N° d'ordre : 158
N° attribué par la bibliothèque : 00ENSL0158

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon

spécialité : Informatique

au titre de l'école doctorale Mathématiques et Informatique Fondamentale

par **Fabrice RASTELLO**

PARTITIONNEMENT : OPTIMISATIONS DE COMPILATION ET ALGORITHMIQUE HÉTÉROGÈNE

Présentée et soutenue publiquement le 6 septembre 2000

Après avis de : Madame Jocelyne Erhel (Directrice de recherche, INRIA)
Monsieur François Irigoin (Directeur de recherche, ENSMP Paris)

Devant la commission d'examen formée de :
Monsieur Frédéric Desprez (Chargé de recherche, INRIA, co-directeur de thèse)
Madame Jocelyne Erhel (Directrice de recherche, INRIA)
Monsieur François Irigoin (Directeur de recherche, ENSMP Paris)
Madame Claire Kenyon (Professeur, Université Paris Sud)
Monsieur Yves Robert (Professeur, ENS Lyon, directeur de thèse)
Monsieur Denis Trystram (Professeur, INP Grenoble)

Thèse préparée à l'École Normale Supérieure de Lyon
au sein du Laboratoire de l'Informatique du Parallélisme.

Résumé

Le parallélisme consiste à faire usage de plusieurs ressources simultanément, afin de diminuer le temps d'exécution d'un calcul, ou de résoudre des problèmes de plus grande taille. Mais le temps de communication entre les processeurs ainsi que le déséquilibre de charge, rendent complexe la parallélisation d'un code. Ainsi, afin de réduire la latence et d'augmenter la localité, on va chercher à regrouper certains calculs, c'est à dire à augmenter la granularité. C'est le principe du pavage, technique d'optimisation que nous abordons dans la première partie de cette thèse, dans différents contextes : soit le nid de boucles ne contient que des dépendances internes, et nous cherchons la taille et la forme de tuiles optimales minimisant le chemin critique du graphe des tâches ; soit le nid de boucles ne contient que des dépendances externes, et nous cherchons la forme de tuiles qui optimise la réutilisation des données rapatriées ; nous abordons aussi le problème de pavage pour des tuiles non atomiques dans lesquelles les tâches sont reordonnancées afin de minimiser la latence d'exécution.

Les problèmes liés à la parallélisation se posent de manière encore plus complexe lorsque les ressources sont hétérogènes. Nous abordons ainsi dans une seconde partie le problème sous une approche plus algorithmique, visant la constitution de librairies de calculs linéaires sur ressources hétérogènes. Cela pose un certain nombre de problèmes d'optimisation géométrique, souvent montrés comme étant NP-complets, et pour lesquels nous proposons un certain nombre de solutions heuristiques avec garanties.

Mots clés

parallélisation automatique, pavage, ressources hétérogènes, algorithmique, NP-complet, problèmes d'optimisation, graphe des tâches, chemin critique, tuiles, bande passante, mémoire cache, pavage hiérarchique, circuit VLSI, ordonnancement, calcul redondant, chaîne d'oscillateurs, relaxation, modèle de communication, décomposition LU, décomposition QR, produit de matrices, algèbre linéaire, partitionnement.

Abstract

The goal of parallelization is to use several resources simultaneously, so as either to execute a given program faster, or to solve a larger problem. But communication overhead and load imbalance render the problem complicated. In order to increase both the locality of data references and the granularity of the computation, we aggregate neighboring tasks together inside tiles. This optimization technique is called loop tiling, and is dealt with in the first part of this thesis in three different contexts : (i) when the nested loop contains only internal dependences, we look for optimal tile size and shape that minimize the critical path length of the iteration space task graph ; (ii) when the nested loop contains only external dependences, we look for a tile shape that optimizes the reuse of loaded data ; (iii) we consider also tiling when removing the constraint of atomicity, and we look for a reordering of the execution of tasks within a tile, on which the latency is critically dependent.

The difficulties of parallelization become even tougher when targeting heterogeneous computing platforms. The second part of this thesis is devoted to providing the required framework to build an extension of the ScaLAPACK library (linear algebra kernels), capable of running on top of heterogeneous networks of workstations or non-dedicated parallel machines. We investigate several algorithmic issues, we state several NP-complete results (related to some geometrical optimization problems), and we propose some guaranteed heuristics.

Keywords

automatic parallelization, tiling, loop blocking, heterogeneous computing platforms, algorithmic, NP-complete, optimization problems, tasks graph, tiles, bandwidth, cache memory, hierarchical tiling, VLSI chip, scheduling, redundant tasks, dynamical systems, solitary waves, relaxation, communication models, LU decomposition, QR decomposition, Matrix multiplication, linear algebra, partitioning.

Table des matières

Chapitre 1

Introduction

Le parallélisme consiste à faire usage de plusieurs ressources *simultanément* afin de diminuer le temps d'exécution d'un calcul, ou de résoudre des problèmes de plus grande taille [94]. Pour certains codes, la parallélisation peut être triviale. Par exemple, il n'est pas rare qu'un utilisateur veuille exécuter plusieurs fois un même programme (notons le f), en faisant simplement varier les paramètres d'entrée (notons les i). Dans ce cas, il lui suffirait de lancer indépendamment son programme sur les différentes machines puis de rapatrier les résultats à la fin de chacune des exécutions. Si le temps d'exécution du programme f sur chaque processeur est suffisamment important, on dira que le calcul est à *gros grain*, et on aura globalement le schéma de la Figure 1.1. Notons que dans

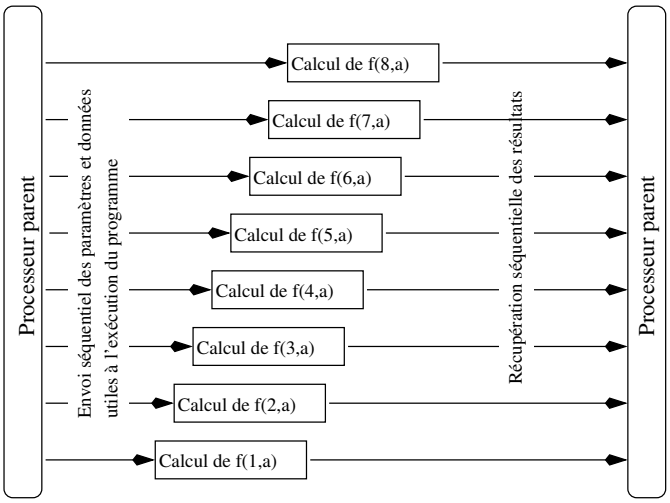


FIG. 1.1: *Exécution parallèle de* $f(1..8, a)$ *sur 8 processeurs.*

ce schéma, l'exécution du programme ne débute *pas* au même moment sur chaque processeur. Cela est dû au fait que chaque processeur P_i doit attendre les données utiles (i et a) au calcul de $f(i, a)$ provenant du processeur parent. Or la *communication* de ces données n'est ni immédiate, ni simultanée. Ainsi, si la granularité du calcul est trop petite (temps de communication comparable ou supérieur au temps d'exécution de f), cette exécution parallèle ne sera pas rentable, car le processeur "parent" va passer plus de temps à communiquer des données qu'il ne lui en aurait fallu pour qu'il exécute lui même l'ensemble des calculs.

Cet exemple, a priori trivial, met en valeur l'un des plus importants obstacles à la parallélisation, le temps de communication entre les processeurs (le déséquilibre de charge en est un autre). Ainsi,

afin de réduire la latence et d'augmenter la localité, on va chercher à regrouper certains calculs, c'est à dire à augmenter la granularité de calcul. Dans notre exemple, les données, représentées par a, nécessaires à l'exécution de f sont communes aux différentes instances d'exécution, et l'adéquation entre l'utilisation des ressources de calcul et la diminution de l'impact des communications est simple : il suffit de minimiser la fonction objective

$$\left[\frac{\text{Temps de calcul de}f}{\text{Nombre de processeurs}} + \text{Temps de communication} \times \text{Nombre de processeurs}\right]$$

Lorsque les différentes instances ne sont pas indépendantes, c'est à dire qu'il y a nécessité du résultat d'un calcul pour débuter un autre calcul et que les données nécessaires à chaque instance ne sont pas identiques, le regroupement n'est pas aussi trivial. Il faut chercher à maximiser le *réutilisation des données communes*, sans pour autant supprimer le parallélisme. Les dépendances entre tâches induisent une véritable notion de localité. Un des principes fondamentaux de *la technique de pavage* est de chercher à favoriser le plus possible cette localité. L'étude de cette technique d'*optimisation de compilation* fait l'objet d'une part importante de ce mémoire de thèse. Le pavage s'applique, comme toute technique de parallélisation automatique, aux nids de boucles constitués de plusieurs boucles "do" imbriquées les unes dans les autres. Elle s'applique en particulier aux nids de boucles dits parfaits et aux dépendances uniformes [11]. Considérons l'exemple[1] suivant :

Algorithme 1. *Exemple de nid de boucles parfait contenant des dépendances internes et externes.* $l > 0$

do $t = 1, W$
  doall $i = 1, H$
    $S_{t,i} : A[i][t] = \frac{1}{3}(A[i-1][t-1] + A[i][t-1] + A[i+1][t-1]) + A[i][t-l] \times \frac{B[t,i]}{l}$

Dans ce code, les références $A[i-1][t-1]$, $A[i][t-1]$ et $A[i+1][t-1]$ mettent en jeu des dépendances *internes*, c'est à dire que le résultat de tâches internes au nid de boucles est utile au calcul d'autres tâches elles-mêmes internes au nid de boucle. Ces dépendances ont un impact important sur le déroulement de l'exécution, son ordonnancement et sa latence. Le Chapitre 2 est consacré à la recherche d'un pavage minimisant la *longueur du chemin critique* du graphe des tâches.

La référence $B[t,i]$ induit une dépendance *externe*, et les données correspondantes peuvent être rapatriées *avant* le début de l'exécution du nid de boucles. Le Chapitre 3 est consacré au pavage de nids de boucles ne présentant que ce type de dépendances. En d'autres termes, tous les calculs sont indépendants, mais nécessitent le rapatriement d'un certain nombre de données. Nous cherchons alors à paver cet espace afin d'optimiser le recouvrement des données rapatriées, c'est à dire minimiser la *bande passante* minimum nécessaire à l'exécution distribuée d'un tel nid de boucles.

Finalement, la référence $A[i][t-l]$ induit une dépendance *variable*, c'est à dire inconnue au moment de l'allocation des données. Dans ce cas partitulier, le pavage initial associé à de telles dépendances pouvant supprimer le parallélisme, il faut soit redistribuer les données, soit réordonnancer les tâches à l'intérieur de chaque groupement (tuile). La recherche d'un tel ordonnancement fait

[1] Dans cette thèse, doall représente une boucle ne contenant aucune dépendance interne, c'est à dire dont l'ordre d'exécution est quelconque. dovect représente une boucle parallèle contenant des dépendances, c'est à dire une boucle dont l'exécution séquentielle nécessite l'utilisation de variables temporaires. Par exemple, la transposition d'une matrice dovect i,j {A[i,j]=A[j,i]}.

l'objet du Chapitre 4. Le but est d'exprimer le parallélisme interne aux tuiles. Cela implique de relâcher la contrainte classique d'atomicité des tuiles, c'est à dire dans notre cas, d'effectuer *des* communications *durant* l'exécution d'une tuile. Les communications doivent donc être suffisamment rapides, ou les tuiles suffisamment grosses, afin de pouvoir assurer une granularité de calcul suffisante. Cela est possible en particulier sur des systèmes de type VLSI, ainsi que sur des machines distribuées dotées d'une mémoire cache importante.

Classiquement, lorsque les dépendances de données sont connues, les techniques de pavage cherchent à partitionner l'espace d'itérations en tuiles *tordues* afin de mettre en valeur le parallélisme, et de minimiser la longueur du chemin critique. Le Chapitre 5 propose une approche différente mettant en jeu des *calculs redondants* : nous appliquons cette technique à un code réel modélisant la chaîne d'oscillateurs du modèle Fermi-Pasta-Ulam. Ayant besoin de très bonnes performances d'exécution de ce code (voué à tourner plusieurs mois sur un système de plus de 20 processeurs), nous avons cherché les meilleurs paramètres et fourni un certain nombre de mesures de performances. Cela fut l'occasion d'illustrer, sur un exemple concret, la technique de pavage ainsi que les modèles classiquement utilisés.

En général, paralléliser un nid de boucle est un problème difficile pour lequel on utilise des modèles de machine simplifiés. Les approches utilisées sont heuristiques et choisissent un schéma d'investigation par étapes. Ces différentes étapes sont l'analyse de dépendances (mise en évidence des parties indépendantes, et d'un ordre de précédence obligatoire), l'ordonnancement et l'allocation des tâches ainsi que le partitionnement (regroupement des tâches), etc. Chacune de ces étapes mettant en jeux un problème généralement NP-complet (soluble en un temps pouvant être exponentiel en la taille du problème), des approches heuristiques sont proposées, et la classe de problèmes résolus est réduite. Une fois l'espace d'itération pavé, la régularité des codes traités dans les quatre premiers chapitres de cette thèse rend l'allocation et l'ordonnancement de tuiles *relativement* simple. Cette régularité est brisée lorsque l'on souhaite exécuter un tel programme sur un ensemble hétérogène de processeurs. Dans un premier temps (Chapitre 6), afin d'illustrer les problématiques liées à l'équilibrage de charge, nous abordons le cas simple d'allocation unidimensionnelle des données : nous traitons le cas d'équilibrage de charges indépendantes, que nous appliquons à l'algorithme de multiplication de matrices. Puis, au travers de l'algorithme de décomposition LU ainsi que de l'algorithme de relaxation traité dans le Chapitre 2, nous montrons les problèmes liés à la présence de dépendances de données. Nous proposons à cet effet, deux stratégies différentes adaptées à chacun des algorithmes.

L'approche pragmatique du parallélisme est la constitution de librairies de noyaux de calculs distribués : le Chapitre 7 est consacré à l'extension de la librairie d'algèbre linéaire ScaLAPACK [15] sur ressources hétérogènes. Pour être efficace une allocation des données se doit d'être bidimensionnelle, et le problème d'optimisation associé est NP-complet. Nous nous restreignons dans ce chapitre à une allocation en *grille parfaite*, et proposons une solution heuristique.

Finalement, nous abordons, dans le Chapitre 8, le cas d'une allocation libre des données. L'équilibrage pouvant alors être parfaitement effectué, nous nous proposons de minimiser le coût des communications engendrées par une telle allocation. En fonction des contraintes imposées, liées au modèle de ressources utilisé, cela pose un certain nombre de problèmes d'optimisation géométrique. Ce dernier chapitre est consacré aux preuves de NP-Complétude de ces problèmes, ainsi qu'au développement de solutions heuristiques.

Chapitre 2

Pavage de nids de boucles avec dépendances internes

Ce chapitre est consacré à la recherche de formes optimales de tuiles dans le cadre de nids de boucles avec dépendances internes. Ce travail est motivé par un article de Högsted, Carter et Ferrante [55] dont le but est de formuler analytiquement le temps total d'inactivité des processeurs pour un pavage donné. Nous proposons dans ce chapitre une approche du problème fondée sur l'évaluation de la longueur du chemin critique du graphe des tâches, qui nous permet de fournir une solution plus précise et plus exhaustive au problème soulevé par Högsted et al.. Ce chapitre est aussi voué à introduire la notion de pavage ainsi que quelques problématiques liées à cette technique. C'est ce qui constitue les paragraphes 2.1, 2.2 et 2.3. Ensuite, nous présentons le problème traité par Högsted et al., dans le paragraphe 2.3. Les paragraphes suivants sont consacrés à la présentation de nos résultats, à leurs preuves ainsi qu'à la conclusion de ce chapitre.

2.1 La technique de pavage

2.1.1 Définition et exemple

Le pavage[1] est une technique de compilation largement répandue pour augmenter la granularité des calculs et la localité des références. Cette technique a été initialement limitée aux nids de boucles parfaits avec des dépendances uniformes (comme défini par Banerjee [11]), mais fut étendue par la suite aux boucles totalement permutables [57, 36, 71, 92]. L'idée fondamentale du pavage est de regrouper les points élémentaires de calcul en tuiles considérées alors comme nouvelles unités de calcul. Plus les tuiles sont larges, plus l'exécution des tâches peut être optimisée, en tenant compte des spécificités de chaque processeur, telles que les unités de calculs pipelinées ou la hiérarchie de la mémoire (cela peut être illustré par l'utilisation des BLAS 3 dans les librairies d'algèbre linéaire pour systèmes distribués [27, 42]).

Un autre avantage du pavage est la diminution du temps de communication, qui est proportionnel à la surface de la tuile, relativement au temps de calcul, qui est proportionnel au volume de la tuile[2]. Le prix à payer peut être un temps de latence accru (s'il y a des dépendances de données, par exemple, nous devons attendre que le premier processeur ait terminé l'exécution entière de la première tuile avant qu'un autre processeur puisse commencer l'exécution de la seconde, et ainsi

[1] *tiling* en anglais

[2] Lorsque l'espace d'itération est de dimension 2, le vocabulaire est différent : on parle de *surface* pour le calcul, et de *périmètre* pour les communications.

de suite), aussi bien que quelques problèmes d'équilibrage de charge (plus les tuiles sont grandes, plus il est difficile de distribuer les calculs équitablement parmi les processeurs).

Afin de fixer les idées, considérons l'exemple suivant :

Programme 1.

$$\begin{array}{l} \mathsf{do}\ t = 1, W \\ \quad \mathsf{do}\ i = 1, H \\ \quad\quad A[i][t] = f(A[i-1][t], A[i][t-1]) \end{array}$$

Il s'agit d'un nid de boucles fréquent. Il est utilisé, par exemple, dans la méthode de relaxation pour la résolution d'équations aux dérivées partielles ou dans des schémas explicites pour la résolution d'équations différentielles(cf [48]). La version pavée de ce code, à l'aide de rectangles de taille $w \times h$, est la suivante :

Programme 2.

$$\begin{array}{l} \mathsf{do}\ T = 1, W : w \\ \quad \mathsf{do}\ I = 1, H : h \\ \quad\quad \mathsf{do}\ t = T, \min(W, T + w - 1) \\ \quad\quad\quad \mathsf{do}\ i = I, \min(H, I + h - 1) \\ \quad\quad\quad\quad \theta_{t,i} : A[i][t] = f(A[i-1][t], A[i][t-1]) \end{array}$$

Une tuile correspond alors au calcul des deux boucles internes (t, i). Notons que dans la littérature, un pavage vérifie classiquement les contraintes suivantes :

les tuiles sont bornées. Pour des raisons de scalabilité, on souhaite que le nombre de points internes à une tuile soit borné par une constante indépendante de la taille du domaine.

les tuiles sont identiques par translation. Deux tuiles qui n'intersectent pas la bordure doivent être identiques à une translation près. Cette contrainte est imposée pour la génération de code.

les tuiles sont atomiques. Une tuile est vue comme une unité de calcul. Chaque point de synchronisation est situé au début ou à la fin d'une tuile. L'ordonnancement à l'intérieur d'une tuile, et l'ordonnancement des tuiles, doivent correspondre globalement à un ordonnancement valide de l'espace d'itérations.

2.1.2 Intérêts du pavage

Le pavage est une technique de compilation largement répandue pour augmenter la localité des références et la granularité des calculs. C'est ce que nous expliquons dans ce paragraphe.

2.1.2.1 Localité des données

La structure de la mémoire d'une machine monoprocesseur est hiérarchique (se référer à l'article de Carter et al. [25]) : registres, mémoire(s) cache (il peut y avoir un, deux ou trois niveaux de cache), mémoire vive, disque dur... La transparence fonctionnelle au niveau utilisateur est assurée à l'aide de techniques de pagination [88] dont il existe plusieurs stratégies, plus ou moins efficaces selon les configurations. Dans tous les cas, le temps d'accès à une donnée, lié au niveau de mémoire dans laquelle elle se trouve alors, est d'autant plus faible que la date de sa dernière utilisation est

récente. Ainsi, dans la version pavée du Programme 1, la donnée `A[i][t]` est utilisée par les tâches $\theta_{t,i}$, $\theta_{t+1,i}$ et $\theta_{t,i+1}$. Si ces 3 tâches sont situées dans la même tuile, le nombre d'itérations entre 2 accès à la donnée `A[t][i]`, correspond au nombre d'itérations entre $\theta_{t,i}$ et $\theta_{t+1,i}$, c'est à dire h. Le bon compromis est celui qui prend une hauteur de tuile (h) suffisamment grande pour assurer le maximum d'accès locaux, mais pas trop pour ne pas briser la localité au sein même de la tuile. La Figure 2.1, issue d'expériences exposées dans le Chapitre 5, illustre la nécessité de favoriser la localité des accès aux données. Dans cette figure[3], le nid de boucle étudié, est assez similaire à celui du Programme 1 : il contient deux boucles, une en temps (t) et une en espace (i). On fait varier la taille de la boucle interne en espace (H) et on compare les performances entre une version pavée ($h \times w = 100 \times 100$) et une version non pavée du code. L'expérience effectuée sur un Pentium-Pro, met en évidence deux niveaux de cache, correspondant aux deux paliers de la courbe du programme non pavé.

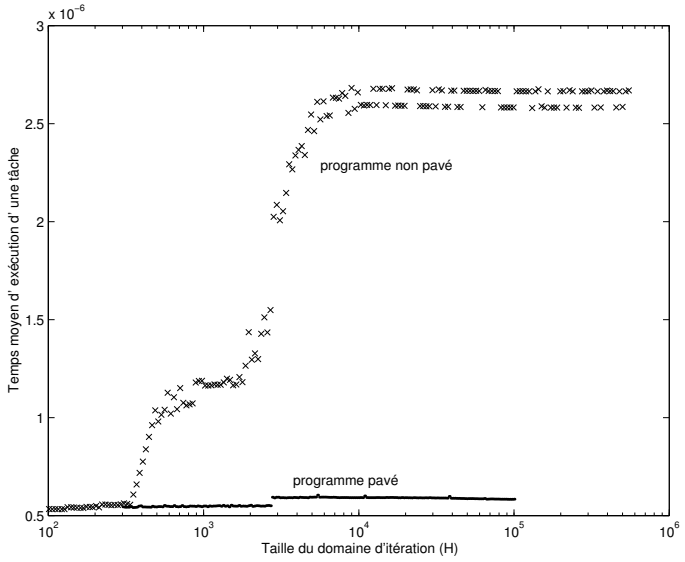


FIG. 2.1: *Courbes extraites de mesures de performance présentées au Chapitre 5. Mise en évidence de deux niveaux de cache, et illustration de la nécessité du pavage.*

Dans le cas où le nid de boucles est distribué (exécution en parallèle), le pavage consiste à regrouper des tâches voisines sur un même processeur. Le terme *localité* prend alors tout son sens : les données accédées par un processeur pour le calcul de sa tuile sont majoritairement situées sur ce processeur, les autres étant situées sur des processeurs voisins.

2.1.2.2 Granularité du calcul

Un autre aspect intéressant du pavage est l'augmentation de la granularité du calcul : il est généralement plus coûteux de faire n communications de taille l qu'une seule communication de taille nl. Ainsi, on a intérêt, dans une certaine mesure, à regrouper les communications. Bien sûr, une taille de tuile trop importante séquentialise le code (latence trop importante), alors qu'une trop petite taille de tuile étouffe le parallélisme par la présence des trop nombreuses communications.

[3]On appelle *temps d'exécution moyen d'un calcul élémentaire* le temps total d'exécution normalisé par le nombre d'itérations, c'est à dire $\tau_{calc} = \frac{T_{exe}}{WH}$.

Considérons l'exemple précédant pour illustrer le compromis ; une parallélisation possible de cette boucle est d'utiliser la méthode du "front" : toutes les tuiles d'une même colonne sont affectées à un même processeur ; les colonnes sont projetées de manière cyclique sur la ligne de processeurs ; chaque colonne est exécutée du bas vers le haut. Cette méthode est illustrée par la Figure 2.2.

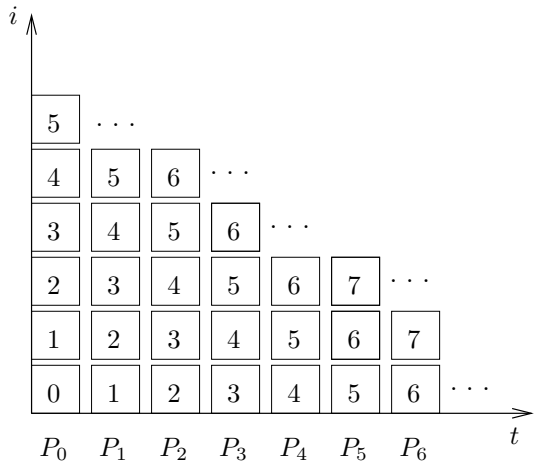


FIG. 2.2: *La méthode du* **front** *: une méthode possible pour ordonner l'exécution des tuiles. L'ordre d'exécution est ainsi du bas à gauche vers le haut à droite. Les tuiles situées sur la même diagonale sont exécutées en parallèle.*

On utilise classiquement comme modèle de communications la formule $\tau_{comm} = \beta + l\tau$: le temps d'une communication de longueur l, correspond au temps d'initialisation du processus de communication (β), suivi du temps de transit des données ($l\tau$). β est souvent beaucoup plus grand que τ. Prenons par exemple $\tau = 1$, $\beta = 100$ et $\tau_{calc} = 1$ (temps de calcul élémentaire). Faisons abstraction de l'impact de la hiérarchie de la mémoire sur τ_{calc} et calculons le temps total d'exécution en fonction de h et de w (on suppose le nombre de processeurs infini) : on obtient $T_{exe} \approx (100+h\tau+hw) \times (\frac{H}{h} + \frac{W}{w})$. Cela correspond au chemin critique (cf Paragraphe 2.5.1 pour une définition du chemin critique) longeant les 2 bords de l'espace d'itérations. Augmenter h ou w diminue localement l'impact des communications. En contrepartie, l'augmentation de w (respectivement h) rajoute une latence au chemin vertical (respectivement horizontal).

En résumé, le pavage permet de réduire le sur-coût de communication, en diminuant le ratio *temps de communications* (approximativement proportionnel au périmètre de la tuile[4]) sur *temps de calculs* (approximativement proportionnel à la surface de la tuile[5]). D'un autre point de vue, le pavage correspond à une augmentation de la granularité, à un moyen d'augmenter la localité des références.
Il faut bien sûr voir que d'une part cette méthode est réduite aux boucles totalement permutables, et que d'autres parts, l'augmentation de granularité implique une latence plus importante (c'est à dire qu'il est plus long d'atteindre l'état de parallélisme total où tous les processeurs travaillent).

[4]nommé surface en dimension supérieure à deux.
[5]nommé volume en dimension supérieure à deux.

2.2 Principales questions en matière de pavage

La technique de pavage a été étudiée par différents chercheurs dans plusieurs contextes [1, 7, 19, 24, 23, 26, 55, 57, 75, 78, 82, 85, 87, 91][6]. La majeure partie des travaux s'attache à partitionner l'espace d'itérations d'un nid de boucles uniforme en tuiles dont la forme et la taille sont optimisées selon certains critères (tel que le rapport *temps de calcul/temps de communication*). Une fois que la forme et la taille des tuiles sont définies, il reste à distribuer les tuiles aux processeurs physiques et à trouver l'ordonnancement des tuiles. Plus précisément :

- Boulet et al. [19] et Calland et al. [24] cherchent la forme de tuile qui minimise le volume de communication par tuile.
- Un problème assez similaire consiste à chercher à minimiser la quantité de données *utilisées* par le calcul d'une tuile. En effet, on peut vouloir, par exemple, que toutes les données tiennent dans la mémoire locale ou dans le cache. Ce problème introduit par Agarwal et al. [1] fait l'objet du Chapitre 3 de cette thèse.
- De manière plus globale, de nombreux travaux [6, 23, 54, 78] cherchent à optimiser la taille et/ou le placement des tuiles pour un nombre de ressources fini. Ces travaux sont restreints à l'étude d'espaces d'itérations et de tuiles rectangulaires.
- Dans cette partie, nous nous appuyons sur les travaux de Högsted, Carter et Ferrante [55]. Pour un domaine d'itérations déjà pavé, ils cherchent à déterminer la somme des temps d'inactivité de tous les processeurs. Cette somme de temps d'inactivité dépend fortement des formes respectives du domaine et des tuiles.
  Högsted, Carter, et Ferrante introduisent un nouveau paramètre, nommé *pente*[7], qui caractérise le rapport entre la forme des tuiles et la forme du domaine d'itérations. Ce paramètre a un impact significatif sur le temps d'inactivité des processeurs. Des domaines d'itérations de forme parallélogramme mais aussi de forme trapézoïdale sont considérés.

Ainsi, après avoir présenté dans le Paragraphe 2.3 un exemple motivant l'étude d'espaces d'itérations non nécessairement rectangulaires, nous résumons dans le Paragraphe 2.4 les résultats de Högsted et al.. Ensuite, nous présentons un modèle légèrement différent, qui nous permet de proposer de nouvelles et beaucoup plus courtes, preuves de ces résultats, et nous les étendons dans plusieurs directions importantes. Plus précisément, nous fournissons une solution précise pour toutes les valeurs de *pente* et pour toutes les distributions possibles de tuiles aux processeurs, alors que Högsted et al. traitent seulement un nombre limité de cas et fournissent les limites supérieures plutôt que les formules exactes. Ces résultats sont présentés dans le Paragraphe 2.5. Le Paragraphe 2.6 est consacré à l'utilisation de ces résultats. En particulier, nous les appliquons au problème de pavage hiérarchique [25, 77].

2.3 Présentation du problème de Fermi-Pasta et Ulam (FPU)

Dans ce paragraphe, nous traitons un exemple de manière informelle, du nid de boucles initial jusqu'à l'espace d'itérations pavé. Nous ne fournissons aucune justification théorique pour les transformations successives du code (se référer à [24, 23, 57, 82, 85]). Nous expliquons plutôt chacune d'elles avec des arguments intuitifs.

La Figure 2.3 illustre le modèle de Fermi, Pasta et Ulam pour un cristal unidimensionnel [84]. Notons par $x_i(t)$ la position de la molécule i au temps t. Ce système dynamique est discret en

[6] Cette courte liste est loin d'être exhaustive.
[7] *rise* dans le texte en anglais.

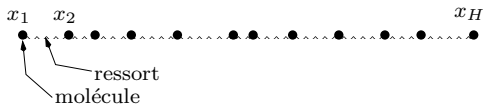


FIG. 2.3: *Chaîne d'oscillateurs de longueur H. Dans le modèle de Fermi, Pasta et Ulam (FPU), le ressort est considéré comme non parfait, et on rajoute un terme perturbatif dans l'expression de la force de rappel. Par conséquent, le principe fondamental de la dynamique s'exprime $m\ddot{x}_i = k(x_{i+1} - 2x_i + x_{i-1}) + K((x_{i+1} - x_i)^3 + (x_{i-1} - x_i)^3)$ où K est petit devant k.*

espace mais continu en temps. Un schéma[8] en temps explicite possible s'exprimerait comme suit :

$$\begin{aligned} x_i(t) &= 2x_i(t-1) + x_i(t-2) + \frac{(dt)^2}{m} f(x_{i-1}(t-1), x_i(t-1), x_{i+1}(t-1)) \\ &= g(x_{i-1}(t-1), x_i(t-1), x_{i+1}(t-1), x_i(t-2)) \end{aligned}$$

Le noyau principal d'un programme correspondant à ce schéma est donc le suivant :

Programme 3. *Evolution d'un système dynamique unidimensionnel*

```
do t = 1, W
    do i = 1, H
        X[i][t] = g(X[i − 1][t − 1], X[i][t − 1], X[i + 1][t − 1], X[i][t − 2])
```

Dans ce noyau,
- $X[i][t]$ correspond à l'orbite de la molécule i au temps t.
- Un traitement particulier effectué sur les bornes, n'est pas représenté dans ce code.
- Normalement le tableau X est unidimensionnel, et la variable t est masquée. Pour plus de clarté, nous l'avons laissé apparente ici.

Les vecteurs de dépendance de cette boucle peuvent être résumés par l'ensemble

$$\left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right\}.$$

La Figure 2.4 donne une représentation schématique du domaine (ou espace) d'itérations

$$\mathcal{I} = \{(t, i) \in \mathbb{N}^2, 1 \leq t \leq W, 1 \leq i \leq H\},$$

avec les vecteurs de dépendance.

