension de la donnée transmise est 1, car seule i est une variable libre pour la partie donnée de R.

2. La deuxième étape du tri consiste à mettre les règles transitives en fin de liste et ce quelle que soit la dimension des données. Une règle R est dite transitive si elle

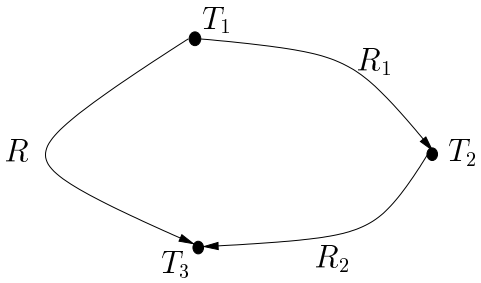


FIG. 4.2: *R une règle transitive*

correspond à un arc de transitivité dans le GTP. La figure 4.2 montre un exemple de règle transitive : R envoie des données de la tâche générique T_1 à la tâche générique T_3, or les règles R_1 et R_2 décrivent la transmission de données de T_1 à T_2 et de T_2 à T_3. Dans ce cas, il est important de considérer R après R_1 et R_2. En effet, comme nous le verrons dans la section suivante il se peut que R soit en conflit avec R_1 et R_2. Dans un tel cas, si R est mis à zéro en premier, alors ni R_1 ni R_2 ne pourront l'être. Cependant, R_1 et R_2 ne peuvent jamais être en conflit. Comme l'on veut mettre à zéro le plus grand nombre de règles possible, il est important de considérer R après avoir essayé de mettre à zéro R_1 et R_2.

4.2.4 Recherche des règles en conflit

Nous nous intéressons maintenant à la sélection de règles bijectives telles que l'ensemble sélectionné permette de construire un regroupement linéaire. Soit R_1 et R_2 deux règles bijectives :

$$R_1 : T_a(\vec{u}) \rightarrow T_b(\vec{v})|P_1$$

et

$$R_2 : T_c(\vec{w}) \rightarrow T_d(\vec{z})|P_2$$

Pour savoir si R_1 et R_2 sont en conflit il faut considérer deux cas :

- On peut avoir un conflit de type *fork*, si pour une instance valide de R_1 et de R_2, on a la même tâche d'envoi mais deux tâches de réception différentes. Formellement :
  1. $T_a = T_c$.
  2. Il existe une instance de $\vec{u}$ appelée $\vec{u_1}$, $\vec{u_1} \in P_1$ et une instance de $\vec{w}$ appelée $\vec{w_1}$, $\vec{w_1} \in P_2$ telles que $\vec{u_1} = \vec{w_1}$.
  3. Soient $\vec{v_1}$ l'image[1] de $\vec{u_1}$ suivant la règle R_1 et $\vec{z_1}$ l'image de $\vec{w_1}$ suivant la règle R_2 telles que $T_b(\vec{v_1}) \neq T_d(\vec{z_1})$.
- On peut avoir un conflit de type *join*, si pour une instance valide de R_1 et R_2 on a deux tâches d'envoi différentes et la même tâche de réception. Formellement :
  1. $T_b = T_d$

[1] Cette image existe et est unique car R_1 est une règle bijective

2. Il existe une instance de $\vec{v}$ appelée $\vec{v_1}$, $\vec{v_1} \in P_1$ et une instance de $\vec{z}$ appelée $\vec{z_1}$, $\vec{z_1} \in P_2$ telles que $\vec{v_1} = \vec{z_1}$.
3. Soient $\vec{u_1}$ la source de $\vec{v_1}$ suivant la règle R_1 et $\vec{w_1}$ la source de $\vec{z_1}$ suivant la règle R_2 telles que $T_a(\vec{u_1}) = T_c(\vec{w_1})$.

Pour déterminer automatiquement si deux règles sont en conflit de type *fork* (le cas du type *join* est symétrique), nous procédons comme suit. Concernant l'étape 2 nous calculons l'ensemble I qui est l'intersection de $P_1\backslash\vec{u}$ (P_1 restreint à $\vec{u}$) avec $P_2\backslash\vec{w}$. Si I est vide il ne peut y avoir de conflit de type *fork*. Sinon, si $T_b = T_d$ et que $R_1\backslash I$ (la restriction de R_1 à l'ensemble I) est différent de $R_2\backslash I$, alors on a un conflit. On peut calculer de manière symbolique I $R_1\backslash I$ et $R_2\backslash I$ grâce au Calculateur Omega comme montré section 2.5.4.

Exemples Considérons les deux règles suivantes :

$$R_1 : T_1(k, j) \rightarrow T_2(j) | k = 1, 1 \leq j \leq n$$

et

$$R_2 : T_1(k, j+1) \rightarrow T_2(j) | k = 1, n \leq j \leq 2n$$

R_1 et R_2 sont en conflit de type *join* (pour $j = n$). En revanche les règles :

$$R_1 : T_1(k, j) \rightarrow T_2(j) | k = 1, 1 \leq j \leq n$$

et

$$R_2 : T_1(k, j) \rightarrow T_2(j) | k = 1, n \leq j \leq 2n$$

ne sont pas en conflit, car pour $j = n$, l'arc décrit est le même (c'est un multi-arc). Si nous revenons à l'exemple de l'élimination de Gauss nous avons les deux règles bijectives suivantes :

$$R_1 : T_2(k, j) \rightarrow T_1(k+1) : A(i, k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$$

et

$$R_2 : T_2(k, j) \rightarrow T_2(k+1, j) : A(i, j) | 1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n$$

R_1 et R_2 peuvent être en conflit de type *fork*. Mais R_1 est valide pour $j = k+1$ alors que R_2 est valide pour $k+2 \leq j \leq n+1$. Ainsi aucune instance de $T_2(k, j)$ n'a deux fils (à savoir $T_2(k+1, j)$ et $T_1(k+1)$). R_1 et R_2 ne sont pas en conflit et peuvent toutes les deux être mises à zéro.

4.2.5 Preuve de l'algorithme de mise à zéro

Maintenant que nous avons vu les détails de l'algorithme de la figure 4.1. Nous pouvons énoncer le théorème suivant :

Théorème 4.2 *Pour toute instance des paramètres du GTP mettre à zéro toutes les règles de $\mathcal{Z}$ permet de construire un regroupement linéaire.*

Preuve du théorème 4.2 : Chaque règle $\mathcal{Z}$ est une règle bijective. Mettre à zéro une règle revient, dans le graphe de tâches, à regrouper les tâches deux par deux pour chaque instance de chaque règle. La grappe construite à partir de chaque règle est donc un chemin du graphe de tâches.

Mettre à zéro toutes les règles de $\mathcal{Z}$ conduit, dans le graphe de tâches, à fusionner certaines grappes décrites par les règles. Or, aucune des règles de $\mathcal{Z}$ n'est en conflit. Ainsi, il n'y aura pas de tâches indépendantes dans les nouvelles grappes issues de ces fusions. ∎

4.2.6 Identification des grappes

Nous montrons ici comment construire la fonction $\kappa(T_a, \vec{u})$ qui, pour toute tâche générique T_a et pour toute instance valide de $\vec{u}$, donne un numéro de grappe tel que toutes les tâches dans la même grappe reçoivent le même numéro.

Cas général

Considérons la règle suivante

$$R : T_a(\vec{u}) \rightarrow T_b(\vec{v})|P$$

Si cette règle est mise à zéro nous avons alors :

$$\kappa(T_a, \vec{u}) = \kappa(T_b, \vec{v}) \tag{4.1}$$

Le principe de notre méthode est le même que celui utilisé par Feautrier dans [34]. Comme Feautrier nous supposerons à priori que la fonction d'allocation a une forme affine. Ce choix est surtout justifié par le fait que, tout en étant assez générale, cette forme conduit à un système d'équations dont on sait toujours trouver la solution quand elle existe.

Nous posons que :

$$\kappa(T_a, \vec{u}) = \vec{\alpha_a}\vec{u} + \beta_a + \vec{\gamma_a}\vec{p}$$

où $\vec{p}$ est le vecteur des paramètres du programme, $\vec{\alpha_a}$ et $\vec{\gamma_a}$ sont des vecteurs d'inconnus à trouver et β_a une constante à déterminer.

Nous avons aussi $\kappa(T_b, \vec{v}) = \vec{\alpha_b}\vec{u} + \vec{\beta_b} + \vec{\gamma_b}\vec{p}$. Alors, résoudre l'équation 4.1 revient à résoudre

$$\vec{\alpha_a}\,\vec{u} + \vec{\beta_a} + \vec{\gamma_a}\vec{p} = \vec{\alpha_b}\vec{v} + \vec{\beta_b} + \vec{\gamma_b}\vec{p} \tag{4.2}$$

Si nous répétons ce processus pour toutes les règles de $\mathcal{Z}$ nous obtenons un système S de la forme

$$S : C\Omega = 0$$

Où C est obtenu par des manipulations algébriques des équations du type de 4.2 et Ω est le vecteur des $\vec{\alpha}$ des β et des $\vec{\gamma}$. Comme Feautrier, pour résoudre ce système nous appliquons l'algorithme de l'élimination de Gauss en nombres entiers sur C. Ensuite, nous affectons les variables non éliminées à 1 puis nous effectuons une résolution triangulaire du système. Si l'on trouve $\vec{\alpha} \neq \vec{0}$, alors une dernière phase consiste à simplifier la solution en retranchant à

tous les β ainsi qu'à tous les γ_i le plus petit d'entre eux (en valeur absolue). Par exemple, si $\beta_a = -1$ et $\beta_b = 1$ alors la solution simplifiée est $\beta_a = 0$ et $\beta_b = 2$. Si $\gamma_{a,1} = \gamma_{b,1} = 1$ alors la solution simplifiée est $\gamma_{a,1} = \gamma_{b,1} = 0$. Nous traiterons le cas où l'unique solution du système est $\vec{\alpha} = \vec{0}$ plus loin dans cette section.

Exemple

Retournons tout d'abord à l'exemple de l'élimination de Gauss La règle R_2 est :
$T_2(k,j) \rightarrow T_1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$
Nous avons donc :
$\kappa(T_2,(k,j)) = \alpha_{2,1}k + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n$ et $\kappa(T_1,(k+1)) = \alpha_{1,1}(k+1) + \beta_1 + \gamma_{1,1}n$
Ceci conduit à :

$$\begin{cases} \alpha_{2,1}k + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n = \alpha_{1,1}(k+1) + \beta_1 + \gamma_{1,1}n \\ j = k+1 \end{cases} \tag{4.3}$$

L'équation 4.3 est satisfaite pour : $\alpha_{2,1} + \alpha_{2,2} = \alpha_{1,1}$, $\gamma_{1,1} = \gamma_{2,1}$ et $\beta_2 + \alpha_{2,2} = \beta_1 + \alpha_{1,1}$

En ce qui concerne la règle R_3 ($T_2(k,j) \rightarrow T_2(k+1,j) : \{A(i,j) | 1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n\}$) nous avons :

$$\alpha_{2,1}k + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n = \alpha_{2,1}(k+1) + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n \tag{4.4}$$

l'équation 4.4 est satisfaite pour $\alpha_{2,1} = 0$.

Ceci nous conduit au système suivant :

$$\begin{cases} \alpha_{2,1} + \alpha_{2,2} - \alpha_{1,1} = 0 \\ \beta_2 + \alpha_{2,2} - \beta_1 - \alpha_{1,1} = 0 \\ \gamma_{1,1} = \gamma_{2,1} \\ \alpha_{2,1} = 0 \end{cases} \tag{4.5}$$

Le système 4.5 a une infinité de solution, en particulier celle-ci :

$$\begin{cases} \alpha_{2,2} = \alpha_{1,1} = 1 \\ \beta_1 = \beta_2 = \alpha_{2,1} = \gamma_{1,1} = \gamma_{2,1} = 0 \end{cases} \tag{4.6}$$

Ainsi, une fonction κ possible pour le GTP de l'élimination de Gauss est :

$$\kappa(T2,(k,j)) = j$$

et

$$\kappa(T1,(k)) = k$$

Cas où $\vec{\alpha} = \vec{0}$

Cependant ce n'est pas toujours aussi simple. En effet, deux règles peuvent conduire à des conditions contradictoires sur les α. Considérons par exemple les deux règles suivantes :

$$R_1 : T_1(k,j) \rightarrow T_1(k+1,j) | 1 \leq k \leq n, 1 \leq j \leq k$$

$$R_2 : T_1(k,j) \rightarrow T_1(k,j+1) | 1 \leq k \leq n, k+1 \leq j \leq n$$

La règle R_1 conduit à $\alpha_{1,1} = 0$ et la règle R_2 à $\alpha_{1,2} = 0$. Nous ne voulons pas la solution $\vec{\alpha} = \vec{0}$ car elle n'a pas de parallélisme. Cependant, nous savons que ces règles ne sont pas en conflit, les deux fonctions peuvent donc être appliquées à deux polytopes disjoints, levant ainsi l'incompatibilité :

$$\kappa(T1,(k,j)) = \begin{cases} j & \text{si } 1 \leq k \leq n, 1 \leq j \leq k \\ k & \text{sinon} \end{cases}$$

Dans le cas général nous procédons comme suit. Nous construisons le graphe G à partir du GTP issu de l'algorithme de sélection des règles. Chaque sommet est une tâche générique, chaque arc une règle de communication bijective qui n'entre en conflit avec aucune autre.

Lorsque tous les $\vec{\alpha}$ sont nuls (mais pas forcément les β ni les $\vec{\gamma}$) la solution obtenu a très peu ou pas du tout de parallélisme nous la rejetons. En effet, une tâche générique aura toujours le même numéro de grappe quelle que soit l'instance de son vecteur d'itération. Dans un tel cas, nous découpons G en sous-graphes G_1, $G_2, \ldots$ G_i, puis nous appliquons récursivement le processus de d'identification sur G_1 et G_2. La solution pour G n'est alors plus une fonction affine mais une *fonction affine par morceaux*. A l'exécution, pour déterminer quelle fonction d'identification appliquer pour une tâche $T_b(\vec{v})$ donnée nous remontons, grâce aux règles de communications, à la tâche générique T_a où le graphe a été découpé. Puis, nous déterminons l'instance $T_a(\vec{u})$ qui a conduit à $T_b(\vec{v})$ (si $T_a = T_b$ alors $\vec{u} = \vec{v}$). Enfin, en fonction de la valeur de $\vec{u}$ on détermine le sous graphe pour lequel il faut appliquer la fonction d'identification.

En pratique, la mise en œuvre d'un tel processus a un coût faible. Au pire cas, il est proportionnel au nombre de tâches génériques du graphe multiplié par la profondeur de la récursion (le nombre de fois qu'un graphe a été découpé avant d'obtenir une solution affine).

Lorsque nous devons découper un GTP nous supprimons d'abord les arcs multiples. La figure 4.3 montre comment le GTP G est découpé en deux sous graphe G_1 et G_2. S'il n'y a pas d'arcs multiples nous découpons alors le GTP à partir d'un nœud ayant un degré de sortie supérieur à 1. La figure 4.4 montre comment G_1 peut être à nouveau découpé en G_{1_1} et G_{1_2}.

Dans le pire des cas il peut arriver que l'on obtienne un sous graphe tel que la solution impose que tous les α soient nuls. Nous proposons trois solutions pour résoudre ce problème :

- Si G n'est composé que d'une seule règle de communication et d'une seule tâche générique, nous acceptons la solution. En effet cela signifie que G décrit un chemin dans le graphe. La seule manière de regrouper les tâches de ce chemin est de toutes les mettre sur le même processeur.

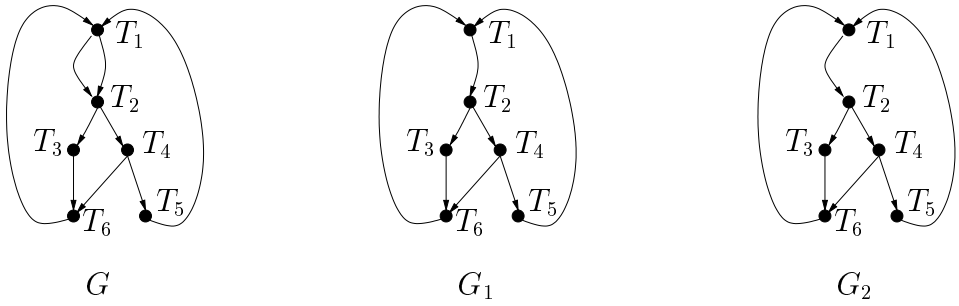


FIG. 4.3: *Le GTP G est découpé en deux sous-graphes G_1 et G_2 de manière à supprimer le multi-arc (T_1, T_2)*

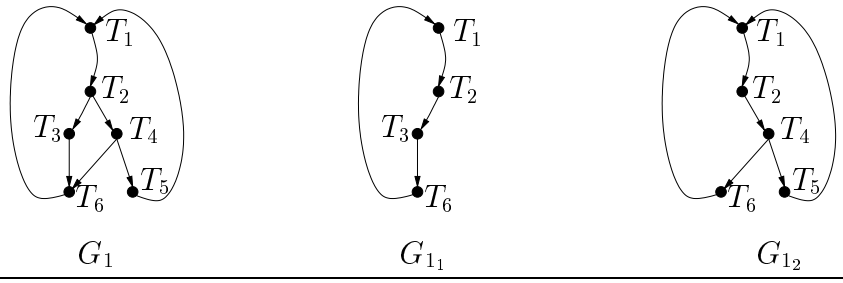


FIG. 4.4: *Le PTG G_1 est découpé en deux sous-graphes G_{1_1} et G_{1_2} au nœud T_2 (On peut encore découper G_{1_2} au nœud T_4)*

- Sinon, une solution en cours d'exécution consiste à affecter à la première tâche d'une grappe un numéro correspondant à la valeur du premier composant de son vecteur d'itération. Puis, lorsque l'on veut exécuter une tâche on calcule, grâce aux règles mises à zéro, l'instance de la première tâche de la grappe et on l'exécute sur le même processeur.
- Enfin, nous proposons une solution qui, «*à la compilation*» consiste à supprimer des règles de l'ensemble $\mathcal{Z}$ en partant de la fin pour recalculer l'allocation jusqu'à ce qu'une solution non triviale soit trouvée.

La deuxième solution permet d'exécuter le regroupement linéaire mais implique beaucoup de calcule lors de l'exécution. La troisième solution est plus simple mais l'allocation des tâches ainsi calculée implique plus de communications car d'avantage de règles ne sont pas mises à zéro.

4.2.7 Trouver les tâches de démarrage et de fin

Pour exécuter l'allocation trouvée par SLC il est nécessaire de connaître l'ensemble des tâches de démarrage (celles qui n'ont pas de père) et l'ensemble des tâches de fin (celles qui n'ont pas de fils). Si l'on exécute un GTP avec les tâches d'entrée et de sortie cela ne pose pas de problème.

En revanche, on peut vouloir se passer de ces deux tâches artificielles. Soit une tâche

générique T_a. Nous calculons le polyèdre $P_{\text{dem}}(T_a)$ qui décrit les valeurs du vecteur d'itération pour lesquelles T_a n'a pas de père et $P_{\text{fin}}(T_a)$ le polyèdre qui décrit les valeurs du vecteur d'itération pour lesquels T_a n'a pas de fils. Une fois ce processus accompli pour toutes les tâches génériques du GTP on obtient les ensembles désirés.

Pour calculer $P_{\text{dem}}(T_a)$ nous procédons comme suit. Soit $\text{emit}(T_a)$ l'ensemble des instances de T_a qui envoie des données. Il s'agit de l'union de toutes les instances valides du vecteur $\vec{u}$ pour toutes les règles d'émission du type : $T_a(\vec{u}) \rightarrow \ldots$ De même, notons $\text{recep}(T_a)$ l'ensemble des instances de T_a qui reçoivent des données. Dans ce cas, il s'agit de l'union de toutes les instances valides du vecteur $\vec{u}$ pour toutes les règles de réception du type $T_a(\vec{u}) \leftarrow \ldots$. L'ensemble de toutes les instances valides de T_a, noté $\text{valid}(T_a)$ est l'union de $\text{emit}(T_a)$ avec $\text{recep}(T_a)$. On a donc $P_{\text{dem}}(T_a) = \text{valid}(T_a) - \text{recep}(T_a)$ l'ensemble des instances valides de T_a qui n'envoient pas de données, donc qui n'ont pas de fils. On a aussi $P_{\text{fin}}(T_a) = \text{valid}(T_a) - \text{emit}(T_a)$ l'ensemble des instances valides de T_a qui ne reçoivent pas de données donc qui n'ont pas de pères.

Pour la tâche T_1 de l'élimination de Gauss on a :

$$\text{emit}(T_1) = \{k | 1 \leq k \leq n-1\}$$

et

$$\text{recep}(T_1) = \{k | 2 \leq k \leq n-1\}$$

ainsi

$$\text{valid}(T_1) = \{k | 1 \leq k \leq n-1\}$$

donc

$$P_{\text{dem}}(T_1) = \{k | 1 \leq k \leq n-1\} - \{k | 2 \leq k \leq n-1\} = \{1\}$$

et

$$P_{\text{fin}}(T_1) = \{k | 1 \leq k \leq n-1\} - \{k | 1 \leq k \leq n-1\} = \emptyset$$

Pour la tâche T_2 de l'élimination de Gauss on a :

$$\text{emit}(T_2) = \{(k,j) | 1 \leq k \leq n-2, k+1 \leq j \leq n+1\}$$

et

$$\text{recep}(T_2) = \{(k,j) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1\}$$

ainsi

$$\text{valid}(T_1) = \{(k,j) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1\}$$

donc

$$\begin{aligned} P_{\text{dem}}(T_2) &= \{(k,j) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1\} \\ &\quad -\{(k,j) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1\} \\ &= \emptyset \end{aligned}$$

et

$$\begin{aligned} P_{\text{fin}}(T_2) &= \{(k,j) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1\} \\ &\quad -\{(k,j) | 1 \leq k \leq n-2, k+1 \leq j \leq n+1\} \\ &= \{(k,j) | k = n-1, k+1 \leq j \leq n+1\} \end{aligned}$$

Ainsi l'ensemble des tâches de démarrage est : $\{T_1(1)\}$ et l'ensemble des tâches de fin est : $\{T_2(n-1,j)|n \leq j \leq n+1\}$.

4.2.8 Calculer le nombre de grappes

Il est utile de connaître le nombre de grappes en fonction de la valeur des paramètres. Pour chaque tâche $T(\vec{u})$ ayant une fonction de placement $\kappa(T,\vec{u}) = \vec{\alpha}\vec{u} + \beta + \vec{\gamma}\vec{p}$, on calcule l'ensemble $E_T = \{\kappa|\exists\vec{u} \in \text{valid}(T), \vec{\alpha}\vec{u}+\beta+\vec{\gamma}\vec{p} = \kappa\}$. E_T est l'ensemble des valeurs valides de la fonction d'allocation pour la tâche T. $E = \cup_T E_T$, l'union pour chaque tâche générique des ensembles de ce type, donne l'ensemble des valeurs valides de la fonction d'allocation pour toutes les tâches de l'application. Le nombre d'éléments de E donne le nombre de grappes différentes.

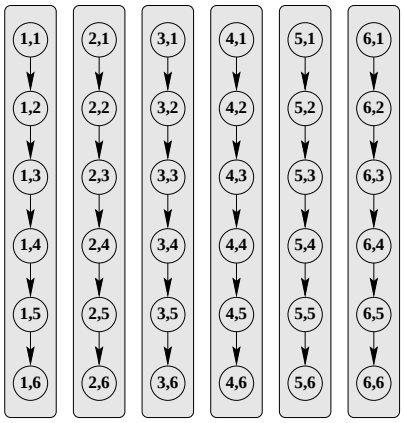
On peut calculer E symboliquement grâce au Calculateur Omega. Pour chaque partie de E qui est un polytope convexe on construit le polynôme de Ehrhart correspondant. Il se peut que, dans certaines parties de E, la condition sur κ ainsi calculé contienne un quantificateur existentiel. Dans ce cas on ne peut pas générer le polynôme de Ehrhart pour évaluer le nombre de points entiers de cette partie de E. On génère alors un nid de boucles qui parcourt E et on incrémente un compteur pour chaque itération valide.

4.3 Résultats

Nous avons testé SLC sur un certain nombre de noyaux de calculs. Les détails sur ces noyaux pourront être trouvés dans l'annexe 7. Tout d'abord, nous montrons les fonctions d'allocation des grappes trouvées par SLC pour ces différents noyaux. Puis, nous comparons SLC avec deux autres algorithmes d'ordonnancement statique DSC de Yang et Gerasoulis [76] et CASS-II de Liou et Palis [56]. Nous avons choisi de comparer SLC à des algorithmes conçus pour ordonnancer des graphes de tâches instanciés. L'intérêt de SLC réside dans le fait que, comme il est symbolique, il peut ordonnancer de très gros graphes. Cependant cette méthode perd une grande part de son attrait si les solutions trouvées s'avérent beaucoup moins bonnes que que celles fournies par des algorithmes d'ordonnancement statique non symbolique. En effet, une méthode alternative pour ordonnancer les très gros graphes consisterait alors de découper le graphe en tranches puis d'ordonnancer efficacement chacune des tranches.

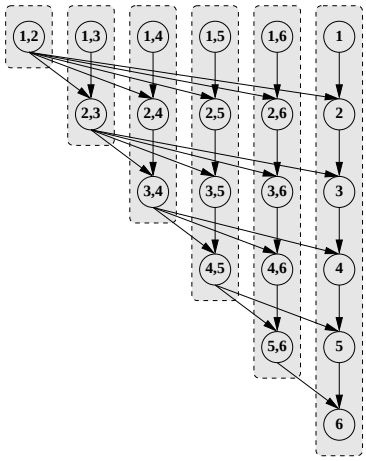
4.3.1 SLC sur des noyaux de calcul intensif

Les figures 4.5, 4.6, 4.7, 4.8 donnent la fonction d'allocation et sa correspondance dans une instance du graphe de tâches pour la multiplication de matrices, la diagonalisation de Jordan, l'algorithme de Givens et celui de Cholesky. Nous ne montrons pas l'élimination de Gauss car cela a été détaillé dans la section précédente. La figure 4.9 montre l'élimination de Gauss suivie d'une résolution triangulaire. Dans ce cas, la fonction d'allocation requiert des paramètres. Sinon, les règles qui décrivent les communications des tâches $T_1(i,j)$ aux tâches $T_1(i+1,j)$ et des tâches $T_1(j-1,j)$ aux tâches $T_2(n-j+1)$ ne peuvent pas être mises à zéro.



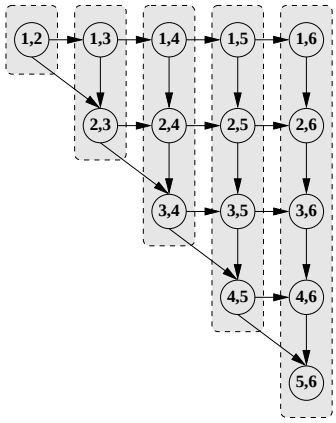
$$\kappa(T_1, (i,k)) = i$$

FIG. 4.5: *Résultat de SLC sur le programme de multiplication de matrices ($n = 6$)*



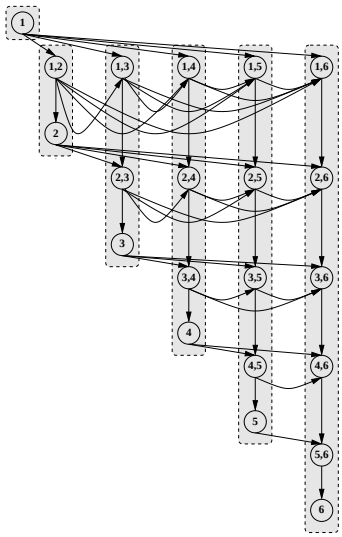
$$\kappa(T_1, (k,j)) = j$$
$$\kappa(T_2, (k)) = k$$

FIG. 4.6: *Résultat de SLC pour la diagonalisation de Jordan ($n = 6$)*



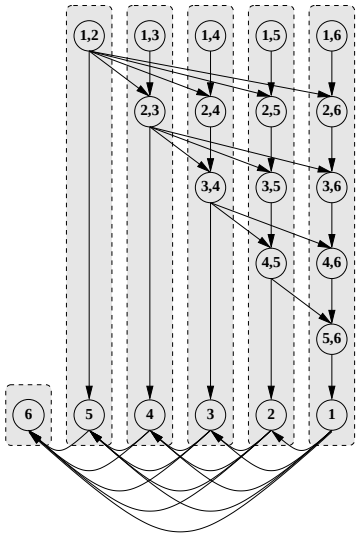
$$\kappa(T_1, (i,j)) = j$$

FIG. 4.7: *Résultat de SLC pour l'algorithme de Givens ($n = 6$)*



$\kappa(T_1,(k)) = k$
$\kappa(T_2,(k,i)) = i$

FIG. 4.8: *Résultat de SLC pour l'algorithme de Cholesky ($n = 6$)*



$\kappa(T_1,(i,j)) = n - i + 1$
$\kappa(T_2,(i)) = i$

FIG. 4.9: *Résultat de SLC pour l'élimination de Gauss suivie d'une substitution arrière ($n = 6$)*

4.3.2 Comparaison avec des algorithmes d'ordonnancement statiques

Nous comparons SLC avec deux algorithmes d'ordonnancement statiques pour un nombre non borné de processeurs. Les deux algorithmes en question sont DSC de Yang et Gerasoulis et CASS-II de Liou et Palis. Ces deux algorithmes nécessitent le graphe de tâches instancié comme entrée. Ainsi, ils ne peuvent pas traiter des problèmes de grande taille (pas plus de

400 en pratique). Nous utilisons les codes fournis par les auteurs respectifs des algorithmes. Ces codes permettent de simuler l'exécution de l'ordonnancement sur une machine parallèle à mémoire distribuée dans le modèle macro-data flow. Le programme Pyrros [74] qui exécute DSC offre en outre deux autres fonctionnalités :

- il permet de simuler l'exécution d'un graphe où les tâches ont déjà été allouées à des grappes. C'est ainsi que nous avons simulé l'ordonnancement de SLC,
- il permet de fusionner les grappes pour en obtenir un nombre inférieur ou égal à une certaine borne. Cela permet de construire un ordonnancement pour un nombre fixé de processeurs et de calculer sa durée. Plusieurs heuristiques de fusion des grappes sont disponibles. Le principe est le suivant : les grappes sont triées suivant leurs durées séquentielles. Elles sont ensuite réparties de manière à équilibrer la charge sur les processeurs. L'utilisateur a le choix entre un équilibrage cyclique (par défaut), par bloc, par repliage. Le lecteur se convaincra rapidement que les fonctions d'allocation des tâches pour les différents noyaux présentés ici sont toutes telles que l'ordre sur le numéro des grappes est le même (ou exactement l'inverse) que l'ordre sur la durée séquentielle (en effet, dans nos cas, plus il y a de tâches dans une grappe plus celle-ci dure longtemps). Ainsi, la fusion cyclique de DSC est la même que la fusion cyclique par numéro de grappe (si P est le nombre de processeurs disponibles et i le numéro de la grappe alors, après fusion la grappe se retrouve sur le processeur $i \bmod P$). Une fois les grappes fusionnées chaque tâche est ordonnancée de manière gloutonne sur le processeur à l'aide de l'heuristique de liste RCP* [75] (*Ready Critical Path*), une variante de CP (*Critical Path list scheduling*) de Adam, Chandy et Dickson [1]. Nous avons utilisé cette fonctionnalité pour comparer DSC avec SLC sur un nombre borné de processeurs.

Comparaison pour un nombre non borné de processeurs

Les tables 4.1 et 4.2 montrent la comparaison entre le regroupement trouvé par SLC avec celui trouvé par DSC et CASS-II pour une valeur du paramètre de 100, 200 et 400. Dans le cas de l'algorithme de Cholesky, il n'a pas été possible de réaliser de test pour des matrices de taille supérieur à 200 pour DSC. Il n'a été possible d'en faire que pour des petites matrices en ce qui concerne CASS-II. Cela vient du fait que le graphe de tâche de Cholesky est très gros (pour $n = 400$ il y a 80 200 nœuds et 10 746 600 arêtes, soit un fichier d'entrée de taille de 117 Mo). Les résultats pour la multiplication de matrices ne sont pas montrés car SLC trouve l'optimal (celui qui supprime toutes les communications).

La différence entre la table 4.1 et la table 4.2 est que le coût des arcs a été réduit de manière à simuler une machine rapide ce qui accroît la granularité. Pour la machine lente on a $\beta/\omega = 2384$ et $\alpha/\omega = 18$. Pour la machine rapide on a $\beta/\omega = 1$ et $\alpha/\omega = 2$.

La colonne *Tps séq* donne la somme de la durée des tâches du graphe. Les colonnes *Nb. Grappes* donnent le nombre de grappes qui a été trouvé par chaque algorithme. Les colonnes *Tps //* donnent la durée de l'ordonnancement trouvé par chaque algorithme. La colonne R_{DSC} donne le rapport entre la longueur de l'ordonnancement de DSC sur la longueur de l'ordonnancement de SLC. La colonne R_{CASSII} donne le rapport entre la longueur de l'ordonnancement de CASS-II sur la longueur de l'ordonnancement de SLC. Si R est plus

Graphe	n	Temps séq.	Nb. grappes DSC	Tps // DSC	Nb. grappes CASSII	Tps // CASSII	Nb. grappes SLC	Tps // SLC	R_{DSC}	R_{CASSII}
Gauss	100	1009800	92	364532	86	358041	102	402572	0.91	0.89
Gauss	200	8039600	196	994960	191	986910	202	1033735	0.96	0.95
Gauss	400	64159200	402	2979410	393	2949852	402	2990280	0.98	0.99
Jordan	100	1509950	92	495149	77	675231	101	779170	0.64	0.87
Jordan	200	12039900	199	1325682	186	1892735	201	1978317	0.67	0.96
Jordan	400	96159800	395	3895200	391	5556294	401	5645048	0.69	0.98
Givens	100	3720750	114	896350	77	725051	101	913707	0.98	0.79
Givens	200	39214956	371	2654398	269	2074518	201	2539794	1.05	0.82
Givens	400	257418656	1269	6964054	472	6286062	401	7845122	0.89	0.80
Gauss + Res. Tri	100	1024260	91	259714	56	272960	101	681721	0.38	0.4
Gauss + Res. Tri	200	8100500	183	913650	166	935657	201	1612021	0.57	0.58
Gauss + Res. Tri	400	64401002	386	2993491	379	3036111	401	4192621	0.71	0.72
Cholesky	100	510150	93	241918			101	312250	0.77	
Cholesky	200	4040300	212	521662			201	687350	0.76	

TAB. 4.1: *Comparaison de SLC avec DSC et CASS-II sur un nombre non borné de processeurs (machine lente/ faible granularité)*

Graphe	n	Temps séq.	Nb. grappes DSC	Tps // DSC	Nb. grappes CASSII	Tps // CASSII	Nb. grappes SLC	Tps // SLC	R_{DSC}	R_{CASSII}
Gauss	100	1009800	101	25093	101	29797	102	30001	0.84	0.99
Gauss	200	8039600	201	100193	201	119597	202	120001	0.83	1.00
Gauss	400	64159200	401	400393	401	479197	402	480001	0.83	1.00
Jordan	100	1509950	101	40002	101	40202	101	40202	1.00	1.00
Jordan	200	12039900	201	160002	201	160402	201	160402	1.00	1.00
Jordan	400	96159800	401	640002	401	640802	401	640802	1.00	1.00
Givens	100	3720750	98	166393	99	170841	101	175906	0.95	0.97
Givens	200	39214956	318	671163	297	688567	201	755580	0.89	0.91
Givens	400	257418656	496	2668550	497	2737167	401	2844880	0.94	0.96
Gauss + Res. Tri	100	1025250	99	36038	100	36038	101	36335	0.99	0.99
Gauss + Res. Tri	200	8100500	199	142088	200	142088	201	142685	1.00	1.00
Gauss + Res. Tri	400	64401000	399	564188	400	564188	401	565385	1.00	1.00
Cholesky	100	510150	100	32156			101	30100	1.07	
Cholesky	200	4040300	239	121759			201	120200	1.01	

TAB. 4.2: Comparaison de SLC avec DSC et CASS-II sur un nombre non borné de processeurs (machine rapide / granularité grossière)

grand que 1, cela signifie que SLC trouve un meilleur ordonnancement que l'ordonnanceur statique.

A la vue du graphe de Choleky, on pourrait penser qu'un ordonnancement horizontal ($\kappa(T_2,(k,j)) = k$) serait plus performant car réduirait d'avantage les communications. Les expériences prouvent qu'une telle allocation est en fait moins bonne. En effet, la longueur de l'ordonnancement pour un regroupement horizontal est de 399469 pour n=100 et 1041769 pour n=200 dans le cas d'une faible granularité et de 34951 pour n=100 et 139901 pour n=200 dans le cas d'une granularité plus grosse. Ces nombres sont tous supérieurs à ceux obtenus avec le regroupement de SLC. Ceci est dû au fait que les arêtes verticales correspondent à des communications plus importantes que les arêtes horizontales.

Les résultats montrent que R (à part pour l'élimination de Gauss avec résolution triangulaire) n'est jamais plus petit que 0.64. Pour une granularité grossière, R n'est jamais plus petit que 0.83. Dans certain cas SLC égale ou surpasse les deux autres algorithmes. En particulier pour la diagonalisation de Jordan, SLC trouve exactement le même regroupement que CASS-II. On voit que, en ce qui concerne l'élimination de Gauss avec résolution triangulaire la performance de SLC se rapproche très rapidement des performances de DSC et CASS-II quand a taille du problème augmente.