Du fait des vecteurs de dépendance $(1, 1)^t$ et $(1, -1)^t$, un pavage rectangulaire introduirait des dépendances cycliques entre les tuiles. Ainsi, l'espace d'itérations doit-t-il être *tordu*[9] avant d'être pavé rectangulairement. La Figure 2.5 représente l'espace d'itérations après *torsion*. Le nouveau nid de boucles s'écrit (on pose $i' = i + t$) :

Programme 4. *Programme 3 une fois tordu*

```
do t = 1, W
    do i' = 1 + t, H + t
        X[i' − t][t] = g(X[i' − (t − 1) − 2][t − 1], X[i' − (t − 1) − 1][t − 1],
                         X[i' − (t − 1)][t − 1], X[i' − (t − 2) − 2][t − 2])
```

[8] Ce schéma en temps, très instable numériquement, n'est pas celui décrit dans le Chapitre 5.
[9] *skewed* en anglais

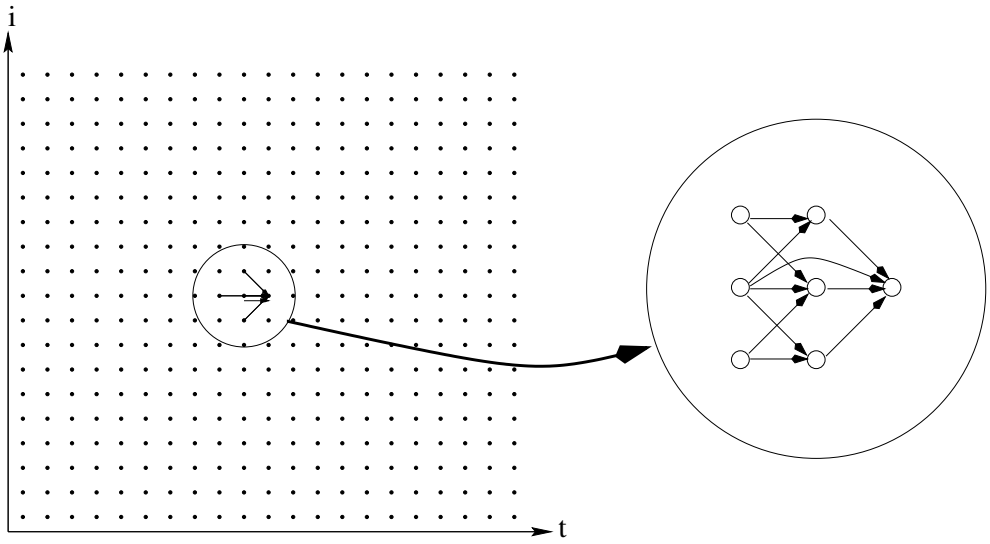


FIG. 2.4: *Espace d'itérations du code FPU et vecteurs de dépendance associés.*

Les vecteurs de dépendance deviennent

$$\left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \end{pmatrix} \right\}.$$

Maintenant, toutes les composantes des vecteurs de dépendance étant positives, les deux boucles deviennent permutables, et l'espace d'itérations peut être pavé, comme schématisé dans la Figure 2.5.

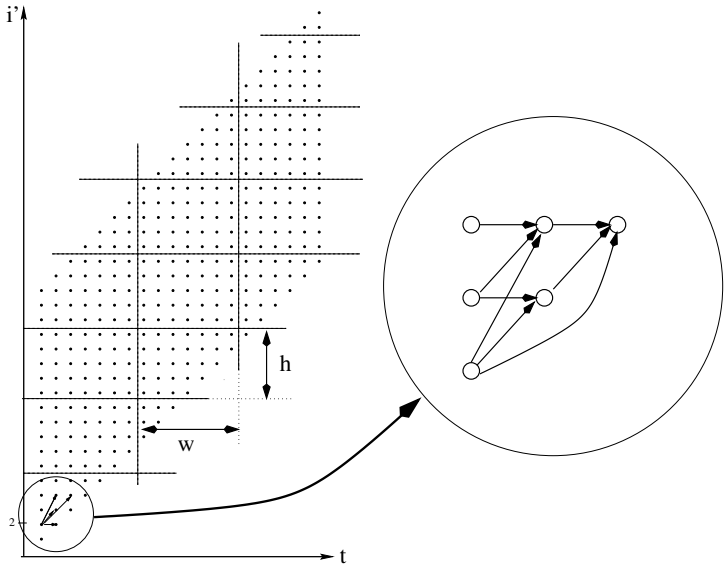


FIG. 2.5: *Le domaine d'itérations* tordu.

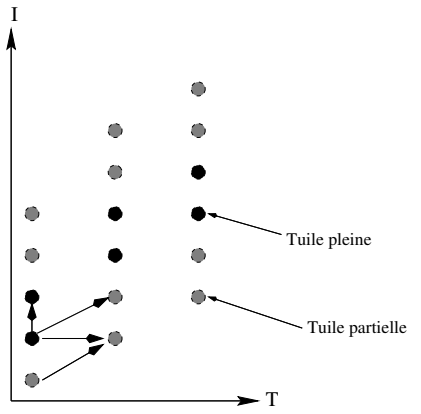


FIG. 2.6: *L'espace d'itérations pavé.*

En changeant ensuite de granularité, on introduit un nouvel espace d'itérations représenté Figure 2.6. Dans cette figure, chaque cercle représente une tuile au lieu d'un nœud élémentaire de

calcul. Supposons que les tuiles sont de taille $w \times h$, alors la version pavée du nid de boucles s'écrit :

Programme 5. *Programme 4 une fois pavé*

$$
\begin{array}{l}
\mathsf{do}\ T = 1, W : w \\
\quad \mathsf{do}\ I = T, T + H - 1 : h \\
\quad\quad \mathsf{do}\ t = T, \min(T + w - 1, W) \\
\quad\quad\quad \mathsf{do}\ i = \max(I - t, 1), \min(I - t + h - 1, H) \\
\quad\quad\quad\quad X[i][t] = g(X[i-1][t-1], X[i][t-1], X[i+1][t-1], X[i][t-2])
\end{array}
$$

Les vecteurs de dépendance entre les tuiles de taille $w \times h \geq 2 \times 2$ sont représentés dans la Figure 2.6. Ils correspondent à l'ensemble

$$\left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}.$$

En d'autres termes, l'exécution d'une tuile ne peut pas débuter tant que la tuile de gauche et la tuile du dessous n'ont pas été terminées : la dépendance diagonale est donc redondante. Du reste, les communications selon cette direction peuvent être routées[10], soit horizontalement puis verticalement, soit l'inverse. Elles peuvent être combinées avec les autres communications (induites par les vecteurs de dépendance $(0,1)^t$ et $(1,0)^t$).

Ainsi, on aboutit à un espace d'itérations pavé, de forme parallélogramme, et de vecteurs de dépendance $(0,1)^t$ et $(1,0)^t$. Cela s'avère être un cas de configuration très général, et motive ainsi le travail de Högsted, Carter, et Ferrante [55] : étant donné un tel domaine d'itérations pavé, comment allouer les tuiles aux processeurs, et ainsi calculer dans cette configuration le temps total d'exécution ?

2.4 Présentation du problème ; article de Högsted et al.

Ce paragraphe est consacré à la présentation des résultats de Högsted, Carter, et Ferrante [55], dont les hypothèses sont les suivantes :

(H1) On dispose de P processeurs connectés en anneau. Les processeurs sont numérotés de 0 à $(P-1)$.

(H2) Les tuiles sont de forme parallélogramme avec les cotés gauche et droit verticaux. La forme et la taille des tuiles sont des données du problème (cf Figure 2.7).

(H3) L'étude porte sur des nids de boucles bidimensionnels correspondant à un espace d'itération de forme soit parallélogramme soit trapézoidale, dont les frontières de gauche et de droite sont verticales. M est le nombre de tuiles de la première colonne.

(H4) Les tuiles sont projetées sur l'espace des processeurs avec une distribution bloc ou bloc-cyclique.

- Pour une distribution bloc-cyclique, on a bP colonnes *de tuiles* numérotées de 0 à $(bP-1)$. Toutes les tuiles de la colonne j, $0 \leq j \leq (bP-1)$ sont projetées sur le processeur j **modulo** P.
- Une distribution bloc est un cas particulier de distribution bloc-cyclique dans lequel $b = 1$.

(H5) La *pente* (r) est un paramètre qui reflète l'angle entre l'inclinaison de la frontière horizontale d'une tuile et la frontière horizontale de l'espace d'itérations. Plus précisément on définit

[10] *routed* en anglais

r comme étant la tangente de cet angle (cf Figure 2.8).
Pour un espace d'itérations trapézoïdal, on distingue la pente inférieure (r_b) correspondant à l'angle fait avec la borne inférieure de l'espace d'itérations, et la pente supérieure (r_t) correspondant à l'angle fait avec la borne supérieure de l'espace d'itérations.

(H6) Chaque tuile dépend de deux tuiles voisines situées respectivement au dessous et à gauche. Ainsi, les dépendances entre les tuiles peuvent être résumées par la paire de vecteurs suivante :

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$

(H7) L'ordre d'exécution des tuiles s'effectue par colonne : chaque processeur positionné sur une colonne exécute les tuiles de cette colonne, du bas vers le haut, et dès que possible, après avoir exécuté une colonne de tuiles, le processeur se déplace sur sa colonne suivante ($col + P$). Le temps nécessaire à l'exécution d'une tuile pleine est T_{calc}. Remarquons que la partie non recouverte de la communication horizontale est incluse dans T_{calc}. Si l'on suppose les communications totalement recouvertes par les calculs, avec les notations de Högsted et al., $T_{calc} = hw\tau_{calc}$, où h et w sont respectivement la hauteur et la largeur d'une tuile et τ_{calc} le temps de calcul d'une tâche élémentaire. On définit le sur-coût de communication c, tel que la partie recouverte de la communication des données d'une tuile à sa voisine droite est $T_{comm} = cT_{calc}$. Bien sûr, c est positif, et le fait que cela corresponde à la partie recouverte des communications, nécessairement[11] on a $c \leq 1$.

Remarque 1. Avec les notations introduites ci-dessus, un espace d'itérations de hauteur gauche M, de largeur bP, de pente inférieure r_b, de pente supérieure r_t et de taille de tuile $w \times h$ correspond au nid de boucles suivant :

$$\begin{array}{l} \mathsf{do}\ t = 0, bPw - 1 \\ \quad \mathsf{do}\ i = 1 + \frac{h}{w} r_b t, \left(M - \frac{r_t - r_b}{2}\right) h + \frac{h}{w} r_t t \\ \quad\quad \theta_{t,i} \end{array}$$

La dernière colonne contient donc $M + (bP - 1)(r_t - r_b)$ tuiles.

Le problème que se posent les auteurs de cet article, est de déterminer la somme des temps d'inactivité de chacun des processeurs ($T_{\Sigma inactif}$). Un processeur pouvant être inactif pour deux raisons :

- La première est liée à la présence des contraintes de dépendance : un processeur peut avoir à attendre des données provenant d'un processeur voisin.
- La seconde est liée à la répartition des tâches sur les processeurs (équilibrage de charge) : un processeur est considéré comme inactif s'il a fini sa charge de travail alors que d'autres processeurs travaillent encore.

Högsted et al. ont cherché à exprimer la somme des temps d'inactivité de tous les processeurs en fonction des différents paramètres exprimés ci-dessus, dans le but de montrer l'influence non négligeable de la pente (r), sur l'efficacité de l'exécution en parallèle d'un nid de boucles. En fait, Högsted et al. ne résolvent que partiellement le problème tel qu'ils l'ont posé. Leurs résultats sont rassemblés dans un tableau récapitulatif (Tableau 2.1). Dans ce tableau, $T_{\Sigma inactif}$ représente la somme des temps d'inactivité des P processeurs. La condition (C) correspond à la condition $M \geq (1+c+r)P$. Elle traduit le fait qu'aucun processeur ne doit attendre entre l'exécution de deux

[11] Une discussion sur ce sujet est fournie au Paragraphe 2.6.2 Page 32.

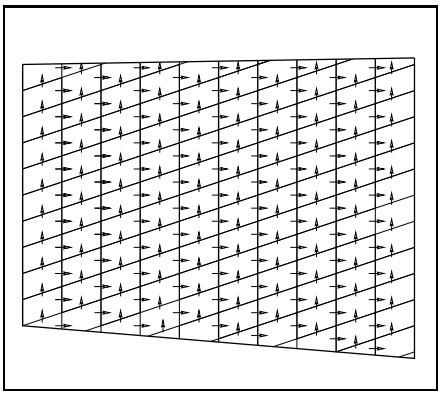


FIG. 2.7: *Exemple d'espace d'itérations trapézoïdal avec des tuiles de forme parallélogramme. Les flèches représentent les dépendances entre les tuiles. Ici,* $r_t = -0.47$, $r_b = -\frac{5}{8}$, $M = 10$, $P = 5$ *et* $b = 2$.

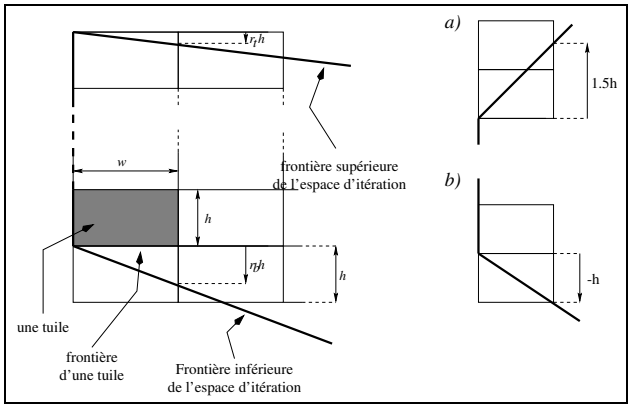


FIG. 2.8: *Forme de l'espace d'itérations. Des tuiles rectangulaires, une pente positive* ($r_t = 1.5$) *en (a) et négative* ($r_b = -1$) *en (b).*

colonnes consécutives. En d'autres termes, la hauteur d'une colonne est suffisamment importante pour maintenir le processeur occupé pendant que l'information utile à l'exécution de la colonne suivante traverse les autres processeurs.

2.5 Temps parallèle, temps séquentiel et temps d'inactivité

Dans ce chapitre, nous développons, sans restreindre les hypothèses de Högsted et al., une méthode de raisonnement plus appropriée que la simple recherche de temps d'inactivité. En effet, comme on peut le voir sur la Figure 2.9, le produit du *temps parallèle* (T_{exe_para}) par le *nombre de processeurs* (P) est égal à la *somme des temps d'inactivité* ($T_{\Sigma inactif}$) et du *temps séquentiel* (T_{exe_seq}). T_{exe_seq} étant le temps nécessaire à un processeur pour exécuter l'ensemble du nid de

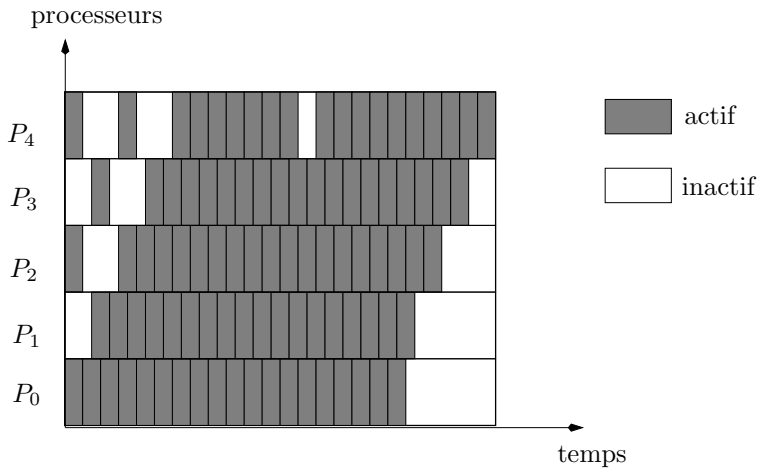


FIG. 2.9: *Activité des processeurs illustrant la formule* $P \times T_{exe\_para} = T_{\Sigma inactif} + T_{exe\_seq}$.

boucles, c'est à dire le temps séquentiel d'exécution. Ainsi, le temps séquentiel est donné par l'aire

Forme parallélogramme • Distribution bloc	
$r \geq -1$	$T_{\Sigma inactif} = P(P-1)(1+r+c)T_{calc}$
$r \leq -2$	$T_{\Sigma inactif} \leq \max(c - \frac{1}{2r}, \frac{-r}{2})PT_{calc}$

Forme parallélogramme • Distribution cyclique	
$r \geq -1$ et condition (C)	$T_{\Sigma inactif} = P(P-1)(1+r+c)T_{calc}$
$r \leq -2$	$T_{\Sigma inactif} \leq \max(c - \frac{1}{2r}, \frac{-r}{2})bPT_{calc}$

Forme trapézoïdale • Distribution bloc • $r_b < r_t$	
$r_b \geq -1$	$T_{\Sigma inactif} = P(P-1)(1+\frac{r_t+r_b}{2}+c)T_{calc}$
$r_b \leq -2 < -1 \leq r_t$	$T_{\Sigma inactif} \leq (\max(c - \frac{1-r^2}{2r}, 0) + (P-1)\frac{r_t+r_b}{2})PT_{calc}$
$r_b < r_t \leq -1$, $r_b \leq -2$	$T_{\Sigma inactif} \leq (\max(c - \frac{1-r^2}{2r}, 0) + (P-1)\frac{r_t+r_b}{2} - \frac{r_t}{2})PT_{calc}$

Forme trapézoïdale • Distribution bloc • $r_t < r_b$	
$-1 \leq r_t \leq r_b$	$T_{\Sigma inactif} = P(P-1)(1+\frac{r_t+r_b}{2}+c)T_{calc}$
$-(1+c) \leq r_t < -1 \leq r_b$	$T_{\Sigma inactif} \leq ((P-1)(1+\frac{r_t+r_b}{2}+c) - \frac{r_t}{2})PT_{calc}$
$r_t \leq -(1+c) < -1 \leq r_b$	$T_{\Sigma inactif} \leq ((P-1)\frac{r_b-r_t}{2} - \frac{r_t}{2})PT_{calc}$
$r_t < r_b \leq -2$, $r_b \leq -2$	$T_{\Sigma inactif} \leq (\max(c - \frac{1-r^2}{2r}, 0) - (P-1)\frac{r_t-r_b}{2} - \frac{r_t}{2})PT_{calc}$

TAB. 2.1: *Résumé des résultats d'Högsted, Carter, et Ferrante : calcul du temps total d'inactivité* $T_{\Sigma inactif}$.

de l'espace d'itérations, il vaut : $(bPM + \frac{bP(bP-1)}{2}(r_t - r_b))$.

2.5.1 Graphe de dépendance. Recherche de chemin critique

La formalisation est simple : à chaque tuile est associé un sommet de poids T_{calc}. Chaque contrainte de dépendance est caractérisée par une arête entre deux sommets ; une dépendance verticale a un poids nul, et une dépendance horizontale a un poids valant cT_{calc}. Si $b > 1$, on rajoute à ce graphe des arêtes de disponibilité (il faut qu'un processeur soit libre pour pouvoir lui assigner du travail) de poids nul entre la tâche de fin d'une colonne de tuiles (i) et la tâche de début de la colonne de tuiles suivante $(i + P)$. Le problème consiste à trouver le chemin de poids maximum de ce graphe. La Figure 2.11 représente tous les chemins critiques du problème de la Figure 2.7.

Restrictions. La seule hypothèse simplificatrice que l'on fait ici est d'effectuer une approximation rationnelle dans la recherche de notre chemin critique. Cela consiste à supposer que si l'on prend deux chemins de même origine et de même cible de type *horizontal-vertical* et *vertical-horizontal* (Figure 2.10), alors ils ont même longueur. Comme nous pouvons le constater sur la Figure 2.10, cette hypothèse n'est pas vraie dans le cas général. En revanche, on pourrait aisément montrer que l'erreur engendrée est négligeable, puisque majorée par $\frac{|r_t|}{2}$ dont la valeur est généralement pas très grande.

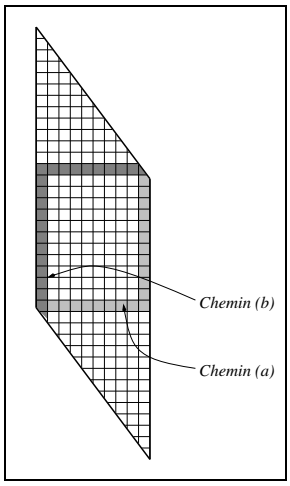


FIG. 2.10: *Le chemin (a) a, dans cet exemple, une aire inférieure à l'aire du chemin (b). On considère pourtant dans notre étude que le chemin (a) est critique. L'erreur engendrée, inférieure à la différence d'aire c'est à dire à $\left|\frac{r_t}{2}\right|$, est un effet de bords, lié à la présence de tuiles incomplètes.*

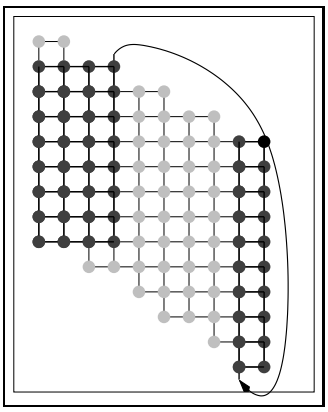


FIG. 2.11: *Chemins critiques du problème $M = 10$, $P = 5$, $b = 2$, $r_b = -0.625$, $r_t = -0.47$, $c = 0.5$, $h = 1.5$ et $w = 2$. Les contraintes qui font partie d'un chemin critique sont représentées en plus foncé. Un trait vertical représente une contrainte de disponibilité, et un trait horizontal une contrainte de dépendance. Il faut aussi noter la contrainte de disponibilité entre la dernière tuile de la quatrième colonne et la première tuile de l'avant-dernière colonne.*

Comme explicité dans les hypothèses, le nombre de colonnes de l'espace d'itérations est un multiple de P, mais les formules présentées ci-dessous peuvent être généralisées sans difficulté.

Pour finir, on peut remarquer que si, dans le cas où $r_b = r_t = 0$, l'allocation des processeurs par colonne est optimale [23], il n'en est rien dans le cas général. Par contre, comme démontré par Boulet et al. [20], elle est clairement optimale de manière asymptotique lorsque le nombre de colonnes bP tend vers l'infini.

Les résultats. Nos formules sont résumées dans le Tableau 2.2. Elles sont présentées comme pour les résultats de Högsted et al.. En contre-partie nous ne fournissons dans ce tableau que le temps total d'exécution T_{exe_para}, le temps total d'inactivité $T_{\Sigma inactif}$ pouvant être obtenu grâce à la formule : $T_{\Sigma inactif} = PT_{exe\_para} - (bPM + \frac{bP(bP-1)}{2}(r_t - r_b))T_{calc}$. Par ailleurs, nous utiliserons par la suite la notation a^+ pour représenter la partie positive d'un nombre réel a :

$$a^+ = \begin{cases} a \text{ si } a \geq 0 \\ 0 \text{ si } a < 0 \end{cases}$$

Forme parallélogramme • Distribution bloc • $T_{exe\_seq} = PMT_{calc}$	
$M + (P-1)r \geq 0$	$T_{exe\_para} = [M + (P-1)(1 + r + c)^+]T_{calc}$
$M + (P-1)r < 0$	$T_{exe\_para} = M \left\lfloor 1 + \frac{(1+r+c)^+}{\lvert r \rvert} \right\rfloor T_{calc}$

Forme parallélogramme • Distribution cyclique • $T_{exe\_seq} = bPMT_{calc}$	
$1 + r + c \leq 0$	$T_{exe\_para} = bMT_{calc}$
$1+r+c > 0$, $M \geq (1+r+c)P$, $M + (P-1)r \geq 0$	$T_{exe\_para} = [bM + (1 + r + c)(P-1)]T_{calc}$
$1+r+c > 0$, $M \geq (1+r+c)P$ et $M + (P-1)r < 0$	$T_{exe\_para} = M \left\lfloor b + \frac{1+r+c}{-r} \right\rfloor T_{calc}$
$1+r+c > 0$, $M < (1+r+c)P$ et $M + (bP-1)r \geq 0$	$T_{exe\_para} = [M + (1 + r + c)(bP-1)]T_{calc}$

Forme trapézoïdale • Distribution bloc • $T_{exe\_seq} = (PM + \frac{P(P-1)}{2}(r_t - r_b))T_{calc}$	
$M + (P-1)r_t \geq 0$	$T_{exe\_para} = [M + (P-1)((1 + r_b + c)^+ + r_t - r_b)^+]T_{calc}$
$1 + r_b + c \leq 0$	
$M + (P-1)r_t < 0$, et $1 + r_b + c > 0$	$T_{exe\_para} = (M + (P-1)(r_t - r_b)^+) \left\lfloor 1 + \frac{(1+r_m+c)^+}{-r_m} \right\rfloor T_{calc}$ où $r_m = \min(r_b, r_t)$

Forme trapézoïdale • Distribution cyclique • $T_{exe\_seq} = (bPM + \frac{bP(bP-1)}{2}(r_t - r_b))T_{calc}$	
$1 + r_b + c \leq 0$	$T_{exe\_para} = [bM + \frac{b(b-1)}{2}P(r_t - r_b) + (bP-1)(r_t - r_b)^+]T_{calc}$
$1 + r_b + c > 0$ et $M + (bP-1)r_t \geq 0$	Voir théorème 4, Page 30

TAB. 2.2: *Résumé de nos résultats.*

2.5.2 Le cas distribution en bloc, forme parallélogramme

C'est le cas le plus simple, et nous allons l'expliquer en détail.

Théorème 1 *Soit un espace d'itérations de forme parallélogramme formé de P colonnes de de M tuiles chacune (distribution bloc), et de pente r.*

- *Si $M+(P-1)r \geq 0$, alors $T_{exe\_para} = [M+(P-1)(1+r+c)^+]T_{calc}$. De manière équivalente, $T_{\Sigma inactif} = P(P-1)(1+r+c)^+T_{calc}$*
- *Si $M+(P-1)r < 0$, alors* $\begin{cases} T_{exe\_para} = M\left[1+\frac{(1+r+c)^+}{|r|}\right]T_{calc} \\ T_{\Sigma inactif} = \frac{PM}{|r|}(1+r+c)^+T_{calc} \end{cases}$

Preuve. Tous les processeurs ont la même quantité de travail à effectuer, MT_{calc}. A cause des dépendances, le processeur $(P-1)$ est toujours le dernier à terminer son exécution. Séparons pour la discussion les deux cas $r<0$ et $r \geq 0$.

- **Cas** $r<0$, $M \geq (P-1)|r|$**.** La tuile d'ordonnée 0 du processeur $q=(P-1)$ (même hauteur que la tuile du coin bas-gauche de l'espace d'itérations) doit attendre
  - d'une part, que toutes les tuiles situées sur sa gauche sur l'axe horizontal soient exécutées. Cela prend un temps $(P-1)(1+c)T_{calc}$.
  - d'autre part, que toutes les tuiles situées au dessous sur l'axe vertical soient exécutées. Cela prend un temps $(P-1)(-r)T_{calc}$.

  Ensuite, il reste $M+(P-1)r$ tuiles à exécuter, ce qui prend $(M+(P-1)r)T_{calc}$ unités de temps. Ainsi, le plus long chemin dans le graphe de dépendances a pour longueur

  $$[(P-1)\max(-r, 1+c) \; + \; (M+(P-1)r)]\, T_{calc}.$$

  La Figure 2.13 représente un exemple de chemin critique dans cette configuration. L'existence d'un chemin horizontal traversant tout l'espace d'itération est nécessaire à assurer la validité de la preuve. En effet, si tel n'est pas le cas, il n'est alors pas possible de construire un chemin critique allant du coin bas-gauche de l'espace d'itération à son coin haut-droit. Remarquons que si un chemin critique ne peut suivre une diagonale descendante (il doit nécessairement suivre les arêtes de dépendances $\{(1,0)^t, (0,1)^t\}$), il peut par contre suivre une diagonale montante, et la condition $M \geq (P-1)|r|$ n'est nécessaire que si $r_t < 0$.
- **Cas** $r<0$, $M < (P-1)|r|$. Un des chemins critiques va de la première tuile du premier processeur à la dernière tuile du processeur $q=Q$ où $M=(Q-1)|r|$; le temps parallèle vaut donc dans ce cas

  $$T_{exe\_para} = [M+(Q-1)(1+r+c)^+]T_{calc}.$$

  Notons qu'ici, Q doit être plus petit que P. Le temps d'inactivité devient alors

  $$T_{\Sigma inactif} = \frac{PM}{|r|}(1+r+c)^+T_{calc}$$

- **Cas** $r \geq 0$ Le processeur $(q=(P-1))$ doit attendre $(P-1)rT_{calc}$ unités de temps que le processeur $q=0$ exécute ses $(P-1)r$ premières tâches. Ensuite, le processeur $q=(P-1)$ doit attendre encore $(P-1)(1+c)T_{calc}$ unités de temps correspondant à l'exécution des tuiles et aux communications des données le long de l'axe horizontal qui passe par sa première tuile. Alors seulement, il peut exécuter ses M tâches, et ne plus être retardé dans son exécution. La longueur du chemin critique représenté sur la Figure 2.14 vaut donc, dans ce cas, $(P-1)(1+r+c)T_{calc}+M$.

■

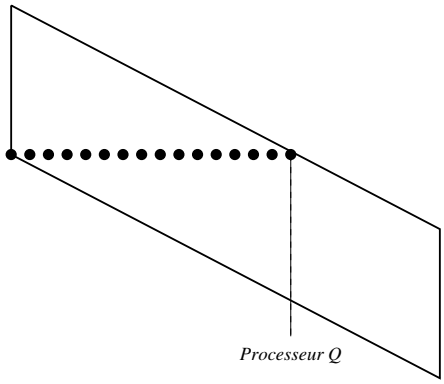


FIG. 2.12: *Chemin critique quand* $M + (P-1)r < 0$.

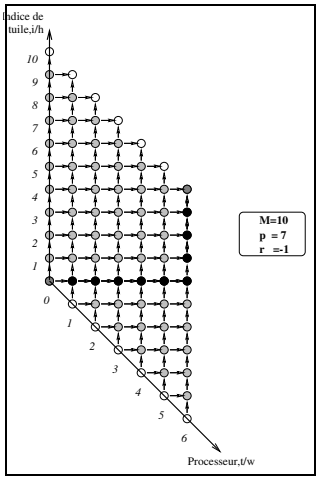


FIG. 2.13: *Chemin critique quand* $r < 0$ *et* $M + (P-1)r \geq 0$.

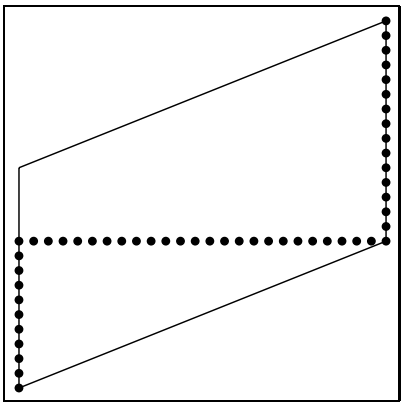


FIG. 2.14: *Chemin critique quand* $r \geq 0$.

Remarque 2. On remarque que les résultats de Högsted et al. reportés dans la Table 2.1 sont imprécis : une faible pente (r) ne préserve pas nécessairement d'un temps d'inactivité quadratique ; la condition nécessaire précise est $1 + r + c \leq 0$. Résultat compréhensible, puisqu'il est normal que le rapport c du *temps de communication/temps de calcul* joue un rôle dans l'expression du temps total d'exécution. En résumé, quand $1 + r + c$ est négatif (ce qui est toujours vérifié quand $r \leq -2$), la pente est suffisamment faible pour que les processeurs puissent travailler indépendamment. D'un autre coté, lorsque $1 + r + c$ est positif (ce qui est d'ailleurs toujours vérifié quand $r \geq -1$) le temps d'inactivité total croît de manière quadratique avec le nombre de processeurs.

2.5.3 Le cas distribution en bloc, domaine de forme trapézoïdale

Théorème 2 *Considérons un espace d'itérations de forme trapézoïdale de hauteur gauche M, de largeur P (distribution bloc), de pente inférieure r_b et de pente supérieure r_t.*

- *Si $M + (P-1)r_t \geq 0$ ou bien $1 + r_b + c \leq 0$, alors $T_{exe\_para} = [M + (P-1)((1 + r_b + c)^+ + r_t - r_b)^+]T_{calc}$.*
- *Si $M+(P-1)r_t < 0$ et $1+r_b+c > 0$, alors $T_{exe\_para} = (M+(P-1)(r_t-r_b)^+) \left[1 + \frac{(1+r_m+c)^+}{-r_m}\right] T_{calc}$ où $r_m = \min(r_b, r_t)$.*

Preuve.

- **Cas** $1+r_b+c \leq 0$ **ou** $[M+(P-1)r_t \geq 0$]. Sous l'une de ces conditions, on est pas confronté au problème de non existence d'un chemin descendant. Soit $H(j) = M + j(r_t - r_b)$ pour $0 \leq j \leq P-1$, la quantité de travail à effectuer par le processeur j, c'est à dire la hauteur de la colonne j.
  - **Cas** $1+r_b+c \leq 0$**.** Toutes les colonnes peuvent être exécutées indépendamment. Le temps total est donc donné par le maximum des quantités de travail de tous les processeurs, soit $T_{exe\_para} = H(0)T_{calc} = MT_{calc}$ si $r_t \leq r_b$ et $T_{exe\_para} = H(P-1)T_{calc} = (M + (P-1)(r_t - r_b))T_{calc}$ dans le cas contraire.
  - **Cas** $1+r_b+c > 0$**.** Tous les processeurs ont des temps d'inactivité liés aux communications horizontales. Chaque processeur j (pour $0 \leq j \leq P-1$) termine l'exécution de sa colonne à l'instant $T_j = [(1 + c + r_b)j + M + j(r_t - r_b)]T_{calc}$.
    En fonction du signe de $(1 + c + r_b) + (r_t - r_b) = 1 + c + r_t$, cette quantité est maximum soit pour $j = 0$ soit pour $j = P-1$. On en déduit la formule présentée plus haut.
- **Cas** $[M + (P-1)r_t < 0$ **et** $1 + r_b + c > 0]$. Remarquons tout d'abord que la condition $M + (P-1)r_t < 0$ implique nécessairement que $r_t < 0$ et $r_b < 0$.
  - **Cas** $r_t \geq r_b$**.** Comme $1 + r_b + c > 0$, le chemin critique est une droite horizontale partant de la bordure inférieure et atteignant le sommet en haut à droite de l'espace d'itération, de longueur $T_{exe\_para} = \frac{M+(P-1)(r_t-r_b)}{-r_b}(1 + c)T_{calc}$. Rappelons que si l'on avait $1 + r_b + c \leq 0$, alors $T_{exe\_para} = (M + (P-1)(r_t - r_b))T_{calc}$.
  - **Cas** $r_t < r_b$**.** Le problème peut être résolu par symétrie, en intervertissant r_b et r_t et en remplaçant M par $M + (P-1)(r_t - r_b)$. Ainsi,
    - si $1 + r_t + c \leq 0$, $T_{exe\_para} = \frac{M}{-r_t}(1 + c)T_{calc}$,
    - sinon, $T_{exe\_para} = MT_{calc}$.

■

Une fois de plus, on remarque que la condition $1 + r_b + c \leq 0$ est celle qui permet de minimiser le temps total d'inactivité : si cette condition est vérifiée, les temps d'inactivité sont dus à la répartition inégale de la quantité de travail sur les processeurs et non à l'attente de données due aux contraintes de dépendance.

2.5.4 Le cas distribution cyclique, forme parallélogramme

Théorème 3 *Considérons un espace d'itérations de forme parallélogramme, formé de bP colonnes de M tuiles chacune (distribution cyclique) et de pente r, alors*

- *si $1 + r + c \leq 0$, $T_{exe\_para} = bMT_{calc}$*
- *sinon, si $M \geq (1 + r + c)P$, $T_{exe\_para} = \left(bM + (1 + r + c)\frac{\min(M, -r(P-1))}{-r}\right) T_{calc}$,*
- *sinon, si $M + (bP-1)r \geq 0$, $T_{exe\_para} = ((1 + r + c)(bP - 1) + T_{calc}$.*

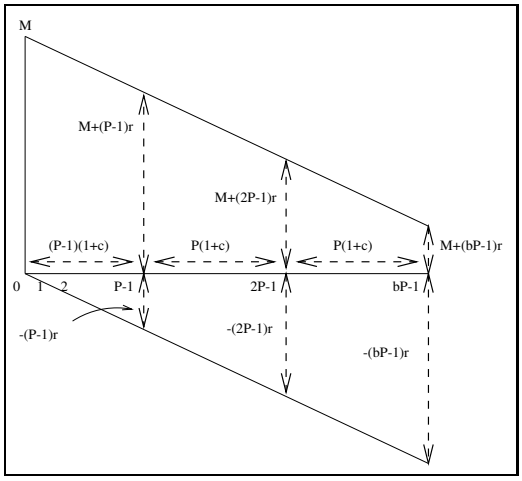


FIG. 2.15: *Schéma de la preuve pour $r < 0$.*

Preuve. Tous les processeurs ont la même charge de travail, bMT_{calc}.

- **Cas $1 + r + c \leq 0$.** Toutes les colonnes de tâches peuvent être exécutées indépendamment, et la première partie du résultat en découle.
- **Cas $1+r+c > 0$ et existence d'un chemin horizontal.** Le processeur $(P-1)$ est toujours le dernier à finir son exécution. Analysons séparément les cas $r < 0$ et $r \geq 0$.
  - **Cas $r < 0$.** Le travail du processeur $q = P - 1$ peut être décomposé de la manière suivante (Figure 2.15) :

$$
\begin{array}{ll}
T_{exe\_para} = & \\
\max(1+c, -r)(P-1) & \textit{maximum entre la propagation des données le long de l'axe horizontal et le calcul des tuiles situées sous l'axe dans la colonne } (P-1) \\
+\sum_{k=1}^{b-1} \max((1+c)P, M - Pr) & \textit{calcul des tuiles restantes dans la colonne } kP-1 \textit{ et tuiles sous l'axe dans la colonne } (k+1)P-1 \\
+M + (bP-1)r & \textit{tuiles restantes dans la colonne } bP-1
\end{array}
$$

  - Lorsque le temps de propagation des données est plus grand que le temps de calcul $(M < (1+r+c)P)$, l'expression ci-dessus n'est vérifiée que s'il existe un chemin horizontal traversant tout l'espace d'itération c'est à dire lorsque $M + (bP-1)r \geq 0$.
  - Par contre, lorsque $M \geq (1+r+c)P$, l'expression ci-dessus n'est vérifiée que s'il existe un chemin du sommet bas-gauche de l'espace d'itération à la colonne $P-1$. Ceci n'est vérifié que si $M + (P-1)r \geq 0$. Dans le cas contraire, le temps que doit attendre le processeur $P-1$ avant de pouvoir travailler est le même que dans le cas d'une distribution bloc c'est à dire $\frac{M}{|r|}(1+c)$.
  - **Cas $r \geq 0$.** La même décomposition conduit à (voir Figure 2.16)

$$
\begin{array}{lll}
T_{exe\_para} = & (1+r+c)(P-1) & \textit{temps de démarrage} \\
& +\sum_{k=0}^{b-2} \max((1+c+r)P, M) & \textit{tuiles de la colonne } j+kP \\
& +M & \textit{tuiles de la colonne } j+(b-1)P
\end{array}
$$

Il s'avère que les deux expressions de T_{exe_para} coïncident du fait que $1+r+c > 0$. En d'autres termes, la dernière expression est valable à la fois pour le cas $r < 0$ et $r \geq 0$. Ce qui nous conduit directement au résultat.

■

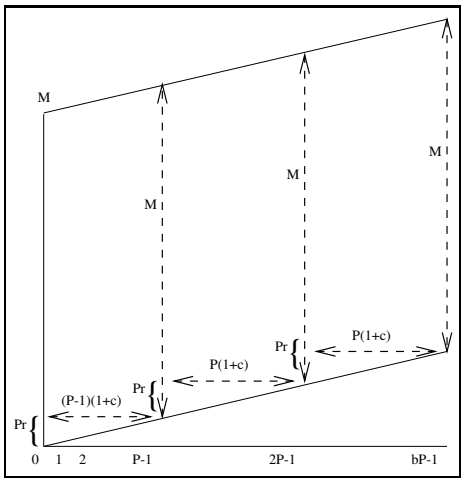


FIG. 2.16: *Schéma de la preuve pour $r \geq 0$.*

2.5.5 Le cas distribution cyclique, forme trapézoïdale

Le cas distribution cyclique, forme trapézoïdale est le cas le plus complexe. On obtient les résultats suivants :

Théorème 4 *Considérons un espace d'itérations de forme trapézoïdale, de hauteur gauche M, de largeur bP (distribution cyclique), de pente inférieure r_b et de pente supérieure r_t.*

- *Si $1 + r_b + c \leq 0$, alors $T_{exe\_para} = [bM + \frac{b(b-1)}{2}P(r_t - r_b) + (bP-1)(r_t - r_b)^+]T_{calc}$.*
- *Sinon, si $M + (bP-1)r_t \geq 0$ alors $T_{exe\_para} = \max(T_{exe}(P_j), 0 \leq j \leq P-1)$. $T_{exe}(P_j) = [(1+c)j]T_{calc} + \left[\sum_{k=0}^{b-2} \max((1+c)P, M - Pr_b + (j+kP)(r_t - r_b))\right] T_{calc} + [M + (j + (b-1)P)r_t]T_{calc}$.*

Preuve.

- **Cas** $1 + r_b + c \leq 0$. Toutes les colonnes peuvent être exécutées indépendamment les unes des autres. Dans ce cas, le processeur le plus chargé est le processeur $q = 0$ si $r_t \leq r_b$ et le processeur $q = P - 1$ dans le cas contraire.
  La charge de travail du processeur j est $\sum_{k=0}^{b-1} H(j+kP)T_{calc}$ où $H(j+kP) = M + (j + kP)(r_t - r_b)$ représente la hauteur de la colonne $j + kP$ ($0 \leq j \leq P-1$ et $0 \leq k \leq b-1$), ce qui conduit à la première partie du résultat :
  - **Cas** $r_b \geq r_t$. C'est le processeur $q = 0$ qui se voit assigner le plus de travail. Alors $\sum_{k=0}^{b-1} H(kP)T_{calc} = \left(bM + \frac{b(b-1)}{2}P(r_t - r_b)\right) T_{calc}$.
  - **Cas** $r_b \leq r_t$. C'est le processeur $q = P - 1$ qui a le plus de charge de travail, et alors $\sum_{k=0}^{b-1} H(P-1+kP)T_{calc} = \left[bM + \left(\frac{b(b-1)}{2}P + (bP-1)\right)(r_t - r_b)\right] T_{calc}$.

– **Cas** $1 + r_b + c > 0$. Il est plus compliqué de déterminer le plus long chemin.
  – **Cas** $r_b < 0$. On utilise une décomposition analogue à celle donnée dans la preuve du Théorème 3 pour exprimer le travail du processeur j $(0 \leq j \leq P-1)$:

$$\begin{array}{ll}
\frac{T_exe(P_j)}{T_{calc}} = & \\
\max(1+c, -r_b)j & \textit{maximum entre la propagation des données le long de l'axe horizontal et le calcul des tuiles situées sous l'axe dans la colonne } j \\
+\sum_{k=1}^{b-2} \max((1+c)P, M - Pr_b + (j+kP)(r_t - r_b)) & \textit{tuiles restantes dans la colonne } j+kP \textit{ et tuiles sous l'axe dans la colonne } j+(k+1)P \\
+M + (j+(b-1)P)r_t & \textit{tuiles restantes dans la colonne } j+(b-1)P
\end{array}$$

Il reste à prendre la valeur maximum de ces quantités pour obtenir le temps parallèle :

$$T_{exe\_para} = \max_{j=0}^{P-1} T_{exe}(P_j)$$

Cette expression n'est vérifiée que si les chemins horizontaux utilisés existent. C'est en particulier le cas si $M + (bP-1)r_t \geq 0$.

  – **Cas** $r_b \geq 0$. Identiquement, on obtient la décomposition suivante pour le processeur j :

$$\begin{array}{ll}
\frac{T_exe(P_j)}{T_{calc}} = & \\
(1+r_b+c)j & \textit{temps de démarrage} \\
+\sum_{k=0}^{b-2} \max((1+c+r_b)P, M + (j+kP)(r_t - r_b)) & \textit{tuiles de la colonne } j+kP \\
+M + (j+(b-1)P)(r_t - r_b) & \textit{tuiles de la colonne } j+(b-1)P
\end{array}$$

De même, ces deux expressions coïncident avec l'expression du cas $r_b \leq 0$. D'où le résultat. ■

Remarque 3. Il est assez aisé de simplifier l'expression de $T_{exe}(P_j)$ en réduisant la somme de maxima en une expression plus compacte. C'est une simple discussion en fonction du signe de $r_t - r_b$ et de la valeur de k à partir de laquelle l'expression du maximum s'inverse. Ensuite, il suffit d'évaluer analytiquement la valeur de j qui maximise $T_{exe}(P_j)$.

2.6 Forme, taille et grains optimaux

Trouver la pente optimale pour une hauteur de tuile donnée, ou déterminer la hauteur et la largeur de tuiles qui minimisent le temps total d'exécution, sont les questions naturelles qui viennent en complément de notre étude.

2.6.1 Pente optimale

Approche analytique. Si l'on regarde attentivement les formules du temps total parallèle (T_{exe_para}) d'exécution du nid de boucles en fonction de la pente et de la distribution des données, on remarque que, pour une distribution donnée, T_{exe_para} est constant pour $1+c+r_b < 0$ et strictement décroissant pour $1 + c + r_b \geq 0$.

Ainsi, le minimum est clairement atteint pour $1 + c + r_b \leq 0$, c'est à dire pour $r_b \leq -(1 + c)$. En particulier, comme $c \leq 1$, $r_b = -2$ convient, et en pratique c'est cette valeur de pente qui devra être choisie. En effet, on a tout intérêt à prendre une pente à valeur entière car tout le bénéfice d'une pente négative est perdu par la présence de parties entières à effectuer, de multiplications réelles, etc.
Rappelons que l'espace de liberté de la valeur de la pente est limité par les vecteurs de dépendance, c'est à dire qu'il ne sera pas toujours possible de choisir une telle pente. En contrepartie, on peut retenir qu'il est souhaitable que r_b soit le plus petit possible.

Simulation. D'après les formules précédemment établies, la courbe du gain devrait prendre une valeur constante pour $1 + r_b + c \leq 0$. Il n'en est rien et cette erreur est liée, rappelons le, à l'approximation présentée Figure 2.10. Mais, comme le montre l'exemple de la Figure 2.17, l'impact de cette approximation est négligeable.

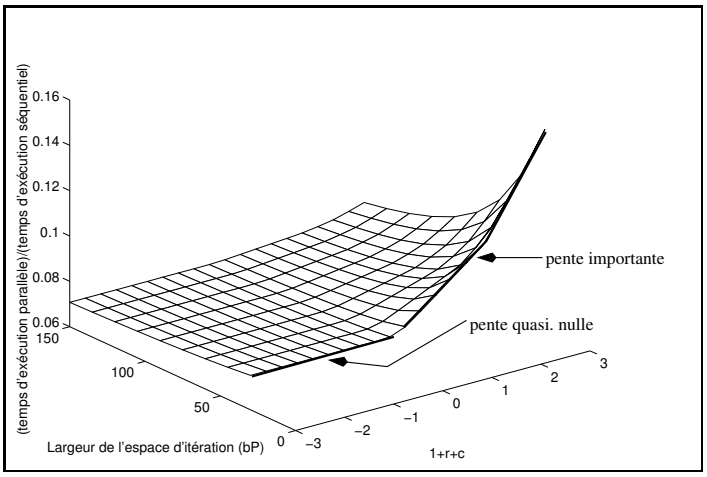


FIG. 2.17: *Valeurs obtenue à l'aide d'une simulation, dans laquelle $c = 0.5$; l'espace d'itération est de forme parallélogramme ; la distribution est cyclique ; b varie de 2 à 10 ; P vaut 15 ; $1+r+c$ varie de -4 à 4. Le temps d'exécution parallèle ainsi rapporté, est normalisé par le temps d'exécution séquentiel : est donc tracé le rapport $\frac{T_{exe\_para}}{T_{exe\_seq}}$.*

2.6.2 Taille de tuile optimale : futilité des modèles

Déterminer la forme ou la taille des tuiles est un problème majeur dans le domaine du pavage. En fixant la pente, nous avons, dans le cadre de nos restrictions, fixé la forme des tuiles. Reste logiquement à trouver la taille optimale. Une telle étude a été récemment menée par Andonov et al. [5]. Plutôt que de présenter ici une solution exhaustive du problème, nous formulons plutôt quelques remarques relatives aux modèles de machines usuellement employés.

Choisir un bon modèle de communications est une question délicate :

– La communication est-elle recouverte par les calculs ? L'est-elle entièrement ou seulement partiellement ? Cela dépend bien évidemment de la plateforme considérée, mais même sur un système censé a priori recouvrir les communications par les calculs, on a malheureusement souvent de mauvaises surprises. Il est en fait difficile d'obtenir des mesures stables.
Supposons néanmoins que nous disposions d'un système stable permettant le recouvrement total des communications par les calculs. Dans ce cas, le temps de communications pouvant

a priori dépasser le temps de calculs ($c > 1$), la formule de temps de calcul d'une tuile de taille $h \times w$ doit s'écrire $T_{calc} = \max(hw\tau_{calc}, \tau_{comm}(h))$ et non $hw\tau_{calc}$. Le cas échéant, on obtiendrait une taille de tuile optimale de l'ordre de 1×1 (cf [23]) : résultat absurde.

– Ensuite, on peut vouloir quantifier le temps de communication en fonction du volume de données communiquées. Mais, comme en témoigne la courbe de la Figure 2.18, la réalité est souvent bien différente du modèle "$\tau_{comm}(l) = \beta + l\tau$" usuellement utilisé (cf Chapitre 5 pour une discussion détaillée sur le sujet). Le temps de communication dépend fortement de la quantité de données manipulées par le programme, de la taille des tuiles, de la longueur de la ligne de cache, etc. Réciproquement, la vitesse de calcul est, elle aussi, dépendante de la fréquence et de la taille des communications.

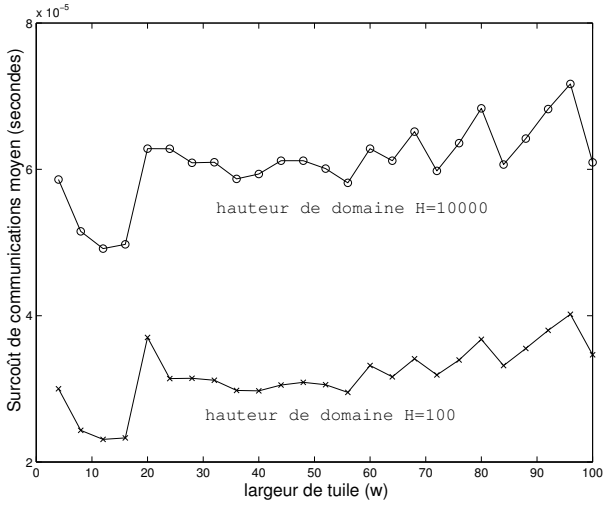


FIG. 2.18: *Courbe expérimentale extraite du Chapitre 5 rapportant en secondes le temps moyen de communication de 48 octets en fonction de la taille des paquets (*$48w$ *octets), de leur fréquence d'envoi (*$\frac{1}{w}$*) et de la quantité de données traitées entre chaque pas d'itération (*H*).*

On ne peut donc pas, dans ce cadre, proposer de manière sérieuse une taille de tuile optimale à la virgule près. La bonne approche consisterait probablement à raisonner en termes d'intervalles, en se donnant une marge d'erreur. Elle s'effectuerait étape par étape, en commençant par exemple par la vérification des contraintes de localité au niveau du cache, puis par la minimisation de la longueur du chemin critique, etc. On obtiendrait ainsi un ensemble de solutions possibles, assurant un temps proche de l'optimal, parmi lesquelles on choisirait la solution la plus simple.

2.6.3 Pavage hiérarchique

Comme l'a été très justement souligné par Högsted et al., la méthode de pavage peut être utilisée dans le cadre de mémoire à niveaux multiples et de parallélisme hiérarchique [25]. Une importante motivation pour déterminer les temps d'inactivité de l'exécution dans [55], était de montrer qu'un tel temps d'inactivité pouvait avoir un impact non négligeable sur les performances de nombreuses applications.

Nous réutilisons l'exemple donné dans [55] pour illustrer ce point : un large espace d'itérations avec des *dépendances verticales et horizontales* est partitionné en tuiles. De plus, chaque tuile est elle-même partitionnée en sous-tuiles que l'on projette sur les processeurs (cf Figure 2.19, où les tuiles et les sous-tuiles sont rectangulaires, en opposition avec la Figure 2.20 où les tuiles sont de

forme parallélogramme et les sous-tuiles toujours rectangulaires.).
Considérons l'exemple donné ci-dessous : les données, stockées sur le disque sont chargées tuile par tuile et distribuées sur les mémoires principales de chacun des processeurs. La taille des tuiles est choisie en fonction de la taille des mémoires, et on suppose une synchronisation implicite entre deux tuiles consécutives.

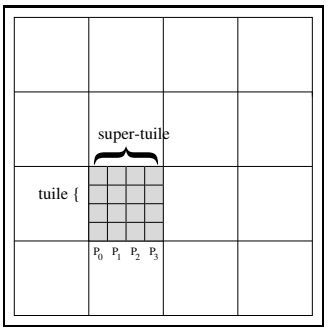


FIG. 2.19: *Partitionnement de l'espace d'itérations en tuiles de forme rectangulaire et en sous-tuiles. La pente vaut $r = 0$.*

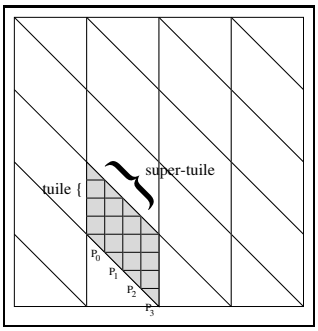


FIG. 2.20: *Partitionnement de l'espace d'itérations en tuiles de forme parallélogramme et en sous-tuiles. La pente vaut $r = -1$.*

Quelle est la meilleure stratégie ? Des tuiles rectangulaires comme sur la Figure 2.19, ou des tuiles parallélogrammes comme sur la Figure 2.20 ?
Comme établi dans [3], les tuiles parallélogramme de la Figure 2.20 contiennent un temps substantiel d'inactivité inférieur aux tuiles rectangulaires de la Figure 2.19.
Les résultats des paragraphes précédents nous permettent de répondre précisément au problème : on peut déterminer analytiquement la meilleure répartition, comme une fonction des paramètres de l'espace d'itérations et des caractéristiques de la machine.
Soient h et w respectivement la hauteur et la largeur normalisées des sous-tuiles. Le temps d'exécution est de $T_{calc} = hw\tau_{calc}$. Considérons une distribution bloc des sous-tuiles sur les processeurs de telle manière qu'une tuile soit de taille $Mh \times Pw$: en d'autres termes, une tuile contient P colonnes de M sous-tuiles chacune. Par exemple, sur les figures 2.19 et 2.20, on a $M = 4$ et $P = 4$. Notons $D_h = d_h M$ et $D_w = d_w P$ tels que la taille de l'ensemble de l'espace d'itérations soit $D_h h \times D_w w$. Avec une répartition rectangulaire, il y a $d_h d_w$ tuiles. Avec une répartition parallélogramme, il y a $(d_h + \lceil |r| \rceil) d_w$ tuiles (une tuile partielle est considérée comme une tuile pleine du fait des barrières de synchronisation). Le lemme suivant est une conséquence directe des résultats établis dans ce chapitre.

Lemme 1 *Avec les notations courantes, le temps total d'exécution est*

$$\begin{cases} T_{exe\_rect} = [M + (P-1)(1+c)] d_h d_w T_{calc} & \text{pour des tuiles rectangulaires} \\ T_{exe\_pente}(r) = M(d_h + \lceil |r| \rceil) d_w T_{calc} & \text{pour des tuiles parallélogrammes de pente } r \leq -(1+c) \end{cases}$$

Pour des tuiles rectangulaires, si on réécrit T_{exe_rect} comme $T_{exe\_rect} = [M+(P-1)(1+c)] \frac{D_h}{M} \frac{D_w}{P} T_{calc}$, on voit que c'est une fonction décroissante en M. En d'autres termes, M doit être choisi le plus grand possible et tel que Mhw données tiennent dans la mémoire d'un seul processeur. Une fois la valeur de M fixée, on choisit comme pente r une valeur suffisamment petite telle que $(1+r+c)^+ = 0$. Prenons $r = -2$. On trouve alors que $T_{exe\_pente}(-2) = M(\frac{D_h}{M} + 2) \frac{D_w}{P} T_{calc}$. On en déduit le Théorème suivant :

Théorème 5 *Si $2M \leq (P-1)(1+c)\frac{D_h}{M}$ alors $T_{pente}(-2) \leq T_{rect}$.*

La condition du Théorème 5 sera toujours vérifiée pour des domaines d'itérations suffisamment larges. En d'autres termes, des tuiles de forme parallélogramme donnent de meilleures performances.

2.7 Conclusion

Dans ce chapitre, nous avons étendu les résultats d'Högsted, Carter, et Ferrante [55]. Nous avons ainsi déterminé le temps d'inactivité associé à un pavage, pour des domaines d'itérations de forme parallélogramme et trapézoïdale. Nous avons fourni une expression de forme close du temps d'inactivité, pour toute valeur du paramètre de pente (r, r_b, r_t) et pour une distribution bloc aussi bien que pour une distribution bloc-cyclique. Mais, ces résultats restreints au cas bidimensionnel devraient à présent être généralisé au cas multidimensionnel. Par ailleurs, le cas d'un espace d'itération de forme triangulaire est très fréquent et mériterais une attention toute particulière.

Nous avons appliqué nos résultats dans le contexte du pavage hiérarchique. Bien que nous ayons traité seulement un exemple particulier du problème de pavage récursif, nous pensons que notre approche est assez générale pour être appliquée dans plusieurs autres situations (comme celles décrites dans [25]).

Par ailleurs, un sujet important est la compilation de langages de haut niveau sur machines hétérogènes, comme des clusters de machines SMP[12] par exemple. L'hétérogénéité des réseaux, des éléments de calculs et des hiérarchies mémoires, augmente la complexité des solutions. Il est maintenant admis que l'obtention de performances importantes sur ce type de machine passera par l'utilisation d'un système d'exploitation performant et complet, utilisant notamment les processus légers. Malgré tout, la présence d'un système aussi évolué qu'il soit, ne dispensera pas les compilateurs de générer un code à gros grains. Les résultats présentés dans ce chapitre doivent donc être étendus à ce type de plateformes. Ce problème est abordé dans le Chapitre 6 de cette thèse.

[12] *Shared Memory Processors* : machines à mémoire partagée

Chapitre 3

Pavage de nids de boucles sans dépendances internes

3.1 Introduction

Comme indiqué dans le Chapitre 2 de cette thèse, il existe plusieurs approches dans le domaine du pavage : minimisation du chemin critique [6, 23, 54, 78], minimisation du volume ou du nombre de communications inter-processeurs [19, 24]... Ces techniques s'appliquent au pavage de nids de boucles uniformes dans le contexte de machines à mémoire distribuée. C'est la présence de dépendances uniformes, internes au nid, qui crée des communications entre les processeurs voisins. Il existe aussi bien sûr, des nids de boucles comprenant des dépendances externes. Il y a par exemple le cas où les références à un même tableau ne sont qu'en lecture : c'est alors le rapatriement initial des données vers chaque processeur qui doit être optimisé. Ces données peuvent être soit distribuées sur la grille de processeurs (distribution CYCLIC de HPF par exemple), soit située sur une mémoire partagée externe.

Dans ce cadre, Agarwal, Kranz et Natarajan [1] cherchent à minimiser le volume de données utilisées par le calcul d'une tuile (nommée *empreinte*[1]). Plus précisément, leur but est de trouver un pavage hyper-parallélépipédique du domaine d'itérations sous le critère d'optimisation suivant : étant donnée une taille de tuile fixée (liée par exemple à la taille du cache), déterminer la forme de tuile qui minimise la valeur de l'empreinte, c'est à dire maximise la réutilisation de données. Cet objectif n'est que partiellement atteint puisque la méthode donnée par Agarwal et al. n'est pas totalement constructive, et n'est illustrée qu'à l'aide d'exemples de cas particuliers.

Le but de ce chapitre est donc d'étendre les résultats d'Agarwal et al. : dans un premier temps (Paragraphe 3.2), nous résumons l'approche d'Agarwal, Kranz et Natarajan [1]. Ensuite, (Paragraphe 3.3), nous introduisons une nouvelle formulation de la taille de l'empreinte. Puis (Paragraphe 3.4), nous fournissant une heuristique résolvant le problème d'optimisation associé. Nous donnons plusieurs exemples dans le Paragraphe 3.5, puis nous formulons quelques remarques finales dans le Paragraphe 3.6.

3.2 Travaux d'Agarwal, Kranz et Natarajan

Dans ce paragraphe, nous résumons l'approche d'Agarwal, Kranz et Natarajan [1]. Nous utilisons exactement le même formalisme et les mêmes hypothèses que ces derniers.

[1] *footprint* en anglais

3.2.1 Pavage optimal pour minimiser les communications

Débutons avec l'exemple suivant :

Programme 6.

doall $i = 0, 99$
  doall $j = 0, 99$
    $A[i, j] = B[i, j] + B[i + 1, j - 2] + C[2i, 2i + j] + C[2i + 1, 2i + j]$

Supposons que les tableaux de données B et C soient stockés dans une unité de mémoire externe. Afin d'augmenter la granularité du calcul, et la localité des références, la technique du pavage peut être employée. Le nid de boucles ne présentant aucune dépendance interne, toute forme de tuile est acceptable. Prenons par exemple des tuiles rectangulaires de taille 10×5. On obtiendrait alors le code pavé suivant :

Programme 7. *Version pavée du Programme 6*

doall I=0,9
  doall J=0,19
    do i=0,9
      do j=0,4
        $A[I * 10 + i, J * 5 + j] = B[I * 10 + i, J * 5 + j]$
          $+B[I * 10 + i + 1, J * 5 + j - 2]$
          $+C[2(I * 10 + i), 2(I * 10 + i) + J * 5 + j]$
          $+C[2(I * 10 + i) + 1, 2(I * 10 + i) + J * 5 + j]$

Naturellement deux pavages différents ne mènent pas au même temps d'exécution : le volume et la forme des tuiles sont des paramètres importants qui doivent être déterminés. Le problème est ici le suivant : "on veut d'une part que l'ensemble des données utilisées par le calcul d'une tuile tienne dans le cache, et d'autre part que le volume de calcul correspondant soit maximal." En d'autres termes, étant donnée une limite sur le volume de données accédées par tuile, on cherche la tuile de volume maximum. Ce problème étant difficile, on lui préfère l'approche suivante [26, 55, 85, 92] : étant donnée une taille (ou volume) de tuile fixée, déterminer la forme de tuile qui minimise la quantité de données (accédées par tuile). C'est ce problème d'optimisation qui est traité par Agarwal et al. [1]. Notons que pour des tuiles non-singulières, comme a pu l'indiquer Irigoin dans sa thèse [56], la recherche de forme peut être traitée indépendamment de la taille.

3.2.2 Notations

Evaluer la quantité de données accédées par le calcul d'une tuile n'est pas un problème simple. Agarwal et al. [1] supposent que les références aux données sont des fonctions affines injectives des indices de boucles. Le cas échéant, le problème est beaucoup plus complexe. En contre-partie, il est inutile de considérer des espaces de données supérieurs à l'espace d'itération : il suffit de réduire les fonctions de référence au sous-espace de données effectivement utilisé. Dans ce cas, la réduction d'une fonction affine injective est une fonction inversible.

Ainsi, a partir de maintenant, nous ne considérons que des fonctions de référence inversibles.

Pour une référence isolée, l'empreinte d'une tuile (quantité de données chargées) correspond au volume de cette tuile ; lorsqu'il y a plusieurs références, mais chacune à un tableau différent, l'empreinte cumulée est la somme des empreintes séparées. *En revanche*, lorsque l'on a plusieurs références au même tableau (Comme pour B dans le Programme 6), l'empreinte cumulée doit être évaluée avec attention : en effet, une même donnée peut être utilisée par différentes références, et ne doit donc être comptabilisée qu'une fois. Ainsi, le Programme 6 ne contient qu'une seule dépendance de réutilisation : $(-1, 2)$ entre $B[i, j]$ et $B[i + 1, j - 2]$. En effet, les dépendances d'entrée des références $C[2i, 2i+j]$ et $C[2i+1, 2i+j]$ ne se recouvrent pas. La Figure 3.1 représente les empreintes de ces quatre références.