Comparaison pour un nombre borné de processeurs

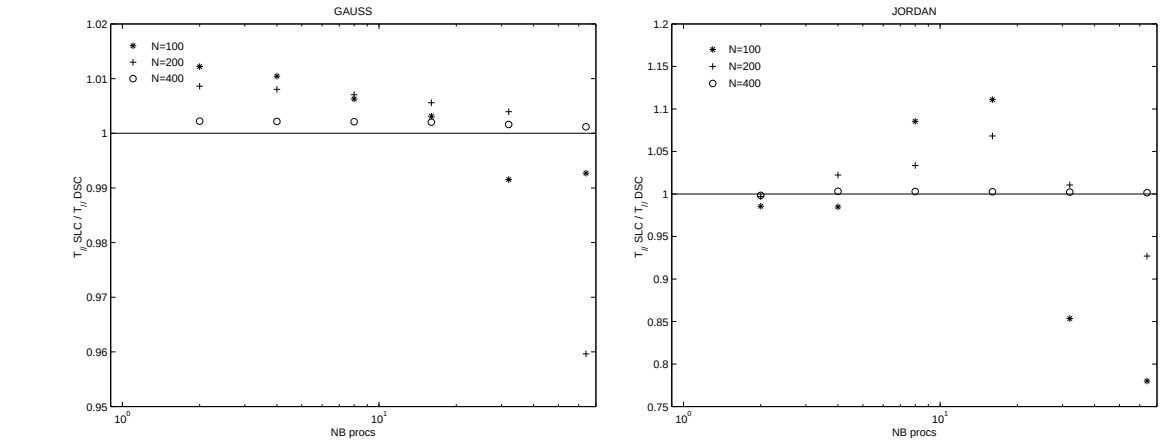


FIG. 4.10: *Rapport entre la durée de l'ordonnancement calculée par DSC et par SLC pour 2 à 64 processeurs pour l'ëlimination de Gauss et la diagonalisation de Jordan. La fusion des grappes est effectuée par RCP**

Nous avons utilisé le programme Pyrros pour simuler l'exécution de l'ordonnancement trouvé par DSC et par SLC sur un nombre borné de processeurs. Comme expliqué précédemment la fusion des grappes se fait à l'aide de l'algorithme RCP*. Les figures 4.10 4.11 4.12 montrent le rapport entre l'ordonnancement trouvé par DSC et celui trouvé par SLC dans le cas d'une machine rapide. Ce rapport est toujours compris entre 0.75 et 1.4.

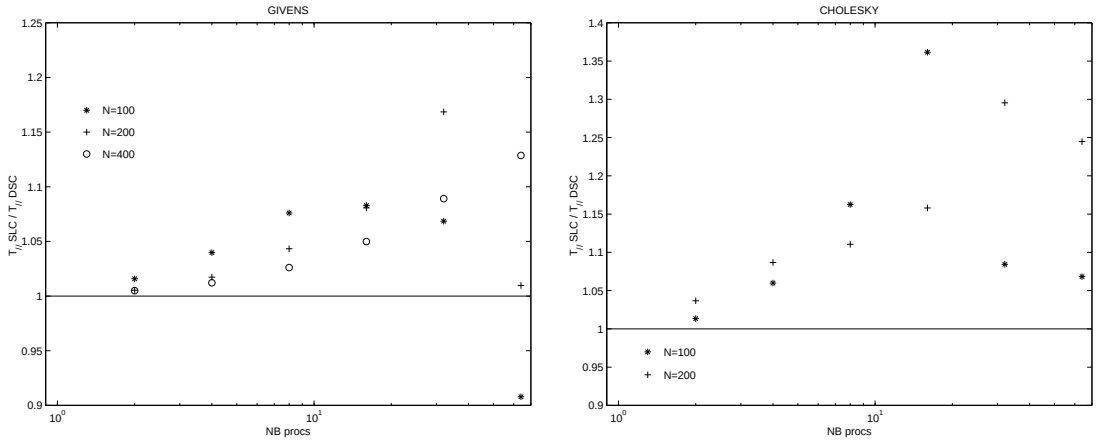


FIG. 4.11: *Rapport entre la durée de l'ordonnancement calculé par DSC et par SLC pour 2 à 64 processeurs pour l'algorithme de Givens et de Cholesky. La fusion des grappes est effectuée par RCP**

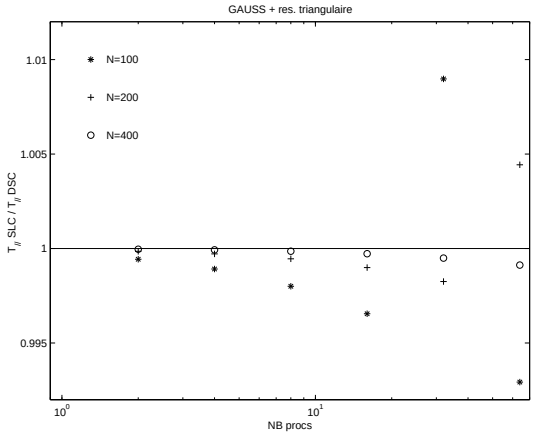


FIG. 4.12: *Rapport entre la durée de l'ordonnancement calculé par DSC et par SLC pour 2 à 64 processeurs pour l'élimination de Gauss avec résolution triangulaire. La fusion des grappes est effectuée par RCP**

4.4 Conclusion

Dans ce chapitre nous avons présenté SLC, une heuristique pour allouer symboliquement un GTP. Cette méthode permet d'obtenir un temps d'allocation et un coût mémoire indépendants de la taille du problème.

Notre contribution se résume en deux points :

1. Nous avons montré comment on peut extraire un regroupement linéaire à partir d'un GTP. Pour cela nous avons introduit la notion de règle bijective et de conflit.
2. En ce qui concerne l'identification des grappes, nous avons utilisé le principe d'une

fonction d'allocation affine. Nous l'avons transformée en fonction affine par morceaux afin qu'elle prenne en compte les conditions qui peuvent apparaitre dans le modèle.

Nous avons réalisé un programme qui implante SLC et nous l'avons testé sur un certain nombre de noyaux de calcul intensif. Nous avons étudié ces performances en le comparant à deux algorithmes d'ordonnancement destinés au même type d'applications (les noyaux de calculs intensifs) et réputés pour leurs efficacités aussi bien en terme de qualité de la solution que de rapidité d'exécution. Les résultats montrent que les ordonnancements trouvés par SLC sont très compétitifs par rapport à ceux trouvés par des ordonnanceurs statiques.

Aujourd'hui, ce prototype est intégré à l'environnement *pyrros+* [51] de Gerasoulis et al. à l'université de Rutgers, New-Jersey comme alternative à DSC.

Chapitre 5

Génération de code et optimisation

5.1 Introduction

Dans le chapitre précédent nous avons montré comment, à partir d'un GTP, on peut trouver une fonction d'allocation des tâches, de manière à obtenir un regroupement linéaire. Nous étudions ici le prototype de générateur de code que nous avons réalisé. Il permet de construire un programme qui exécute sur une machine parallèle à mémoire distribuée un programme source modélisé par un GTP. Il se conforme à l'allocation trouvée par SLC pour placer les instances des tâches. Il utilise les règles de communications pour générer les communications. Chaque tâche est exécutée de manière multithreadée dans l'environnement PM2 (voir section 2.5.2).

Dans tout ce chapitre, nous considérerons que le programme d'entrée respecte les contraines nécessaires pour être analysé par PlusPyr. Ainsi, on aura à notre disposition le GTP modélisant le programme, ainsi que le code.

Observations préliminaires

Nous avons choisi de réaliser ce prototype pour deux raisons :

- nous voulons montrer qu'il est possible de paralléliser de manière semi-automatique un programme en se basant sur le GTP et l'allocation trouvée par SLC,
- nous voulons évaluer l'algorithme SLC sur de véritables programmes et pas seulement à l'aide de simulations. Dans cette optique, nous pensons qu'exécuter des programmes générés automatiquement est plus significatif de l'intérêt de la méthode que si l'on avait écrit chaque programme «*à la main*».

Notre but n'était pas de réaliser un compilateur. Une des raisons est que la classe des programmes qui peuvent être modélisés par un GTP est trop restreinte pour qu'un compilateur ait un quelconque intérêt. De plus, le modèle que nous utilisons n'est pas adapté à l'implantation d'optimisations basées sur le texte du programme source (restructuration de code, etc ...), techniques qui ont fait leurs preuves dans les compilateurs.

Nous générons du code PM2. L'avantage est que ce code est portable. En effet, la couche de communication de PM2 fonctionne sous MPI pour un très vaste choix de machines parallèles. La contrepartie de la portabilité est qu'il n'est pas possible d'optimiser le code en

fonction de l'architecture de la machine cible. En particulier, nous considérerons toujours que le temps de transmission d'un message ne dépend que de sa taille et de la rapidité du réseau (il ne dépend pas de la distance inter-processeur, ni de la charge du réseau).

Organisation du chapitre

Ce chapitre est organisé comme suit. La section 5.2 décrit les environnements de génération de code en parallélisme de contrôle déjà existant. La section 5.3, justifie notre choix d'un environnement à base de processus légers et décrit de manière générale le fonctionnement du code généré. La section 5.4 présente un travail préliminaire qui a consisté à générer du code Athapascan 1. Le code parallèle et la génération automatique de celui-ci sont décrits section 5.5. Les optimisations sont présentées section 5.6. Les expériences que nous avons menées sur les programmes parallèles ainsi générés sont décrites section 5.7. Enfin, nous concluons section 5.8.

5.2 Génération de code en parallélisme de contrôle

Un certain nombre d'outils permettant de générer du code à partir d'un graphe de tâches ont été développés ces dernières années. C'est le cas de Pyrros de Yang et Gerasoulis [74] de Hypertool de Wu et Gajsky [72] qui a été étendu dans CASCH (*Computer Aided SCHeduling*) de Ahmad et al. [3].

Ces outils fonctionnent tous suivant le même principe :

- L'utilisateur décrit son algorithme sous forme d'appel à des routines à partir du programme principal. Dans l'en-tête de chaque routine, l'utilisateur décrit les variables lues et les variables écrites. Il est alors très facile pour un pré-processeur de déterminer les dépendances entre routines.
- On instancie la valeur des paramètres et on construit le graphe de tâches.
- Ce dernier est ordonnancé. Ici les outils divergent. Pyrros utilise uniquement DSC puis fusionne les grappes pour en obtenir un nombre égal au nombre de processeurs, en équilibrant la charge.

  Hypertool utilise MCP (*Modified Critical Path*) ou MD (*Mobility Directed scheduling*) qui tous deux ordonnancent le graphe sur un nombre non borné de processeurs virtuels. Le placement de ces processeurs virtuels se fait de manière à essayer de minimiser les communications résiduelles.

  CASCH propose un grand nombre d'heuristiques d'ordonnancement. Celles-ci sont aussi bien des heuristiques pour un nombre borné de processeurs, pour un nombre non borné de processeurs, ou encore des heuristiques qui tiennent compte de la topologie et de la contention du réseau.
- Une fois le graphe ordonnancé, les outils génèrent une table qui décrit l'ordonnancement.
- Le code parallèle est ensuite généré. Dans Pyrros il s'agit d'un code qui lit la table d'ordonnancement en cours d'exécution pour déterminer l'ordre d'exécution des tâches

et les envois de messages. Dans Hypertool et CASCH, le code est généré à partir de la table d'ordonnancement. Cette dernière méthode a l'avantage de construire un code très rapide. En revanche la taille du code généré est proportionnelle à la taille du graphe de tâches ce qui est complètement irréaliste pour des applications sérieuses.

Dans tous les cas, les codes ainsi générés obéissent au modèle macro-dataflow : chaque tâche reçoit les données de ses pères, s'exécute, puis envoie les données à ses fils.

5.3 Exécution multithreadée des grappes

5.3.1 Les raisons du choix d'un environnement multithreadé

La caractéristique principale d'un environnement à base de processus légers est que plusieurs procédures d'un même programme peuvent, s'exécuter concurremment, de manière transparente. Nous étudions ici l'intérêt d'une telle spécificité et justifions ainsi l'utilisation de PM2.

Le programme généré va essentiellement exécuter des tâches. Une fois qu'elle possède ses données, une tâche est complètement indépendante. Elle ne peut plus être perturbée par l'environnement et elle ne peut plus modifier l'environnement. Il est donc possible d'exécuter plusieurs tâches en même temps. En pratique, on est limité par les ressources du processeur et par le nombre de tâches exécutables à l'instant «*t*» (celles qui n'attendent plus de données). Dans la pratique, utiliser plusieurs files d'exécution présente un double intérêt :

1. il est possible de faire du recouvrement calcul/communication. En effet, lorsqu'une tâche est terminée, elle peut envoyer de manière autonome les données à ses fils pendant que d'autres tâches continuent à s'exécuter. Si dans la machine parallèle, l'interface réseau est découplée du processeur de calcul alors envoyer un message coûte peu de ressource CPU.

2. l'exécution se fait en temps partagé. Il n'est plus nécessaire de choisir une tâche à exécuter parmi celles qui sont prêtes. L'ordonnancement entre les tâches est laissé au système. Si, en plus, on possède une gestion des priorités, il est possible de favoriser des tâches «*importantes*» pour qu'elles terminent plus tôt.

Il existe différentes manières de mettre en œuvre une exécution parallèle des tâches sur un même processeur.

Une première idée est de créer un processus par tâche et de faire communiquer chaque processus à l'aide de «*socket*». Ceci peut facilement être mis en œuvre à l'aide de PVM [43]. Cette méthode a cependant deux désavantages :

– le coût de création d'un processus est très long (sous Solaris 2.7 la création et la destruction d'un processus prend environs 8,5 millisecondes sur un Pentium II à 350 MHz (source J.F. Mehaut)). De plus, le processus est créé avec de nombreux attributs inutiles en calcul parallèle (table des signaux, table des fichiers, etc ...). Or, dans nos applications nous allons exécuter des millions de tâches, d'une durée bien plus courte que le temps de création d'un processus.

– les communications sous PVM sont en général très lentes. Énormément de recopies du tampon de communication ont lieu entre l'envoi et la réception d'un message, même si celui-ci à lieu sur le même processeur.

Ainsi, compte tenu de la quantité de messages à envoyer et du nombre de tâches à exécuter, une solution basée sur PVM n'est pas envisageable si l'on souhaite obtenir des performances.

Dans l'optique de permettre l'envoi et la réception de messages en même temps que l'exécution des tâches, une approche avec MPI [49] n'est pas non plus satisfaisante. En effet, chaque processus MPI est intrinsèquement séquentiel. Il n'est pas possible d'envoyer ou de recevoir des messages et en même temps d'exécuter des tâches. Plus précisément, le programmeur doit définir explicitement les instants où les messages seront pris en compte (en envoi ou en réception) par le programme.

Nous avons choisi d'utiliser un environnement à base de processus légers. En effet, le coût de gestion d'un processus léger est très faible et il est possible d'en exécuter un très grand nombre sur un processeur. Nous avons utilisé l'environnement de processus légers PM2 [63] décrit dans le premier chapitre de cette thèse. Comme PM2 fonctionne sur un très grand nombre de plateformes, le code obtenu est très portable. De plus, les communications se faisant à l'aide d'appel de procédures à distance, l'envoi et la réception de messages sont réalisés de manière complètement asynchrone, ce qui rend beaucoup plus simple l'écriture du code.

5.3.2 Présentation générale

En ce qui concerne l'exécution, nous distinguons trois parties principales : la réception des messages, l'exécution des tâches et l'envoi des messages.

Le grain des applications que nous allons traiter étant relativement élevé, chaque tâche va être exécutée par un processus léger. Dans PM2 les communications étant elles aussi effectuées par des processus légers particuliers il sera possible de réaliser en temps partagé les calculs et les communications, la réception des messages étant totalement asynchrone. Comme PM2 supporte beaucoup de couche de communications (MPI, BIP, TCP, PVM, ...) et qu'il s'agit d'une sur-couche de C, le code généré sera très portable.

Réception des messages

L'algorithme de réception des messages est donné figure 5.1 Cet algorithme gère deux ensembles :

– l'ensemble des *tâches en attente*. Il s'agit de tâches qui ont déjà reçu des messages, mais qui en attendent d'autres.

– l'ensemble des *tâches prêtes*. Il s'agit de tâches qui ont reçu tous leurs messages et qui peuvent être exécutées.

Nous avons aussi implanté de manière assez stricte le modèle macro-data flow. A savoir que, dans le principe, les données sont privées aux tâches. Aussi bien celles qui sont nécessaires aux calculs que celles qui sont calculées. On appellera ces données, les données *rattachées* à la tâche. Cela permet d'éviter de gérer de manière globale les données et d'avoir à les rechercher en mémoire lors de l'exécution d'une tâche. En revanche, si une même donnée est lue par

pour chaque nouveau message arrivant sur le nœud **faire**
  si le message est destiné à une tâche T_{recp} qui est dans l'ensemble des *tâches en attente* **alors**
    Ajouter les données du message à T_{recp} ;
  sinon
    Créer la tâche T_{recp} ;
    Ajouter les données du message à T_{recp} ;
  si ce message est le dernier que doit recevoir T_{recp} **alors**
    Mettre T_{recp} dans l'ensemble des *tâches à exécuter*;
  sinon
    Mettre T_{recp} dans l'ensemble des *tâches en attente*;

FIG. 5.1: *Réception des messages*

plusieurs tâches cela peut impliquer des recopies. Nous verrons dans la section consacrée aux optimisations ce que nous avons mis en œuvre pour surmonter cet inconvénient.

Exécution des tâches

Le nombre des processus légers chargés d'exécuter les tâches est fixé à l'exécution. L'algorithme qu'ils exécutent est décrit figure 5.2. Quand un processus léger a fini d'exécuter une

tant que toutes les tâches de fin n'ont pas été exécutées **faire**
  Prendre T une tâche dans l'ensemble des *tâches prêtes* ;
  Exécuter T ;
  si T est une tâche de fin **alors**
    Décrémenter le nombre de tâches de fin à exécuter ;
  sinon
    Utiliser les règles de communications pour envoyer les messages aux fils de T ;
  Détruire la tâche et les données qui lui sont rattachées.

FIG. 5.2: *Exécution d'un processus léger*

tâche, il en choisit une autre dans l'ensemble des tâches prêtes. Si cet ensemble est vide, il se bloque jusqu'à ce qu'une tâche devienne prête. Pour exécuter la tâche, il appelle la procédure qui correspond au code de celle-ci et passe en paramètre les données reçues précédemment et qui lui sont rattachées.

Envoi des messages

Le modèle du graphe de tâches est tel que seules les données calculées ou modifiées par une tâche sont transmises à ses fils. Pour envoyer les messages, le processus léger parcourt l'ensemble des règles d'émission, détermine celles qui doivent être appliquées et envoie les

messages nécessaires. En principe, la transmission des données se fait par recopie. Ainsi, une fois que les messages sont envoyés on peut détruire les données rattachées à la tâche. Nous verrons dans la partie consacrée aux optimisations comment dans certains cas on se passe d'une telle recopie et donc de la désallocation de données.

L'envoi d'un message est synthétisé par la figure 5.3 :

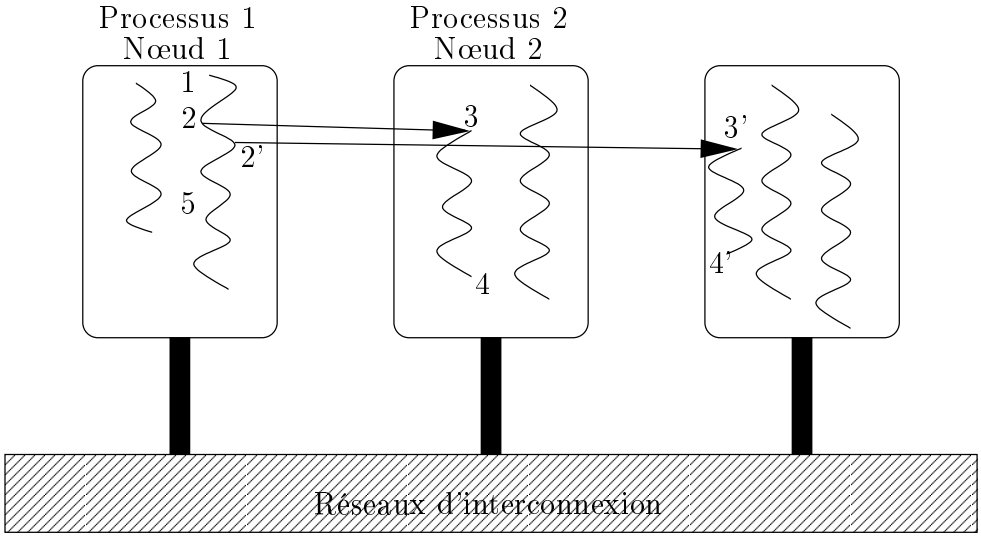


FIG. 5.3: *Les différentes étapes asynchrones d'un transfert de message*

1 Un processus léger commence à exécuter une tâche T.

2 et 2' L'exécution de la tâche étant terminée, le processus léger empaquette et envoie les données aux fils de T.