Afin de formuler le problème d'évaluation (ou approximation) de l'empreinte d'une tuile, nous devons tout d'abord introduire quelques notations :

1. *La tâche* "$A[i, j] = ...$" est représentée par le vecteur colonne $\overrightarrow{\imath} = \begin{pmatrix} i \\ j \end{pmatrix}$.
2. Comme les références aux données sont supposées être des fonctions affines des index de boucle $\overrightarrow{\imath}$, elles peuvent être représentées par l'expression $\mathbf{G}\overrightarrow{\imath} + \overrightarrow{a}$, où $\mathbf{G}$ est une matrice et $\overrightarrow{a}$ un vecteur de translation. Ainsi, dans le Programme 6,
   - `B[i,j]` peut être représentée par le couple $(\mathbf{G}_1, \overrightarrow{a_{1,1}}) = \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right)$
   - `B[i+1,j-2]` peut être représentée par le couple $(\mathbf{G}_1, \overrightarrow{a_{1,2}}) = \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix} \right)$
   - `C[2i,2i+j]` peut être représentée par le couple $(\mathbf{G}_2, \overrightarrow{a_{2,1}}) = \left(\begin{pmatrix} 2 & 0 \\ 2 & 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right)$
   - `C[2i+1,2i+j]` peut être représentée par le couple $(\mathbf{G}_3, \overrightarrow{a_{3,1}}) = \left(\begin{pmatrix} 2 & 0 \\ 2 & 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right)$

   Notez que nous avons utilisé le même nom G_1 pour la matrice représentant la référence $B[i, j]$ et pour la matrice représentant la référence $B[i+1, j-2]$. Ceci est lié au fait que les données accédées correspondant à ces deux références *s'intersectent*. En d'autres termes, il existe une dépendance de réutilisation entre ces deux référence. En contrepartie, nous avons utilisé des noms différents, pour caractériser les références $C[2i, 2i + j]$ et $C[2i + 1, 2i + j]$, ceci bien qu'elles correspondent au même tableau et à la même matrice $\begin{pmatrix} 2 & 0 \\ 2 & 1 \end{pmatrix}$: en effet, dans ce cas les données utilisées par ces deux références ne s'intersectent pas (cf Figure 3.1).
3. Le volume de données utilisées par le calcul d'une tuile, correspondant à une référence isolée, est appelé "*empreinte* de cette référence". Le volume de données correspondant à toutes les références cumulées est appelé "*empreinte cumulée* de ces références". L'exemple précédent montre que l'empreinte cumulée de plusieurs références ne correspond pas nécessairement à la somme des empreintes de toutes les références. Considérons le cas particulier, mais néanmoins important, où un tableau est accédé deux fois, par les références $(\mathbf{G}, \overrightarrow{a_1})$ et $(\mathbf{G}, \overrightarrow{a_2})$. En d'autres termes, il existe une dépendance de réutilisation de $(\overrightarrow{a_2} - \overrightarrow{a_1})$ entre ces deux références. Alors si $G^{-1}(\overrightarrow{a_2} - \overrightarrow{a_1})$ est à composantes entières, chaque couple d'empreintes à une intersection significative, et l'empreinte cumulée doit être évaluée attentivement. Notons que cela se produit, en particulier, à chaque fois qu'une donnée est accédée deux fois par la même matrice unimodulaire (car comme G^{-1} est à composantes entières, $G^{-1}(\overrightarrow{a_2} - \overrightarrow{a_1})$ est nécessairement à composantes entières) ; c'est le cas pour les deux premières références du Programme 6.

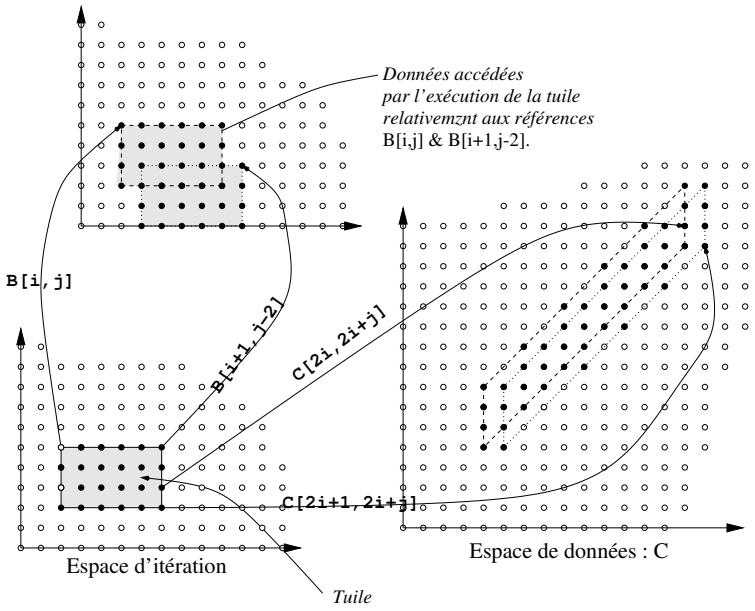


FIG. 3.1: *Cette figure représente une tuile de l'espace d'itérations (ici, un rectangle de taille* 6×4*) et ses données correspondantes utilisées. Les références* $B[i, j]$ *et* $B[i+1, j-2]$ *touchent des données communes qui ne doivent être comptabilisées qu'une fois. En contrepartie, les références* $C[2i, 2i+j]$ *et* $C[2i+1, 2i+j]$ *ont une intersection vide. En effet, l'indice* $2i$ *ne fait référence qu'aux lignes paires du tableau* C*, alors que l'indice* $2i+1$ *ne fait référence qu'aux lignes impaires.*

Dans tous les autres cas ($G^{-1}(\overrightarrow{a_2} - \overrightarrow{a_1})$ à une composante non-entière), les empreintes ont une intersection vide ou négligeable [1].

4. Une tuile est un hyper-parallélépipède déterminé par n vecteurs libres $\overrightarrow{u_1}, ..., \overrightarrow{u_n}$, où n est la profondeur du nid de boucles (nombre de boucles)—cf Figure 3.2 pour un exemple avec $n = 2$. Une tuile peut donc être représentée par une matrice H non singulière de taille $n \times n$, formée des vecteurs colonnes $\overrightarrow{u_1}, ..., \overrightarrow{u_n}$. Le *volume* de la tuile est alors $|\det(H)|$. En fait, cette matrice correspond exactement à la matrice P définie par Irigoin et Triolet [57], inverse de la matrice définie à partir des vecteurs normaux aux faces de la tuile. Considérons la référence ($\mathbf{G}$, $\overrightarrow{a}$) où $\mathbf{G}$ est unimodulaire, ainsi que la tuile représentée par la matrice $\mathbf{H}$: la tuile correspondant aux données accédées peut être représentée par la matrice $\mathbf{GH}$ et par le vecteur de translation $\mathbf{G}\overrightarrow{a}$. Ainsi, l'empreinte vaudrait dans ce cas $|\det(GH)| = |\det(H)|$ (car $\mathbf{G}$ est unimodulaire).

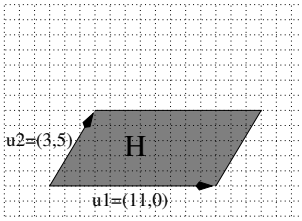


FIG. 3.2: *La tuile peut être représentée par la matrice* $H = \begin{pmatrix} 11 & 3 \\ 0 & 5 \end{pmatrix}$*. Sa taille vaut* $|\det(H)| = 55$.

Le premier objectif d'Agarwal, Kranz et Natarajan [1] a été de trouver une évaluation précise

de l'empreinte cumulée. Ensuite, fixant la taille de tuile, ils ont cherché la forme qui minimise leur expression d'empreinte cumulée.

Remarquons que la taille de l'empreinte cumulée des références $(\mathbf{G}, \overrightarrow{a_1}), \ldots, (\mathbf{G}, \overrightarrow{a_i})$ est identique à la taille de l'empreinte cumulée des références $\left(\frac{\mathbf{G}}{\det G}, \frac{\overrightarrow{a_1}}{\det G}\right), \ldots, \left(\frac{\mathbf{G}}{\det G}, \frac{\overrightarrow{a_i}}{\det G}\right)$.

Ainsi, à partir de maintenant nous ne considérons que des références unimodulaires.

3.2.3 Résultats

Comme souligné précédemment, la difficulté réside en l'estimation de l'empreinte cumulée de plusieurs références à une donnée commune (cf Figure 3.3). La solution proposée par Agarwal et al. [1] est la suivante : considérons les références $(G, \overrightarrow{a_1}), ..., (G, \overrightarrow{a_k})$, où G est unimodulaire.

- Notons par $(\overrightarrow{x_1}, ..., \overrightarrow{x_n})$ la base canonique de $\mathbb{R}^n$, et par $\overrightarrow{a}$ "*l'écartement*[2]" entre les k vecteurs $\overrightarrow{a_1}, ..., \overrightarrow{a_k}$:
$$\overrightarrow{a} = \left(\max_{1 \leq j < j' \leq k} | < \overrightarrow{x_i}, \overrightarrow{a_j} - \overrightarrow{a_{j'}} > | \right)_{1 \leq i \leq n}$$
où $< \overrightarrow{x}, \overrightarrow{y} >$ représente le produit scalaire des vecteurs $\overrightarrow{x}$ et $\overrightarrow{y}$. Intuitivement, les composantes de $\overrightarrow{a}$ correspondent au décalage maximal, entre les différents vecteurs, dans chacune des différentes directions.
- Si $D = (\overrightarrow{d_1} \cdots \overrightarrow{d_n})$ est une matrice de taille $n \times n$ composée des n vecteurs colonnes $\overrightarrow{d_i}$, alors $\det(D_{j \to \overrightarrow{a}})$ représente le déterminant de la matrice $D_{j \to \overrightarrow{a}}$ obtenue en remplaçant la j-ième colonne de D par $\overrightarrow{a}$.

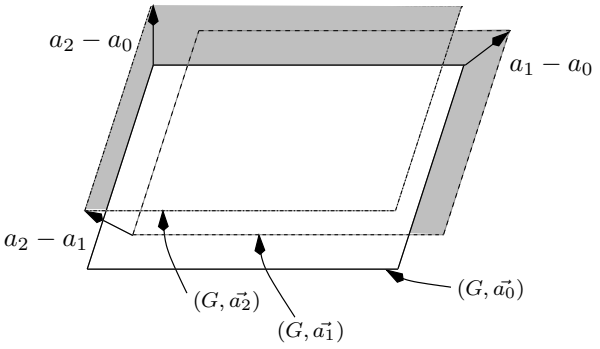


FIG. 3.3: *Empreinte cumulée de trois références à un tableau commun : $(G, \overrightarrow{a_0})$, $(G, \overrightarrow{a_1})$ et $(G, \overrightarrow{a_2})$. Si V_{calc} représente le volume de la tuile, V_{com} le volume de la partie grisée, alors l'empreinte cumulée de ces trois références est $V_{calc} + V_{com}$. Le but est de trouver une bonne approximation pour l'expression de V_{com}.*

Ainsi, comme le montrent intuitivement les figures 3.3 et 3.4, l'empreinte cumulée des références $(G, \overrightarrow{a_1}), ..., (G, \overrightarrow{a_k})$ peut être approchée par

$$|\det(D)| + \sum_{j=1}^{n} |\det(D_{j \to \overrightarrow{a}})|, \quad \text{où } D = GH.$$

Avec les notations de la Figure 3.3, puisque G est unimodulaire, $V_{calc} = |\det(D)| = |\det(H)|$ représente le volume de tuile (volume de données d'une seule référence), et V_{com} $(= \sum_{j=1}^{n} |\det(D_{j \to \overrightarrow{a}})|)$

[2] *spread* en anglais

représente le volume de données supplémentaire (lié à la présence d'autres références). On utilise le nom intuitif V_{com} car la partie grisée de la Figure 3.3 correspondrait à des communications inter-processeurs dans le contexte du pavage de nid de boucles sur machines à mémoire distribuée [19, 24], alors que cela correspond à des accès mémoire dans le contexte présent.

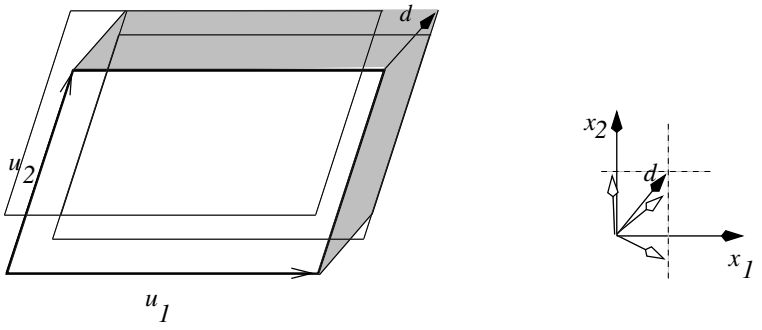


FIG. 3.4: *Si* $\overrightarrow{d} = (d_i)_{1\leq i\leq n}$, d_i *est obtenu en prenant les valeurs maximales de* $|<\overrightarrow{x_i}, \overrightarrow{a_k} - \overrightarrow{a_l}>|$ *où* $\overrightarrow{x_i}$ *correspond au vecteur colonne* $(\delta_{ij})_{1\leq j\leq n}$ *(base canonique). Avec ces notations,* V_{com} *peut être approché par* $|\det(\overrightarrow{u_1}, \overrightarrow{d})| + |\det(\overrightarrow{u}_2, \overrightarrow{d})|$.

En fait, l'approximation d'Agarwal, Kranz et Natarajan [1] est plus élaborée : ils utilisent l'expression précédente seulement quand la matrice D définit un parallélépipède aux arêtes parallèles aux axes de la base canonique $(\overrightarrow{x_1}, ..., \overrightarrow{x_n})$. En général, ce n'est pas le cas, et l'expression d'écartement $\overrightarrow{a}$ doit être adaptée en conséquence. Pour cela, ils utilisent un changement de base afin de prendre en compte les directions de D. Formalisons cette démarche : on définit pour chaque référence $\overrightarrow{a_j}$, $1 \leq j \leq k$ le vecteur $\overrightarrow{a'_j} = D^{-1}\overrightarrow{a_j}$ associé. Maintenant que les références sont "normalisées", on considère l'écartement $\overrightarrow{a'} = \left(\max_{1\leq j<j'\leq k} |<\overrightarrow{x_i}, \overrightarrow{a'_j} - \overrightarrow{a'_{j'}}>|\right)_{1\leq i\leq n}$. On pose donc finalement $\overrightarrow{a} = D\overrightarrow{a'}$, et c'est cette valeur que l'on utilise dans la formule de $V_{com} = \sum_{j=1}^{n} |\det(D_{j\rightarrow\overrightarrow{a}})|$.

En résumé, le problème d'optimisation lié aux références $(G, \overrightarrow{a_1}), ..., (G, \overrightarrow{a_k})$, où G est unimodulaire, s'écrit de la manière suivante :

$$\textbf{Trouver } H \textbf{ qui minimise } \; |\det(D)| + \sum_{i=1}^{n} |\det(D_{i\rightarrow\overrightarrow{a}})|$$

$$\text{avec} \begin{cases} |\det D| = V_{calc} \text{ est fixé} \\ D = GH \\ \overrightarrow{a} = D\overrightarrow{a'} \\ \overrightarrow{a'} = \left(\max_{1\leq j<j'\leq k} |<\overrightarrow{x_i}, \overrightarrow{a'_j} - \overrightarrow{a'_{j'}}>|\right)_{1\leq i\leq n} \\ \overrightarrow{a'_j} = D^{-1}\overrightarrow{a_j}, 1 \leq j \leq k \end{cases} \qquad \text{(CFP)}$$

S'il y a plusieurs matrices d'accès à considérer, on somme les empreintes cumulées correspondant à chaque matrice. L'approximation (CFP) est assez générale et précise, puisqu'elle est valide pour des tuiles de forme parallélépipède arbitraires. Malheureusement, Agarwal, Kranz et Natarajan [1] ne résolvent pas ce problème d'optimisation de manière générale. Cela est dû au fait que D et donc l'écartement $\overrightarrow{a}$ sont inconnus. Agarwal, Kranz et Natarajan [1] résolvent ce problème dans un cadre plus restrictif : ils limitent leur recherche à des matrices H diagonales, c'est à dire au cas où les tuiles sont rectangulaires. Dans ce cas, la forme de H est fixée à n coefficients homothétiques

près (la longueur de chaque direction), et la recherche est fortement réduite, comme expliqué dans le Paragraphe 3.4.1. Mais même dans ce cas, les auteurs ne donnent aucune expression analytique ; seuls quelques exemples spécifiques sont résolus à la main.

Dans le paragraphe suivant, nous étendons les résultats d'Agarwal, Kranz et Natarajan [1] : grâce à une nouvelle formulation de leur expression de l'empreinte cumulée,

- nous résolvons analytiquement le problème d'optimisation réduit à la recherche de tuiles de forme rectangulaire,
- nous donnons une heuristique générale pour la résolution du problème d'optimisation à la recherche de tuiles de forme parallélépipède, c'est à dire pour des matrices H quelconques.

3.3 Nouvelle expression de l'empreinte cumulée

L'expression du problème d'optimisation (CFP) ne permet pas de déduire un algorithme donnant la meilleure forme de tuile pour un ensemble de références donné. Elle permet juste de résoudre le problème à la main, dans les cas simples, comme cela est fait dans [1]. Le but de ce paragraphe est de reformuler l'Expression (CFP) à l'aide de manipulations algébriques classiques afin d'obtenir une expression utilisable.

Nous avons besoin pour cela du résultat d'algèbre linéaire suivant [58] :

Lemme 2 *Si A est une matrice de taille $n \times n$, non singulière et $\overrightarrow{u}$ un vecteur quelconque de dimension n, alors*

$$\det(A_{j\rightarrow\overrightarrow{u}}) = \det(A)\times < \overrightarrow{b_j}, \overrightarrow{u} >$$

où $\overrightarrow{b_j}$ représente la j-ième ligne de A^{-1}.

Considérons la formule (CFP) du Paragraphe 3.2.3.
Avec les mêmes notations, on a $\overrightarrow{a'} = \left(\max_{j<j'} | < \overrightarrow{x_i}, \overrightarrow{a'_j} - \overrightarrow{a'_{j'}} > |\right)_{1\leq i\leq n}$.
Soit, $\overrightarrow{a'_j} = D^{-1}\overrightarrow{a_j} = H^{-1}(G^{-1}\overrightarrow{a_j}) = H^{-1}\overrightarrow{b_j}$, où $\overrightarrow{b_j} = G^{-1}\overrightarrow{a_j}$.
Ainsi,

$$\begin{aligned} < \overrightarrow{x_i}, \overrightarrow{a'_j} - \overrightarrow{a'_{j'}} > &= < \overrightarrow{x_i}, H^{-1}(\overrightarrow{b_j} - \overrightarrow{b'_{j'}}) > \\ &= < H^{-T}\overrightarrow{x_i}, \overrightarrow{b_j} - \overrightarrow{b'_{j'}} > \\ &= < \overrightarrow{e_i}, \overrightarrow{b_j} - \overrightarrow{b'_{j'}} > \end{aligned}$$

où $\overrightarrow{e_1}, ..., \overrightarrow{e_n}$ sont les vecteurs lignes de la matrice $E = H^{-1}$.

Notons I_n la matrice identité de dimension n.

L'expression de l'empreinte cumulée $V = |\det(D)| + \sum_{i=1}^{n} |\det(D_{i\rightarrow\overrightarrow{a}})|$ devient donc

$$V = |\det(D)|(1 + \sum_{i=1}^{n} |\det(D^{-1}).\det(D_{i\rightarrow\overrightarrow{a}})| = |\det(D)|(1 + \sum_{i=1}^{n} |\det((I_n)_{i\rightarrow\overrightarrow{a'}})|.$$

Utilisons le Lemme 2, $V = |\det(D)| \left(1 + \sum_{i=1}^{n} | < \overrightarrow{x_i}, \overrightarrow{a'} > |\right)$.
Mais,

$$| < \overrightarrow{x_i}, \overrightarrow{a'} > | = \max_{j<j'} | < \overrightarrow{x_i}, \overrightarrow{a'_j} - \overrightarrow{a'_{j'}} > | = \max_{j<j'} | < \overrightarrow{e_i}, \overrightarrow{b_j} - \overrightarrow{b_{j'}} > |.$$

Ainsi, minimiser l'expression (CFP) correspond à trouver une matrice E non singulière telle que $|\det(E^{-1})| = |\det(H)| = V_{calc}$ et qui minimise $V = V_{calc} + V_{com}$, où

$$V_{com} = V_{calc}.\sum_{i=1}^{n} \max_{j<j'} \left| < \vec{e_i}, \vec{b_j} - \vec{b_{j'}} > \right|.$$

Jusqu'à présent, nous avons considéré l'accès avec une matrice commune G unimodulaire. Si l'on a m matrices G_i unimodulaires distinctes, posons $\overrightarrow{b_{i,j}} = G_i^{-1}\overrightarrow{a_{i,j}}$ et $\overrightarrow{c_{i,j}}$ les éléments de $C_i = \{\overrightarrow{b_{i,j}} - \overrightarrow{b_{i,j'}},\ j \neq j'\}$ L'expression à minimiser devient

$$\begin{aligned} V &= \sum_{i=1}^{m} \left(V_{calc} + V_{calc}.\sum_{k=1}^{n} \max_{j<j'} \left(| < \vec{e_k}, \overrightarrow{b_{i,j}} - \overrightarrow{b_{i,j'}} > | \right) \right) \\ &= mV_{calc} + V_{calc}.\sum_{k=1}^{n}\sum_{i=1}^{m} \max_{j\neq j'} < \vec{e_k}, \overrightarrow{b_{i,j}} - \overrightarrow{b_{i,j'}} > \\ &= mV_{calc} + V_{calc}.\sum_{k=1}^{n}\sum_{i=1}^{m} \max_{j} < \vec{e_k}, \overrightarrow{c_{i,j}} > \end{aligned}$$

La somme peut être décalée à l'intérieur du max, au prix d'une augmentation notable du nombre de termes du max : pour $i \in [1, m]$ on note d_i le nombre de vecteurs $\overrightarrow{c_{i,j}}$, $1 \leq j \leq d_i$. On pose

$$\sigma : \left(\begin{array}{rcl} \{1, .., m\} & \longrightarrow & \mathbb{N} \\ i & \longmapsto & 1 \leq j \leq d_i \end{array} \right)$$

et

$$\overrightarrow{l_\sigma} = \sum_{i=1}^{m} \overrightarrow{c_{i,\sigma(i)}}$$

Alors

$$V = mV_{calc} + V_{calc}.\sum_{k=1}^{n} \max_{\sigma} < \vec{e_k}, \overrightarrow{l_\sigma} > \tag{3.1}$$

3.4 Résolution du problème d'optimisation

La nouvelle expression de (CFP) ne permet pas de donner une solution analytique dans la configuration la plus générale. En revanche, nous pouvons,

- donner une expression analytique de la solution dans le cas restreint de tuiles de forme rectangulaire,
- utiliser l'expression (3.1) afin de construire une heuristique au problème général avec des tuiles de forme arbitraire.

Commençons par un exemple afin de montrer le gain potentiel (théorique) lié à l'utilisation de tuiles de forme parallélépipédique plutôt que simplement rectangulaires.

Programme 8. *Exemple de code nécessitant un pavage de forme parallélépipède*

```
do i=0,N
    do j=0,M
        A[i,j] = B[i,j] + B[i+1,j+1]
```

Supposons que l'on veuille avoir $V_{calc} = 100$. Alors la tuile rectangulaire qui minimise l'expression $|\det(H)| + |\det(H_{1\to\overrightarrow{a}})| + |\det(H_{2\to\overrightarrow{a}})|$ avec $\overrightarrow{a} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ est une tuile de forme carrée : $H = 10.I_2 = \begin{pmatrix} 10 & 0 \\ 0 & 10 \end{pmatrix}$. Cette tuile (cf Figure 3.5) conduit à $V_{com} = 19$. Alors que la tuile de forme parallélépipède : $H' = \frac{1}{\sqrt{2}} \begin{pmatrix} 100 & 1 \\ 100 & -1 \end{pmatrix}$ conduirait à $V_{com} = 1$.

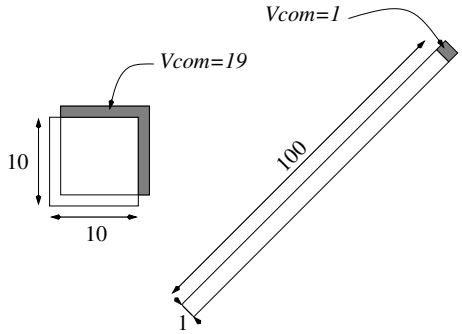


FIG. 3.5: *Comparaison des empreintes cumulées des tuiles H et H' pour un volume commun valant $V_{calc} = 100$.*

Utiliser une tuile parallélépipédique au lieu d'une tuile rectangulaire risque de créer un surcoût lors de la génération de code ou même de l'exécution. Ce surcoût étant fortement dépendant de l'application, il est souhaitable de tester et comparer les deux solutions dans chaque configuration.

3.4.1 A la recherche de tuiles rectangulaires

Avant d'introduire une heuristique pour résoudre le problème d'optimisation dans le cadre général, nous donnons une solution analytique dans le cas de tuiles de forme rectangulaire.

Considérons pour cela la nouvelle formulation de (CFP) pour les références $(G, \overrightarrow{a_1}), ..., (G, \overrightarrow{a_k})$ où G est unimodulaire : $V = V_{calc}\left(1 + \sum_{i=1}^{n} \max_{j<j'} \left| < \overrightarrow{e_i}, \overrightarrow{b_j} - \overrightarrow{b_{j'}} > \right|\right)$. Supposons que l'on restreigne notre recherche aux matrices diagonales $H = \mathrm{diag}(h_1, \cdots, h_n)$, où $\det(H) = \prod_{i=1}^{n} h_i = V_{calc}$. On a $\overrightarrow{e_i} = \frac{1}{h_i}\overrightarrow{x_i}$, d'où $V = V_{calc}(1 + \sum_{i=1}^{n} \frac{1}{h_i} c_i)$, où $c_i = \max_j b_{ij} - \min_j b_{ij}$ (b_{ij} représente la i-ième composante de $\overrightarrow{b_j}$). En d'autres termes, il faut minimiser $\sum_{i=1}^{n} \frac{c_i}{h_i}$ tel que $\prod_{i=1}^{n} h_i = V_{calc}$. La solution est alors

$$h_i = c_i \times \frac{V_{calc}}{\prod_{j=1}^{n} c_j}.$$

Cette solution s'étend directement au cas où l'on a plusieurs matrices d'accès (on a la même expression, où c_i est maintenant la somme de toutes les contributions dans la direction i). Cette notre nouvelle formulation de (CFP) permet donc de donner une solution analytique au problème d'optimisation introduit par Agarwal, Kranz et Natarajan [1].

3.4.2 Problèmes connexes

La minimisation de l'Expression (3.1) Page 44 est un problème difficile. En fait, nous savons résoudre les problèmes connexes suivant :

Problème 1 Si $(\overrightarrow{a_1}, ..., \overrightarrow{a_m})$ sont m vecteurs libres, et $|\det E| = \frac{1}{V_{calc}}$, Boulet et al. [19] proposent une solution au problème de minimisation de l'expression suivante :

$$\sum_{i=1}^{m} \sum_{k=1}^{n} < \overrightarrow{e_k}, \overrightarrow{a_i} >$$

La solution est simple ($E = A^{-1}$) si $m = n$ mais devient très complexe si $m \neq n$, avec un coût exponentiel en m. Notons que les $\overrightarrow{a_i}$ représentent des dépendances dans le contexte du pavage de nids de boucles totalement permutables, donc de dépendances aux composantes non négatives, propriété qui n'est pas vérifiée ici.

Problème 2 Notons $\mathcal{H}_n$ la matrice de Hadamard de taille n, c'est à dire une matrice carré composée de coefficients valant 0 ou 1 et de déterminant maximal [22]. Si $A = (\overrightarrow{a_1} \cdots \overrightarrow{a_n})$ est non singulière, alors $E = \mathcal{H}_n A^{-1}$ minimise l'expression (cf [24]) :

$$\sum_{k=1}^{n} \max_{1 \leq j \leq n} < \overrightarrow{e_k}, \overrightarrow{a_j} >$$

De nouveau, si A n'est pas carré, le problème devient très difficile et les seules solutions connues ont un coût exponentiel [24].

Problème 3 Si N est la taille de l'espace d'itération, Irigoin [56] formalise le problème d'évaluation des volumes référencés dans un tableau de dimension $N - P$. Il exprime analytiquement cette valeur pour une référence et $P = N - 1$ ou $P = 1$ (cas dual). Le volume de stockage nécessaire et et le volume référencé sont différenciés. Différents cas sont discutés en fonction de l'ordonnancement et de la machine cible.

Grossièrement, notre problème se situe entre le Problème 1 et le Problème 2. Pour éviter un coût exponentiel, nous introduisons une heuristique qui s'inspire de la solution du Problème 2 donnée par Calland et al. [24]. Dans un premier temps, nous réduisons le problème au sous-espace vectoriel généré par l'ensemble des vecteurs $\{\overrightarrow{l_\sigma}\}$; notons n' la dimension de ce sous-espace. Ensuite, nous choisissons n' vecteurs libre dans $\{l_\sigma\}$ (base du sous-espace). Nous voulons que ces vecteurs soient le plus représentatif possible de la famille initiale. Pour cela, nous proposons de choisir n' vecteurs qui maximisent (ou presque) le volume du polytope qu'ils définissent (notez que c'est une heuristique dans l'heuristique). Ensuite, nous résolvons ce problème en utilisant la solution du Problème 2 appliquée au sous-espace de vecteurs ainsi choisis (on est dans le cas particulier d'une matrice carrée). Finalement, pour les directions restantes (espace complémentaire), nous choisissons des vecteurs orthonormaux.

3.4.3 Heuristique

Pour décrire notre heuristique, nous avons besoin d'introduire préalablement quelques notations :

- $L = (\overrightarrow{l_1} \cdots \overrightarrow{l_m})$ est une matrice constituée de m vecteurs colonnes de taille n.
- $\mathcal{H}_n$ est la matrice de Hadamard de dimension n [22].

- Si $D = (\overrightarrow{d_1} \cdots \overrightarrow{d_m})$ est une matrice rectangulaire constituée de m vecteurs de taille n qui génére un espace vectoriel de dimension n, alors $C_{maxvol}(D)$ est une sous-matrice de D de taille $n \times n$ de déterminant maximum.
- Le rang de la matrice D est noté par rang(D).
- Si D est une matrice composée de $m \geq n$ vecteurs colonne de dimension n générant un espace vectoriel de dimension n, alors $O_{GS}(D)$ est la matrice de taille $n \times n$ constituée de n vecteurs orthonormés obtenu par l'orthonormalisation de Gram-Schmidt des vecteur $(\overrightarrow{d_1}, ..., \overrightarrow{d_m})$.
- Si C est une matrice, alors $up_n(C)$ est la sous-matrice de C constituée des n premières lignes de C.

La solution approchée du problème d'optimisation (3.1) est donnée par l'algorithme suivant :

Algorithme 2. *Renvoie la valeur de H minimisant l'Expression 3.1*

Procédure H(V_{calc}, L)
$O = O_{GS}(L, I_n)$
$r =$ rang(D)
$C = up_r(O^T L)$
$P = C_{maxvol}(C)\mathcal{H}_r$
$P = \frac{V_{calc}}{|\det P|^{\frac{1}{r}}} P$
$P = \begin{pmatrix} P & 0 \\ 0 & I_{n-r} \end{pmatrix}$
$H = OP$
Calcule V_{com} pour H
Calcule V_{com} pour la meilleure matrice diagonale H_{diag} (Paragraphe 3.4.1)
Retourne meilleure solution entre H et H_{diag}

En comparant le résultat de notre heuristique avec celui de la meilleure tuile rectangulaire (comme montré dans le Paragraphe 3.4.1), nous sommes sûr de trouver une solution avec un volume de communication au moins aussi petit que celui obtenu par Agarwal, Kranz et Natarajan [1]. Notons néanmoins que notre heuristique donne, dans le cas particulier du Programme 8 du Paragraphe 3.4 la solution optimale.

Heuristique pour trouver une famille génératrice de volume maximal. Nous donnons une description sommaire de l'heuristique dans l'heuristique. Etant donné une famille de m vecteurs générant un sous-espace vectoriel de dimension n, trouver une sous-famille génératrice de volume maximal peut être fait en comparant le volume de chaque sous-famille de dimension n. Il y a $\begin{pmatrix} m \\ n \end{pmatrix} = \frac{m!}{n!(m-n)!}$ sous-familles, ce qui rend cette méthode inutilisable si m est grand. Ainsi, nous proposons un algorithme glouton : nous commençons par le plus grand vecteur. Puis, itérativement, nous ajoutons un vecteur à la sous-famille construite, tel que le volume correspondant dans le sous-espace généré soit maximisé. Quand la famille construite contient n vecteurs libres, nous essayons d'échanger un de ces vecteurs par un autre ; si cette permutation augmente le volume, nous l'honorons. Nous continuons le processus jusqu'à ce qu'aucun échange ne puisse plus augmenter le volume.

Le volume de p vecteurs de dimension n quand $p < n$ peut être calculé en utilisant la matrice de Gram : si D est une matrice composée des vecteurs colonne $(\overrightarrow{d_1}, ..., \overrightarrow{d_p})$, $Gram(\overrightarrow{d_1}, ..., \overrightarrow{d_p}) = D^T D$.

Alors, le volume du polytope généré par $(\overrightarrow{d_1}, ..., \overrightarrow{d_m})$ vaut $\sqrt{\det(D^T D)}$.

L'algorithme correspondant est l'Algorithme 9 décrit ci-dessous. Dans cette procédure, nous notons par B_i la i-ième colonne de la matrice B, et par $[B, C]$ la concaténation horizontale des deux matrices B et C.

Programme 9. *Heuristique pour trouver une famille génératrice de volume maximum.*

Procédure C_{maxvol}(U)
$B = 0$
for $i = 1, n$
    $B = [B, U_1]$
    $j = 1$
    $V_{max} = Volume(B)$
    $h = m - i + 1$
    for $k = 2, h$
        $B_i = U_k$
        $V = Volume(B)$
        if $V > V_{max}$
            $V_{max} = V$
            $j = k$
        $B_i = U_j$
        $U_j \leftrightarrow U_h$
        $h = h - 1$
$may.increase$ = True
while $may.increase$ do
    $E = B^{-1}$
    $V_{max} = 1$
    $may.increase$ = False
    for $k = 1, h$
        for $i = 1, n$
            $V = |E_i^T.U_k|$
            if $V > V_{max}$
                $V_{max} = V$
                $j = k$
                $l = i$
                $may.increase$ = True
    if $may.increase$
        $B_l = U_j$
        $U_j \leftrightarrow U_{m-l+1}$
Retourne(B)

3.5 Exemples

Pour évaluer la qualité de notre heuristique, nous l'appliquons tout d'abord sur les deux exemples donnés par Agarwal et al. [1], puis sur un jeu d'exemples générés aléatoirement. Notre but est de

mesurer le gain apporté par notre heuristique en comparaison avec la meilleure solution de forme rectangulaire. Dans les deux paragraphes suivant, nous comparons donc le volume de communication obtenu en limitant la recherche à des tuiles de forme rectangulaires, à celui obtenu par notre heuristique. Pour chaque solution, nous calculons le volume *exact* de communications, afin d'examiner la précision de l'approximation.

3.5.1 Premier exemple

Cet exemple correspond à "l'Exemple 7" de l'article d'Agarwal et al. :

Programme 10.

```
doall i = 1, N
    doall j = 1, N
        doall k = 1, N
            A[i,j,k] = B[i-1,j,k+1] + B[i,j+1,k] + B[i+1,j-2,k-3]
```

Avec nos notations, il y a ici une seule matrice d'accès $G_1 = I_2$, $a_{1,1} = (-1,0,1)^T$, $a_{1,2} = (0,1,0)^T$ et $a_{1,3} = (1,-2,-3)^T$. Comme $G_1 = I_2$ et $b_{1,j} = a_{1,j}$, pour une valeur donnée de V_{calc}, l'expression à minimiser est

$$V_{calc}\left(1+\sum_{i=1}^{3}\max\left(|<\overrightarrow{e_i},(-1,-1,1)^T>|,|<\overrightarrow{e_i},(-2,2,4)^T>|,|<\overrightarrow{e_i},(-1,3,3)^T>|\right)\right)$$

où la matrice E vérifie $|\det(E)| = \frac{1}{V_{calc}}$.
Notre algorithme conduit à la solution $H = E^{-1} = \left(\overrightarrow{h_1},\overrightarrow{h_2},\overrightarrow{h_3}\right)$, où

$$\left(\overrightarrow{h_1},\overrightarrow{h_2}\right) = \sqrt{V_{calc}}\begin{pmatrix} -0.7331 & -0.3656 \\ 0.7331 & 1.0967 \\ 1.4622 & 1.0967 \end{pmatrix} \text{ and } \overrightarrow{h_3} = \begin{pmatrix} -0.8018 \\ 0.2673 \\ -0.5345 \end{pmatrix}.$$

Dans ce cas, si l'on prend $V_{calc} = 1000$ (par exemple), on obtient

$$\begin{aligned} V_{com} &= V_{calc}\sum_{i=1}^{3}\max\left(|<\overrightarrow{e_i},(-1,-1,1)^T>|,|<\overrightarrow{e_i},(-2,2,4)^T>|,|<\overrightarrow{e_i},(-1,3,3)^T>|\right) \\ &= 173 \end{aligned}$$

Alors que la solution donnée par Agarwal et al. [1] est $H = \sqrt[3]{\frac{V_{calc}}{24}}\begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{pmatrix}$

L'importance du gain (Table 3.1) entre notre solution et celle d'Agarwal et al. [1] est lié au fait que dans cet exemple le sous-espace vectoriel généré par les vecteurs $(b_{i,j})_{i,j}$ est un sous-espace vectoriel de dimension 2 alors que l'espace vectoriel initial ($\mathbb{R}^3$) est de dimension 3. Notre algorithme est capable de tenir compte de cette information en résolvant d'abord le problème dans le sous-espace puis en généralisant la solution à l'espace tout entier.

	Approximation (CFP)	Valeur exacte
Solution d'Agarwal et al.	865	756
Solution de notre heuristique	173	173

TAB. 3.1: *Table de comparaison pour l'Exemple 7.*

3.5.2 Second exemple

Cet exemple correspond à "l'Exemple 8" présenté dans [1] :

Programme 11.

doall $i = 1, N$
  doall $j = 1, N$
    $A[i,j] = B[i-2,j] + B[i,j-1] + C[i+j-1,j] + C[i+j+1,j+3]$

Avec nos notations il y a deux matrices d'accès $G_1 = I_2$ et $G_2 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.
On a $a_{1,1} = (-2,0)^T$, $a_{1,2} = (0,-1)^T$, $a_{2,1} = (-1,0)^T$ et $a_{2,2} = (1,3)^T$.
Ainsi, $b_{1,j} = a_{1,j}$, $b_{2,1} = (-1,0)^T$ et $b_{2,2} = (-2,3)^T$.
D'où, $c_{1,1} = (-2,1)^T$, $c_{2,1} = (1,-3)^T$ et $c_{2,2} = (-1,3)^T$.
Alors $l_1 = (-1,-2)$ et $l_2 = (-3,4)$.

L'expression à minimiser est donc

$$V_{calc}\left(1 + \sum_{i=1}^{2} \max\left(| < \overrightarrow{e_i}, (-1,-2)^T > |, | < \overrightarrow{e_i}, (-3,4)^T > |\right)\right)$$

Notre algorithme conduit à :

$$H = \sqrt[2]{V_{calc}}. \begin{pmatrix} -0.9487 & -0.3162 \\ 1.2649 & -0.6325 \end{pmatrix}$$

Et, si l'on prend $V_{calc} = 1000$, alors $V_{com} = 195$.

D'autre part, la solution donnée par Agarwal et al. est

$$H = \sqrt[2]{\frac{V_{calc}}{12}}. \begin{pmatrix} 3 & 0 \\ 0 & 4 \end{pmatrix},$$

ce qui conduit à $V_{com} = 214$ (pour également $V_{calc} = 1000$).

3.5.3 Exemples générés aléatoirement

Dans ce paragraphe, nous générons un jeu de 10000 tests avec les caractéristiques suivantes : le problème est de dimension 3 ($n = 3$) ; il y a une unique matrice commune unimodulaire G ($G = Id_3$ sans perte de généralités) ; il y a entre 2 et 5 vecteurs de référence $\overrightarrow{a_j}$. Les composantes des vecteurs de référence sont générées selon une distribution normale de moyenne 0 et de variance

	Approximation (CFP)	Valeur exacte
Solution d'Agarwal et al.	219	214
Solution donnée par notre heuristique	200	195

TAB. 3.2: *Table de comparaison pour l'Exemple 8.*

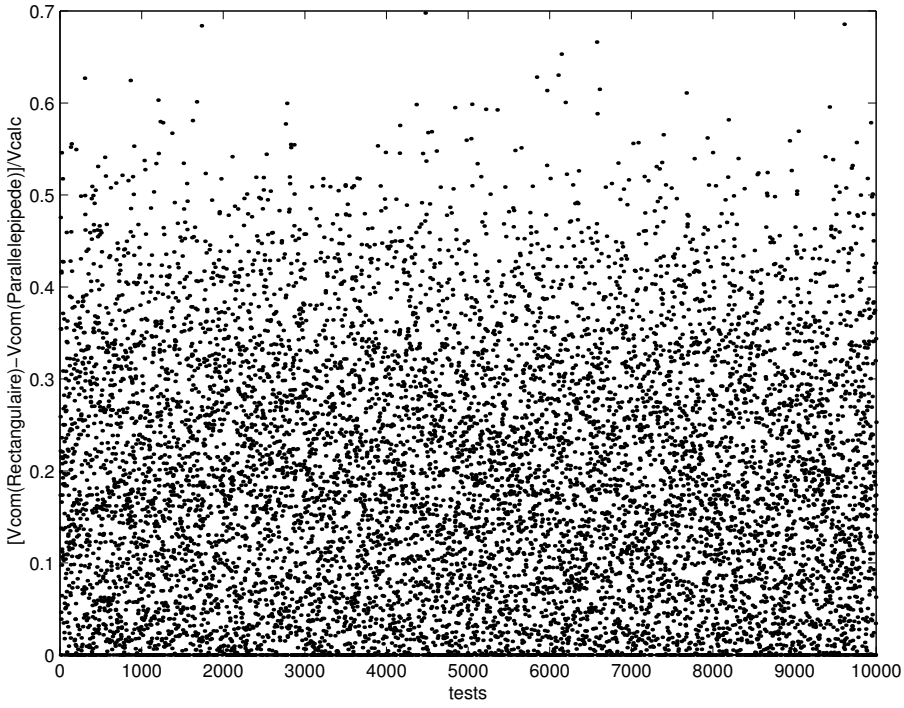


FIG. 3.6: *Distribution (densité) des résultats des expériences.*

5 (nous avons utilisé la fonction `randn` de Matlab), puis arrondie pour obtenir des valeurs entières. Le volume d'une tuile est fixé à $V_{calc} = 256000$.

Nous donnons les résultats, suivant deux points de vue, Figure 3.6 et Figure 3.7. Dans chaque figure, nous avons reporté la quantité

$$\frac{V_{com}(\text{Rectangulaire}) - V_{com}(\text{Parallélépipède})}{V_{calc}},$$

où V_{com}(Rectangulaire) est le volume de communications obtenu pour la meilleure tuile rectangulaire, et V_{com}(Parallélépipède) est le volume de communications obtenu par notre heuristique. Le gain moyen est égal à $0.18 \times V_{calc}$.

3.6 Conclusion

Dans ce chapitre, nous avons présenté un travail améliorant les résultats de pavage d'Agarwal, Kranz et Natarajan [1]. Nous avons reformulé leur évaluation de l'empreinte cumulative, permettant de ce fait la dérivation d'une solution analytique pour les tuiles rectangulaires. Nous avons également

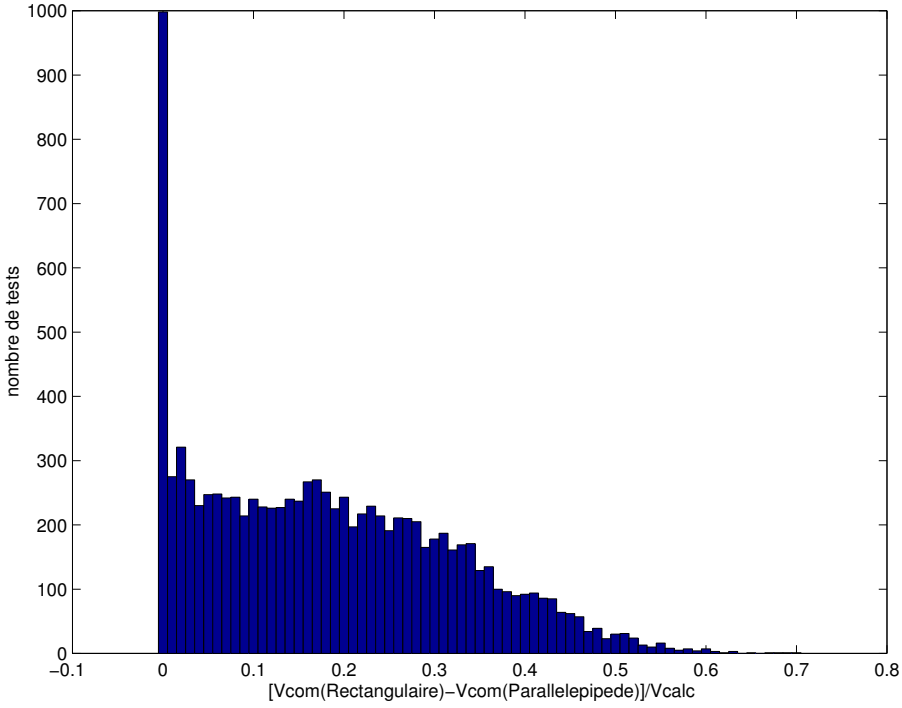


FIG. 3.7: *Distribution (histogramme) des résultats des expériences.*

proposé une heuristique pour résoudre le problème d'optimisation général (sans pour autant réduire rigoureusement l'espace de recherche). Cette heuristique est inspirée de résultats récents dans le domaine du pavage. Toutes les formules utilisées étant multi-linéaires, il est alors aisé d'utiliser le résultat obtenu pour la résolution du problème d'optimisation suivant : étant donné un volume de données limité, trouver la forme de tuile de volume de calcul maximal. Remarquons que le résultat s'applique aussi au problème de pavage dans le cadre de circuits VLSI [69] où la bande passante est limitée et où il faut minimiser le rapport du nombre de données rapatriées sur la quantité de calcul effectués.

Plusieurs améliorations peuvent être apportées à notre heuristique, et un certain nombre d'autres résultats expérimentaux en plus des exemples et des cas aléatoires traités ici seraient nécessaires pour évaluer entièrement l'utilité de notre approche.

Il faut par ailleurs noter que notre évaluation de l'empreinte n'est pas tout à fait générale : en effet, si la matrice G est singulière, l'ensemble des données accédées par une référence isolée est généralement un polytope plus général qu'un simple parallélépipède. Notre approche ne recouvre pas ce cas de figure.

Par ailleurs, il semble logique de vouloir généraliser les résultats de ce chapitre au pavage d'un nid de boucles présentant à la fois des dépendances internes et externes : il est toujours possible de rajouter dans notre ensemble C de vecteurs de dépendances de réutilisation les vecteurs de dépendances de flot internes D. Le problème dans ce cas est que les dépendances internes impliquent des contraintes sur la forme limite des tuiles ($H^{-1}D \geq 0$). Il faut donc s'assurer dans le choix, pour l'instant effectué par la procédure C_{maxvol}, des vecteurs P représentatifs de $\{\overrightarrow{l_\sigma}\}$ que D soit bien contenu dans le cône généré par P.

Dans tous les cas, nous pensons que notre nouvelle approche est un complément utile au travail

décrit dans [1].

Chapitre 4

Tuiles non atomiques : réordonnancement des tâches pour exploiter le parallélisme interne.

4.1 Introduction

La technique de pavage est utilisée pour de nombreuses architectures différentes telles que les machines à mémoire partagée avec caches, les systèmes monoprocesseur avec caches où encore les systèmes à mémoire distribuée avec caches. Sur les machines à *mémoire partagée*, l'atomicité des tuiles est imposée afin d'éviter des synchronisations "inter-tuiles", permettant ainsi à un parcours simple de l'espace d'itérations d'assurer la légalité des calculs. De nombreuses approches [1, 33, 62, 78, 82, 93] utilisent cette contrainte afin de trouver la taille et la forme de tuiles optimales. L'atomicité des tuiles est justifiée sur un système *monoprocesseur* par le fait que la taille des tuiles est choisie pour que les données utilisées tiennent dans le cache ; à la fin du calcul de chaque tuile, le cache est vidé[1] et les données utiles au calcul de la nouvelle tuile sont rapatriées. En revanche, dans le cadre de systèmes *multiprocesseurs* à mémoire distribuée, l'atomicité des tuiles est souvent restrictive. Dans ce cadre, plusieurs paramètres entrent en jeu : la taille de la *mémoire cache*, la *granularité* du calcul face au temps de communications, mais aussi l'*exploitation du parallélisme.* Sur un système capable de recouvrir les calculs et les communications, et sur lequel le surcoût de synchronisation n'est pas trop élevé, l'impact du surcoût des communications est réduit. Dans ce cas, briser la contrainte d'atomicité peut permettre d'exploiter le parallélisme interne aux tuiles et ainsi réduire d'un facteur important la *latence du programme.*

Ainsi, sans remettre en cause l'approche classique introduite dans [57], nous proposons, dans ce chapitre, une transgression de la contrainte d'atomicité : on se propose de maintenir cette contrainte lors de l'étape de distribution des données, mais par contre, de la briser lors de l'étape d'ordonnancement des calculs et des communications. C'est en quelque sorte du pavage hiérarchique, la difficulté résidant dans le réordonnancement initial des tâches à l'intérieur de chaque tuile afin de mettre en évidence le parallélisme.

Cette approche est surtout justifiée sur les machines VLSI, par le fait que les contraintes y sont plus importantes, et que d'autres part, sur ce type de système les communications peuvent être pipelinées. Mais on peut aussi envisager que dans l'avenir cette approche devienne probante sur des plateformes plus classiques. En effet, il semble qu'actuellement, la taille de mémoire cache,

[1] *flushed* en anglais

croisse plus vite que le rapport du temps de calcul sur le temps de communication. Ainsi, en terme de localité, on sera amener à utiliser des tuiles de plus en plus larges, et donc de granularité "trop" importante. On tend donc, *a priori* vers des architectures ayant des propriétés favorables à l'approche présentée dans ce chapitre.

Partitionner les nids de boucles uniformes de telle manière qu'il n'y ait plus de communications, ou bien pour que le nombre de communications soit minimal, ou encore pour favoriser le recouvrement des calculs avec les communications, a motivé récemment un nombre significatif de recherches [89, 1, 79, 82, 91]. Ces approches impliquent des *redistributions de données* et un partitionnement *minimisant* les *communications* [37], mais aussi maximisant la localité des données, c'est à dire minimisant *le temps de calcul élémentaire*. D'un autre point de vue, on peut ne faire aucune redistribution, mais plutôt réordonner les tâches à l'intérieur d'une même tuile afin de minimiser le temps de latence entre les tuiles. Exploiter le parallélisme interne aux tuiles peut réduire de manière significative le temps d'exécution d'un nid de boucles. C'est ce qui motive le travail présenté dans ce chapitre.

Les approches précédentes de Chou et Kung [29] et de Dion et al. [39] au problème de recherche d'un ordonnancement optimal interne aux tuiles, n'apportent que des solutions heuristiques au problème, même dans le cas de dimension 1. La contribution principale de ce chapitre est de proposer une solution optimale au problème de dimension 1 pour une permutation de tâches constante[2]. Ensuite, nous prenons en compte la possibilité d'avoir des permutations de tâches non constantes pour laquelle nous proposons aussi une solution optimale dans le cas de dimension 1. Finalement, nous appliquons les résultats trouvés au cas de dimensions supérieures.

Ainsi, après avoir formulé et motivé le problème dans le Paragraphe 4.2, nous présentons (Paragraphe 4.3) les résultats antérieurs et discutons sommairement des différentes solutions. Ensuite (Paragraphe 4.4), nous introduisons le formalisme utile au développement des démonstrations, puis nous présentons quelques résultats intermédiaires fondamentaux. Nous décrivons alors (Paragraphe 4.5) les différents algorithmes dont nous prouvons l'optimalité (relative). Nous avons effectué des mesures de performance que nous décrivons dans le Paragraphe 4.6, paragraphe dans lequel nous faisons aussi quelques remarques succinctes ayant trait à la généralisation n-dimensionnelle des résultats présentés. Finalement, nous terminons ce chapitre (Paragraphe 4.8) par quelques remarques générales.

4.2 Formulation et motivation du problème

4.2.1 Exemple

Considérons l'exemple suivant :

Programme 12.

$$\begin{array}{l} \mathsf{do}\ j = 1, L \\ \quad \mathsf{do}\ i = 0, N \\ \quad\quad A[i + l[j]] = f(A[i]) \end{array}$$

Il serait possible de distribuer ce nid de boucles de telle manière que toutes les tâches dépendant les unes des autres soient exécutées par un même processeur (un processeur par sous-graphe connexe

[2]On définit une permutation de tâches constante comme étant un ordonnancement, *identique* pour chaque tuile, à une translation près.

du graphe des tâches) afin d'éliminer toutes les communications relatives au tableau A. Cette opération peut être effectuée en allouant les tâches d'indice i et $i+l[j]$ au même processeur. Mais le partitionnement de la boucle et la distribution des données seraient effectués pour toutes les valeurs de $l[j]$, ce qui est, en pratique, une solution trop coûteuse.

Une autre solution consiste à partitionner la boucle i de l'espace d'itérations en tuiles consécutives et à allouer cycliquement les tuiles aux processeurs. Ensuite, on réordonne les tâches à l'intérieur des tuiles afin d'exploiter le parallélisme interne et réduire la latence entre deux tuiles. L'exécution prend alors la forme d'un pipeline dans lequel, si le système le permet, calculs et communications seront recouverts.

On oppose donc deux solutions,

- l'une dans laquelle, des redistributions sont effectués permettant ensuite aux tâches d'être exécutées en parallèle sans plus de communications,
- l'autre n'effectuant aucune redistribution, mais impliquant de petites communications répétitives entre les processeurs voisins.

la première solution implique un large mouvement de données effectué d'un bloc, et la seconde induit *beaucoup moins* de mouvements de données, répartis durant l'exécution et que l'on espère pouvoir être recouverts par les calculs.

Le problème consistant à déterminer la meilleure permutation des tâches à l'intérieur d'une tuile afin de minimiser le temps total d'exécution, a d'abord été abordé par Chou et Kung [29] puis par Dion et al. [38, 39]. La solution qu'ils proposent n'est pas optimale, et n'est adaptée qu'à une certaine classe de réseaux de type VLSI. En fait, leur recherche est restreinte aux permutations constantes.

Notre cadre d'étude est l'implémentation de nids de boucles aux *dépendances inconnues statiquement*. Dans ce cadre, la qualité des permutations et la vitesse de génération sont des paramètres importants pour avoir un code pavé efficace. C'est pourquoi nous proposons plusieurs solutions "intermédiaires". Cet équilibre étant fortement dépendant de l'application et de la machine cible, les différentes solutions doivent être comparées en pratique.

4.2.2 Formulation du problème

Notre étude porte sur les nids de boucles parfaits avec des dépendances uniformes (éventuellement inclus dans des boucles externes) inconnues statiquement. Si un tel nid de boucles doit être exécuté sur un anneau ou une grille de processeurs, il est pavé, afin de favoriser la localité des données ainsi que la granularité de calculs. Nous supposons que la granularité de calcul est forte, et nous cherchons, suivant une direction donnée, à diminuer le temps de latence en exploitant le parallélisme interne à la boucle. Nous supposons que les distances de dépendance à l'intérieur de cette boucle ne sont pas connues au moment de la compilation. Nous cherchons donc, durant l'exécution, une permutation générique des tâches au sein même de chaque tuile, permettant de débuter le plus tôt possible l'exécution de la tuile suivante.

Nous considérons donc le schéma simplifié du Programme 13. Nous supposons que la boucle d'indice i, de longueur N, est pavée par des tuiles de longueur n et que les tâches internes portent une dépendance interne de longueur l, inconnue au moment de la compilation.

Programme 13.

$$\begin{array}{l}\mathsf{do}\ i = 0, N-1\\ \qquad A[..,i+l,..] = f(A[..,i,..])\end{array}$$

Restreignons pour l'instant notre présentation au cas de recherche d'une *permutation constante.* Dans ce cadre, le problème se pose de la manière suivante :

- L'espace d'itérations est noté $\tau = \{t_0, t_1, ..., t_{N-1}\}$ où t_i est la tâche d'indice i. Ainsi, la boucle est de longueur N. On considère pour l'instant qu'il n'y a qu'une seule dépendance uniforme de longueur l. On fait l'hypothèse que $l < n$: en effet, le cas échéant, on peut se ramener au cas $l' = l \mod n$, le temps de latence considéré n'étant alors plus entre deux tuiles voisines mais entre les tuiles T_j et $T_{j+\lceil \frac{l}{n} \rceil}$ pour tout j.
- La boucle interne est partitionnée en tuiles de longueur n. *Insistons sur le fait que l est inconnue statiquement et il n'est pas donc pas possible d'ajuster a priori n en conséquence.* Ainsi, pour $j \in \{0, ..., \lceil \frac{N}{n}, \rceil - 1\}$ la tuile T_j est l'ensemble $\{t_{nj}, t_{nj+1}, t_{nj+2}, ..., t_{nj+n-1}\} \cap \tau$ avec $j \in \{0, ..., \lceil \frac{N}{n} \rceil - 1\}$. Les tâches d'une même tuile sont exécutées par le même processeur.
- On note τ_{calc} le temps d'un calcul élémentaire (d'une tâche), et τ_{comm} le temps d'une communication élémentaire. Ainsi, on appelle *période de l'ordonnancement* (T), le temps séparant le début d'exécution de deux tuiles consécutives. La tuile T_i est donc exécutée entre le temps iT et le temps $iT + n\tau_{calc}$
- Dans chaque tuile, la permutation des tâches est identique modulo n : formellement, notons σ la permutation dans la tuile T_0.

$$\{0, 1, ..., n\} \xrightarrow{\sigma} \{0, 1, ..., n\}$$

  Alors, pour tout $i \in \{0, 1, ..., N-1\}$, t_i est exécuté par le processeur $j\lfloor \frac{i}{n} \rfloor$ entre l'instant $\tau_i = jT + \sigma^{-1}(i - nj)\tau_{calc}$ et l'instant $\tau_i + \tau_{calc}$.
- Une période T et une permutation σ constituent une solution valide à notre problème s'ils correspondent à un ordonnancement global valide, c'est à dire si pour tout i, t_{i+l} est exécuté après t_i.
- Pour n, l, τ_{comm} et τ_{calc} fixés, T_{min} représente la période valide minimale atteignable.

Le but, est donc de trouver la période T_{min}, et la permutation σ qui lui est associée.

La Figure 4.1 illustre les notations utilisées dans ce chapitre.

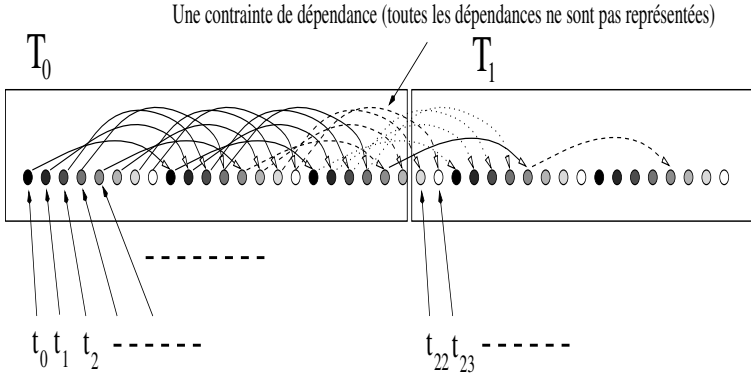


FIG. 4.1: *Pavage d'un nid de boucles unidimensionnel. La distance de dépendance vaut $l = 8$. La taille des tuiles vaut $n = 22$. Les pointillés montrant que les dépendances qui ne sont pas toutes représentées doivent être reproduites répétitivement vers la droite.*

4.3 Travaux antérieurs.

4.3.1 Solution de Chou et Kung

Dans ce paragraphe, nous présentons la solution de Chou et Kung [29]. Ils proposent l'ordonnancement suivant :

Pour tout $i \in \{0, .., N-1\}$, $j = \lfloor \frac{i}{n} \rfloor$ (on a $nj \leq i \leq nj+n-1$), t_i est exécuté par le processeur P_j entre l'instant $\tau_i = (i-nj) \times \tau_{calc} + j \times T$ et l'instant $\tau_i + \tau_{calc}$. La période de l'ordonnancement vaut :

$$T = (n - l + 1) \times \tau_{calc} + \tau_{comm}$$

Ainsi, pour tout i, la tuile T_i est exécutée entre l'instant $i \times T$ et l'instant $i \times T + n \times \tau_{calc}$.

La solution de Chou et Kung au problème $n = 7$ et $l = 3$ est présentée Figure 4.2. Dans cette figure, l'ordonnancement présenté est donné pour $\tau_{calc} = 1$ et $\tau_{comm} = 0$.

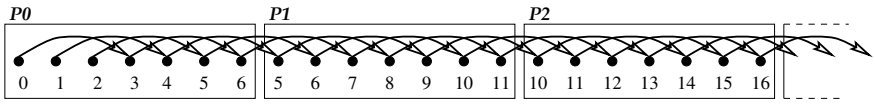


FIG. 4.2: *Ordonnancement de Chou et Kung pour le cas $n = 7$, $l = 3$ et $T = 5$. Dans ce cas, $T = 5\tau_{calc} + \tau_{comm}$.*

Comme on peut le voir, la solution obtenue n'est pas optimale, l'exécution des tâches étant (dans le cas unidimensionnel) quasiment séquentielle.

4.3.2 Solution de Dion et al..

Dion et al. [38] ont cherché un ordonnancement réduisant la période T, et donc, introduisant du parallélisme dans l'exécution. Dans ce paragraphe, nous présentons leurs résultats. Utilisons pour cela la notations $a \bigodot b$ pour représenter le pgcd de a et de b.

Lemme 3 *La permutation optimale permettant d'atteindre la période minimale T_{min} est indépendante des valeurs de τ_{calc} et τ_{comm}. De plus, si $T^{1,0}_{min}$ est la période minimale pour $\tau_{calc} = 1$ et $\tau_{comm} = 0$, alors $T^{\tau_{calc},\tau_{comm}}_{min} = T^{1,0}_{min} \times \tau_{calc} + \tau_{comm}$.*

Lemme 4 *Si $n \wedge l \neq 1$ alors le problème est équivalent au problème réduit où $n' = \frac{n}{n \wedge l}$ et $l' = \frac{l}{n \wedge l}$.*

Ainsi, à partir de maintenant, nous prendrons (sans perte de généralités) n et l tels que $n \wedge l = 1$, τ_{calc} égal à 1, et τ_{comm} égal à 0.

La permutation proposée par Dion et al. conduit à une période plus petite que celle obtenue à partir de la permutation de Chou et Kung. En fait, leur solution est optimale pour le cas particulier où $l = 2$. Par exemple pour $n = 9$ et $l = 2$, leur solution conduit à $T = 7 = T_{min}$ alors que la solution de Chou et Kung conduit à $T = 8$. Leurs solutions respectives sont présentées Figure 4.3.

En dimension 1, n petit est un cas d'école. Dans la pratique n est de l'ordre de 1000 ou plus. Or plus n est grand, plus le rapport entre la période obtenue par Dion et al. et la période obtenue par Chou et al. est grande, donc plus le rapport entre les temps d'exécution de ces deux solutions est grand.

Le théorème suivant résume le résultat de Dion et al. pour le cas $l = 2$,

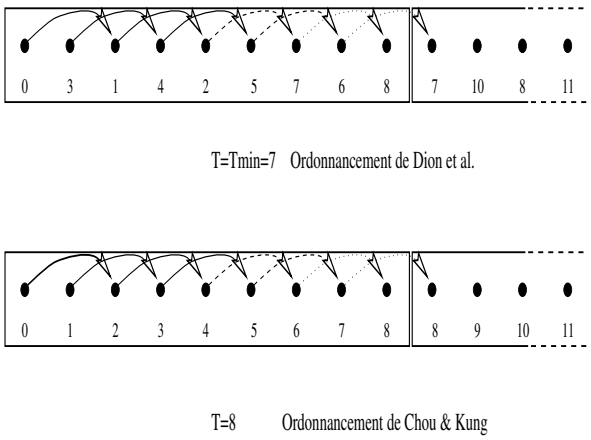


FIG. 4.3: *Comparaison de l'ordonnancement de Dion et al. avec l'ordonnancement de Chou et Kung pour* $n = 9$ *et* $l = 2$.