3 et 3' Le processeur qui va exécuter un fils de la tâche T reçoit la donnée. Il crée un processus léger pour la stocker.

4 et 4' Une fois la donnée stockée dans la tâche de réception, le processus léger se détruit.

5 Le processus léger d'exécution choisit une nouvelle tâche prête et l'exécute.

5.4 Approche préliminaire : Génération de code Athapascan 1

Pour générer du code PM2, plusieurs étapes sont nécessaires. Les premières concernent le typage des messages, la fonctionnalisation du programme, la détection des données propres à chaque tâche ainsi que leurs modes d'accès. Une fois que l'on a réalisé ces étapes, il est possible de générer le code d'entrée des outils d'ordonnancement décrits plus haut (Hypertool, Pyrros, CASCH). Nous pouvons aussi générer du code Athapascan-1. L'avantage d'un programme écrit en Athapascan-1 est qu'il est générique : une fois compilé, il fonctionne pour n'importe quelle valeur des paramètres.

Dans le chapitre 2 nous avons décrit l'interface de programmation Athapascan-1. Nous décrivons ici comment générer du code à partir d'un programme à contrôle statique et de son GTP. Nous montrons ici :

- comment analyser les règles de manière à en tirer des informations cohérentes,
- comment fonctionnaliser un programme découpé en tâches,
- comment typer les données transmises,
- comment calculer les modes d'accès aux variables.

5.4.1 Analyse des règles

Lorsqu'il analyse le code d'un programme à contrôle statique, un expert peut déterminer quelles données sont lues ou écrites par une tâche. De plus, il est capable de préciser ces informations en décrivant de manière symbolique les accès aux variables à l'aide du vecteur d'itération de la tâche et des paramètres.

Par exemple, dans l'élimination de Gauss, la tâche $T_1(k)$ modifie la colonne k de la matrice A, la tâche $T_2(k, j)$ modifie la colonne j et lit la colonne k de la matrice A.

Nous prétendons que le modèle de programmation que nous avons choisi, nous permet de déterminer automatiquement ces informations et cela, grâce aux règles de communications.

Hypothèses

Notre méthode est basée sur la syntaxe des expressions. Ainsi, outre le fait que le programme doit être analysable par PlusPyr, nous supposerons :

- qu'aucune variable qui est affectée à une expression constituée d'une fonction des paramètres et du vecteur d'itération de la tâche n'est utilisée dans d'autres expressions de la tâche. Ainsi, si dans un programme de paramètre n, on trouve dans une tâche de vecteur d'itération (k, j) l'affectation : `x=3*k+n-j` alors, les occurrences de `x` seront remplacées, dans la tâche, par l'expression en question.
- qu'aucune tâche n'accède à une même case de tableau par l'intermédiaire de deux fonctions d'accès différentes. Ainsi, si une tâche accède au tableau `a(i,k)` puis au tableau `a(i,n-j)` et qu'il est toujours vrai que $k + j = n$, alors on substituera l'un des accès par l'autre.

Ces hypothèses sont une restriction sévère du style de programmation, mais elles n'entachent en rien l'expressivité du langage.

Ces informations nous permettrons de nommer sans équivoque les données qui seront rattachées aux tâches.

De plus, nous nous limitons aux tableaux de dimensions deux. Ceci est est dû au fait que nous sommes amenés à comparer les dimensions des données transmises entre les tâches. Suivant les cas, des fusions de messages ou la création de nouveaux tableaux doivent être effectués. Tous les tests pour les dimensions inférieures à deux ont été écrits dans ce prototype, mais pas les autres.

Identification des données accédées

Pour générer le programme parallèle, il est nécessaire de déterminer, pour chaque tâche, les données accédées (en lecture ou en écriture). L'analyse des règles et du code d'une tâche nous permet de déterminer ces données. Cette analyse permet de préciser, lorsqu'il s'agit d'un tableau, si c'est seulement une ligne ou une colonne qui est accédée et de laquelle il s'agit.

Notons tout d'abord, que les accès aux tableaux sont des fonctions affines des indices de boucles englobantes et des paramètres du programme. Ainsi, si la fonction d'accès d'une des dimensions du tableau ne dépend que des variables du vecteur d'itération de la tâche et des paramètres alors cette fonction est constante pendant toute l'exécution de la tâche. Suivant la dimension du tableau et celles qui sont constantes, on peut déterminer à qui la tâche accède. Par exemple, dans un tableau de dimension 2, si la première dimension est constante cela signifie que la tâche accède, au plus, à une ligne. Si les deux dimensions sont constantes, la tâche accède à un élément de matrice. Si aucune des dimensions ne sont constantes la tâche accède, au plus, à la matrice entière.

On réécrit les expressions du programme et des règles de manière à ce que chacune des variables soit écrite dans l'ordre alphabétique suivi de la constante. On obtient alors une expression normalisée notée $\bar{e}$. Par exemple, l'expression normalisée de $1 + n + 3k - j - 2n$ est $\bar{e} = -j + 3k - n + 1$. Il est facile de transformer chaque expression normalisée en une chaîne de caractères, notée $\tilde{e}$, de telle sorte que deux expressions différentes soient transformées en deux chaînes de caractères différentes. Pour l'expression $\bar{e}$ on obtient $\tilde{e}$ = "`moins_j_plus_3_k_moins_n_plus_1`".

On identifiera donc les éléments accédés par une tâche en construisant une variable dont le nom décrit exactement l'élément accédé. Ce nom est constitué de trois parties :

1. le type de l'élément : `c` pour une colonne, `r` pour une ligne, `v` pour un vecteur, `m` pour un élément de matrice, etc ...
2. la chaîne $\tilde{e}$ qui correspond à l'expression. Dans le cas d'un élément de matrice, la chaîne $\tilde{e_1}$ qui correspond à la ligne et la chaîne $\tilde{e_2}$ qui correspond à la colonne sont séparées par un double soulignage : `__`.
3. le nom de la variable tel qu'il est déclaré dans le programme source.

Par exemple, si c'est la colonne $1+n+3k-j-2n$ de la matrice A qui est accédée le nom de la variable qui la représentera sera : `c_moins_j_plus_3_k_moins_n_plus_1_a`. Le rôle de cette variable est de pointer sur les éléments d de la colonne dans le programme généré.

Déterminer les données lues et les données écrites

Une fois déterminé l'ensemble des données accédées par une tâche, il faut préciser si ces données sont lues ou écrites.

Dans notre modèle, une donnée est reçue par une tâche si et seulement si elle est lue et une donnée est envoyée par une tâche si et seulement si elle est écrite. On va donc pouvoir se servir des règles d'émission pour déterminer l'ensemble des données écrites et des règles de réception pour déterminer l'ensemble des données lues par une tâche.

Notons aussi, que PlusPyr génère automatiquement des tâches d'entrée et de sortie. La première écrit toutes les variables, et la dernière les lit toutes. Il génère le GTP modélisant le programme source auquel on a ajouté ces tâches. Ainsi, si une variable est lue par une tâche, une règle décrira sa transmission à partir de la dernière tâche qui l'a écrite (la tâche d'entrée ou une autre). De même, si une tâche écrit une variable, une règle décrira sa transmission vers la prochaine tâche qui la lira (la tâche de sortie ou une autre).

Déterminer l'ensemble des données lues par une tâche ne pose pas de véritable problème. En effet, les règles de réception sont construites à partir des dépendances à grain fin du programme. Ainsi, chaque règle de réception vérifie les propriétés suivantes :

- la tâche gauche de la règle (la tâche de réception), a comme vecteur d'itération le vecteur d'itération de la tâche générique,
- le vecteur d'accès aux données est, pour chaque dimension, syntaxiquement le même que celui du programme.

Par exemple, si l'on a une règle de réception suivante :

$$T_1(k,j) \leftarrow T_2(k+2) : A(k+2n-j,i)| \ldots$$

cela signifie que (k,j) est, dans le code source, le vecteur d'itération de la tâche T_1 et que son code est constitué d'une lecture de la variable $\texttt{a}(\texttt{k} + \texttt{2n} - \texttt{j}, \texttt{i})$. On associera donc à la tâche T_1 une variable dont le nom est `r_moins_j_plus_k_plus_2_n_a` qui est un pointeur sur la ligne de `a`.

Il arrive qu'une règle décrive la réception d'une partie d'une ligne (colonne, matrice, vecteur, ...), et qu'une autre règle décrive la réception d'une autre partie. Il faut veiller à ne pas rattacher à chaque tâche des variables redondantes. Par exemple, si une règle décrit la réception de la colonne `k` et qu'une autre règle décrit la réception de l'élément `(k,k)`, alors il ne faut pas rattacher à la tâche ce dernier élément car il est déjà décrit par la colonne. De même, si une tâche reçoit à la fois une ligne et une colonne de la même matrice, l'élément d'intersection, s'il est modifié, pose des problèmes. Dans ce cas, notre attitude est conservative : on associe la matrice entière à la tâche. D'une manière générale, on regarde pour chaque élément décrit par une règle de communication s'il est décrit par un élément plus général dans le cadre d'une autre règle. Seul l'élément le plus général sera rattaché à la tâche.

Déterminer l'ensemble des données écrites par une tâche est plus délicat. En effet, chaque règle d'émission est construite en inversant une règle de réception. Ainsi, il est possible que le vecteur d'itération de la tâche gauche (la tâche d'émission) soit constitué d'expressions où les variables ne sont pas celles du vecteur d'itération. De même, rien ne garantit que le vecteur d'accès aux données est syntaxiquement le même que celui du programme.

Par exemple, dans l'élimination de Gauss la règle d'émission suivante

$$T_1(k) \leftarrow T_2(k-1,k) : A(l,k)|2 \leq k \leq n-1, k+1 \leq l \leq n$$

une fois inversée donne

$$T_2(k-1,k) \rightarrow T_1(k) : A(l,k)|2 \leq k \leq n-1, k+1 \leq l \leq n$$

La première étape consiste donc à renommer les variables du vecteur d'itération de la tâche gauche en fonction du vecteur d'itération de la tâche générique. Après un tel changement de variables,on obtient :

$$T_2(k,j) \rightarrow T_1(k+1) : A(l,k+1) | 2 \leq k \leq n-1, k+1 \leq l \leq n, j = k+1$$

Une fois le changement de variables effectué, il se peut que des égalités subsistent entre les variables du vecteur d'accès aux données. On a donc le choix entre plusieurs expressions qui sont sémantiquement identiques mais syntaxiquement différentes. Pour trouver celle qu'il faut utiliser, il est nécessaire, dans le cas général, d'analyser le code source. Typiquement, dans notre exemple, comme $j = k + 1$ il est impossible de savoir (sans le code source) si la variable qui décrit la colonne doit être `c_k_plus_1_a` ou `c_j_a`. Une telle analyse du code source bien qu'indispensable pour traiter le cas général n'a pas été implantée dans notre générateur de code. Dans tous les cas que nous avons rencontrés nous trouvons la bonne solution en choisissant toujours la plus simple (ici `c_j_a`).

5.4.2 Fonctionnaliser

Dans ce qui suit, nous décrivons comment transformer un programme découpé en tâches en un programme où chaque tâche est devenue une fonction appelée par le programme. Si le programme est à contrôle statique, la fonctionnalisation ne pose pas de problème théorique. En effet, dans ce cas :

- il n'y a pas d'alias dans les accès aux tableaux. Ainsi, lorsque l'on écrit dans un tableau seul celui-ci est modifié,
- la portée des variables est connue. On peut déterminer si une variable est seulement lue ou écrite dans le corps d'une tâche,
- il n'y a pas de branchement et chaque instruction de contrôle est contenue dans le corps d'une tâche. Ainsi chaque tâche a un point d'entrée et un point de sortie unique.

La fonctionnalisation est le contraire de *l'inlining*, très utilisé en compilation. Nous procédons comme suit :

- on transforme le programme écrit en *Tiny*, le langage d'entrée de PlusPyr, en programme écrit en *C*. Pour cela, nous avons réalisé un traducteur à l'aide du parser de PlusPyr. La seule difficulté dans la réalisation d'un tel traducteur est que les tableaux commencent à l'indice 1 en Tiny et à l'indice 0 en C (pour résoudre ce problème nous avons décalé tous les accès aux tableaux de 1),
- pour chaque tâche, on crée une fonction où le code de celle-ci est une copie syntaxique du corps de la tâche,
- en se servant du GTP, on détermine les variables qui sont lues ou écrites par la tâche. Ces variables (il ne s'agit que de tableaux ou de scalaires) ainsi que celles du vecteur d'itération sont alors déclarées comme paramètres de la fonction,
- dans le corps de la fonction, on déclare les variables utilisées uniquement par la tâche. Les autres variables restent globales,

```
void T1(int k, float **a){
  int l ;
  float s ;
  s=1/a[k-1][k-1] ;
  for(l=k+1 ;l<=n ;l++){
    a[l-1][k-1]=a[l-1][k-1]*s ;
  }
}

void T2(int k, int j, float **a){
  int i ;
  for(i=k+1 ;i<=n ;i++){
    a[i-1][j-1]=a[i-1][j-1]-a[k-1][j-1]*a[i-1][k-1] ;
  }
}

main(){
  float a[n][n+1] ;
  int k,j ;
  for(k=1 ;k<=n-1 ;k++){
    T1(k,a) ;
    for(j=k+1 ;j,+n+1 ;j++){
      T2(k,j,a) ;
    }
  }
}
```

FIG. 5.4: *Fonctionnalisation de l'élimination de Gauss*

– dans le programme principal, chaque tâche est remplacée par un appel à la fonction correspondante.

La figure 5.4 donne l'exemple de la fonctionnalisation de l'élimination de Gauss.

5.4.3 Typage des données transmises

Nous venons de montrer comment transformer un programme découpé en tâches en un programme où chaque tâche est remplacée par une fonction. Dans l'optique où ces fonctions vont être exécutées en parallèle, les paramètres de celles-ci seront transmis à l'aide de messages. Nous décrivons ici comment affiner ces paramètres pour extraire du parallélisme de l'application.

Il est clair que transmettre une simple colonne de matrice prend moins de temps que de transmettre toute la matrice. De plus, en *C*, les accès au tableaux se font ligne par ligne (lire

séquentiellement une colonne prend plus de temps que lire séquentiellement une ligne).

Comme montré section 5.4.1 sur l'analyse des règles, le GTP nous donne de précieuses indications au niveau du type de données qui sont transmises. L'analyse des règles décrites plus haut nous permet en effet de savoir si c'est un scalaire, un vecteur, une ligne ou une colonne de matrice ou une matrice toute entière qui doit être transmis.

Dans certains cas, une telle analyse nous permet d'améliorer le code :

- si le programme accède une matrice uniquement par colonne, on inverse alors les dimensions de cette matrice ainsi que les accès pour qu'il n'y ait alors plus que des accès par ligne,
- si chaque instance d'une tâche donnée accède uniquement une ligne ou une colonne de matrice, alors seulement celle-ci est passée en paramètre de la fonction.

```
void T1(int k, float *c_k_a){
  int l ;
  float s ;
  s=1/c_k_a[k-1] ;
  for(l=k+1 ;l<=n ;l++){
    c_k_a[l-1]=c_k_a[l-1]*s ;
  }
}

void T2(int k, int j, float *c_j_a,float *c_k_a){
  int i ;
  for(i=k+1 ;i<=n ;i++){
    c_j_a[i-1]=c_j_a[i-1]-c_j_a[k-1]*c_k_a[i-1] ;
  }
}

main(){
  float a[n+1][n] ;
  int k,j ;
  for(k=1 ;k<=n-1 ;k++){
    T1(k,ligne(k-1,a)) ;
    for(j=k+1 ;j,+n+1 ;j++){
      T2(k,j,ligne(j-1,a),ligne(k-1,a)) ;
    }
  }
}
```

FIG. 5.5: *Typage des paramètres de l'élimination de Gauss (dans* `main` *les colonnes et les lignes de* `a` *sont inversées)*

L'application de ces techniques sur l'élimination de Gauss est montrée figure 5.5

La fonction `ligne(int l,float **matrice)` construit un pointeur sur le début de la ligne numéro `l` de la matrice `a`. Dans notre exemple, une telle fonction est triviale : il s'agit de l'adresse `a[l]`. Dans le corps de la fonction, les accès aux tableaux sont remplacés par des accès aux variables construites à l'aide des expressions normalisées décrites précédemment.

Ainsi, non seulement on gagne du temps en ce qui concerne la transmission des messages, mais aussi on extrait du parallélisme. En effet, après une telle passe on précise l'ensemble des données accédées par chaque instance de la tâche. Si, grace à cette analyse, on découvre que deux tâches indépendantes écrivent dans deux colonnes différentes, alors elles pourront être parallélisées (ce n'est pas le cas si l'on sait juste qu'elles écrivent toutes les deux dans la même matrice).

5.4.4 Calcul du mode d'accès aux variables

En Athapascan-1 les dépendances entre tâches sont déterminées en fonction du mode d'accès aux paramètres des fonctions. Ces modes d'accès peuvent être : *lecture*, *écriture* ou *lecture/écriture*. De plus, lorsque nous générerons du code PM2 il nous sera utile de savoir si une variable est lue, écrite ou bien les deux.

Ces modes d'accès sont déterminés à l'aide des règles de communications comme montré plus haut. Grâce aux règles de réception, on détermine les données qui sont lues, les règles d'émission nous permettant de connaître les variables écrites.

5.4.5 Génération du code

Une fois que les trois étapes précédentes ont été réalisées il faut encore se conformer à la syntaxe de Athapascan et du C++. Il faut aussi créer les types qui correspondent aux lignes, aux colonnes, aux vecteurs, etc ... des paramètres des fonctions. Basé sur ces techniques, nous avons réalisé un programme qui produit du code Athapascan-1 parallèle. Il n'a malheureusement pas été possible d'obtenir des performances. En effet, dans le code tel que nous l'avons généré, c'est chaque élément de la matrice qui est virtuellement partagé. Ce grain est trop fin et implique trop de contrôle pour qu'Athapascan-1 puisse être efficace.

5.5 Le programme parallèle

La génération de code Athapascan-1 décrite ci-dessus est une étape qui simplifie la génération de code PM2. Les expressions normalisées vont nous permettre de construire les variables pour les structures de données rattachées aux tâches ainsi que les variables dans le code du programme (code des tâches, empaquetage/dépaquetage des données, ...). La fonctionnalisation va nous permettre de créer les procédures calculant les tâches. La détermination du mode d'accès aux variables permet d'optimiser le stockage et la transmission de celles-ci.

Le code produit est toujours composé de deux parties. Une partie *statique*, qui ne dépend pas du programme source. Il s'agit des structures de contrôle et de leurs fonctions de gestion, ainsi que l'implantation des algorithmes de contrôle, d'envoi et de réception des messages. La deuxième partie du code est dépendante de l'application. Il s'agit du code des tâches,

des structures de données qui sont rattachées aux tâches des fonctions d'empaquetage et de dépaquetage des données, etc ...

5.5.1 Partie statique

Nous décrivons ici l'ensemble des fonctions qui ne dépendent pas de l'application à paralléliser.