Théorème 6 *Pour* $n = 2k + 1,\ k > 0$ *et* $l = 2$*, l'ordonnancement optimal a une période*

$$T_{min} = \left\lceil \frac{3n-1}{4} \right\rceil$$

Ils donnent aussi un algorithme pour ordonnancer les tâches dans ce cas particulier :
Soit $j \in \{0, \cdots n-1\}$.

Algorithme 3. *Permutation constante optimale dans le cas* $l = 2$.

Procédure CalculPermutationConstante(n)
for x=0,n-1
if $(x \mod 2) = 1$ then $\sigma[\lceil \frac{3n-1}{4} \rceil - \frac{n-x}{2}] = x$
else if $x \leq 2(\lceil \frac{3n-1}{4} \rceil + 1) - n$ then $\sigma[\frac{x}{2}] = x$
else $\sigma[\frac{x+n-1}{2}] = x$
Retourne(σ)

Finalement, ils proposent une borne asymptotiquement optimale pour le cas général :

Théorème 7 *Pour* $l \geq 3$ *et* $n \wedge l = 1$*, la période* T *est contrainte par*

$$2\lfloor \frac{n}{l} \rfloor - 1 \leq T \leq 2\lfloor \frac{n}{l} \rfloor + 2$$

Pour $l \geq 3$, Dion et al. donnent un algorithme qu'ils nomment *algorithme cyclique* qui fournit une permutation correcte et asymptotiquement optimale ($T = 2\lfloor \frac{n}{l} \rfloor + 2$). Mais cette solution n'est pas optimale, et comme on pourra le voir dans le Paragraphe 4.7, le cas où l'espace d'itérations est de dimension supérieure à 1, met en jeu des tailles de tuiles pouvant être petites ($n_i \approx 10$ pour un espace d'itérations de dimension 3). C'est la limitation la plus notable de la solution donnée par Dion et al.. L'autre limitation est l'utilisation de manière similaire au travail de Chou et Kung, de la contrainte (restrictive) de permutation constante. Nous verrons dans les paragraphes qui suivent que la solution peut être fortement améliorée si l'on enlève cette contrainte.

4.3.3 Comparaison des solutions

Les Figure 4.4 et 4.5 fournissent une comparaison visuelle des différentes solutions proposées pour le cas $N = 700$, $n = 22$ et $l = 8$: à chaque fois, on trace un graphe "*espace/temps*", c'est à dire que l'exécution de la tâche t_i durant l'intervalle de temps $[j, j+1]$ est matérialisée par un cercle de coordonnées (i, j). Notons que dans les figures 4.4(b) et 4.5, les tâches connexes sont grisées avec la même intensité.

La Figure 4.4(a) (respectivement Figure 4.4(b)) représente la solution de Chou et Kung (respectivement Dion et al.). La Figure 4.5 montre les solutions fournies par nos deux algorithmes (cf paragraphe 4.5.2 et 4.5.3).

Ces figures montrent que la complexité des permutations augmente de la manière suivante :

Algorithme de Chou et Kung < Algorithme cyclique (Dion et al.) < Algorithme 4 < Algorithme 5.

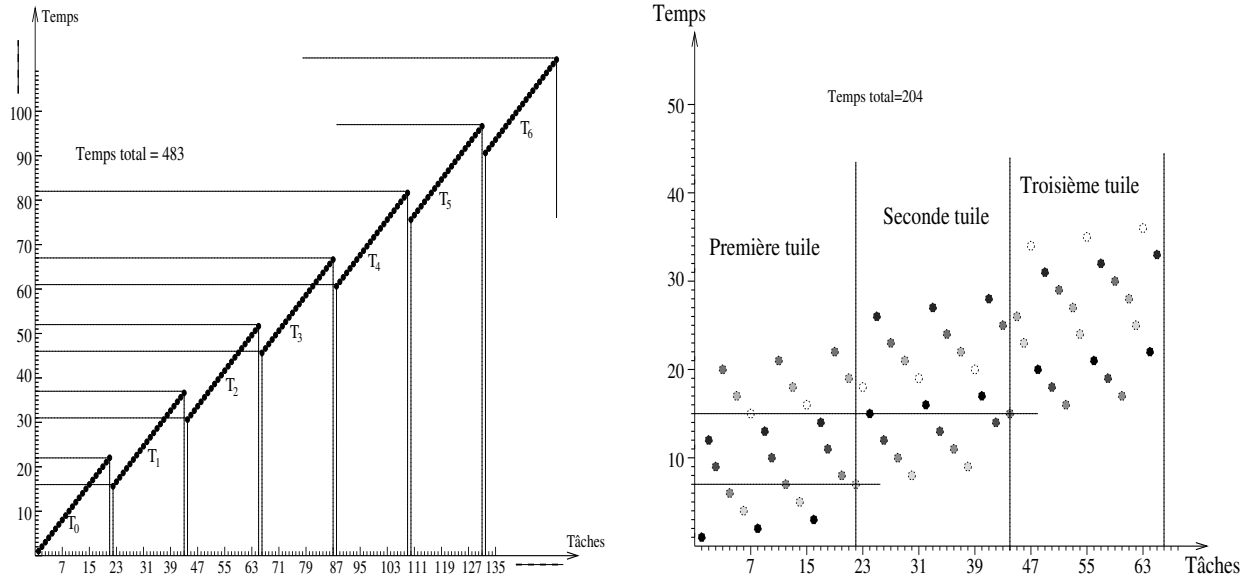


FIG. 4.4: *Graphe Espace/Temps montrant la solution de Chou et Kung (a) et de Dion et al.(b) pour* $N = 700$*,* $n = 22$ *et* $l = 8$.

4.4 Résultats préliminaires

4.4.1 Complexité du problème

Avant toute chose, examinons la combinatoire du problème, afin de motiver une recherche plus sophistiquée qu'une simple approche exhaustive.

Soit une tuile de taille n et une distance de dépendance de longueur l. Dénombrons le nombre de permutations possibles à l'intérieur de la tuile. Pour $n = 22$ et $l = 8$, la Figure 4.6 représente le graphe de dépendance pour une tuile.

Ce graphe a l composantes connexes. $k = (n \mod l)$ composantes comportent $(p + 1)$ tâches chacune (où $p = \lfloor \frac{n}{l} \rfloor$),le plus tôt possible et les $(l - k)$ autres composantes comportent p tâches chacune. Le nombre total de permutations valides de ce graphe est le nombre d'extensions linéaires du préordre associé, c'est à dire

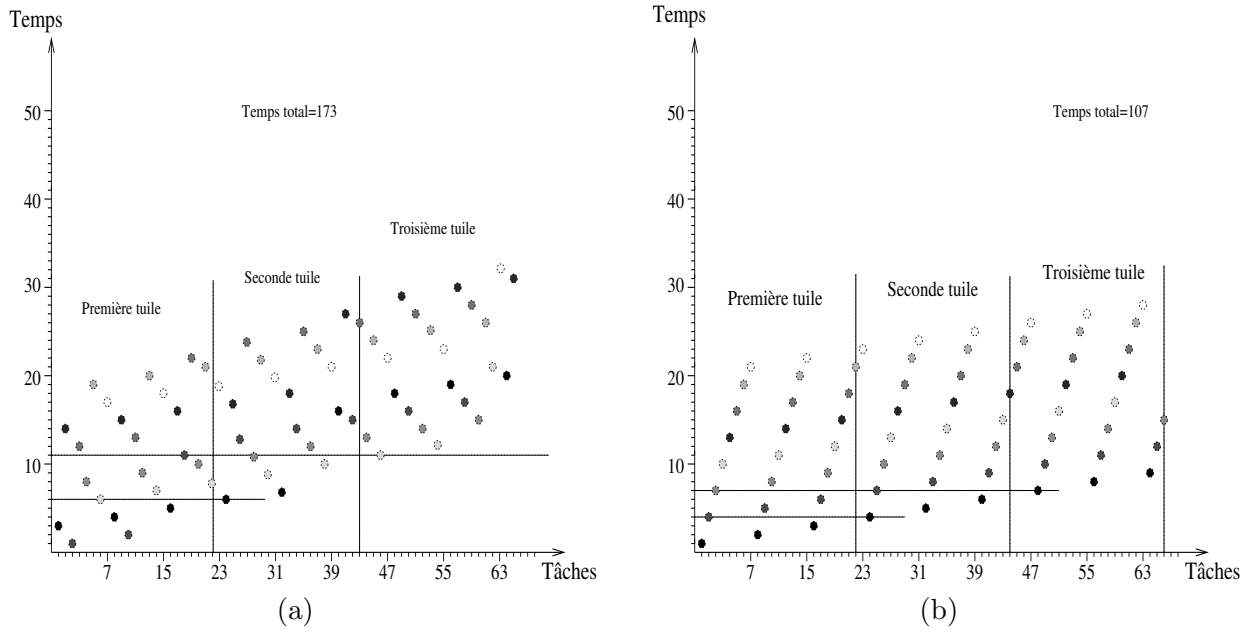


FIG. 4.5: *Graphe Espace/Temps montrant la solution fournie par l'Algorithme 4 (a) et par l'Algorithme 5 (b) pour* $N = 700, n = 22$ *et* $l = 8$.

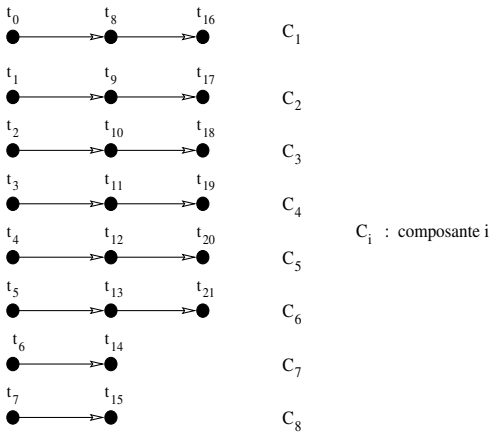


FIG. 4.6: *Graphe de dépendance des tâches de la première tuile.*

$$\frac{n!}{((p+1)!)^k(p!)^{l-k}} = \frac{n!}{(p+1)^k(p!)^l}$$

Pour, $n = 22,\ l = 8$, le nombre de permutations possibles est de 6.02×10^{15}. Même si n est petit (comme nous cherchons une permutation durant l'exécution) la nature combinatoire du problème interdit toute approche exhaustive.

Afin de trouver une solution acceptable, nous introduisons un formalisme qui nous permet de fournir la période optimale T_{min} pour toute valeur de $l \leq n$ en fonction de n, k et l, où $n = lp + k$ et $0 \leq k < l$. Cette solution est générée par un algorithme linéaire. C'est la présentation de ce formalisme qui va faire l'objet du paragraphe suivant.

4.4.2 Classes d'équivalence

Rappelons que n et l sont supposés premiers entre eux, que l est supposé inférieur à n et que $n = lp + k$.

Commençons par quelques définitions

1. L'ensemble des tâches de la première tuile T_0 est noté par $\Omega = \{t_0, t_1, ..., t_{n-1}\}$. De plus, on dit que $t_i \equiv t_j$ si $i \equiv j[l]$. Ainsi, pour simplifier les notations, la tâche t_i est désormais notée par l'entier i.
2. L'opérateur $\equiv$ est une relation d'équivalence qui définit l classes d'équivalence dans $\{0, 1, ..., n-1\}$, $X_0, X_1, ..., X_{l-1}$. On note alors l'ensemble des classes d'équivalence par $\Psi = \{X_0, X_1, ..., X_{l-1}\}$.
3. On dit que $X \to Y$ si et seulement si,

$$\exists (x, y) \in (X, Y) \cdot x \equiv y + n[l]$$

4. Les indices des classes d'équivalence sont choisis de telle manière que,
   - $X_0 = \{i \in \Omega \cdot i \equiv 0[l]\}$
   - $X_0 \to X_1 \to X_2 \to \cdots \to X_{l-1} \to X_0$

Avec les définitions précédentes, on vérifie que :

$$\forall X_i \in \Psi, \quad p \leq |X_i| \leq p + 1$$

De plus, il y a k classes de taille $p + 1$ et $l - k$ de taille p.

Définition 1 *Pour tout i dans $\{0, 1, \cdots, l-1\}$, on définit $\lambda_i = |X_i| - p$. En d'autres termes, si $|X_i| = p + 1$ alors $\lambda_i = 1$, sinon $|X_i| = p$ et $\lambda_i = 0$.*

Définition 2 *Pour tout $i \geq l,\ \lambda_i = \lambda_{(i \mod l)}$*

Ainsi, le mot formé des λ_i peut être étendu à une chaîne infinie périodique de période l, nommée λ à partir de maintenant. La permutation des tâches, et donc la période d'ordonnancement, dépendent des valeurs relatives des λ_i, c'est à dire des propriétés de la chaîne λ.

4.4.3 Mots sur l'alphabet $\{0,1\}$—mot bien équilibré.

Nous allons, par la suite, effectuer des manipulations sur les *mots*[3]. Voici quelques définitions classiques dans le domaine :

Soit $\sum = \{0,1\}$ l'alphabet du mot $\lambda_0\lambda_1\lambda_2\cdots\lambda_{l-1}\cdots$. Sur cet alphabet, on note par uv la concaténation des deux mots u et v (si $u = u_0u_1u_2.....u_n$ et $v = v_0v_1.....v_m$ alors $uv = u_0u_1.....u_nv_0v_1.....v_m$).

Définition 3 (sous-mot) *on dit que v est un sous-mot de u, s'il existe deux mots α et β, tels que $u = \alpha v\beta$*

Définition 4 (longueur) *La longueur d'un mot $u = u_1u_2u_3....u_n$ est le nombre de caractères qui le composent c'est à dire l'entier $n = |u|$.*

Définition 5 (poids) *Soit $\alpha \in \sum$. Si $u = u_1u_2.....u_n \in \sum^*$ est un mot de longueur n, alors $|u|_\alpha = |\{i \in \{1,...,n\}, u_i = \alpha\}|$.*

Définition 6 (bien-équilibré) *Soit $u \in \sum^*$. u est dit bien équilibré, si pour toute paire de sous-mots de u, (v,v') on a,*

$$|v| = |v'| \Longrightarrow ||v|_1 - |v'|_1| \leq 1$$

4.4.4 Propriétés de la chaîne $\boldsymbol{\lambda = \lambda_0\lambda_1\lambda_2\cdots\lambda_{l-1}\cdots}$

Comme noté lors de la Définition 1, la taille d'une classe d'équivalence X_i est donnée par $p+\lambda_i$, où $p = \lfloor \frac{n}{l} \rfloor$. Comme nous le verrons plus loin, la période optimale dépend des valeurs relatives des λ_i. Le but de ce paragraphe est donc de discuter des propriétés de la chaîne $\lambda_0, \lambda_1, ..., \lambda_{l-1}.....$ en fonction de la taille des tuiles (n) et de la distance de dépendance (l). Les résultats sont résumés par le théorème suivant :

Théorème 8 *Si $l \geq 3$, $n \wedge l = 1$, $n = pl + k$ et $\Lambda = \min_{i\in\mathbb{N}}[\max_{i+1\leq j,j+1\leq i+l-1}(\lambda_j + \lambda_{j+1})]$, alors*

- *Si $l = 4$ et $k = 3$, alors $\Lambda = \max_{2\leq j,j+1\leq 4}(\lambda_j + \lambda_{j+1}) = 1$*
- *Sinon,*

$$\Lambda = \max_{1\leq j,j+1\leq l-1}(\lambda_j + \lambda_{j+1}) = \left| \begin{array}{ll} 0 & si\ k = 1 \\ 1 & si\ 1 < k \leq \lceil \frac{l}{2} \rceil \\ 2 & si\ k > \lceil \frac{l}{2} \rceil \end{array} \right.$$

Notons que Λ est défini sous forme de min-max. Le Théorème 8 fournit la valeur de Λ pour toute taille de tuile et pour toute distance de dépendance. Le rapport entre la valeur de Λ et la période optimale (T_{min}) est présenté dans le Théorème 9.

Illustrons tout d'abord les résultats du Théorème 8 à l'aide de quelques exemples dans lesquels le sous-mot $F_i = \lambda_{i+1}\ldots\lambda_{i+l-1}$ qui minimise $\max_{\lambda_j\lambda_{j+1}}$ sous mot de $F_i(\lambda_j + \lambda_{j+1})$, est matérialisé en caractères gras.

Exemple 1

Si $n = 7,\ l = 3$ ($k = 1$) alors $\lambda = \mathbf{100}100100...$, et $\Lambda = 0$.

[3] *string* en anglais

Exemple 2

Si $n = 7,\ l = 4\ (k = 3)$ alors $\lambda = 11\mathbf{101}110...$, et $\Lambda = 1$

Exemple 3

Si $n = 7,\ l = 5\ (k = 2 < \lceil \frac{5}{2} \rceil)$ alors $\lambda = 1\mathbf{0100}10100...$, et $\Lambda = 1$

Exemple 4

Si $n = 9,\ l = 5\ (k = 4 > \lceil \frac{5}{2} \rceil)$ alors $\lambda = 1\mathbf{1110}11110...$, et $\Lambda = 2$

Afin de prouver le Théorème 8 quelques résultats préliminaires sont nécessaires.

Le Lemme 5 fournit une définition de X_i équivalente à celle proposée dans le Paragraphe 4.4.2. Le Lemme 6 établit les conditions que la taille de tuile et la distance de dépendances doivent vérifier, pour que la chaîne λ contienne le sous-mot 11. Le Lemme 7 établit la propriété de bon équilibre de la chaîne λ, propriété utilisée à la fois pour la preuve du Théorème 8 et pour la preuve du Théorème 9.

Lemme 5 *Si* $X'_i = \{x \in \Omega \mid \exists \alpha \in \mathbb{N} \cdot (x \equiv \alpha l[n]) \wedge (in \leq \alpha l < (i+1)n)\}$, *alors* $\forall i,\ X'_i = X_i$.

Preuve. Remarquons dans un premier temps que $x \in \Omega = \{0, 1, \cdots, n-1\}$. Ainsi, $(\forall i, in \leq x + in < (i+1)n,)$ et donc,

$$\forall i, X'_i = \{x \in \Omega \mid \exists \alpha \in \mathbb{N} \cdot x + in = \alpha l\}$$

Nous avons les résultats suivants :

1. Soit $in = (\alpha - 1)\ l + r$ la division euclidienne de in par l. On a $l - r \in X'_i$.
   Ainsi, pour tout $i, X'_i \neq \emptyset$.

2. Soit $x_i \in X'_i$ et $x_i + in = \alpha_i\ l$. Alors,

$$\begin{aligned} X'_i &= \{x \in \Omega | \ \exists \alpha \in \mathbb{Z} \cdot x + in = (\alpha_i + \alpha)\ l\} \\ &= \{x \in \Omega \mid \exists \alpha \in \mathbb{Z} \cdot x = x_i + \alpha l\} \\ &= \{x \in \Omega \mid x \equiv x_i[l]\} \end{aligned}$$

   En conséquence, pour tout $i, X'_i \in \Psi$.

3. Clairement, $X'_0 = X_0$.

4. Soit $x_i \in X'_i$ and X'_{i+1}.
   Comme il existe α_i et α_{i+1}, tels que $x_i + in = \alpha_i l$ et $x_{i+1} + (i+1)n = \alpha_{i+1} l$,
   On a, $x_i \equiv x_{i+1} + n[l]$.
   D'où, pour tout $i, X'_i \rightarrow X'_{i+1}$.

■

Lemme 6

$$\begin{aligned} \exists i \in \{1, ..., l-1\} \mid \lambda_i\lambda_{i+1} = 11 &\iff 2k > l+1 \\ &\iff k > \lceil \frac{l}{2} \rceil \end{aligned}$$

Preuve. La preuve de $(2k > l+1 \iff k > \lceil \frac{l}{2} \rceil)$ est décomposée en deux cas : l est soit pair, soit impair. La preuve est immédiate, on a les résultats suivants :

1. Si $2k = l+1$, alors

$$\begin{aligned} |X_i| + |X_{i+1}| &= |X_i + X_{i+1}| \\ &= |\{\alpha \in \mathbb{N} \mid in \leq \alpha l < in + 2n\}| \\ &= |\{\alpha \in \mathbb{N} \mid in \leq \alpha l < in + (2p+1)l + 1\}| \end{aligned}$$

   Ainsi, comme $l \wedge n = 1$, $|X_i| + |X_{i+1}| = 2p+2 \iff i \equiv 0[l]$.

2. Si $2k < l+1$, alors

$$\begin{aligned} |X_i| + |X_{i+1}| &\leq |\{\alpha \in \mathbb{N} \mid in \leq \alpha l < in + (2p+1)l\}| \\ &\leq 2p+1 \end{aligned}$$

3. Si $2k > l+1$, considérons $i \in \{1, ..., l-1\}$ tel que $n-1 \in X_i$. Alors $i - 1 \neq 0$, en effet,

$$\begin{aligned} l+1 < 2k < 2l &\implies 2pl + 2k \not\equiv 1[l] \\ &\implies 0 \not\equiv (n-1) + n[l] \end{aligned}$$

   De plus, $\lambda_{i-1}\lambda_i = 11$.

■

Lemme 7 *La chaîne $\lambda = \lambda_0\lambda_1\lambda_2.....$ est bien équilibrée.*

Preuve. Considérons $\lambda' = \lambda_i\lambda_{i+1}....\lambda_{i+\alpha-1}$ une sous-chaîne de λ de longueur α. Il suffit de montrer que $|\lambda'|_1 \in \{\beta, \beta+1\}$ où $\alpha n = (\alpha p + \beta)l + r$ tel que $0 \leq r < l$. On a,

$$\begin{aligned} |X_i| + |X_{i+1}| + ... + |X_{i+\alpha-1}| &= |\{k \in \mathbb{N}, in \leq kl < (i+\alpha)n\}| \\ |X_i| + |X_{i+1}| + ... + |X_{i+\alpha-1}| &\in \{\alpha p + \beta, \alpha p + \beta + 1\} \end{aligned}$$

■

Nous pouvons maintenant présenter la preuve du Théorème 8,

Preuve du Théorème 8. Notons d'abord que $\lambda_{\mathbf{0}} = \mathbf{1}$. On a,
$|X_0| = |X'_0| = |\{\alpha \in \mathbb{N}, 0 \leq \alpha l < n\}| = p + 1$.
D'où,

$$\begin{aligned} |X_{l-1}| &= |\{\alpha \in \mathbb{N}, (l-1)n \leq \alpha l < ln\}| \\ &= |\{\alpha \in \mathbb{N}, (l-1)n \leq \alpha l \leq ln\}| - 1 \\ &\leq (p+1) - 1 \end{aligned}$$

Et, $\lambda_{\mathbf{l-1}} = \mathbf{0}$.

Considérons maintenant les différents cas suivants :
Cas $k = 1$:

- Comme $k = |\lambda_0\lambda_1....\lambda_{l-1}|_1 = 1$ et $\lambda_0 = 1$ alors, $\Lambda \leq \max_{1\leq p,p+1\leq l-1}(\lambda_p + \lambda_{p+1}) = 0$
- En conséquence, $\mathbf{\Lambda = 0}$.

Cas $1 < k \leq \lceil \frac{l}{2} \rceil$:

- $\forall i, |\lambda_i\lambda_{i+1}....\lambda_{i+l-1}| = k \geq 2$ alors, $\forall i, \exists p \in \{i, ..., l+i-2\} \mid \lambda_p = 1$. D'où, $\Lambda \geq 1$.
- D'après le Lemme 6, on a

$$\Lambda \leq \min_{1\leq p,p+1\leq l-1} \lambda_p + \lambda_{p+1} \leq 1$$

- En conséquence, $\mathbf{\Lambda = 1}$.

Cas $l > k > \lceil \frac{l}{2} \rceil$:

- On a $2n = 2pl + 2k \geq (2p+1)l + 1$. donc,

$$\begin{aligned} |X'_0| + |X'_1| &= |\{\alpha \in \mathbb{N}, 0 \leq \alpha l < 2n\}| \\ &= 2p + 2 \end{aligned}$$

  Ainsi, $\lambda_{\mathbf{0}}\lambda_{\mathbf{1}} = \mathbf{11}$
- Supposons qu'il existe $i \in \{2, 3, ..., l-1\}$ tel que $\lambda_i\lambda_{i+1} = 11$.
  Comme $i \neq l-1$ on aurait alors $\mathbf{\Lambda = 2}$
- Supposons qu'un tel i n'existe pas, alors $\lambda_{l-1}\lambda_0\lambda_1\lambda_2 = 0111$ ($\lambda_2 = 0$ violerait la condition $k > \lceil \frac{l}{2} \rceil$). Ainsi, $l \geq 4$.
  Supposons $l > 4$. D'après le Lemme 7 on sait que $\lambda_0\lambda_1.....\lambda_{l-1}$ ne contient pas le sous-mot 00. Ainsi, l est nécessairement pair ($l \geq 6$)

$$\begin{aligned} \lambda_0\lambda_1....\lambda_{l-1} &= 111(01)^{\frac{l-4}{2}}0 \\ &= 111010(10)^{\frac{l-6}{2}} \end{aligned}$$

  Mais ceci contredit le Lemme 7, du fait de la présence des sous-mots 111 et 010.
  Finalement, $\mathbf{l = 4}$ et $\mathbf{k = 3}$ donne $\lambda_0\lambda_1\lambda_2\lambda_3\lambda_4\lambda_5\lambda_6\lambda_7 = 11\mathbf{101}110$ et,
  $\mathbf{\Lambda} = \max_{2\leq p,p+1\leq 4} \lambda_p + \lambda_{p+1} = 1$

■

4.5 Algorithmes

4.5.1 Borne inférieure sur la période d'ordonnancement

Dans ce paragraphe, nous donnons une borne inférieure sur la période d'ordonnancement. Nous verrons dans le Paragraphe 4.5.1 que cette borne inférieure est atteignable. Le résultat peut être résumé par le théorème suivant dans lequel Λ correspond à la définition préalablement donnée dans le Théorème 8 :

Théorème 9 *Pour un ordonnancement valide et $l > 2$, si on note $p = \lfloor \frac{n}{l} \rfloor$, on a $T \geq 2p + \Lambda$.*

En d'autres termes, on ne perdrait qu'un facteur 2 à ne pas redistribuer les données.
Afin de prouver ce théorème, nous introduisons quelques notations :

1. Considérons un ordonnancement valide tel que $T < 2p + \Lambda$ (démonstration par l'absurde).
2. Notons $A_1, \ldots, A_l$ les classes d'équivalence. Pour $A_i \in \Psi$, notons b_i (respectivement e_i) l'instant pendant lequel le premier élément de A_i (respectivement le dernier élément) est exécuté.
3. Notons A_l la classe d'équivalence ayant le plus grand b_i. Puis, indexons les autres classes d'équivalence $(A_1, A_2, \ldots, A_{l-1})$ telles que $A_l \rightarrow A_{l-1} \rightarrow .... \rightarrow A_1$.
4. On pose $\lambda_i = |A_i| - p$.
5. On notera $\mathbf{B}(A)$ le fait que l'ensemble de tâches A soit exécuté d'un bloc.

Preuve. L'idée de la démonstration est de montrer, par récurrence *décroissante*, que pour tout $2 \leq i \leq l-2$ on a $\mathbf{B}(A_i, A_{i+1}, \ldots, A_l)$ (le cas $i = l-1$ est un cas particulier faisant l'objet du Lemme 10). De cela (Lemme 8), on en déduit que pour tout $i \geq 2$, $\Lambda \geq \lambda_{i-1} + \lambda_i + 1$, ce qui contredit la définition de Λ.
Le moteur de la récurrence est le Lemme 9, son initialisation correspond au Lemme 10.

Lemme 8 *Si $b_{i-2} < b_{i-1} < b_i$ alors* $\begin{cases} \mathbf{B}(A_i, A_{i-1}) \\ \Lambda \geq \lambda_{i-1} + \lambda_i + 1 \end{cases}$

Lemme 9 *Si pour $3 \leq i < l$ on a $\mathbf{B}(A_i, A_{i+1}, \ldots, A_l)$ alors $\mathbf{B}(A_{i-1}, A_i, \ldots, A_l)$*

Lemme 10 *On a soit $\mathbf{B}(A_{l-1}, A_l)$, soit $\mathbf{B}(A_{l-2}, A_{l-1}, A_l)$. Dans le second cas,* $\begin{cases} \Lambda \geq \lambda_{l-1} + \lambda_l + 1 \\ \Lambda \geq \lambda_{l-2} + \lambda_{l-1} + 1 \end{cases}$

Remarque 4. La présence de la dépendance entre la dernière tâche de A_i et la première tâche de A_j pour tout $A_i \rightarrow A_j$, nous fournit les inégalités suivantes :

$$1 \leq i \leq l-1 \quad e_{i+1} < b_i + T \tag{4.1}$$

$$e_1 < b_l + T \tag{4.2}$$

Preuve du Lemme 8. Considérons l'intervalle $[b_{i-1}, b_{i-1} + T[$ de taille $T = 2p + \Lambda - 1$.

- Comme $b_i > b_{i-1}$, d'après la Remarque 4, les $p + \lambda_i$ tâches de A_i doivent être exécutées dans cet intervalle.
- Comme $b_{i-2} < b_{i-1}$, $e_{i-1} < b_{i-2} + T < b_{i-1} + T$, les $p + \lambda_{i-1}$ tâches de A_{i-1} doivent être exécutées dans cet intervalle.
- Ainsi, $p + \lambda_i + p + \lambda_{i-1} \leq 2p + \Lambda - 1$, soit $\Lambda \geq \lambda_i + \lambda_{i-1} + 1$.
- Finalement, comme $\Lambda \leq \lambda_i + \lambda_{i-1} + 1$, $\mathbf{B}(A_{i-1}, Ai)$

■

Preuve du Lemme 9. Posons $b = \min(b_i, b_{i+1})$. On a $b_{i-1}, b_{i-2} < b$ et $\mathbf{B}(A_i, A_{i+1})$.

- Considérons la classe A_{i-1}. On a, $e_{i-1} < b_{i-2} + T \leq b - 1 + 2p + \Lambda - 1 \leq \max(e_i, e_{i+1})$
  Ainsi, comme $\mathbf{B}(A_i, A_{i+1})$, $e_{i-1} < b \leq b_i$.
- Considérons maintenant l'intervalle $[b_{i-1}, e_i]$ de taille inférieure à $T = 2p+\Lambda-1$. D'après ce qui précède, A_{i-1} et A_i doivent être exécutées dans cet intervalle, soit $p+\lambda_{i-1}+p+\lambda_i \leq 2p+\Lambda-1$.
- Comme $2p + \lambda_{i-1} + \lambda_i \geq 2p + \Lambda - 1$, on a $\mathbf{B}(A_{i-1}, A_i)$, d'où le résultat.

■

Preuve du Lemme 10. On a $[b_{l-1}, b_{l-2} < b_l]$. Si $b_{l-2} < b_{l-1} < b_l$ le Lemme 8 fournit le résultat. Supposons donc que $b_{l-1} < b_{l-2} < b_l$

- Considérons l'intervalle $[b_{l-1}, b_l + T[$.
- Comme $[b_{l-3}, b_{l-2} < b_l]$, ainsi $[e_{l-1}, e_{l-2} < b_l + T]$.
- Ensuite, $e_l < B_{l-1} + T$ donc $b_l \leq e_l - p - \lambda_l < b_{l-1} + \Lambda - 1 - \lambda_l + p$ et alors $b_l + T \leq b_{l-1} + 2\Lambda - 3 - \lambda_l + 3p$
  Cet intervalle est donc de taille inférieure ou égale à $2\Lambda - 3 - \lambda_l + 3p$
- Ainsi, A_l, A_{l-1} et A_{l-2} doivent être exécutées dans un intervalle de taille $2\Lambda - 3 - \lambda_l + 3p$, d'où $\mathbf{B}(A_l, A_{l-1}, A_{l-2})$ et $2\Lambda \geq 2\lambda_l + \lambda_{l-1} + \lambda_{l-2} + 3$.
- Cette dernière inégalité n'étant vérifiée que si $\Lambda = 2$ et $\lambda_{l-2}\lambda_{l-1}\lambda_l = 010$.