Envoi et réception des messages

La manière dont sont envoyés et reçus les messages ne diffère pas d'une application à l'autre (ce qui diffère est le contenu de chaque message). Dans le premier chapitre nous avons décrit comment des processus légers communiquent par l'intermédiaire d'appel de procédure à distance léger. L'envoi de messages est un cas particulier de ce schéma puisque le rôle du service appelé ne renvoie pas de données et que son code consiste à lire, à stocker les données dans la bonne tâche et à déterminer si cette dernière est devenue prête à être exécutée.

Les différentes informations dont est constitué un message sont résumées dans le figure 5.6. Supposons qu'un message de ce type soit construit. L'envoyer revient à appeler,

1. le type du message. Si c'est une diffusion ou un message point à point,
2. la tâche d'envoi et la tâche de réception (en cas de message point à point),
3. la liste des nœuds destinations en cas d'une diffusion,
4. le nœud de destination en cas de message point à point,
5. le nombre de nœuds auquel est destiné le message,
6. une liste chaînée de données envoyées, chaque élément de la liste est composé :
   - du nombre d'éléments,
   - d'un pointeur sur le premier élément du tampon d'envoi,
   - de la règle de communication concernée par cet envoi,
   - du type de transmission (par pointeur ou par copie),
   - d'un pointeur sur l'élément suivant de la liste.

FIG. 5.6: *Les différents champs d'un message*

sur le nœud où il est destiné, le service de réception de messages, le paramètre de cet appel étant le message lui-même.

Nous avons écrit deux services de réception. Un service pour les diffusions et un service pour les communications point à point. La seule différence est que les données sont transmises à un ou plusieurs nœuds. Le service appelé est choisi en fonction du type de message.

Lorsqu'un service de réception est appelé, outre les données, il reçoit les informations utiles à leurs stockage. Comme il connaît la tâche de réception, il sait à quelle tâche sont

destinées les données. Les données qu'utilise une tâche lui sont en principe privée. En général une tâche en utilise plusieurs (par exemple la ligne j et la ligne k de la matrice pour la tâche $T_2(k, j)$ de l'élimination de Gauss). Comme on connaît la règle de communication qui correspond au message et que la règle décrit quelles données sont transmises, on détermine où les ranger.

Gestion des tâches prêtes et des tâches en attente

Lorsqu'une donnée arrive pour une tâche, deux cas se présentent. Soit la tâche a déjà reçue des données. Dans ce cas elle a déjà été créée : c'est une tâche en attente. Soit elle n'a jamais reçu de données et il faut la créer. L'ensemble des tâches en attente de données est rangé dans une table de hachage [23]. Ce n'est pas la structure de données la plus efficace mais l'expérience montre que le temps passé à gérer cette structure est négligeable (en général moins de 1% du temps total d'exécution). Le problème principal de cette structure de données est qu'il faut choisir une bonne fonction de hachage. Pour une tâche $T_a(\vec{u})$, la fonction de hachage choisie est :

$$\kappa(T_a, \vec{u}) \bmod H$$

où H est la taille de la table et κ la fonction de placement de la tâche. En effet, étant donné que les tâches qui ont la même valeur de κ vont, à priori, être exécutées séquentiellement, il est peu probable que l'on ait beaucoup de tâches en attente de données ayant la même valeur de fonction de hachage au même instant. Ainsi, chaque entrée de la table de hachage n'ayant pas beaucoup d'éléments, les temps d'accès seront réduits. Par défaut, nous avons choisi 1531 comme taille de notre table. Il s'agit du nombre premier le plus proche de $(10^2 + 11^2)/2$. En effet si H est une puissance de 2 la fonction de hachage sera simplement constituée des premiers bits de la fonction de placement. De plus, le choix d'un nombre premier est justifié par le fait en que $\mathbb{Z}/p\mathbb{Z}$ est un corps lorsque p est premier. De plus le nombre de clé étant de l'ordre de quelques milliers, en général, il faut que H soit proche de ce nombre.

Lorsqu'il faut créer une tâche, on calcule le nombre de messages qu'elle doit recevoir. A chaque message reçu ce nombre est décrémenté. Lorsqu'il arrive à zéro la tâche devient prête à être exécutée. Elle est alors mise en fin de liste des tâches à exécuter. Lorsqu'un processus léger cherche à exécuter une tâche, il prend la première tâche prête dans la liste.

Il est à noter que ces structures peuvent être accédées concurremment. En effet, un nœud peut recevoir plusieurs messages en même temps. De même, plusieurs processus légers peuvent accéder à la liste des tâches prêtes en même temps. Pour régler les problèmes de cohérence de ces structures nous avons utilisé les sémaphores fournit par PM2.

5.5.2 Code généré automatiquement

Nous décrivons ici le code qui dépend de l'application et nous montrons comment il est généré automatiquement. Toutes les générations de nid de boucles pour parcourir des polyèdres sont faites par Enum. Le calcul du nombre de points d'un polyèdre est réalisé grâce aux polynômes de Ehrhart. Les opérations sur les règles et les ensembles sont menées grâce au Calculateur Omega.

Les données rattachées à chaque tâche

Nous avons opté pour le principe selon lequel les données sont privées aux tâches. Dans cette optique, il est nécessaire, pour chaque tâche générique de créer une structure qui décrive les données rattachées à cette tâche.

Comme nous l'avons montré dans la section 5.4 de ce chapitre, une analyse des règles de communication nous permet de déterminer, pour chaque tâche générique, les données qu'elle va lire et/ou écrire. Cela nous permet de créer une structure qui correspond à ces données et qui sera mise à jour (remplie) au fur et à mesure de l'arrivée des messages.

Exemple Dans l'élimination de Gauss les règles de réception et d'émission normalisées nous permettent de construire pour les tâches T_1 et T_2 la structure de donnée décrite figure 5.7.

```
typedef struct _T1_data{
  double *c_k_a ; /* READ_WRITE */
}T1_data ;

typedef struct _T2_data{
  double *c_j_a ; /* READ_WRITE */
  dounle *c_k_a ; /* READ */
}T2_data ;
```

FIG. 5.7: *Structures de données rattachées aux tâches génériques de l'élimination de Gauss*

Lorsque la règle d'émission normalisée $T_2(j-1,j) \rightarrow T_1(j) : A(i,j)|1 \leq k \leq n-2, k+1 \leq i \leq n$ est exécutée, les éléments $k+1$ à n sont extraits du tableau pointé par `c_j_a`. Puis une fois que toutes les règles correspondant à l'instance de la tâche T_2 sont exécutées la tâche est désallouée ainsi que les pointeurs `c_j_a` et `c_k_a`. Lorsque la donnée arrive sur le processeur qui exécute la tâche $T_1(j)$ la règle d'émission normalisée correspondante est exécutée : $T_1(k) \leftarrow T_2(k-1,k) : A(l,k)|2 \leq k \leq n-1, k+1 \leq l \leq n$. Le tableau `c_k_a` est alors alloué en mémoire et les données transmises sont stockées entre les éléments $k+1$ et n de ce tableau.

Le code des tâches

Le code de chaque tâche correspond à une fonction. Chaque fonction est générée de manière similaire à ce qui a été dit section 5.4 concernant la fonctionnalisation. Lorsqu'un processus léger veut exécuter une tâche qu'il vient de sélectionner dans la liste des tâches prêtes, il détermine de quelle tâche générique il s'agit, puis appelle la fonction correspondante en transmettant le vecteur d'itération et les données de cette tâche.

Les fonctions d'allocation et désallocation des tâches

Lorsqu'une tâche reçoit un message pour la première fois, il faut la créer. Lorsqu'une tâche a fini d'être d'exécutée et que toutes ses données sont envoyées, il faut la désallouer ainsi que les données qui lui sont rattachées. Pour chaque tâche, nous avons les informations suivantes :

- son identificateur : ce nombre permet de distinguer les tâches génériques entre elles,
- son vecteur d'itération,
- l'ensemble des données qui lui sont rattachées,
- le nombre de messages qui lui reste à recevoir.

Ainsi, la création et la destruction d'une tâche dépendent de l'application. Le code est généré en s'aidant de l'analyse des règles qui explicite quelles données sont lues et/ou écrites par une tâche. Le code de création d'une tâche réalise l'allocation des données et des autres informations. Pour allouer une donnée qui n'est pas un scalaire, on extrait du code source les informations relatives à la taille des lignes et des colonnes des matrices ainsi que la taille des vecteurs : il s'agit soit d'une constante, soit d'une fonction affine des paramètres. Le code de destruction de la tâche désalloue les données et les autres informations.

Empaquetage des données

Une fois qu'une tâche est exécutée, il faut envoyer les données qu'elle vient de calculer à ses fils. Pour cela, on parcourt l'ensemble des règles susceptibles de concerner cette tâche. Si, pour une règle donnée, les vecteurs d'itération de la tâche d'émission et de la tâche de réception ainsi que les paramètres vérifient le prédicat de la règle, celle-ci est exécutée.

Une règle d'émission décrit les données qui doivent être transmises. En général, les données calculées ne sont pas toutes transmises par la même règle. Par exemple, si une tâche a mis à jour la diagonale d'une matrice, il se peut qu'une première règle décrive l'envoi d'une partie de la diagonale et qu'une autre règle décrive l'envoi du reste de la diagonale. Pour respecter la sémantique des règles et du programme, il est nécessaire d'envoyer exactement les données décrites par la règle (à savoir ici, uniquement les éléments de la diagonale) Il faut donc être capable d'extraire parmi les données rattachées à une tâche celles qui sont décrites par la règle en question.

Or, les données qui sont rattachées à une tâche sont déterminées grâce aux règles de communications. Retrouver quelles données correspondent à une règle ne pose donc pas de difficultés particulières.

Pour chaque règle, on génère alors du code qui recopie ces données dans un tampon d'envoi qui calcule le nombre de données envoyées. Cela nous permet de générer ce qui correspond à la partie 6 d'un message et qui est décrit figure 5.6.

Exemple Nous avons vu que la tâche T_2 de l'élimination de Gauss lit la colonne k et écrit la colonne j de la matrice A. Les données rattachées à la tâche T_2 sont donc le tableau `c_k_a` et le tableau `c_j_a`. Ces deux tableaux sont unidimensionnels de longueur n, le paramètre du programme. Lorsque T_2 est terminée, la colonne j a été mise à jour et doit être transmise.

Supposons que ce soit la règle normalisée

$$T_2(k,j) \rightarrow T_1(k+1) : A(i,j) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$$

qui doive être appliquée. Dans ce cas, on alloue un tampon de taille $n-k$ et on y recopie les éléments de $k+1$ à n du tableau `c_j_a`.

Un cas particulier est l'extraction d'éléments de dimension 1 (ligne, colonne, ...) à partir d'un bloc de matrice. Dans ce cas là, il faut construire une boucle qui parcourt la matrice en question, pour extraire les bons éléments.

Dépaquetage des données

Lorsqu'un message est reçu, il faut le ranger au bon endroit, parmi les données rattachées à la tâche. Or, le numéro de la règle d'émission qui a servi à empaqueter les données est joint au message. On en déduit donc la règle de réception qui sert à dépaqueter les données.

Pour chaque règle on génère un code qui lit le tampon de données transmises et qui les recopie dans les données rattachées à la tâche.

Exemple La règle de réception normalisée qui correspond à l'exemple précédent est :

$$T_1(k) \leftarrow T_2(k-1,k) : A(l,k) | 2 \leq k \leq n-1, k \leq l \leq n$$

Le code généré pour cette règle correspond à la recopie des $n-k+1$ valeurs du tampon dans le tableau `c_k_a` rattaché à la tâche T_1 à partir de l'élément k.

Dans tous les cas, on crée un tampon d'envoi qui contient toutes les données à envoyer et seulement celles-ci.

La génération des messages

Les messages générés sont composés outre du tampon d'envoi, du fils (la liste des fils, s'il s'agit d'une diffusion) à qui sont destinées ces données. Un message est généré pour chaque règle valide. Nous verrons dans la section consacrée aux optimisations comment on peut concaténer des messages qui sont destinés à la même tâche.

Le calcul du nombre de messages d'une tâche

Le nombre de messages que doit recevoir une tâche est différent du nombre de père de cette tâche. Nous réalisons donc une fonction qui compte le nombre de messages que doit recevoir une tâche avant de devenir prête à être exécutée. Pour chaque instance d'une tâche, on détermine les règles de réception qui s'appliquent. Pour chaque règle, on construit un polynôme de Ehrhart qui évalue le nombre d'instances valides de la tâche d'envoi.

5.6 Optimisations

Le premier programme que nous avons généré ne comportait pas d'optimisation. L'exécution sur un seul processeur était beaucoup plus lente que le programme source. De plus,

l'exécution parallèle ne montrait pas une bonne extensibilité. Il s'est avéré indispensable de procéder à des optimisations pour obtenir des performances, aussi bien en terme de rapidité qu'en terme d'utilisation mémoire.

5.6.1 Utilisation des communications globales

De nombreuses règles de communication décrivent des diffusions. Il est nécessaire de traiter différemment ces règles de celles qui décrivent seulement des communications point à point. Lorsqu'une tâche envoie des données à n successeurs, il est possible d'envoyer ces données n fois en faisant une communication point à point. Dans une telle approche, la quantité de messages est très importante ; ce qui augmente la contention du réseau. Une meilleure approche est d'envoyer le message à chaque processeur qui exécute au moins un des n fils, en utilisant des communications optimisées. Notons que nous sommes capable de déterminer si une règle est bijective ou non. Ainsi, la détection de règles impliquant des communications globales se fait naturellement.

De plus, PM2 permet de réaliser des diffusions de manière transparente. Il utilise, quand c'est possible, les routines de communication globales disponibles sur la plateforme. Dans la partie statique du code, il y a un service de réception pour les messages point à point et un service de réception pour les diffusions. Lorsque le service de réception d'une diffusion est activé, c'est que le nœud vient de recevoir un message destiné à plusieurs tâches qui s'exécuteront plus tard sur celui-ci. Une analyse de la règle qui implique cette diffusion nous permet de générer le code qui calcule l'ensemble des nœuds à qui il faut distribuer le message et qui, sur chacun de ces nœuds, construit l'ensemble des tâches à qui il faut rattacher les données.

5.6.2 Fusion des règles

Lorsqu'il analyse un programme, PlusPyr, génère un grand nombre de règles. Certaines d'entre elles peuvent être fusionnées. C'est le cas lorsqu'elles décrivent l'envoi de données contigües. La fusion de règles est importante car elle réduit leurs nombres. Lorsqu'une tâche est terminée on doit tester toutes les règles pour déterminer lesquelles s'appliquent. Plus on a de règles plus le nombre de tests sera important et plus le nombre de règles valide risque de l'être aussi. Comme chaque règle valide provoque l'envoi d'un message, on comprend que réduire le nombre de règles permet à la fois de réduire le contrôle du programme parallèle et aussi de diminuer le temps de communication. Cette étape est indispensable pour obtenir de la performance. L'algorithme de fusion des règles est donné dans le chapitre 2.

5.6.3 Fusion des messages

Lorsqu'une tâche envoie deux messages différents à une même tâche de réception, il est n'est pas nécessaire d'envoyer les deux messages à la suite. Pour gagner du temps nous fusionnons à l'exécution ces deux messages en un seul. C'est le rôle de la liste chaînée qui constitue le champ 6 de la figure 5.6. Chaque élément de la liste est transmis dans un seul message physique et la liste est reconstituée sur le nœud d'arrivée.

5.6.4 Transmissions des données sur un même nœud

Le placement trouvé par SLC essaye autant que possible de placer les tâches qui communiquent beaucoup entre elles sur le même processeur. Cependant lors de l'exécution il est nécessaire d'exécuter toutes les règles. Ainsi un grand nombre d'instances de règles décrivent la transmission de données sur un même processeur. Nous avons donc traiter ce cas particulier à part.

Lorsqu'une tâche T doit transmettre des données à une tâche T' et que ces deux tâches s'exécutent sur le même nœud, il n'est pas nécessaire de construire un tampon en empaquetant les données transmises puis de les dépaqueter dans les données associées à la tâche T'. En effet, ceci induit des recopies de données et des allocations mémoire superflues. Pour chaque règle bijective nous générons une fonction qui :

- extrait T' de l'ensemble des tâches en attente de données, ou la crée si elle n'est pas dans cet ensemble,
- copie les données décrites par la règle de la tâche T à la tâche T' sans tampon intermédiaire,
- range T' dans l'ensemble des tâches prêtes ou dans l'ensemble des tâches en attentes selon qu'elle attende ou non d'autres messages.

Cette fonction est activée à l'exécution, si une règle décrit la transmission de données sur un même nœud.

5.6.5 Transmission par pointeur des données

Les règles de communications de PlusPyr décrivent seulement la transmission de messages. Elles sont issues des dépendances de flot du programme. Les autres dépendances ne sont pas prises en compte. Cela induit une recopie partielle des données sur les processeurs. Comme en plus les données sont, en principe, privées à chacune des tâches, la recopie partielle des données se produit entre les tâches qui s'exécutent sur un même processeur. L'optimisation que nous présentons ici à pour but de palier à la recopie d'une même donnée sur différentes tâches qui s'exécute sur un même processeur. En contre partie, ces données ne seront plus strictement privées aux tâches : on aura pour les tâches correspondantes, un pointeur sur la zone mémoire correspondant à la donnée. Cela va permettre à la fois de réduire le coût mémoire et d'accélérer l'exécution en diminuant le nombre de dépaquetages (une donnée distribuée sur n tâches n'étant plus dépaqueter qu'une fois au lieu de n fois).

Afin que l'exécution soit correcte, plusieurs tâches peuvent pointer sur une même donnée seulement si cette donnée est lue et n'est pas écrite. Grâce aux règles d'émission nous savons, pour chaque donnée qui est lue par une tâche, si elle est aussi écrite (modifiée). Nous connaissons aussi l'ensemble des règles qui transmettent ces données. Ainsi, dans le cas particulier où une donnée est transmise par une diffusion et qu'elle est seulement lue, nous pouvons nous passer de la dupliquer : sur chaque nœud toutes les tâches concernées par la diffusion pointeront sur la même donnée.

La donnée étant pointée par plusieurs tâches, il faut aussi mettre en place un mécanisme qui empêche de désallouer celle-ci lorsque l'on désalloue une tâche qui pointe sur elle. On ne peut la désallouer qu'une fois que toutes les tâches qui doivent la lire sont terminées.

L'implantation de ce mécanisme se fait en rajoutant un compteur qui est initialisé au nombre de pointeurs sur cette donnée et décrémenté chaque fois qu'une tâche qui pointe dessus est désallouée. Lorsque le compteur arrive à zéro la donnée est elle aussi désallouée.

On peut aussi réaliser l'envoi de données par pointeur. La difficulté est d'éviter que la tâche émettrice ne désalloue cette donnée une fois qu'elle la transmise. Pour ce faire on affecte a `NULL` le pointeur de la tâche émettrice une fois les données transmises.

On peut penser que la mise en œuvre de cette optimisation est coûteuse. C'est sans compter tout ce que l'on gagne :

- il n'y a plus de copie de données dans le tampon d'envoi, ni d'allocation du tampon d'envoi
- il n'y a plus de duplication de données
- sur le processeur qui la reçoit, la donnée n'est dépaquetée, allouée et désallouée qu'une fois

Il est possible de mettre en place la copie par pointeur lors de communications point à point. Cela est particulièrement intéressant pour les règles qui ont été mises à zéro par SLC. En effet, dans ce cas, on est sur que la tâche d'émission et la tâche de réception seront toujours exécutées sur le même processeur. Dans ce cas, il est inutile d'empaqueter les données dans un tampon de communication puis de les dépaqueter dans la tâche d'arrivée puisque, dès la compilation, on sait que ce tampon ne sera pas envoyé sur le réseau.

Ici il faut être très prudent pour éviter des ruptures de cohérence entre les données transmises aux tâches. La figure 5.8, montre un cas où une donnée est transmise deux fois de suite par pointeur sur un même processeur. Si cette donnée est modifiée la deuxième tâche qui possédera la copie subira elle aussi les modifications.

- La tâche T_1 calcule la colonne pointée par `c_k_a`.
- T_1 transmet par pointeur la colonne `c_k_a` à T_2, un de ses fils, sur le même processeur.
- T_2, qui vient de recevoir son dernier message s'exécute et modifie la colonne qui pointe sur les mêmes cases mémoire que `c_k_a`.
- T_1 transmet par pointeur la colonne `c_k_a` à T_3, un de ses fils, sur le même processeur.

FIG. 5.8: *Exemple de rupture de cohérence dans les données : T_3 reçoit les données une fois modifiées par T_2 et non celles calculées par T_1*

Pour palier à ce problème, il est nécessaire, avant d'autoriser le transfert par pointeur de données, de s'assurer que la donnée ne sera pas à nouveau transmise. Pour chaque règle on vérifie donc si des règles qui seront exécutées après enverront la même donnée, si ce n'est pas le cas la donnée peut être transmise par pointeur. Sinon seule la dernière règle qui transmet une donnée peut le faire par pointeur. Les dernières règles à être exécutées étant celles qui pourront permettre la transmission par pointeur, il est important de générer le code de manière à ce que les règles mises à zéro soient exécutées en dernier. Les autres règles sont triées suivant la quantité de données transmises.

Nous avons mis au point une dernière optimisation concernant la réception des données. Lorsqu'une ligne ou une colonne est transmise on essaye d'éviter la recopie de cette donnée entre le tampon de communication et la tâche de réception en faisant, ici aussi, une affectation de pointeur. Ceci n'est possible que si la tâche réceptrice n'a pas encore reçu une autre partie de la ligne ou de la colonne (dans un tel cas on est obligé de faire des copies). En effet, si la tâche a déjà reçu une partie de la colonne, cela signifie que la variable qui décrit la colonne dans les données rattachées à la tâche pointe déjà sur des données en partie valide. Il est nécessaire de conserver ces données lorsque une nouvelle partie de la colonne arrive pour cette tâche.

5.7 Résultats

Nous avons généré automatiquement le code parallèle de plusieurs noyaux de calcul intensif et nous avons mesuré leurs vitesse d'exécution sur deux plateformes parallèles.

5.7.1 Les plateformes de test

Nous avons utilisé des machines parallèles à mémoire distribuée pour évaluer les programmes. Il s'agit de :

- l'IBM SP2 du LaBRI à Bordeaux. C'est une machine composée de 16 processeurs IBM RS6000 reliés par un réseau «*High Performance Switch*» et qui fonctionne sous PM2-MPI
- La POM du LIP à Lyon. Il s'agit d'une pile composée de 14 PowerPC reliés par Myrinet et fonctionnant sous PM2-BIP.

Nous avons mesuré la vitesse du réseau de chacune des machines à l'aide de *ping-pong*. Le résultat est montré figure 5.9. Les performances obtenues sont 61 μs de latence et 39 Mo/s de débit pour l'IBM SP2 et 7 μs de latence et 64 Mo/s de débit pour la POM.

En ce qui concerne la vitesse d'exécution, l'exécution d'une élimination de Gauss d'une matrice d'ordre 2000 sur un seul processeur dure 508 secondes sur la SP2 (soit environs 15,7 Mflops) et 257 secondes sur la POM (soit environs 31,2 Mflops).

Enfin, les 8 premiers nœuds de la SP2 possèdent 128 Mo de mémoire et les 8 suivants 64 Mo. Chaque nœud de la POM a une mémoire de 64 Mo

5.7.2 Le placement des grappes

SLC donne une fonction d'allocation qui ne dépend pas du nombre de processeurs de la machine. Nous avons plusieurs possibilités pour répartir les grappes sur les P processeurs :

- **Cyclique**. Soit C le numéro de grappe de la tâche Tet p le processeur qui doit l'exécuter. La première possibilité est de répartir chaque grappe uniformément sur les processeurs (cyclique) :

  $$p = c \bmod P$$

  Une telle fonction a de bonne propriété d'équilibrage de charge.

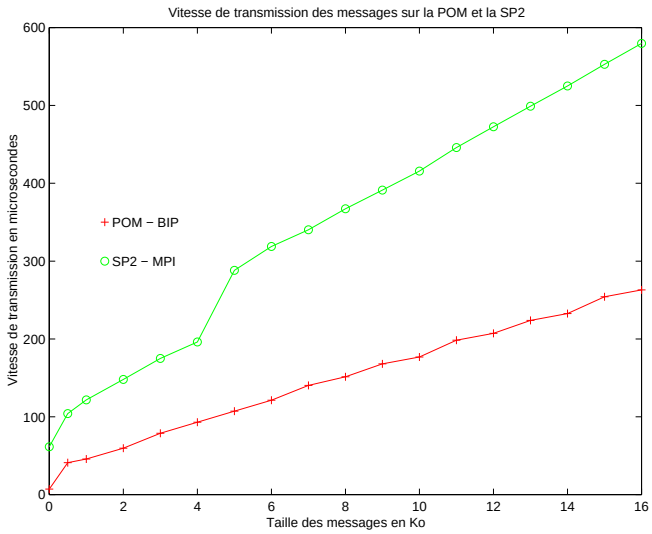


FIG. 5.9: *Comparaison de la vitesse réseau de la POM et de la SP2*

– **Bloc-cyclique**. Cependant on peut vouloir regrouper des grappes contiguës en bloc de taille B pour supprimer des communications résiduelles (bloc-cyclique) :

$$p = (c \div B) \bmod P$$

Les deux fonction présentées ci-dessus on l'avantage de ne pas réclamer de connaissances sur le nombre de grappes.

– **Bloc**. Si ce nombre est connu (N) on peut supprimer un maximum de communications en répartissant par bloc les grappes : dans ce cas il suffit de poser $B = N \div P$ dans l'allocation bloc-cyclique. Une allocation par bloc n'a pas, en général, de bonne propriété d'équilibrage de charge. Obtenir un bon compromis entre l'équilibrage de charge et la suppression des communications demande dans le cas bloc-cyclique de calculer la taille de bloc optimale, ce qui réclame de l'expertise de la part du programmeur.

– **Réflexion**. Le placement par *réflexion* est un placement assez proche du placement par bloc mais, qui, à priori, a de meilleures propriétés d'équilibrage de charge. On pose $B = N \div (2P)$ et si $c \geq N/2$ alors $c = N - c$. On pose alors $p = (c \div B) \bmod P$. Cela revient à créer $2P$ bloc de taille $N/(2P)$ et à les allouer en repliant une moitié sur l'autre. Un exemple de placement par réflexion est donné figure 5.10 pour un domaine triangulaire.

5.7.3 Résultats d'accélération

Nous avons choisi d'exécuter 3 programmes sur les plateformes décrites plus haut. Il s'agit de l'élimination de Gauss, de l'algorithme de Givens, et de la diagonalisation de Jordan.

Le choix de l'élimination de Gauss se justifie par le fait qu'il s'agit d'une application standard qui nous a servit d'exemple tout au long de cette thèse.

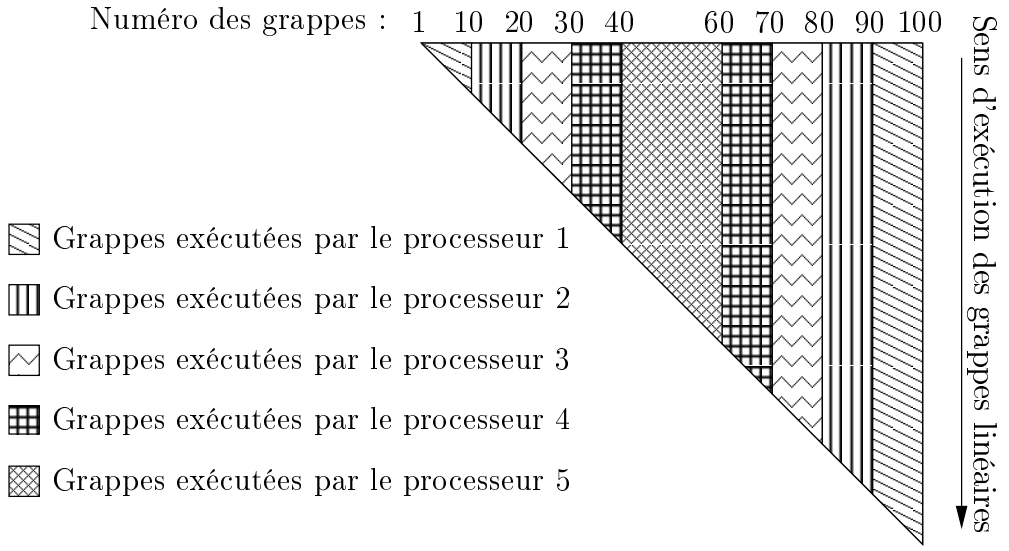


FIG. 5.10: *Allocation des 5 processeurs d'une machine parallèle lors d'un placement par réflexion de 100 grappes, pour un domaine triangulaire.*

Le regroupement du graphe de tâches de l'algorithme de Givens laisse subsister beaucoup de communications. Pour obtenir des performances, il faut regrouper les grappes par bloc. Cela nous permettra d'évaluer les différentes méthodes de placement des grappes décrites précédemment.

Le GTP de la diagonalisation de Jordan contient énormément de règles. La parallélisation de cette application permet de mettre en évidence l'importance de la fusion des règles.

Les temps d'exécution s'entendent sans l'exécution des tâches d'entrée et de sortie. La distribution des données est supposée avoir déjà eu lieu. Comme le temps d'exécution des programmes ne dépend pas de la valeur des données, celles-ci sont initialisées aléatoirement.

Les figures 5.11 5.12 et 5.13, montrent les courbes d'accélération pour les trois applications sur chacune des machines. Il s'agit du rapport entre le temps d'un programme séquentiel et celui du programme parallèle. Le programme séquentiel correspond à la traduction en *C* du programme *Tiny* compilé avec les mêmes options que le programme PM2. Le *C* accédant les tableaux par colonnes, ceux-ci sont transposés si nécessaire. Notre programme séquentiel n'est clairement pas le «*meilleur programme séquentiel*». Cependant, dans une optique de génération automatique de code, il est naturel de comparer le programme d'entrée avec le programme généré en utilisant, pour construire l'exécutable, le même compilateur dans les deux cas. Le nombre de processus légers exécutant les tâches est 1. On obtient toujours de moins bonnes performances avec plus d'un processus léger d'exécution. Cela signifie que le surcoût de gestion de plusieurs processus légers dépasse le gain dû au recouvrement calcul/communication. Cela est compréhensible, car l'envoi des messages est totalement asynchrone et, en ce qui concerne la réception, il y a création d'un processus léger chaque fois qu'un service de réception est exécuté. Il est possible d'empêcher la création d'un processus léger et de faire exécuter les appels de procédure à distance par un processus spécial, qui tourne sans cesse dans le processus et qui bloque tous les autres processus légers lorsqu'il

reçoit un message. Dans cette configuration, on constate alors une légère augmentation du temps d'exécution (quelques pourcents). Cela signifie que bien qu'aucun processus léger ne soit créé dynamiquement à la réception des messages, le fait de stopper le processus qui exécute les tâches pour réceptionner les messages augmente le temps d'exécution. Cela prouve qu'il y a donc bien, dans une faible mesure, recouvrement des calculs avec les communications.

Comme La SP2 possède plus de mémoire que la POM, il a été possible d'y exécuter de plus grand problème. En revanche, sur la POM, il n'a pas été possible de faire fonctionner les programmes sans qu'ils «*swappent*» sur le disque pour une taille de matrice 3000 sur un seul nœud. Le temps séquentiel est alors obtenu en ne tenant compte que du temps CPU du programme.

D'une manière générale, on constate que plus la taille de la matrice augmente plus le programme parallèle est performant. Cela vient du fait que la granularité des tâches augmente avec la taille du graphe.

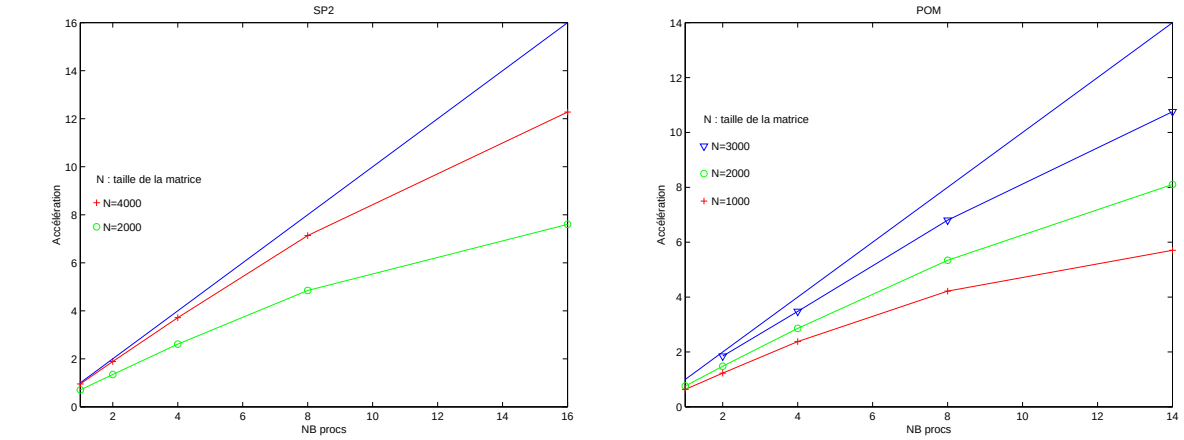


FIG. 5.11: *Mesure d'accélération pour l'élimination de Gauss*

En ce qui concerne l'élimination de Gauss, on constate que le programme se comporte bien et que les courbes montrent une assez bonne extensibilité. On obtient une accélération de 12,28 pour N=4000 sur la SP2 avec 16 processeurs.

En ce qui concerne l'algorithme de Givens, les résultats sont moins bons que ceux de l'élimination de Gauss. On obtient une accélération de 6,76 pour N=4000 sur la SP2 avec 16 processeurs. La répartition des grappes est réalisée suivant la politique de *réflexion*. Le problème est que, bien que chaque processeur a à peu près la même charge de travail, dans un tel schéma, le premier processeur à commencer l'exécution est aussi le dernier à finir. Il est bien évidement possible de calculer la taille de bloc optimal théorique pour chaque problème. Cela est cependant contraire à notre optique d'une génération automatisée. En effet, il semble difficile de calculer la fonction qui, pour chaque GTP et chaque regroupement trouvé par SLC, donne la taille de bloc optimal en fonction des paramètres de la machine cible.

L'exécution de la diagonalisation de Jordan a un bon comportement lorsqu'un nombre maximum de règles sont fusionnées, comme le montre la figure 5.13. On obtient une accéléra-

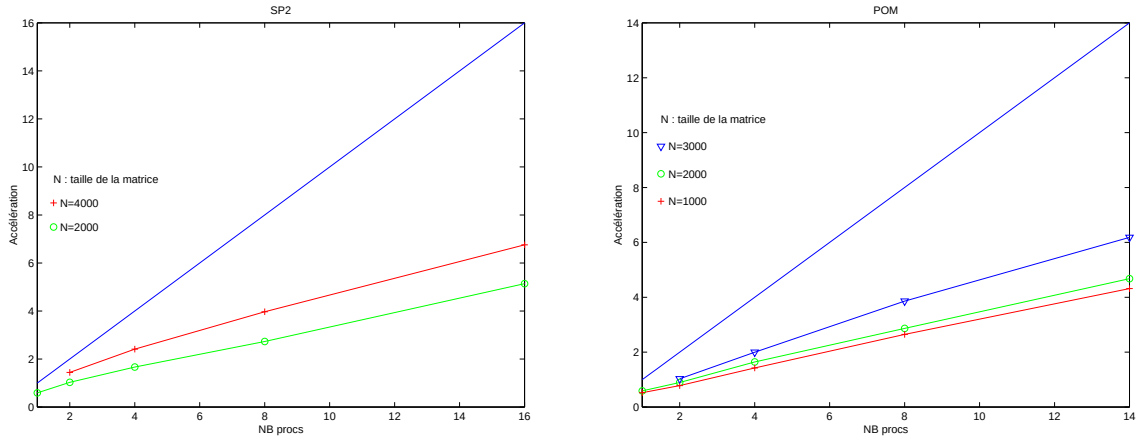


FIG. 5.12: *Mesure d'accélération pour l'algorithme de Givens*

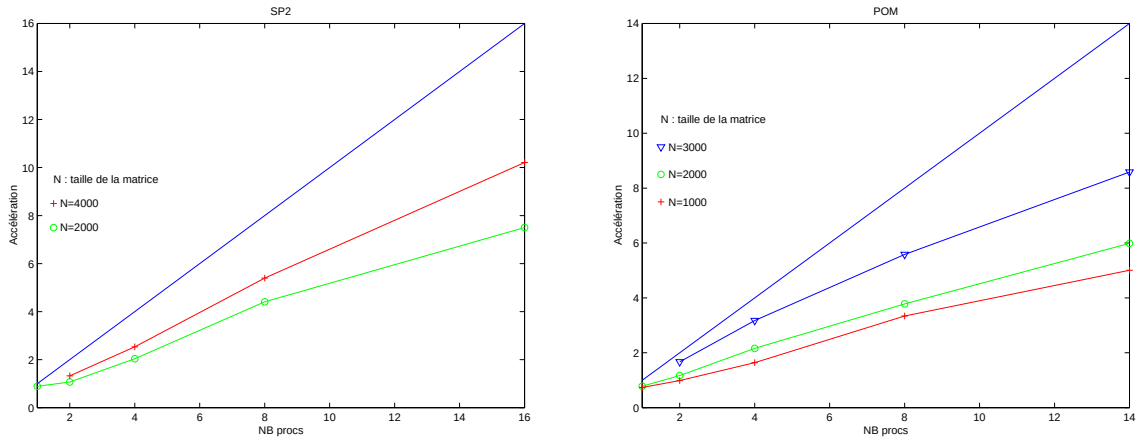


FIG. 5.13: *Mesure d'accélération pour la diagonalisation de Jordan*

tion de 10,21 pour N=4000 sur la SP2 sur 16 processeurs. Lorsque les règles sont fusionnées elles sont au nombre de 4 dont 2 ne sont pas en conflit. Lorsque les règles ne sont pas fusionnées elles sont au nombre de 12 dont 6 ne sont pas en conflit. Dans le deuxième cas, le surcoût du contrôle fait que le temps d'exécution pour une matrice d'ordre 1000 est plus que doublé.

5.7.4 Chronométrage des différentes parties du programme

Pendant l'exécution du programme parallèle nous mesurons le temps que prennent les différentes parties du programme. Leurs proportions par rapport au temps total sont montrés figure 5.14 pour l'élimination de Gauss sur une matrice d'ordre 2000 sur la POM et figure 5.15 pour l'algorithme de Givens sur une matrice d'ordre 2000 sur la POM. Lorsque le processus léger est préempté, pour que le processus exécute un service de réception le chronomètre de la fonction dans laquelle ce processus se trouvait continue à tourner. Cela fausse un peu les

mesures, mais on estime que statistiquement, les fonctions où les processus légers sont le plus interrompus sont celles où ils passent le plus de temps.