■

4.5.2 Algorithme pour une permutation constante

Dans le cadre du formalisme introduit dans les paragraphes précédents, nous proposons ici un algorithme fournissant une permutation constante optimale. Rappelons que $l \geq 3$ et $n \wedge l = 1$.

Algorithme 4. *Permutation constante optimale dans le cas $l > 2$.*

Procédure CalculePermutationConstante (n, l)
{ Initialise variables p et k }
    $p = \lfloor \frac{n}{l} \rfloor$
    $k = n \mod l$
{ Trouve la tâche f à être exécutée en premier }
    if $(l = 4$ et $k = 3)$ then $f = l - k$ { car $0 \equiv (f + n)[l]$ }
    else $f = 0$
{ Exécute les p premières tâches de X_f }
    $x = f$
    do $t = 0, p - 1$
        $\sigma[t] = x$
        $x = x + l$
{ Exécution des classes dans l'ordre inverse de $(\rightarrow)$}
{Ici $t = p$}
    $x = (x + n) \mod l$
    while $x \neq f$ do
        repeat
            $\sigma[t] = x$
            $x = x + l$
            $t = t + 1$
        until $x \geq n$
        $x = (x + n) \mod l$
{ Exécute la dernière tâche de X_f }
    $\sigma[t] = f + p \times l$
Retourne (σ)

Montrons que cet algorithme est correct et optimal : les dépendances internes à chaque classe sont clairement vérifiées. Supposons donc une période $T = 2p + \Lambda$, et montrons que la permutation est valide au sens des dépendances externes aux classes.

Avec les notations du Paragraphe 4.5.1, on a

$$\begin{array}{ll} b_0 = 0 & \\ b_1 = p & e_1 = p + |A_1| - 1 \\ b_2 = p + |A_1| & e_2 = p + |A_1| + |A_2| - 1 \\ \vdots & \vdots \\ b_j = p + \sum_{i=1}^{j-1} |A_i| & e_j = p + \sum_{i=1}^{j} |A_i| - 1 \\ \vdots & \vdots \\ b_{l-1} = p + \sum_{i=1}^{l-2} |A_i| & e_{l-1} = n - 2 \\ & e_0 = e_l = n - 1 \end{array}$$

De plus,

$$\forall j \in [0, l-2], A_{j+1} \rightarrow A_j$$

Cas $n = 14$ et $l = 6$															
Tuile 0	Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Tâches	0	6	2	8	4	10	12	1	7	3	9	5	11	13
Tuile 1	Temps	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	Tâches	14	20	16	22	18	24	26	15	21	17	23	19	25	27
Tuile 2	Temps	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	Tâches	28	34	30	36	32	38	40	29	35	31	37	33	39	41

TAB. 4.1: *Ordonnancement optimal avec une permutation constante dans chaque tuile.*

Une condition nécessaire pour que T soit une période valide est,

$$\forall j \in [0, l-1],\ T \geq e_{j+1} - b_j + 1$$

Or, $\forall j \in \{1, \cdots, l-2\}$

$$\begin{aligned} e_{j+1} - b_j + 1 &= p + \sum_{i=1}^{j+1} |A_i| - 1 - (p + \sum_{i=1}^{j-1} |A_i| + 1) \\ &= |A_j| + |A_{j+1}| \\ e_1 - b_0 + 1 &= p + |A_1| \\ e_l - b_{l-1} + 1 &= n - 1 - (p + \sum_{i=1}^{l-2} |A_i|) + 1 \\ &= 1 + |A_{l-1}| \end{aligned}$$

Finalement,

– Si $l = 4$ et $k = 3$, on a,

$$\begin{aligned} (|A_0|, |A_1|, |A_2|, |A_3|) &= (|X_1|, |X_0|, |X_3|, |X_2|) \\ &= (\lambda_5 + p, \lambda_4 + p, \lambda_3 + p, \lambda_2 + p) \end{aligned}$$

Ainsi, $T = 2p + \max_{2 \leq q, q+1 \leq 4} \lambda_q + \lambda_{q+1}$ est valide.

– Sinon $(|A_0|, |A_1|, \cdots, |A_{l-1}|) = (|X_0|, |X_{l-1}|, \cdots, |X_1|) = (\lambda_l + p, \lambda_{l-1} + p, \cdots, \lambda_1 + p)$
Ainsi, $T = 2p + \max_{1 \leq q, q+1 \leq l-1} \lambda_q + \lambda_{q+1}$ est valide.

■

Dans le cas où $n \wedge l \neq 1$, comme nous l'avons vu, la solution optimale peut être construite aisément à partir de la solution du problème associé $(n', l') = (\frac{n}{n \wedge l \neq 1}, \frac{l}{n \wedge l \neq 1})$. Le Tableau 4.1 fournit un exemple d'ordonnancement pour $n = 14$ et $l = 6$. Comme $n \wedge l = 2$, la tuile T_0 a deux composantes {0,2,4,6,8,10,12} et {1,3,5,7,9,11,13}. Chacune d'elle forme une sous-tuile avec $n' = 7$ et $l' = 3$. La première composante contient les classes d'équivalence $X_0 = \{0, 6, 12\}, X_1 = \{4, 10\}, X_2 = \{2, 8\}$. Au départ, p tâches de X_0 sont exécutées suivies des tâches de X_2 et X_1. Finalement, la dernière tâche de X_0 est exécutée. La seconde composante est exécutée dans le même ordre.

4.5.3 Algorithme pour une permutation non constante

Si on enlève la contrainte de permutation constante, la solution peut être fortement améliorée. Cela permet d'atteindre en théorie une période à peu près deux fois plus petite.

Cas $n = 5$ et $l = 3$						
Tuile 0	Temps	0	1	2	3	4
	Tâches	0	3	1	4	2
	Décalage	2				
Tuile 1	Temps	2	3	4	5	6
	Tâches	6	9	7	5	8
	Décalage	2				
Tuile 2	Temps	4	5	6	7	8
	Tâches	12	10	13	11	14
	Décalage	1				

TAB. 4.2: *Ordonnancement optimal avec permutations non constantes dans chaque tuile.*

Cas où $n \wedge l = 1$. Reprenons les notations du Paragraphe 4.4.2 : $X_0 \to X_1 \to \cdots \to X_{l-1} \to X_0$. En fait, l'algorithme optimal est l'algorithme le plus naturel. Le premier processeur exécute les tâches de X_0 puis X_1, X_2 et finalement X_{l-1}. Le second processeur commence à travailler $|X_0|\tau_{calc} + \tau_{comm}$ unités de temps après le premier, et exécute les tâches de X_1 puis $X_2, X_3 \cdots X_{l-1}$ et finalement X_0. Le troisième processeur commence à travailler $|X_1|\tau_{calc} + \tau_{comm}$ unités de temps après le second, et exécute les tâches X_2 puis $X_3, X_4 \cdots X_0$ et finalement X_1.

Le temps de décalage O_i est le nombre d'unités de temps entre le début de l'exécution de la tuile i et le début de l'exécution de la tuile $i+1$. Contrairement à la période, le temps de décalage n'est pas le même pour toutes les tuiles. Il correspond (en unité de temps de calcul) à la taille de l'ensemble $C_i = \{m \in \mathbb{N}, in \leq ml \leq (i+1)n\}$. Ainsi,

$$O_i = \left\lceil \frac{n(i+1) - \lceil \frac{ni}{l} \rceil l}{l} \right\rceil \tau_{calc} + \tau_{comm}.$$

Le Tableau 4.2 illustre le fonctionnement de l'algorithme dans le cas $n = 5$ et $l = 3$.

Cas général—Algorithmes et exemple. Dans le cas général où $n \wedge l = d$, la méthode est identique à celle utilisée dans le cas de permutation constante. Voici l'algorithme fournissant la

permutation pour une tuile i donnée :

Algorithme 5. *Permutation non constante optimale dans le cas où $l > 2$*

Procédure CalculePermutationNonConstante (n, l, i)
{ Initialise variables d, n_d et l_d }
$d = n \wedge l$
$n_d = \frac{n}{d}$
$l_d = \frac{l}{d}$
{ Initialise m et f}
$m = \lceil \frac{n_d i}{l_d} \rceil$
$f = (m l_d) \mod n_d$
do $i_d = 0, d-1$
{ Exécute la première tâche }
$t = 0$
$\sigma[t] = f \times d + i_d$
{ Exécution des classes dans l'ordre inverse de $(\rightarrow)$}
$x = (f + l_d) \mod n_d$
while $x \neq f$ do
$t = t + 1$
$\sigma[t] = x \times d + i_d$
$x = (x + l_d) \mod n_d$
Retourne σ

Ensuite, l'algorithme fournissant le décalage O_i pour une tuile fixée i est le suivant :

Algorithme 6. *Décalage valide minimal pour chaque tuile, associé à l'Algorithme 5*

Procédure CalculeDécalage (n, l, i)
{ Initialise d, n_d et l_d }
$d = n \wedge l$
$n_d = \frac{n}{d}$
$l_d = \frac{l}{d}$
{ Initialise m et f}
$m = \lceil \frac{n_d i}{l_d} \rceil$
$f = (m l_d) \mod n_d$
{ Calcule le décalage }
$O_i = \left\lceil \frac{n_d - f}{l_d} \right\rceil$
retourne O_i

Le Tableau 4.3 illustre l'algorithme dans le cas $n = 15$ et $l = 9$.

4.6 Evaluation de performances

Comme souligné dans le Paragraphe 4.3.3, la permutation générée par l'Algorithme 5 est plus complexe (mais théoriquement meilleure) que celle générée par l'Algorithme 4, elle même plus com-

Cas $n = 15$ et $l = 9$																
Tuile 0	Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	Tâches	0	5	10	2	7	12	4	9	14	1	6	11	3	8	13
	Décalage	2														
Tuile 1	Temps	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	Tâches	18	23	28	15	20	25	17	22	27	19	24	29	16	21	26
	Décalage	2														
Tuile 2	Temps	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	Tâches	31	36	41	33	38	43	30	35	40	32	37	42	34	39	44
	Décalage	1														

TAB. 4.3: *Ordonnancement optimal avec une permutation non constante dans chaque tuile.*

l	Temps (micro-seconde)		
	Dion	Algo 4	Algo 5
3	11937	17153	19451
5	13108	18352	19546
7	13923	18422	19711
9	14966	18783	19627

TAB. 4.4: *Temps de génération des différentes permutations.* ($N = 256$, $n = 64$, $P = 4$).

plexe que celle générée par l'Algorithme cyclique (Dion et al.). Selon l'architecture cible, l'impact de la complexité de l'algorithme de génération sera plus ou moins important. Il semble fort probable que sur une architecture de type VLSI, cet impact soit très important, et que l'algorithme de Dion et al. soit préférable à l'Algorithme 5. Il est même probable qu'en dimension 2 ou supérieure le concepteur d'un circuit choisisse, pour des raisons d'utilisation de l'espace, d'implémenter l'algorithme de Chou et Kung. En contrepartie sur une architecture plus classique, l'impact de la complexité de l'algorithme est moins important, et l'Algorithme 5 fournit très probablement des performances supérieures aux deux précédents. C'est ce que l'on peut vérifier à l'aide des expériences présentées dans le Paragraphe suivant.

4.6.1 Expériences sur un Cray T3E

Les mesures effectuées dans ce paragraphe ont été effectuées sur un Cray T3E, système à mémoire distribuée (se référer à `http ://oscinfo.osc.edu/hardware` pour plus de détails).

Complexité des différentes permutations. Dans un premier temps, afin d'illustrer la hiérarchie de complexité des différents algorithmes, nous avons mesuré le temps de génération des différentes permutations, sans effectuer aucun calcul. Les résultats sont présentés dans le Tableau 4.4.

Performance globale. La comparaison a été effectuée sur l'exemple suivant :

Programme 14.

```
do i=0,N-1
    Tâche(i)
```

où Tâche() induit une dépendance uniforme de longueur l. Ici Tâche() représente une boucle. Les modifications présentées ci-dessous ont été incorporées à l'intérieur du compilateur SUIF [53] en tant que phase d'optimisation. Nous avons ensuite effectué nos expériences en faisant varier la taille de tuile (n) et la distance de dépendance (l). Nous avons utilisé 16 processeurs, alloués cycliquement sur les tuiles.
Dans le cas de permutations constantes, la permutation est générée à l'entrée de la boucle comme montré ci-dessous :

Programme 15. *Code utilisé pour tester les algorithmes de permutation constante*

```
GénèrePermutation(n, l)
do I = 0, N − 1 : n
    do i = I, min(N, I + n) − 1
        { Si nécessaire réceptionne les données du processeur de gauche }
        Tâche(I + σ(i − I))
        { Si nécessaire envoie les données au processeur de droite }
```

Dans le cas de permutations non constantes, la permutation est générée avant chaque tuile, comme montré ci-dessous : ce qui notons le, n'est probablement pas la manière optimale d'implémenter cette méthode.

Programme 16. *Code utilisé pour tester l'Algorithme 5*

```
do I = 0, N − 1 : n
    GénèrePermutation(n, l, I)
    do i = I, min(N, I + n) − 1
        { Si nécessaire réceptionne les données du processeur de gauche }
        Tâche(σ(i))
        { Si nécessaire envoie les données au processeur de droite }
```

La Figure 4.7 compare les temps d'exécution des différentes stratégies pour de petites tailles de tuile, alors que la Figure 4.8 compare les différentes stratégies sur de plus grandes tailles de tuile. Les résultats confirment l'intuition : sauf dans les cas particuliers où la permutation générée par les trois algorithmes est identique, c'est à dire les cas $l \in 2, 4, 8, 16$ dans notre exemple, on a

$$Temps(\text{Algorithme 5 }) < Temps(\text{Algorithme 4}) < Temps(\text{Algorithme cyclique}).$$

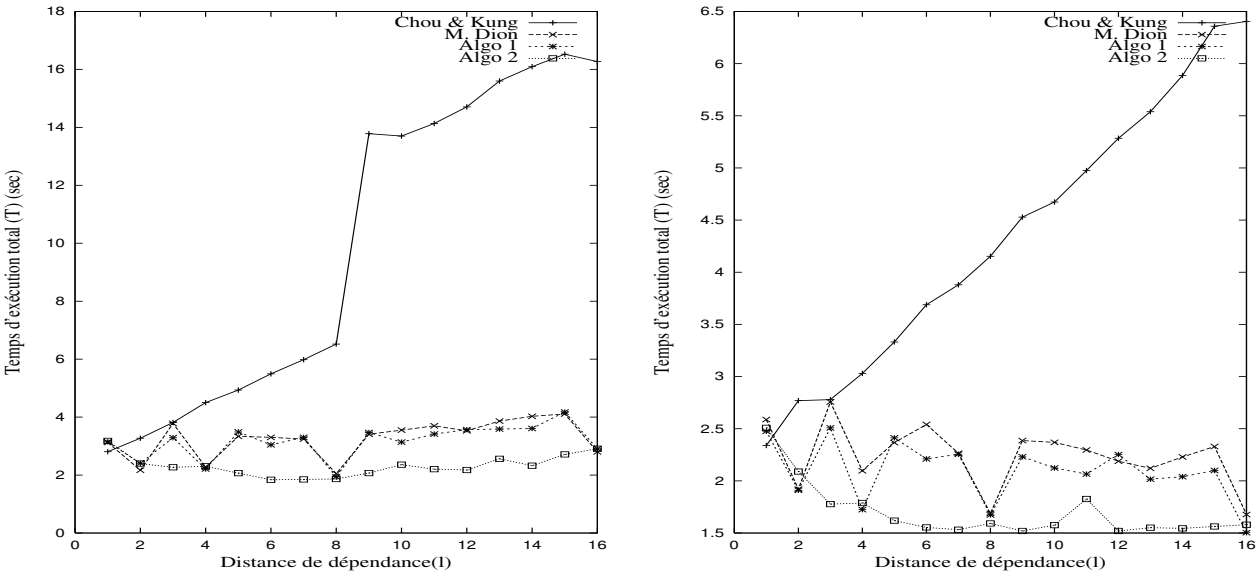


FIG. 4.7: *Temps d'exécution total de la boucle en fonction de la distance de dépendance sur un Cray T3E pour (a) P = 16, n = 32 et N = 4096 et (b) P = 16, n = 64 et N = 4096.*

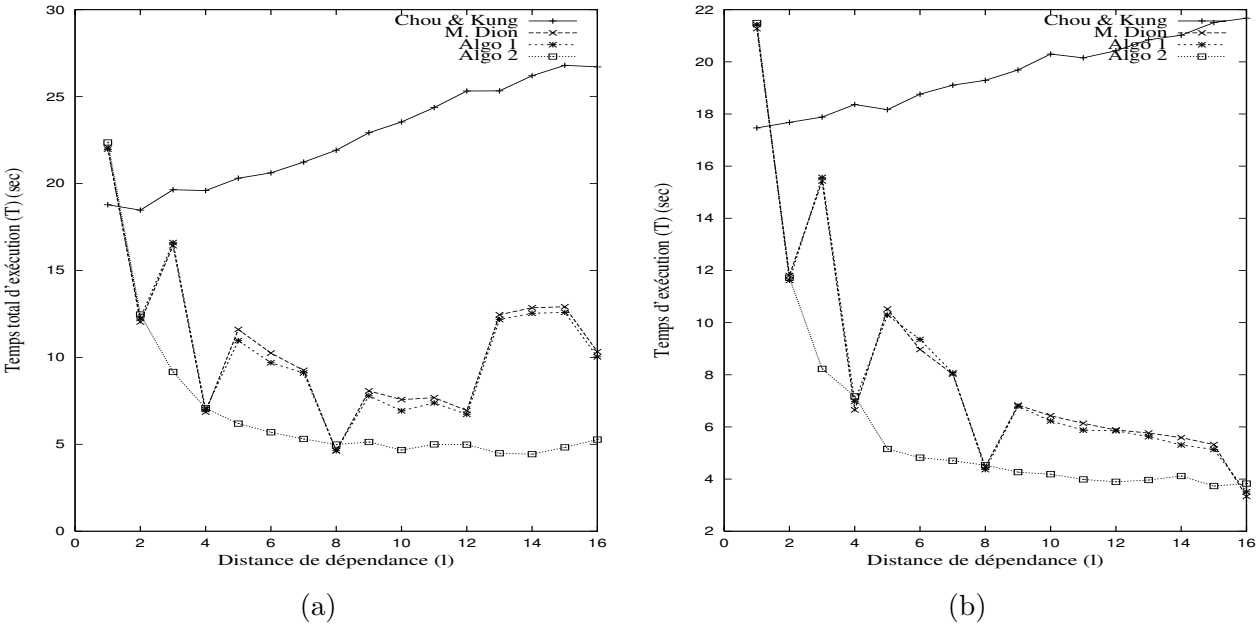


FIG. 4.8: *Temps d'exécution total de la boucle en fonction de la distance de dépendance sur un Cray T3E pour (a) P = 16, n = 512 et N = 65536 et (b) P = 16, n = 1024 et N = 65536.*

4.6.2 Taille de tuile optimale

Comme discuté dans le Chapitre 2, les différents paramètres qui influencent le choix de la taille des tuiles, sont la taille de cache, le rapport *temps de calcul/temps de communication élémentaire*, et la taille du domaine d'itérations. Dans ce paragraphe, nous étudions sommairement le cas où l'espace d'itérations est de dimension 1.

Dans ce cadre, la taille de cache est rarement limitative, et nous pouvons considérer le temps de calcul τ_{calc} comme une constante indépendante de la taille des tuiles n. Avec les mêmes notations que celles utilisées dans les paragraphes précédents, et en utilisant l'Algorithme 5, on obtient la période d'ordonnancement suivante :

$$T \approx \frac{n}{l}\tau_{calc} + \tau_{comm}$$

Le temps total d'exécution de la boucle est donné par l'expression,

$$\begin{aligned} T_{tot} &\approx \frac{N}{n}T + n\tau_{calc} \\ &= \frac{N\tau_{calc}}{l} + n\tau_{calc} + \frac{N\tau_{comm}}{n} \\ &= A + Bn + \frac{C}{n} \end{aligned}$$

Ainsi, la taille de tuile qui minimise cette expression vaut,

$$n_{opt} = \sqrt{\frac{C}{B}} = \sqrt{\frac{N\tau_{comm}}{\tau_{calc}}}$$

Posons $\frac{\tau_{comm}}{\tau_{calc}} = c$. Alors (indépendamment de la valeur de l) ,

$$n_{opt} \approx \sqrt{cN}$$

Finalement, pour que les communications soient recouvertes par les calculs, il faut que

$$\frac{n}{l}\tau_{calc} > \tau_{comm}$$

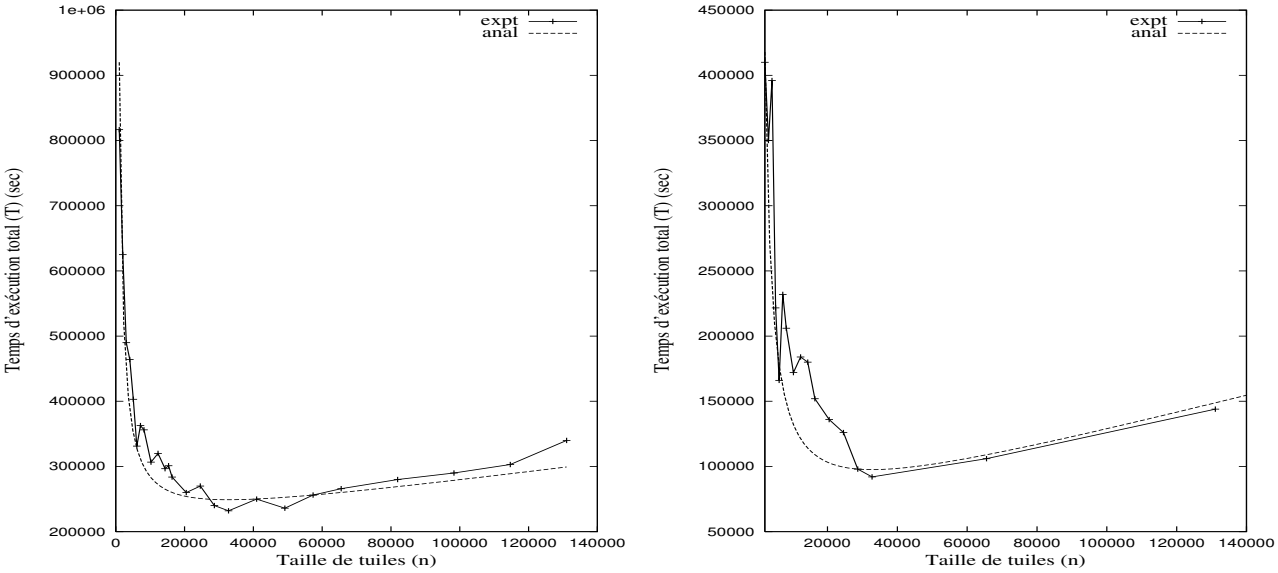
Ce qui donne,

$$n > lc$$

La Figure 4.9 confronte cette solution analytique avec des résultats expérimentaux effectués sur 16 processeurs d'une SGI Power Challenge. Sur ce système, nous avons mesuré pour notre application une valeur de $c = 2048$. Ainsi, pour de petites tailles de tuiles ($n \lesssim lc$), la courbe expérimentale est plus haute que la courbe théorique.

4.7 Généralisation aux dimensions supérieures.

Nous pourrions traiter le problème de dimension d, comme nous l'avons fait pour le problème de dimension 1, et chercher pour toute instance du problème, la permutation constante ou non constante optimale.



(a) (b)

FIG. 4.9: *Temps d'exécution de la boucle en fonction de la taille des tuiles sur une SGI Power Challenge pour (a) $P = 16$, $l = 7$ et $N = 2097152$ et (b) $P = 16$, $l = 7$ et $N = 524288$.*

Notre approche est tout autre. En fait nous souhaitons dans ce paragraphe montrer comment les résultats de dimension 1 peuvent être *appliqués* aux cas de dimensions supérieures. Considérons par exemple le nid de boucles suivant :

Programme 17. *Exemple de code correspondant aux cas de dimension supérieure à 1*

do $k = 0, L-1$
  do $i = 0, N-1$
    do $j = 0, M-1$
      Boucles internes
        $A[\ldots, i, j, \ldots] = f(A[\ldots, i - b[k], j, \ldots], A[\ldots, i, j - c[k], \ldots])$

Remarquons que dans cet exemple, il y a deux dépendances parallèles aux axes. Dans le cas d'une dépendance quelconque, on se ramènerait au cas précédent en projetant le vecteur de dépendance suivant chacune des deux directions principales. Ainsi, la dépendance $(4, 5)$ serait décomposée en deux dépendances $(4, 0)$ et $(0, 5)$. Il faut bien voir que cette simplification est *très restrictive*.
Un fois pavé avec des tuiles de taille $n \times m$, on obtient :

Programme 18. *Programme 17 pavé*

```
do k = 0, L
    do I = 0, N − 1 : n
        do J = 0, M − 1 : m
            do i = I, min(N, I + n) − 1
                do j = J, min(M, J + m) − 1
                    Tâche(i, j)
```

Notons σ_i (respectivement σ_j) la permutation du problème unidimensionnel $(n, b[k])$ (respectivement $(m, c[k])$). L'idée est d'effectuer, dans chacune des deux directions et à l'intérieur de chaque tuile, ces deux permutations. On obtient alors le code suivant :

Programme 19. *Programme 18 après permutation des tâches à l'intérieur de chaque tuile*

```
do k = 0, L
    do I = 0, N − 1 : n
        do J = 0, M − 1 : m
            do i' = I, min(N, I + n) − 1
                { Calcule σ_i et σ_j }
                do j' = J, min(M, J + m) − 1
                    { Si nécessaire réceptionne les données du processeur de dessus et/ou de gauche}
                    Tâche(σ_i(i'), σ_j(j'))
                    { Si nécessaire envoie les données au processeur de dessous et/ou de droite}
```

Mais cette solution favorise clairement la direction j, donc est favorable au cas où $\frac{M}{m} \gg \frac{N}{n}$. Si par contre, $\frac{M}{m} \ll \frac{N}{n}$ il suffirait bien sûr de permuter i' et j' afin de favoriser la direction i. En fait, ces ordonnancements ne sont optimaux que si les dépendances sont de longueur 1 dans chacune des deux directions (pour un k donné, $b[k] = 1$ et $c[k] = 1$).

L'idée consiste en fait à partitionner chaque tuile en sous-tuiles, à effectuer un ordonnancement à l'intérieur de chaque sous-tuile (favorable à la direction la plus grande), puis à trouver un ordonnancement des sous-tuiles qui minimise le chemin critique du graphe des tâches (i.e. $T_j\frac{M}{m} + T_i\frac{N}{n}$ où T_i et T_j représentent les périodes d'ordonnancement dans chacune des directions i et j).

Pour fixer les idées, considérons deux exemples. Dans ces exemples, on notera b et c les distances de dépendance dans chacune des directions i et j (dépendances $(b, 0)$ et $(0, c)$). On supposera aussi que la permutation dans chacune des directions est fournie par l'Algorithme 5.

4.7.1 Exemple 1 : $b = c = 1$

Considérons la période obtenue par un ordonnancement lexicographique des tâches à l'intérieur de la tuile. Cela se note $\sigma_{lex.ij}(j' + mi') = (i', j')$ si la boucle i' englobe la boucle j' (comme dans le programme ci-dessus), et $\sigma_{lex.ji}(i' + nj') = (i', j')$ si la boucle j' englobe la boucle i'.

Dans le premier cas, la tuile du dessous doit attendre que la première colonne soit exécutée avant de démarrer, ce qui donne $T_j = m$. La tuile de droite, elle, doit attendre que les $(n - 1)$ premières colonnes puis que la tâche en haut à droite soient exécutées avant de pouvoir démarrer. Ce qui donne $T_i = m(n - 1) + 1$. En résumé, dans le cas $\sigma_{lex.ij}$ on obtient $(T_i, T_j) = (m(n - 1) + 1, m)$. Symétriquement, dans le cas $\sigma_{lex.ji}$ on obtient $(T_i, T_j) = (n, n(m - 1) + 1)$.

D'autres solutions intermédiaires comme celle présentée Figure 4.10 sont possibles. Mais, comme en témoigne le Théorème 10, les seules solutions minimisant la fonction de coût $T_j\frac{M}{m} + T_i\frac{N}{n}$ sont lexicographiques.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

(a)

1	3	5	7	9
2	4	6	8	10
11	13	15	17	19
12	14	16	18	20

(b)

FIG. 4.10: *Si* $\frac{M}{m} = 90$ *et* $\frac{N}{n} = 120$. *Dans le cas (a) on a* $T_i = 5$ *et* $T_j = 16$. *La longueur du chemin critique vaut donc* $5 \times 120 + 16 \times 90 = 2040$. *Dans le cas (b) on a,* $T_i = 9$ *et* $T_j = 12$, *et la longueur du chemin critique vaut* $9 \times 120 + 12 \times 90 = 2160$. *Ainsi, l'ordre lexicographique est meilleur que l'ordre "pavé".*

Théorème 10 *Si* $b = c = 0$ *alors si* $M \geq N$, $\sigma_{lex.ij}$ *est optimal et si* $M \leq N$, $\sigma_{lex.ji}$ *est optimal.*

Preuve. Supposons, sans perte de généralité, que $M \leq N$. Notons $T_i^{lex.ji}$ et $T_j^{lex.ji}$ les périodes dans chacune des directions i et j de l'ordonnancement $\sigma_{lex.ji}$. Notons T_i et T_j les périodes d'un ordonnancement valide. On a $T_j\frac{M}{m} + T_i\frac{N}{n} = (T_i + T_j)\frac{M}{m} + T_i(\frac{N}{n} - \frac{M}{m})$. La preuve est fondée sur les deux inégalités suivantes :

$$T_i + T_j \geq nm + 1 = T_i^{lex.ji} + T_j^{lex.ji} \tag{4.3}$$

$$T_i \geq n = T_i^{lex.ji} \tag{4.4}$$

La preuve de l'Inégalité (4.3) est donnée par la Figure 4.11. La preuve de l'Inégalité (4.4) est immédiate.

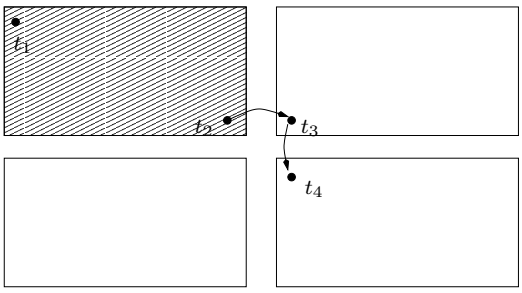


FIG. 4.11: *Afin de pouvoir exécuter la tâche* t_4, *la tâche* t_3 *et donc la tâche* t_2 *doivent être exécutées. Ainsi, le temps entre le début de l'exécution de la tâche* t_1 *et le début de l'exécution de la tâche* t_4, *est supérieur au temps nécessaire à l'exécution de la première tuile suivi de* t_3. *Soit* $T_i + T_j \geq nm + 1$.

■

4.7.2 Exemple 2 : $n = \beta b$ et $m = \gamma c$

L'idée, dans ce cas, est de paver chaque tuile en sous-tuiles. Une sous-tuile correspond à une partie connexe du graphe des tâches. Ainsi, on a $b \times c$ sous-tuiles de taille $\beta \times \gamma$. Les sous-tuiles ayant toutes la même taille, l'ordre dans lequel on les exécute importe peu. Ensuite, selon que $\frac{N}{n} \geq \frac{M}{m}$ ou pas, les tâches à l'intérieur de chaque sous-tuile seront ordonnées lexicographiquement dans le sens (ji) ou dans le sens (ij). Ainsi le code correspondant au cas où $M \geq N$ s'écrirait :

Programme 20.

```
do k = 0, L
    do I = 0, N − 1 : n
        do J = 0, M − 1 : m
            do I' = I, min(N, I + n) − 1 : β
                do J' = J, min(M, J + m) − 1 : γ
                    do i' = I', min(N, I' + β) − 1
                        do j' = J', min(M, J' + γ) − 1
                            { Si nécessaire réceptionne les données du dessus et/ou de la gauche }
                            Tâche(σ_i(i'), σ_j(j'))
                            { Si nécessaire envoie les données au dessus et/ou à droite}
```

Cas "général". Dans le cas plus général où les sous-graphes connexes n'ont pas la même taille, une analyse rapide montre que l'ordre dans lequel les sous-tuiles correspondantes sont exécutées, est important. La résolution de ce problème d'ordonnancement semble être compliquée. De plus, le problème est combinatoire, comme en témoigne le nombre d'extensions linéaires de l'ordre défini par le rectangle[4] $b \times c$ égal à

$$\frac{(bc)! \left(\Pi_{i=1}^{b-1} i!\right) \left(\Pi_{i=1}^{c-1} i!\right)}{\Pi_{i=1}^{b+c-1} i!}$$

Le problème reste donc largement ouvert.