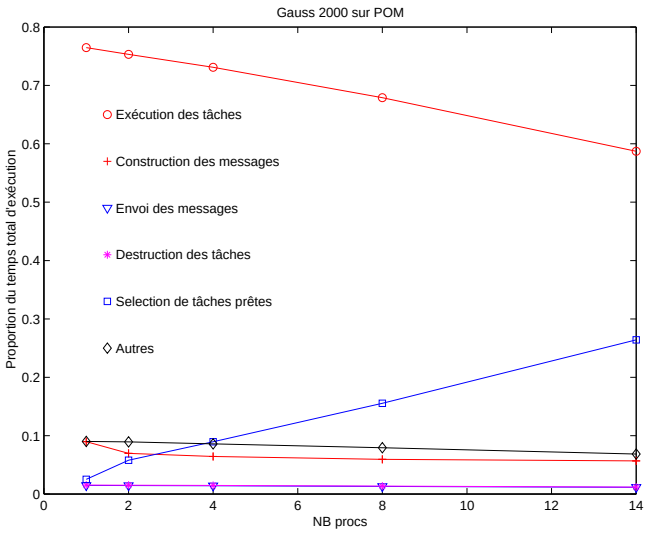


FIG. 5.14: *Proportion du temps total d'exécution des différentes parties du programme pour l'élimination de Gauss*

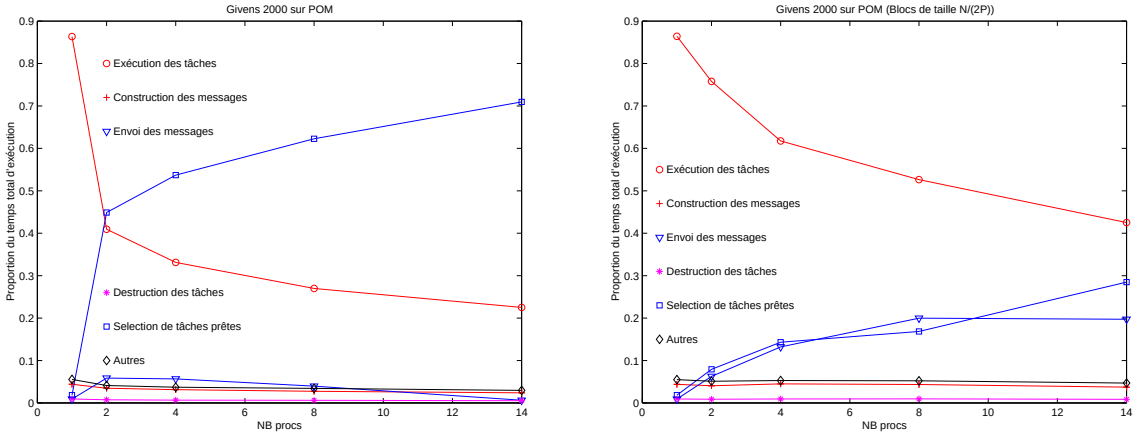


FIG. 5.15: *Proportion du temps total d'exécution des différentes parties du programme pour l'algorithme de Givens (à droite : un placement par réflexion) (à gauche un placement bloc-cyclique)*

On constate que, d'une manière générale, la proportion de temps passée à exécuter des tâches diminue avec le nombre de processeurs, alors que la proportion de temps passée à sélectionner des tâches augmente avec le nombre de processeurs. Cela signifie que, pour une taille de problème constante, le processus léger a de moins en moins de tâches à exécuter,

il passe donc de plus en plus de temps bloqué à attendre des tâches prêtes. Ceci est particulièrement criant pour l'algorithme de Givens où le chronométrage a été réalisé sur le nœud qui commence et termine le calcul, alors que l'exécution sur un processeur montre un très bon comportement. On remarque cependant que dans le cas de l'algorithme de Givens un placement bloc cyclique (avec une taille de bloc de $N/2P$), donne de meilleur résultat. En effet, dans ce cas, les processeurs ont du travail plus régulièrement que dans le cas d'un placement *réflexion*, il est dans ce cas 1,17 fois plus rapide sur 14 processeurs.

En ce qui concerne le contrôle du programme parallèle, on constate qu'il est à peu près de 20 % quel que soit le nombre de processeurs.

5.8 Conclusion

Dans ce chapitre nous avons montré comment le GTP d'une application est utile pour générer automatiquement du code parallèle. Nous avons décrit un prototype qui permet de construire, à partir du code source d'une application et du GTP correspondant, un programme parallèle qui exécute les tâches en respectant le regroupement linéaire trouvé par SLC.

L'utilisation d'un environnement multithreadé nous permet d'avoir un envoi et une réception de messages complètement asynchrone et transparente au niveau du programme.

Nous avons décrit un certain nombre d'optimisations qui se sont révélées indispensables pour obtenir de la performance.

Grâce au GTP, le code produit est générique : il fonctionne pour n'importe quelle valeur des paramètres et n'importe quel nombre de processeurs. L'allocation des tâches est calculée à partir de la fonction trouvée par SLC. Ainsi, elle est décentralisée et s'évalue en temps constant.

L'évaluation du code généré a été réalisée sur deux machines parallèles à mémoire distribuée. Les résultats obtenus sont encourageants et montrent que la méthode proposée est viable. La taille des graphes de tâches ainsi exécutées dépasse de loin ce qui à été réalisé jusqu'à aujourd'hui.

Chapitre 6

Conclusion et perspectives

6.1 Conclusion

Les techniques d'ordonnancement statique échouent lorsque l'on veut analyser des graphes de tâches importants ou construire des programmes génériques en parallèlisme de contrôle.

Pour apporter une réponse à ces deux problèmes, nous avons étudié le modèle appelé graphe de tâches paramétré. Les graphes de tâches paramétrés permettent de décrire symboliquement les graphes de tâches issus de certains noyaux de calculs intensifs que l'on trouve dans les applications scientifiques.

Notre travail se décompose en trois volets :

1. nous avons conçu et étudié un algorithme d'ordonnancement qui utilise directement le graphe de tâches paramétré. Nous avons intégré cet algorithme dans un schéma dynamique pour permettre la construction de programmes génériques. Une étude théorique ainsi que des simulations ont été conduites. Le principal avantage de cette méthode est que le coût mémoire de l'ordonnancement est grandement réduit. Pour la plupart des graphes que nous avons étudiés la quantité de mémoire requise pour les ordonnancer est très inférieure à leur taille. En ce qui concerne l'exécution dynamique de l'ordonnancement, le schéma maître-esclave proposé dans cette thèse permet de recouvrir le calcul de l'ordonnancement avec son exécution. Dans la plupart des exemples que nous avons traité, la quantité des messages de contrôles est inférieur, en ordre de grandeur, à la quantité de calcul. Cependant, il se peut que dans certains systèmes, les messages de contrôles induisent un fort ralentissement de l'exécution. Pour supprimer les messages de contrôles nous avons proposé un autre schéma dynamique basé sur le calcul décentralisé de l'ordonnancement. Cette méthode, en contrepartie, ne permet pas de recouvrir le calcul de l'ordonnancement avec son exécution.
2. Nous avons mis au point une technique d'allocation symbolique du GTP appelée SLC. Nous garantissons que cette allocation forme des grappes linéaires. Pour ce faire, nous avons introduit la notion de règle bijective qui permet de selectionner les communications point à point. Nous avons aussi introduit la notion de conflit qui permet de choisir un ensemble de règles qui décrivent un graphe de tâches formé seulement de chaînes simples. Le temps et le coût mémoire de l'allocation sont alors indépendants

de la taille du problème. Nous avons comparé cette méthode avec des algorithmes d'ordonnancement statique. A notre connaissance, SLC est le seul algorithme qui produit une allocation symbolique des calculs dans le modèle graphe de tâches à gros grain. Ces performances sont comparables à des algorithmes d'ordonnancement statique qui utilisent le graphe de tâches instancié en entrée. En revanche, la vitesse de calcul et la taille des graphes qui peuvent être analysés par SLC sont très supérieures à ce qqu'il est possible de réaliser à l'aide des techniques statiques mais non symboliques déjà existantes.

3. Nous avons décrit le prototype d'un générateur de code qui produit un programme multithreadé lequel se conforme à l'allocation trouvée par SLC. Nous obtenons un code générique qui fonctionne pour toutes les valeurs des paramètres et de la machine cible. Nous utilisons l'environnement de programmation PM2 pour des raisons d'efficacité et de portabilité. Dans le principe, les données qui sont lues ou écrites par une tâche lui sont privées (il n'y a pas d'accès concurrent à une donnée). Une analyse syntaxique du code source et des règles nous permet de fonctionnaliser le programme écrit en tâches et de générer les fonctions d'empaquetage et de dépaquetage des données. Nous avons mis au point des optimisations qui augmentent grandement la performance du code parallèle. Nous utilisons les routines de communications globales pour transmettre les diffusions. Nous autorisons la transmission par pointeur lorsqu'une même donnée est lue par plusieurs tâches. Nous réduisons les copies de données lors de communications intra-processeur.

   Nous avons testé nos programmes sur une pile de Power-PC et sur l'IBM SP2. Les résultats montrent que les programmes parallèles ainsi générés sont efficaces.

6.2 Travaux futurs

1. Il semble indispensable d'étendre la classe des programmes qui peuvent être parallélisés à l'aide des méthodes présentées dans cette thèse. En effet, même si dans le cadre d'applications scientifiques un grand nombre de codes sont à contrôle statique, il est important de généraliser les résultats pour des codes irréguliers. Les travaux de Barthou [8, 9] sur l'analyse du flot de données pour des dépendances non-affines constituent un bon point de départ. Dans ce cadre, il faudrait modifier PlusPyr pour qu'il puisse générer des règles de communications dans le cas de dépendances non-affines.

   Dans un premier temps, on pourraitimposer que les `if` et `while` soient compris entre les délimiteurs de tâches. Il ne serait alors plus possible de connaître la durée de l'instance d'une tâche. Cependant, sous certaines conditions (à préciser), les règles resteraient identiques. Le calcul de l'ensemble de Mandelbrot est un exemple de programme qui vérifie ces propriétées.

2. Une étude intéressante à mener consisterait à placer automatiquement les délimiteurs de début et fin de tâches. Cela permettrait d'automatiser complètement la construction du GTP. La difficulté consiste à trouver un équilibre entre le grain et le parallèlisme exprimé.

   A ce titre plusieurs méthodes peuvent être mises au point.

On peut, par exemple, restructurer le code pour rendre interne les boucles non parallèles et externe les boucles parallèles. Les délimiteurs de tâches peuvent être placés juste avant la première boucle non parallèle (on privilégie le parallèlisme), ou juste après la première boucle parallèle (on privilégie alors, le grain). Un calcul symbolique du grain peut se révéler utile pour bien placer les délimiteurs.

Une autre méthode consiste, dans un premier temps à placer les délimiteurs de tâches entre chaque instruction et de calculer la fonction de placement κ. Si cette fonction ne fait pas apparaitre un indice de boucle et que cette boucle peut-être rendue interne, alors on peut en faire une tâche.

3. Les expériences menées sur les machines à mémoire distribuée montrent que les performances obtenues sont loin des performances crêtes de la machine. Par exemple, sur la SP2, le programme de l'élimination de Gauss est exécuté à la vitesse d'environ 171 Mflops, sur 16 processeurs, alors que la puissance crête d'un seul nœud est de 220 Mflops. Cela signifie que notre programme n'utilise qu'environ 5% de la puissance totale disponible. Comme sur un seul processeur on utilise un peu plus de 6% de la puissance crête, la performance relative (accélération) du programme parallèle est bonne. Pour obtenir des performances proche de la puissance crête de la machine parallèle il est nécessaire d'utiliser des bibliothèques de calcul par bloc comme par exemple les BLAS (Basic Linear Algebra Subroutines [30, 55]).

   En effet, l'utilisation de bibliothèques de calcul permet de s'approcher facilement des performances crêtes de la machine. Par exemple, la multiplication de matrices du projet ATLAS (Automatically Tuned Linear Algerbra Software) optimisée sur la POM utilisant les BLAS3 va environ 8,6 fois plus vite que le programme de la figure 7.6 pour des matrices d'ordre 1000 (une fois que la matrice B est transposée).

   Intégrer les opérations par bloc dans PlusPyr peut se faire très simplement. Il suffit d'ajouter un paramètre de taille de bloc dans chaque programme : les éléments de matrice deviennent alors des blocs de matrice. Seule la sémantique des opérations change (par exemple l'opération de division devient une opération d'inversion). De plus, la durée de calcul et de communication doit être multipliée par un facteur qui dépend du (ou des) paramètre(s) de taille de bloc. Une fois ces changements effectués le GTP est identique à une version sans bloc. SLC ou d'autres algorithmes peuvent alors être appliqués.

   Lors de la génération de code, les opérations et les affectations sont remplacées par des appels aux routines par bloc. Le module de communication doit aussi être réécrit pour envoyer des blocs de matrices à la place d'éléments.

   Les performances du code généré réalisant des appels aux BLAS3 seront ainsi beaucoup plus proches des performances crête de la machine parallèle.

4. Nous avons montré l'importance de la fusion des règles. Moins de règles accélère SLC et réduit le contrôle du programme généré. L'algorithme de la figure 2.6 permet de supprimer les multi-arcs. Cependant, il se peut que deux règles ne décrivent pas exactement le même ensemble d'arcs et peuvent néanmoins être fusionnées. C'est le cas des règles

$$T_1(k) \rightarrow T_1(k+1) : b(i) | 1 \leq k \leq n-1, k+1 \leq i \leq n$$

$$T_1(k) \rightarrow T_1(k+1) : b(k) | 1 \leq k \leq n$$
$$T_1(k) \rightarrow T_1(k+1) : b(i) | 2 \leq k \leq n, 1 \leq i \leq k-1$$

qui peuvent être fusionnées en

$$T_1(k) \rightarrow T_1(k+1) : b(i) | 1 \leq k \leq n, 1 \leq i \leq n$$

L'algorithme de fusion des règles que nous avons mis au point ne fonctionne que pour des règles transmettant des données différentes mais contigües pour un même ensemble d'arcs. Il manque un algorithme qui permette de traîter la fusion de règle dans le cas général, c'est à dire qui puisse analyser à la fois les données transmises et l'ensemble des arcs décris par les règles.

Chapitre 7

Annexe : les noyaux de calcul utilisés

Nous présentons ici les codes utilisés pour les expériences de cette thèse. Il s'agit de l'élimination de Gauss (figure 7.1), l'algorithme de Givens (figure 7.2), de l'élimination de Gauss suivit d'une résolution triangulaire (figure 7.3), de la diagonalisation de Jordan (figure 7.4) de la multiplication de matrice (figure 7.6) et d'un calcul de puissance de matrice (figures 7.7 et 7.8). Pour chaque code un tableau résume les caractéristiques du graphe de tâches correspondant et donne le nombre de million d'opérations flottantes que doit effectuer le programme. Les 6 premiers noyaux sont décrit dans [48]. En ce qui concerne la puissance de matrice, nous avons écrit ce programme à l'aide de plusieurs paramètres de manière à ajuster finement la granularité. Ce programme calcule la puissance $m^{\text{ième}}$ d'une matrice de rang n en la multipliant m fois par elle-même. Un calcul est effectué sur chaque puissance intermédiaire.

```
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)
    for l = k + 1 to n do
      a(l,k) = a(l,k) * s
    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)
      endfor
    endtask
  endfor
endfor
```

Taille de la matrice	2000	4000
Nombre de tâches $(n^2/2 + 3n/2 - 2)$	2 002 998	8 005 998
Nombre d'arcs $(n^2 + n - 4)$	4 001 996	16 003 996
Nombre d'opérations flottantes $((n^3 + n^2 - 2)/10^6)$	8 004	64 016

FIG. 7.1: *Élimination de Gauss et caractéristiques du graphe de tâches*

```
param n
real a(n,n),c,s,a1,a2,d
for i = 1 to n do
  for j = i+1 to n do
    task                    T1(i,j)
      a1 = a(i,i)
      a2 = a(j,i)
      d = sqrt(a1*a1+a2*a2)
      c = a1/ d
      s = a2/ d
      for k = i to n do
        a1 = a(i,k)
        a2 = a(j,k)
        a(i,k) = c * a1 + s * a2
        a(j,k) = -s * a1 + c * a2
      endfor
    endtask
  endfor
endfor
```

Taille de la matrice	2000	4000
Nombre de tâches $(n^2/2 - n/2)$	1 999 000	7 998 000
Nombre d'arcs $(n^2 - 2n)$	3 996 000	15 992 000
Nombre d'opérations flottantes $(11 * (n^3/3 + n^2/2 - \frac{5}{6}n)/10^6)$	29 355	234 754

FIG. 7.2: *Algorithme de Givens et caractéristiques du graphe de tâches*

```
param n
real s, f
real a(n,n+1)
real x(n)
for i = 1 to n - 1 do
  for j = i + 1 to n do
    task                    T1(i,j)
      f = a(j,i) / a(i,i)
      for k = i + 1 to n + 1 do
        a(j,k) = a(j,k) - f * a(i,k)
      endfor
    endtask
  endfor
endfor
for i = 1 to n do
  task                      T2(i)
    s = 0.0
    for j = 1 to i - 1 do
      s = s + a(n - i + 1,n - j + 1)
          * x(n - j + 1)
    endfor
    x(n - i + 1) = (a(n - i + 1,n + 1) - s)
                   /a(n - i + 1,n - i + 1)
  endtask
endfor
```

Taille de la matrice	2000	4000
Nombre de tâches $(n^2/2 + n/2)$	2 001 000	8 002 000
Nombre d'arcs $(\frac{3}{2}n^2 - \frac{5}{2}n + 1)$	5 995 001	23 990 001
Nombre d'opérations flottantes$(n^3 + n^2 + 5n)/10^6)$	8 004	64 016

FIG. 7.3: *Élimination de Gauss puis résolution triangulaire et caractéristiques du graphe de tâches*

```
param n
real a(n,n)
real b(n)
assert n >= 3
real inter_1,inter_2
for k = 1 to n do
  for j = k + 1 to n do
    task                    T1(k,j)
      inter_1 = a(k,j) / a(k,k)
      for i1 = 1 to n do
        a(i1,j) = a(i1,j) - inter_1 * a(i1,k)
      endfor
    a(k,j)=inter_1
  endtask
endfor
task                         T2(k)
  inter_2 = b(k) / a(k,k)
    for i3 = 1 to n do
      b(i3) = b(i3) - inter_2 * a(i3,k)
    endfor
  b(k)=inter_2
  endtask
endfor
```

Taille de la matrice	2000	4000
Nombre de tâches $(n^2/2 + n/2)$	2 001 000	8 002 000
Nombre d'arcs $(n^2 - n)$	3 998 000	15 996 000
Nombre d'opérations flottantes $(3 * (n^3/2 + n^2 + n/2)/10^6)$	12 012	96 048

FIG. 7.4: *Diagonalisation de Jordan et caractéristiques du graphe de tâches*

```
param n
real a(n,n)
for k = 1 to n do
  task                          T1(k)
    a(k,k) = sqrt(a(k,k))
  endtask
  for i = k+1 to n do
    task                        T2(k,i)
      a(i,k) = a(i,k) / a(k,k)
      for j = k+1 to i do
        a(i,j) = a(i,j) - a(i,k) * a(j,k)
      endfor
    endtask
  endfor
endfor
```

Taille de la matrice	2000	4000
Nombre de tâches $(n^2/2 + n/2)$	2 001 000	8 002 000
Nombre d'arcs $(n^3/6 + n^2/2 + \frac{2}{3}n)$	1 335 332 000	10 674 664 000
Nombre d'opérations flottantes $(\frac{4}{3}n^3 - n^2/2 + \frac{7}{6}n)/10^6)$	10 664	85 325

FIG. 7.5: *Algorithme de Cholesky et caractéristiques du graphe de tâches*

```
param n
assert n >= 1
real a(n, n),b(n,n),c(n,n)
for i=1 to n do
  for k=1 to n do
    task                    T1(i,k)
      for j=1 to n do
        c(i,j)=c(i,j)+a(i,k)*b(k,j)
      endfor
    endtask
  endfor
endfor
```

Taille de la matrice	2000	4000
Nombre de tâches (n^2)	4 000 000	16 000 000
Nombre d'arcs ($n^2 - n$)	3 998 000	15 996 000
Nombre d'opérations flottantes ($3 * n^3/10^6$)	24 000	192 000

FIG. 7.6: *Multiplication de matrices et caractéristiques du graphe de tâches*

```
param m,n,l
assert m>=2,n>=1,l>=1
real a(n,n),b(n,n),c(n,n),v(n,l),w(n,l)
task                                                          T0()
  for i=1 to n do
    for j=1 to n do
      c(i,j)=a(i,j)
    endfor
  endfor
endtask
for q=2 to m do
  for i=1 to n do
    task                                                      T1(i,q)
      for j=1 to n do
        b(i,j)=0
        for k= 1 to n do
          b(i,j)=b(i,j)+c(i,k)*a(k,j)
        endfor
      endfor
      for j=1 to n do
        c(i,j)=b(i,j)
      endfor
    endtask
  endfor
  task                                                        T2(q)
    for i=1 to n do
      for j=1 to l do
        w(i,j)=0
        for k= 1 to n do
          w(i,j)=w(i,j)+b(i,k)*w(k,j)
        endfor
      endfor
      for j=1 to l do
        v(i,j)=w(i,j)
      endfor
    endfor
  endtask
endfor
```

FIG. 7.7: *Puissance de $m^{ième}$ d'une matrices d'ordre n*

(m,n)		(100,2000)	(200,2000)	(100,4000)	(200,4000)
Nombre de tâches $((m-1)(n+1))$		198 099	398 199	396 099	796 000
Nombre d'arcs $((m-1)(2n+1))$		396 099	796 199	792 099	1 592 199
Nombre d'opérations flottantes $((-n^2+2mn^2$	l=50	2 436 216	4 897 036	19 248 824	38 692 064
$+3mn^3+2mln+3mln^2-3n^3-2ln-3ln^2)/10^6)$	l=1000	3 565 192	7 166 392	23 763 976	47 767 976

FIG. 7.8: *Caractéristiques du graphe de tâches de la puissance de matrice*

Bibliographie

[1] T. Adam, K. M. Chandy et J. R. Dickson. – A comparison of list schedules for parallel processing systems. *Communication of the ACM*, vol. 17, n° 12, 1974, pp. 685–690.

[2] Vikram S. Adve et Mary K. Vernon. – A Deterministic Model for Parallel Program Performance Evaluation. – (Submited for publication).

[3] I. Ahmad, Y.-K. Kwok, M.-Y. Wu et W. Shu. – Automatic Parallelization and Scheduling on Multiprocessors using CASCH. *In : ICPP'97.*

[4] S.P. Amarasinghe, J. M. Anderson, M. S. Lam et C.W. Tseng. – The SUIF Compiler for Scalable Parallel Machines. *In : seventh SIAM Conference on Parallel Processing for Scientific Computing.*

[5] C. Ancourt et F. Irigoin. – Scanning Polyhedra with DO Loops. *In : 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 39–50.

[6] J. M. Anderson et M. S. Lam. – Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *In : ACM SIGPLAN'93 Conference on Programming Language Design and Implementation.*

[7] U. Banerjee. – An Introduction to a Formal Theory of Dependence Analysis. *The Journal of Supercomputing*, vol. 2, n° 2, 1988, pp. 133–149.

[8] D. Barthou. – *Analyse du Flot de Données pour Tableaux en Présence de Contraintes Non-affines.* – Thèse de doctorat, Univesité de Versailles, 1998.

[9] D. Barthou, J.-F. Collard et F. Feautrier. – Fuzzy Array Data Flow Analysis. *Journal of Parallel and Distributed Computing*, vol. 40, n° 2, février 1997, pp. 210 – 226.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, et R. C. Whaley. – *ScaLAPACK Users' Guide.* – SIAM, 1997.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall et Y. Zhou. – Cilk : An efficient multithreaded runtime system. *In : Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95).* – Santa Barbara, California, juillet 1995.

[12] L. Bougé, P. Hatcher, R. Namyst et C. Perez. – Multithreaded Code Generation for a HPF Data-Parallel Compiler. *In : 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98).* – ENST, Paris, France, octobre 1998.

[13] P. Boulet et F. Feautrier. – Scanning Polyhedra without DO-loops. *In : IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pp. 4–11. – Paris, octobre 1998.

[14] Th. Brandes. – *Adaptor*. Disponible sur
`http ://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`.

[15] Th. Brandes et F. Zimmermann. – ADAPTOR - A Transformation Tool for HPF Programs. *In : Programming Environments for Massively Parallel Distributed Systems*, éd. par K.M. Decker, R.M. Rehmann, pp. 91–96. – Birkhäuser, avril 1994.

[16] Z. Chamski. – *Environnement logiciel deprogrammation d'un accélérateur de calcul parallèle.* – Rennes, Thèse de doctorat, Université Rennes I, 1993.

[17] S. Chintor et R. Enbody. – Performance Degradation in Large Wormhole-Routed Interprocessor Communication Networks. *In : ICPP'90*, pp. 424–428.

[18] F. T. Chong, S. D. Sharma, E. A. Brewer et J. Saltz. – Multiprocessor Runtime Support for Fine-Grained Irregular DAGs. *In : Toward Teraflop Computing and New Grand Challenge" Applications.*, éd. par Rajiv K. Kalia et Priya Vashishta. – New York, 1995.

[19] P. Chretienne et C.Picouleau. – *Scheduling Theory and its Applications*, chap. 4, Scheduling with Communication Delays : A Survey, pp. 65–89. – John Wiley and Sons Ltd, 1995.

[20] P. Clauss et V. Loechner. – Parametric Analysis of Polyhedral Iteration Spaces. *Journal of VLSI Signal Processing*, 1998.

[21] "Ph. Clauss, V. Loechner et D. K. Wilde". – Ehrhart. – Disponible sur `http ://icps.u-strasbg.fr/Ehrhart`.

[22] J.-F. Collard, P. Feautrier et T. Risset. – Construction of DO Loops from System of Affine Constraints. *Parallel Processing Letters*, vol. 5, n° 3, septembre 1995, pp. 421–436.

[23] Th. Cormen, C. Leiserson et R. Rivest. – *Introduction à l'algorithmique.* – Dunod, 1994.

[24] M. Cosnard et M. Loi. – Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, vol. 5, n° 4, 1995, pp. 527–538.

[25] M. Cosnard et M. Loi. – A Simple Algorithm for the Generation of Efficient Loop Structures. *Internationnal Journal of Parallel Programming*, vol. 24, n° 3, juin 1996, pp. 265–289.

[26] D. Culler, R. Karp, R. Patterson, A. Sahay, K. E. Shauser, E. Santos, R. Subramonian et T. Von Eiken. – LogP : Toward a realistic model of parallel computation. *In : 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

[27] A. Darte et Y. Robert. – On the Allignment Problem. *Parallel Processing Letters*, vol. 4, n° 3, 1993, pp. 259–270.

[28] E. Deelman, A. Dube, A. Hoisie, Y. Luo, R. Oliver, D. Sunderam-Stukel, H. Wasserman, V.S. Adve, R. Bagrodia, J.C. Browne, E. Houstis, O. Lubeck, J. Rice, P. Teller et M.K. Vernon. – POEMS : End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *In : First International Workshop on Software and Performance.* – Santa Fe, USA, octobre 1998.

[29] M. Dion et Y. Robert. – Mapping Affine Loop Nests : New Results. *In : Int. Conf. on High Performance Computing and Networking, HPCN'95*, pp. 184–189.

[30] J. J. Dongarra, J. Du Croz, I. S. Duff et and S. Hammarling. – A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, vol. 16, 1990, pp. 1 – 17.

[31] H. El-Rewini, T.G. Lewis et H.H. Ali. – *Task Scheduling in Parallel and Distributed Systems.* – Prentice Hall, 1994.

[32] M. Haghighat et C .Polychronopoulos. – Symbolic Analisys : a Basis for Parallelization, Optimization and Scheduling of Program. *In : 6th annual workshop on Programming Languages and Compilers for Parallel Computing.*

[33] P. Feautrier. – Dataflow analysis of array and scalar references. *Internationnal Journal of Parallel Programming*, vol. 20, n° 1, 1991, pp. 23–53.

[34] P. Feautrier. – Toward Automatic Distribution. *Parallel Processing Letters*, vol. 4, n° 3, 1994, pp. 233–244.

[35] P. Feautrier. – Distribution automatique des données et des calculs. *T.S.I.*, vol. 15, n° 5, 1996, pp. 529–557.

[36] I. Foster et C. Kesselman. – The Globus project : A progress report. *In : Heterogeneous Computing Workshop.*

[37] I. Foster, C. Kesselman et S. Tuecke. – The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, vol. 37, 1996, pp. 70–82.

[38] C. Fu et T. Yang. – Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines. *In : ACM/IEEE Supercomputing'96.* – Pittsburgh, novembre 1996.

[39] C. Fu et T. Yang. – Space and time efficient execution of parallel irregular computations. *In : sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'97).* – Las Vegas, juin 1997.

[40] M. Le Fur. – *Compilation de boucles dirigé par la distribution des donnés.* – Thèse de doctorat, Université de Rennes I, juillet 1995.

[41] G. Blelloch and P. Gibbons and Y. Matias. – Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. *In : Symposium on Parallel Algorithms and Architectures*, pp. 1–12.

[42] F. Gallilée, J.-L. Roch, G. Cavalheiro et M. Doreille. – Athapascan-1 : On-line Building Data Flow Graph in a Parallel. *In : IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98).* – Paris, octobre 1998.

[43] A. Geist, A. Berguelin, W. Jiang J. Dongarra, R. Manchek et V. Sunderam. – *PVM3 User's Guide and Reference Manual.* – ORNL, septembre 1994, TM-12187 édition.

[44] A. Gerasoulis, J. Jiao et T. Yang. – Scheduling of Structured and Unstructured Computation . *In : Interconnections Networks and Mappings and Scheduling Parallel Computation* , éd. par D. Hsu, A. Rosenberg et D. Sotteau. pp. 139–172. – American Math. Society.

[45] A. Gerasoulis, S. Venugopal et T. Yang. – Clustering Task Graph for Message Passing Architectures. *In : ACM International Conference on Supercomputing.*

[46] A Gerasoulis et T. Yang. – A Comparaison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. *Journal of Distributed and Parallel Computing*, vol. 16, n° 4, décembre 1992, pp. 276–291. – special issue on scheduling and load balancing.

[47] A. Gerasoulis et T. Yang. – On the Granularity and Clustering of Direct Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n° 6, juin 1993, pp. 686–701.

[48] G. H. Golub et C. F. Van Loan. – *Matrix Computations.* – Johns Hopkins Univeristy Press, 1989, 2nd édition.

[49] W. Gropp, E. Lusk et A. Skjellum. – *Using MPI : Portable Parallel Programming with the Message Passing Interface.* – The MIT Press, 1994. ISBN 0-262-57104-8.

[50] E. Horowitz, S. Sahni et S. Anderson-Freed. – *Fundamentals of data structures in C.* – New-York, W.H. Freeman and company, 1993.

[51] J. Jiao. – *Software Support For Parallel Processing of Irregular and Dynamic Computations.* – New-Jersey, USA, Thèse de doctorat, Rutgers University, 1996.

[52] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steel Jr. et M. E. Zosel. – *The High Performance Fortran Handbook.* – The MIT Press, 1994.

[53] Y.-K. Kwok et I. Ahmad. – Dynamic Critical-Path Scheduling : An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, n° 5, mai 1996, pp. 506–521.

[54] l. G. Valiant. – A Briging Model for Parrallel Computation. *Communication of the ACM*, vol. 33, n° 8, août 1990, pp. 103–111.

[55] C. L. Lawson, R. J. Hanson, D. Kincaid et F. T. Krogh. – Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Soft*, vol. 5, 1979, pp. 308–323.

[56] J.-C. Liou et M. A. Palis. – A New Heuristic for Scheduling Parallel Programs on Multiprocessor. *In : IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pp. 358–365. – Paris, octobre 1998.

[57] M. Loi. – *Construction et exécution de graphe de tâches acycliques à gros grain.* – Thèse de doctorat, Ecole Normale Supérieure de Lyon, France, 1996.

[58] V. Maslov. – Lazy Array Data-Flow Dependences Analysis. *In : 21st Symposium on Principles of Programming Languages.* ACM SIGPLAN, pp. 311–325.

[59] D. E. Maydan, S. P. Amarasinghe et M. S. Lam. – Array Data Flow Analysis And its use in Array Privatization. *In : 20th Annula ACM Symposium on Principles of Programming Languages.*

[60] C. Mongenet. – Affine Dependence Classification for Communications Minimization. *IJPP*, vol. 25, n° 6, décembre 1997.

[61] R. Namyst. – *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* – Thèse de doctorat, Univ. de Lille 1, janvier 1997.

[62] R. Namyst et J.-F. Méhaut. – PM2 : Parallel Multithreaded Machine. A computing environment for distributed architectures. *In : Parallel Computing (ParCo'95).* pp. 279–285. – Elsevier Science Publishers.

[63] R. Namyst et J.-F. Méhaut. – PM2 : Parallel Multithreaded Machine. A computing environment for distributed architectures. *In : Parallel Computing (ParCo'95).* pp. 279–285. – Elsevier Science Publishers.

[64] D. Oppen. – A $2^{2^{2^n}}$ Upper Bound on the Complexity of Presburger Arithmethic. *Journal of Computer ans System Science*, vol. 16, n° 3, juillet 1978, pp. 323–332.

[65] Loïc Prylli et Bernard Tourancheau. – BIP : a New Protocol Designed for High Performance Networking on Myrinet. *In : Parallel and Distributed Processing, IPPS/SPDP'98.* pp. 472–485. – Springer-Verlag.

[66] W. Pugh. – The Omega Test a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, août 1992. – (website : `http ://www.cs.umd.edu/projects/omega`).

[67] W. Pugh. – Counting Solution to Presburger Formulas : How and Why. *In : 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[68] V. Sarkar. – *Partitionning and Scheduling Parallel Program for Execution on Multiprocessors.* – Cambridge MA, MIT Press, 1989.

[69] N. Tawbi. – Estimation of Nested Loop Execution Time by Integer Arithmetics in Convex Polyhedra. *In : 1994 Intl. Parallel Processing Symposium.*

[70] D. K. Wilde. – *A library for doing polyhedral operations.* – Rapport technique nPI-785, IRISA, 1993.

[71] M. Wolfe. – *High Performance Compiler for Parallel Computing.* – Addison-Wesley Publishing Company, 1996.

[72] M. Wu et D. Gajsky. – Hypertool a programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, n° 3, 1990, pp. 330–343.

[73] T. Yang. – *Scheduling and Code Generation for Parallel Architectures.* – Thèse de doctorat, Rutgers State University of New Jersey, 1993.

[74] T. Yang et A. Gerasoulis. – Pyrros : Static Task Scheduling and Code Generation for Message Passing Multiprocessor. *In : Supercomputing'92.* ACM, pp. 428–437. – Washington D.C., juillet 1992.

[75] T. Yang et A. Gerasoulis. – List scheduling with and without communication delay. *Parallel Computing*, vol. 19, 1993, pp. 1321–1344.

[76] T. Yang et A. Gerasoulis. – DSC Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n° 9, septembre 1994, pp. 951–967.

Allocation de graphes de tâches paramétrés et génération de code

Le graphe de tâches (GdT) est un modèle très utilisé pour la prédiction de performance et l'optimisation d'applications parallèles. Il presente cependant deux désavantages. La taille d'un GdT dépend de la valeur des paramètres de l'application qu'il modélise. Un algorithme d'ordonnancement statique prend en entrée un GdT et affecte, à chaque tâche, un processeur et une date de début d'exécution. La durée du calcul de l'ordonnancement ainsi que le coût mémoire dépendent de la taille du graphe et donc de la valeur des paramètres de l'application. Une telle approche n'est pas extensible car pour les grandes valeurs des paramètres le graphe de tâches correspondant peut ne pas tenir en mémoire. Cette méthode n'est pas non plus adaptative car un changement de machine cible ou des paramètres du programme impose de recalculer l'ordonnancement.
Pour apporter une réponse à ces deux problèmes nous avons étudié un modèle intermédiaire : le graphe de tâches paramétré (GTP). Un GTP est une représentation compacte et symbolique des graphes de tâches issus de certaines applications de calcul scientifique. Il utilise les paramètres du programme qui doivent être instanciés pour construire le GdT.
Nos travaux se décomposent en trois volets. (1) Nous avons conçu un algorithme d'ordonnancement du GTP. Le coût mémoire de l'ordonnancement se trouve alors grandement réduit. Cet algorithme est intégré dans un schéma dynamique pour permettre la construction de programmes génériques. (2) Nous présentons une heuristique d'allocation symbolique du GTP appelée SLC. Nous garantissons que cette allocation forme des grappes linéaires. Le temps et le coût mémoire de l'allocation sont alors indépendants de la valeur des paramètres. (3) Nous avons réalisé un prototype de générateur de code qui produit un programme multithreadé se conformant à l'allocation trouvée par SLC. Nous obtenons ainsi un code parallèle portable et générique qui fonctionne pour toutes les valeurs des paramètres du programme.

Mots Clés : Graphes de tâches paramétrés, ordonnancement dynamique, allocation symbolique, ordinateurs parallèles à mémoire distribuée, parallélisme de tâches, génération de code.

Allocation of parameterized task graph and code generation

The task graph model is a widely used model for performance prediction and scheduling of parallel applications. However it presents two major drawbacks. The size of a task graph depends on the parameter values of the application. A scheduling algorithm takes a task graph and assign each task a processor and a starting time. As a result the task graph could become very large and increase the memory and computational requirements. Therefore, scheduling task graphs to multiprocessors is not scalable with the parameters values. Moreover, it is not an adaptive method since the scheduling solution has to be recomputed when target machine or program parameters change.
In this thesis we address these two problems for regular task graphs by using an an intermediate model called parameterized task graph (PTG). A PTG is a compact and symbolic representation of some scientific applications task graphs. It uses parameters that have to be instantiated when building the task graph.
Our contributions are in three areas : (1) We have designed an algorithm for scheduling the PTG. The memory cost of the scheduling is then greatly reduced. We have integrated this algorithm in a dynamic scheme in order to be able to build generic programs. (2) We present a heuristic called SLC for symbolically allocating PTGs. We guarantee that the found allocation is made of linear clusters. The time and memory cost of the allocation is then independent of to the parameter values. (3) A code generator prototype that builds a multi-threaded program which executes the allocation found by SLC has been carried out. Thus, we obtain a generic portable parallel code that works for all the program parameter values.

Keywords : Parameterized task graphs, dynamic scheduling, symbolic allocation, distributed memory parallel computers, task parallelism, code generation.