4.8 Conclusion

La technique de pavage est habituellement utilisée pour augmenter la granularité de calculs et la localité des accès aux données. Pour les applications à gros grains, le pavage du nid de boucles, nécessaire à la localité des données, risque de tuer une partie du parallélisme par l'introduction d'une latence trop importante entre le calcul de deux tuiles successives. Or, le plus souvent, les tuiles contiennent du parallélisme interne qui s'exprime de manière plus ou moins triviale. Il s'utilise en pratique par une exécution des tuiles sous forme de pipeline.

Le but de ce chapitre a été de mettre en évidence ce parallélisme dans le cas de dépendances uniformes de longueur $l > 1$, en permutant l'ordre d'exécution des tâches au sein de chaque tuile. Nous avons proposé une solution optimale dans le cas de permutations constantes mais aussi de permutations non constantes. L'intérêt de la recherche de permutations constantes peut être motivé

[4] Cette formule découle de la formule des équerres (Hook formula) [46] pour les tableaux de Young standard.

par l'implémentation de nids de boucles sur circuit VLSI, système sur lequel le code exécuté doit être le plus simple possible.

Au travers de quelques exemples, nous avons soulevé quelques problématiques relatives à la généralisation de notre solution au cas multidimensionnel. En fait, cela permet d'entrevoir une méthodologie différente des approches classiques en matière de pavage. Celle-ci consisterait dans un premier temps, à paver l'espace d'itérations afin d'optimiser le temps d'exécution monoprocesseur (favoriser la localité des données), puis dans un deuxième temps, exprimer le parallélisme interne à chaque tuile, et ajuster, dans la limite des degrés de liberté disponibles, la latence dans chacune des directions. Il faudrait alors définir convenablement le domaine d'applicabilité, puis chercher les paramètres optimaux d'un tel pavage (taille de tuiles, ordonnancement interne des calculs et des communications). Notons que le problème de pavage présenté dans le Paragraphe 2.1 (Espace d'itérations de taille $N \times M$ distribué horizontalement en blocs cycliques sur P processeurs), est un cas particulier de cette approche : une tuile est un bloc de colonnes (rectangle de taille $n \times M$) dont la largeur influence la localité des accès aux données ; chaque tuile est pavée en sous-tuiles (rectangles de taille $n \times m$) dont la hauteur influence la granularité du calcul.

Chapitre 5

Application des techniques de pavage à la parallélisation du problème de Fermi, Pasta et Ulam

5.1 Introduction

Dans ce chapitre, nous présentons les résultats de travaux effectués en collaboration avec Thierry Dauxois, chercheur au laboratoire de physique de l'ENS de Lyon. Le but est de paralléliser un code de type SOR[1] le plus efficacement possible sur une pile de PCs. Ainsi, nous avons mis en pratique les techniques de pavage présentées dans le Chapitre 2.

Le problème physique est relié à la théorie des systèmes dynamiques. Dans ce domaine, de nombreux travaux ont été effectués afin d'essayer de caractériser le chaos dans les systèmes dynamiques de grande dimension. Néanmoins, de nombreux points fondamentaux restent incompris, notamment le rapport entre l'analyse d'instabilité de Lyapunov et les propriétés de l'espace des phases, telles que la diffusion d'orbites, la relaxation vers des états d'équilibre, ou encore le développement spatial d'instabilités.

Nous proposons ici une parallélisation, basée sur l'utilisation de calcul redondant, du calcul des exposants de Lyapunov d'une chaîne d'oscillateurs de Fermi, Pasta et Ulam (FPU). Il faut noter, que l'un des buts de ce travail est de montrer qu'il est *aussi* possible d'obtenir d'*excellentes performances*, en effectuant du calcul redondant. Cette approche n'est pas classique, et nous souhaitons montrer ses avantages. Soulignons que le code ainsi généré est *très simple*, et ceci est un critère de poids pour l'utilisateur physicien non spécialiste. Par ailleurs, afin d'évaluer les différents paramètres de notre solution, avec laquelle nous souhaitons implémenter le programme, nous effectuons plusieurs analyses de performance : nous analysons l'impact des accès mémoire, ainsi que le surcoût dû aux communications. Les performances finales obtenues, très proches de l'optimal, valident ainsi nos choix.

Le plan de ce chapitre se décompose de la manière suivante : dans le Paragraphe 5.2 nous introduisons le problème, présentons le système étudié et le programme séquentiel. Dans le Paragraphe 5.3, nous présentons notre stratégie de parallélisation que nous confrontons à une approche plus classique. Ensuite, le Paragraphe 5.4 est consacré à la recherche des paramètres optimaux. Nous y analysons l'impact des accès mémoire et du temps de communication, puis nous y présentons les performances obtenues sur une pile de 25 PCs hétérogènes. Finalement, nous formulons quelques

[1] Successive Over Relaxation

remarques en guise de conclusion au Paragraphe 5.5.

5.2 Formulation du problème

Dans les années 80, Ruffo et al. [73] ont établi que les exposants de Lyapunov ne dépendent que de la densité d'énergie et pas du nombre de sites de la chaîne. Ce résultat majeur a été récemment mis en doute par des simulations de Searles et al. [86] qui semblent mettre en évidence une dépendance logarithmique, ce qui est extrêmement difficile à confirmer ou infirmer numériquement si l'on n'est pas capable d'obtenir des algorithmes particulièrement rapides. Une deuxième question majeure est relative au comportement à très basse énergie de l'exposant de Lyapunov maximal. Dans cette zone totalement inexplorée où l'on a essentiellement des couches chaotiques distinctes, on s'intéresse à la vitesse de convergence vers l'équipartition de l'énergie du système lorsque l'on part de conditions particulières. Cette équipartition prédite par la mécanique statistique à l'équilibre est extrêmement lente à atteindre et nécessite là encore des moyens de calcul importants. C'est un élément essentiel pour conclure le paradoxe de Fermi-Pasta-Ulam débuté dans les années 50.

5.2.1 Problème de Fermi, Pasta et Ulam et exposants de Lyapunov

Comme présenté dans la Figure 2.3 du Chapitre 2 (Page 18), si l'on note $x_i(t)$ (respectivement $v_i(t)$) la position (respectivement la vitesse) du i-éme atome ($i \in [0, H-1]$) de la chaîne d'oscillateurs au temps t, l'équation du mouvement de la chaîne β-FPU s'écrit

$$\left\{ \begin{array}{rcl} \dot{x}_i & = & v_i \\ \dot{v}_i & = & x_{i+1} + x_{i-1} - 2x_i + \beta \ \left[(x_{i+1} - x_i)^3 - (x_i - x_{i-1})^3 \right] \end{array} \right. \tag{5.1}$$

où $\beta = 0.1$, pour des raisons historiques. Nous avons choisi de traiter un problème à bords fixes, mais le problème périodique existe. A priori, l'évolution des deux systèmes est semblable. Le problème est, bien sûr, discrétisé dans le temps, utilisant un pas de temps $dt = 0.01$. L'algorithme d'intégration utilisé est celui de McLachlan-Atela [74], car nous souhaitons préserver le plus possible la structure Hamiltonième du problème. En effet, cet algorithme permet d'obtenir une conservation de l'énergie avec 8 chiffres significatifs.

Ici, nous nous intéressons à l'estimation des N_{lyap} plus grands exposants de Lyapunov λ_j ($1 \leq j \leq N_{lyap}$) en fonction de la densité d'énergie et du nombre H d'atomes dans la chaîne. Les exposants de Lyapunov jouent un rôle important dans la théorie des systèmes dynamiques Hamiltoniens ou dissipatifs [83]. Ils permettent d'évaluer quantitativement le degré de stochasticité d'une trajectoire.

La procédure, pour calculer les exposants de Lyapunov, a été développée par Benettin et al. [44]. Le premier exposant λ_1 est donné par le taux de croissance exponentiel de la différence entre deux trajectoires infiniment proches. Pour mesurer cette croissance, nous étudions l'évolution du vecteur[2] $\begin{pmatrix} \delta x^j \\ \delta v^j \end{pmatrix}$, tangent à l'orbite $\begin{pmatrix} x \\ v \end{pmatrix}$, défini comme suit :

$$\left\{ \begin{array}{rcl} \delta\dot{x}_i^1 & = & \delta v_i^1 \\ \delta\dot{v}_i^1 & = & \left[1 + 3\beta(x_{i+1} - x_i)^2\right] \delta x_{i+1}^1 + \left[1 + 3\beta(x_{i-1} - x_i)^2\right] \delta x_{i-1}^1 \\ & & \quad - \left[2 + 3\beta \left[(x_{i+1} - x_i)^2 + (x_{i-1} - x_i)^2\right]\right] \delta x_i^1 \end{array} \right. \tag{5.2}$$

[2] δx^j (respectivement $\delta v^j, x, v$) représente le vecteur colonne constitué des scalaires δx_i^j (respectivement $\delta v_i^j, x_i, v_i$).

Si on note $\|\delta_1(t)\| = \left\| \begin{pmatrix} \delta x^1(t) \\ \delta v^1(t) \end{pmatrix} \right\|$ la norme euclidienne de la différence à l'instant t des deux trajectoires $\begin{pmatrix} x \\ v \end{pmatrix}$ et $\begin{pmatrix} x + \delta x^1 \\ v + \delta v^1 \end{pmatrix}$, λ_1 est défini par :

$$\lambda_1 = \lim_{t \to \infty} \frac{1}{t} \ln \frac{\|\delta_1(t)\|}{\|\delta_1(0)\|}. \tag{5.3}$$

Le problème est que cette norme croît exponentiellement avec le temps t. Ainsi, devons nous souvent (toutes les n_{ortho} itérations de temps) renormaliser $\delta_1(t)$. La procédure est alors la suivante :

Programme 21. *Schéma de calcul du premier exposant de Lyapunov λ_1*

$lambda[1] = 0$
do $t = 0, T : dt$
  evolution de l'orbite $\begin{pmatrix} x \\ v \end{pmatrix}$ et de la perturbation $\begin{pmatrix} \delta x^1 \\ \delta v^1 \end{pmatrix}$.
  if $\frac{t}{dt} \equiv 0 \ [n_{ortho}]$
    $lambda[1] = lambda[1] + \ln \left\| \begin{pmatrix} \delta x^1 \\ \delta v^1 \end{pmatrix} \right\|$
    $\begin{pmatrix} \delta x^1 \\ \delta v^1 \end{pmatrix} = \dfrac{\begin{pmatrix} \delta x^1 \\ \delta v^1 \end{pmatrix}}{\left\| \begin{pmatrix} \delta x^1 \\ \delta v^1 \end{pmatrix} \right\|}$

Puis nous utilisons la propriété remarquable suivante (si n_{ortho} n'est pas trop grand) :

$$\lambda_1 = \lim_{T \to \infty} \frac{lambda[1]}{T} \tag{5.4}$$

Lorsqu'il y a plusieurs exposants de Lyapunov ($N_{lyap} > 1$), on rencontre le problème suivant : lors de l'évolution, l'angle entre les différents vecteurs tangents ($\begin{pmatrix} \delta x^j \\ \delta v^j \end{pmatrix}$, $1 \leq j \leq N_{lyap}$), tend vers 0 et devient trop petit pour être calculé numériquement. Il faut donc, pour stabiliser le programme numériquement, orthonormaliser à nouveau la famille de vecteurs $\begin{pmatrix} \delta x^j \\ \delta v^j \end{pmatrix}_{1 \leq j \leq N_{lyap}}$. C'est ce que l'on effectue tous les n_{ortho} unités de temps à l'aide de l'algorithme de Gram-Schmidt.

C'est ce programme appliqué à une chaîne de taille variant entre $H = 200$ et $H = 10^6$ et évoluant entre $\frac{T}{dt} = 10^5$ et $\frac{T}{dt} = 10^8$ itérations de temps (temps nécessaire à la convergence du système, observé expérimentalement), que nous nous sommes proposés de paralléliser.

5.2.2 Le programme séquentiel

Le code que nous allons étudier est constitué de trois phases. Après l'initialisation,
- la première phase ($0 \leq t \leq t_1$) est une phase de lancement. On fait évoluer l'orbite, et on ne touche pas aux perturbations.
- Durant la seconde phase ($t_1 < t \leq t_2$), on continue à faire évoluer l'orbite, parallèlement à laquelle évoluent les perturbations. On ne calcule pas encore les exposants de Lyapunov, le système ne s'étant pas encore stabilisé.

– Durant la dernière phase ($t_2 < t \leq t_3 = T$), tous les calculs sont effectués : évolution de l'orbite et des perturbations ainsi que mise à jour des exposants de Lyapunov.

Ainsi, le code[3] s'écrit :

Programme 22. *Le code séquentiel*

Initialisation
do $t = 0, T : dt$
  if $t > t_2$ {phase 2 ou 3}
    do $k = 0, 3$ {L'algorithme de McLachlan-Atela décompose un pas de temps en 4 itérations}
      doall $j \in \{1, \ldots, N_{lyap}\}$
        dovect $i \in \{0, \ldots, H-1\}$
          $\delta v_i^j = \delta v_i^j + c_k \times g_k(x_{i-1}, x_i, x_{i+1}, \delta x_{i-1}^j, \delta x_i^j, \delta x_{i+1}^j)$
          $\delta x_i^j = \delta x_i^j + d_k \times \delta v_i^j$
        } Evolution des perturbations
    if $\frac{t}{dt} \equiv 0 \ [n_{ortho}]$
      Gram_Schmidt $\begin{pmatrix} \delta x^1 & \ldots & \delta x^N \\ \delta v^1 & \ldots & \delta v^N \end{pmatrix}$
      if $t > t_2$ {phase 3} then met à jour les N_{lyap} exposants de Lyapunov
  do $k = 0, 3$
    dovect $i \in \{0, \ldots, H-1\}$
      $v_i = v_i + c_k \times f_k(x_{i-1}, x_i, x_{i+1})$
      $x_i = x_i + d_k \times v_i$
    } Evolution des orbites

Dans ce code,
– H, représente le nombre de particules.
– N_{lyap}, représente le nombre d'exposants de Lyapunov.
– c_k et d_k sont les coefficients de l'algorithme de McLachlan-Atela [74].
– f et g sont les fonctions obtenues à partir des équations de mouvement (5.1) et de leurs dérivées (5.2).

Notons, qu'en pratique, $T = t_3 \gg t_2$, c'est-à-dire que la majorité du temps de calcul est constitué par la troisième phase. La mise à jour des exposants de Lyapunov étant très rapide, les phases 2 et 3 sont donc comparables et nous étudierons à partir de maintenant la phase 2 du programme.

Le graphe des tâches. Le graphe des tâches est légèrement différent de celui (simplifié) présenté dans le Chapitre 2. En fait, le graphe de dépendances réduit, diffère de celui de la Figure 2.4 principalement par la présence d'une orthonormalisation de Gram-Schmidt effectuée tous les n_{ortho} pas de temps. Cette orthonormalisation est assez fréquente et peut être considérée comme une barrière de synchronisation. Ainsi, le pavage de la Figure 2.6 (Page 19) ne constitue pas une solution possible à notre problème de parallélisation.

Par souci de clarté, le graphe de tâches est décomposé en deux schémas.
– Le schéma de la Figure 5.1 représente les dépendances exclusives au calcul de l'orbite. Un calcul d'orbite y est représenté par un cercle.
– Le schéma de la Figure 5.2 représente seulement les dépendances relatives au calcul des perturbations. Le calcul d'une perturbation y est représenté par une croix. Les quatre phases

[3] la notation dovect signifie que le calcul est effectué vectoriellement. Une exécution séquentielle de cette boucle nécessiterait l'utilisation de variables temporaires.

de l'algorithme de McLachlan-Atela *du calcul d'orbite* ne sont pas représentées.
– La superposition de ces deux graphes fournit le graphe des tâches réduit du programme FPU schématisé Figure 5.3 : graphe semblable à celui présenté dans le Chapitre 2 Page 19, qui en diffère par la présence de barrières de synchronisations.

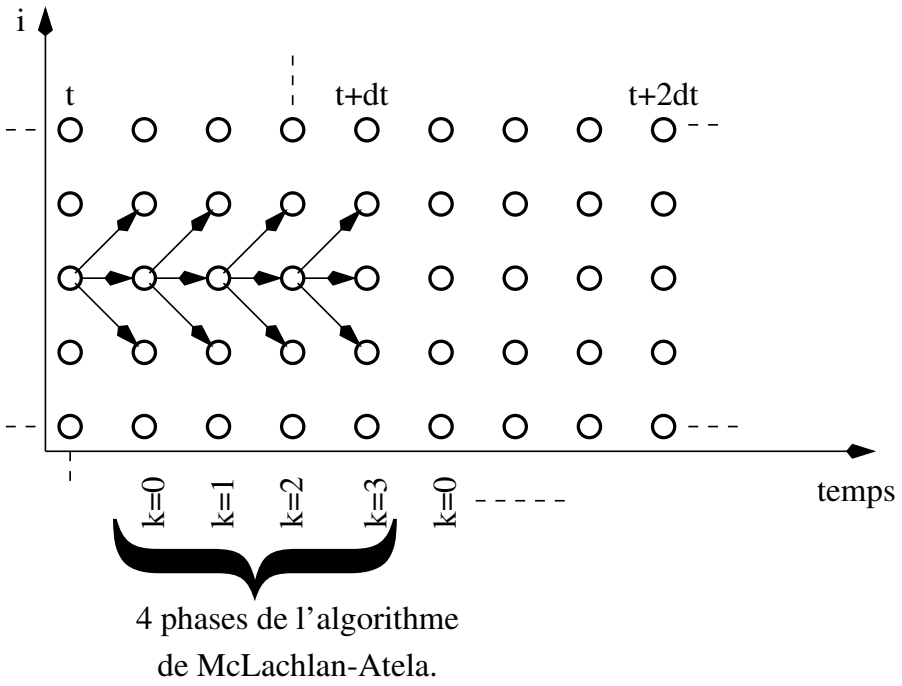


FIG. 5.1: *Graphe de tâches du calcul des orbites. Un cercle représente le calcul de x_i et de v_i. Les dépendances réduites sont $(1,1)^t$, $(1,0)^t$ et $(1,-1)^t$.*

5.3 Stratégies de parallélisation

5.3.1 Intérêt du pavage

Les applications pratiques qui nous intéressent mettent souvent en jeu un nombre de particules très important, de l'ordre de $H = 10^5$, $H = 10^6$, *ou plus si possible*. Ainsi les parallélisations usuelles visant à simplement distribuer la direction d'espace i sur les processeurs, donnent des performances catastrophiques. En effet, l'approche naïve présentée Figure 5.4 fournit, pour de grandes valeurs de H, une granularité suffisante pour que le surcoût de communication soit négligeable devant le temps de calcul. En contrepartie la majorité des accès mémoire n'étant pas locaux, le temps de calcul monoprocesseur sur un super-calculateur risque d'être 5 à 20 fois plus lent que pour une version pavée.

En fait, une bonne parallélisation de ce code doit chercher à paver chaque bande de calcul située entre deux phases d'orthonormalisation afin de favoriser la localité des accès mémoire sans tuer le parallélisme, puis distribuer les tuiles sur les processeurs. Du fait que $n_{ortho} \lesssim 25$ est petit, le partitionnement théoriquement optimal consisterait à partitionner chaque bande de taille $4n_{ortho} \times H$ en parallélogrammes de hauteur $h = \frac{H}{P}$, comme indiqué Figure 5.6. En fait, la solution que nous proposons ici, *beaucoup* plus simple à implémenter, conduit à d'excellentes performances (cf 5.4.5). Elle utilise la notion de réplication de tâches. Elle constitue probablement une solution esthétique

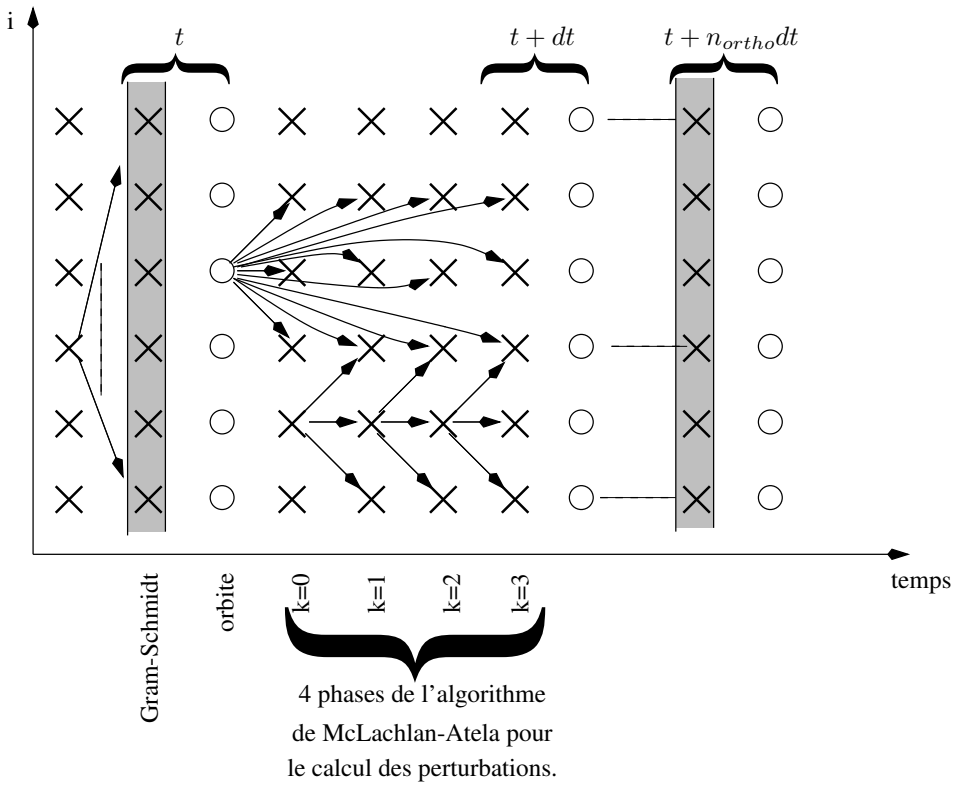


FIG. 5.2: *Graphe de tâches du calcul des perturbations. Un cercle représente x_i et v_i. Une croix représente le calcul de δx_i et de δv_i. Les dépendances réduites sont $(1,1)^t$, $(1,0)^t$ et $(1,-1)^t$. La phase d'orthonormalisation se comporte comme une barrière de synchronisation.*

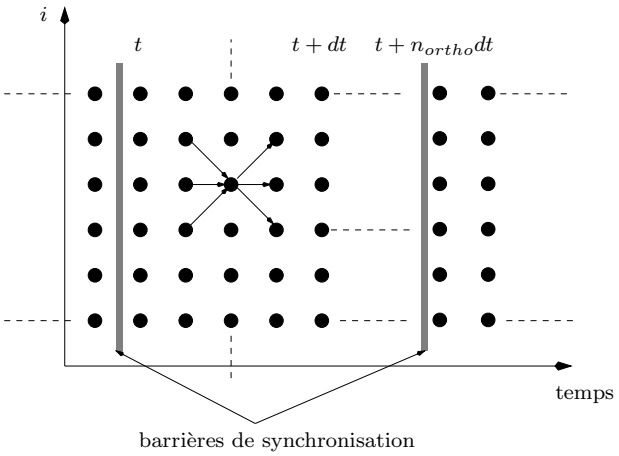


FIG. 5.3: *Graphe de tâches réduit du programme fpu : Le graphe du calcul des perturbations et celui du calcul d'orbite ont été superposés. Les dépendances réduites sont alors $(1,1)^t$, $(1,0)^t$ et $(1,-1)^t$. Afin d'éviter l'utilisation d'un tableau temporaire supplémentaire, les quatre phases du calcul des perturbations au temps $t + dt$ dépendant de l'orbite au temps t, sont considérées comme insécables.*

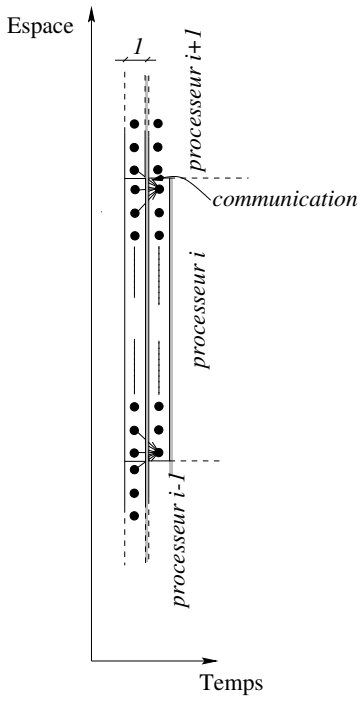


FIG. 5.4: *Approche "naïve" consistant à partitionner l'espace d'itérations en P bandes horizontales. Après chaque étape, chaque processeur communique avec ses voisins les données nécessaires au calculs suivants.*

à la parallélisation de nombreuses applications régulières, contenant un fort degré de parallélisme et traitant de larges domaines de données.

Le paragraphe qui suit est consacré à la description de ces deux solutions.

5.3.2 Tuiles rectangulaires ou tuiles parallélogrammes

Solution parallélogramme. Pour de larges domaines d'itérations, quelle que soit l'approche utilisée, les différents résultats de la littérature s'accordent sur le fait que les bordures des tuiles doivent être parallèles au cône de dépendances. Dans notre exemple, les dépendances réduites étant $(1,1)^t$ et $(1,-1)^t$, les tuiles devraient être de forme losange, comme schématisé Figure 5.5. Si l'on rajoute sur le graphe des tâches après $4n_{ortho}$ itérations, la phase d'orthonormalisation, on voit apparaître des tuiles de forme parallélogramme (non atomiques). Cela correspond à la solution présentée Figure 5.6 : chaque bande diagonale est assignée à un processeur ; chaque processeur gère donc un tableau de dimension $h = H/P$, correspondant à la hauteur de la bande. Le problème de cette solution est, comme en témoigne la Figure 5.7, lié à la complexité de l'ensemble des données communiquées : si le problème traité est à bords fixes, cette stratégie étant, elle, purement cyclique, de nombreux cas de configuration doivent être considérés afin de gérer les effets de bords.

En fait, nous ne sommes pas assurés que cela ne se traduise pas par une *baisse des performances du noyau de calcul élémentaire* : en effet, le surcoût de contrôle lié à la présence de branchements conditionnels, nous obligerait à effectuer du déroulement de boucle. Ceci, ajouté à la gestion des tableaux de communication, alourdit considérablement le code correspondant à la partie la plus exécutée du programme. Cette perte de performances *ne constitue bien entendu qu'une hypothèse*,

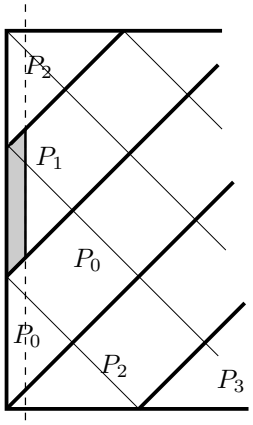


FIG. 5.5: *Lorsque l'espace d'itération est large, le pavage optimal est constitué de tuiles de forme losange. La présence de la phase d'orthonormalisation au bout de $4n_{ortho}$ itérations de temps (représentée par des pointillés) explique la forme parallélogramme de la solution présentée Figure 5.6.*

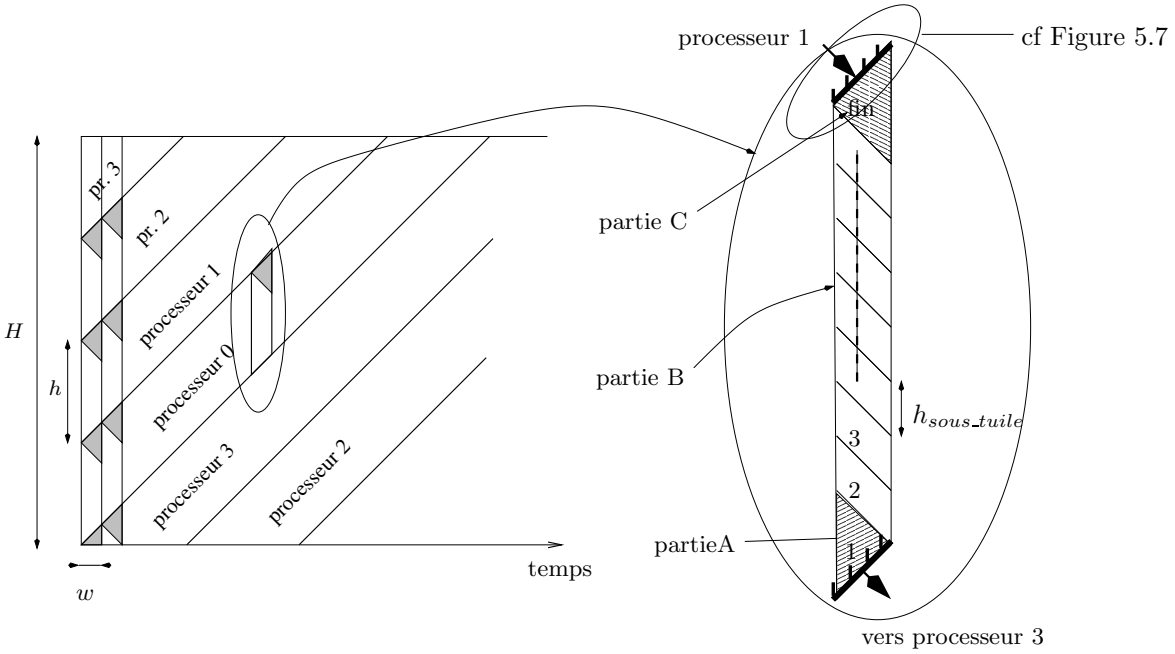


FIG. 5.6: *Solution parallélogramme : l'espace d'itérations est partitionné en bandes diagonales, distribuées cycliquement sur les processeurs. Ainsi, entre deux phases d'orthonormalisation, chaque processeur s'occupe d'une tuile de forme parallélogramme (non atomique). Du fait des dépendances, l'exécution de la Partie C nécessite des résultats de la Partie A exécutée par le processeur du dessus. Ainsi, consécutivement, chaque processeur effectue les calculs correspondant à sa Partie A ; ensuite, les données utiles au calcul de la Partie C sont communiquées entre processeurs voisins ; finalement chaque processeur exécute sa Partie B puis sa Partie C du bas vers le haut. Eventuellement, si h est grand, la Partie B est elle même pavée afin d'assurer la localité des accès aux données.*

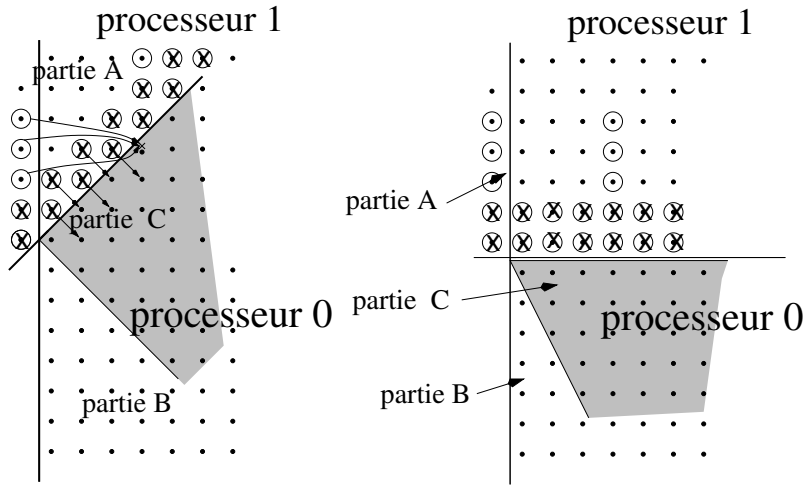


FIG. 5.7: *Partie du graphe de la Figure 5.6 avant et après torsion de l'espace d'itérations. Les données nécessaires au calcul de la Partie C par le processeur i sont matérialisées par un cercle pour les orbites (x_i & v_i) et par une croix pour les perturbations (δx_i^j & δv_i^j).*

et il faudrait mesurer l'impact réel qu'a la complexité du code, sur la vitesse d'exécution.

Solution rectangulaire. Comme expliqué Figure 5.8, la solution consiste simplement à distribuer chaque bande verticale sur les processeurs, en les partitionnant en rectangles de hauteur $h = \frac{H}{P}$ et de largeur w (w étant un multiple de 4 et un diviseur de $4n_{ortho}$). Bien entendu, du fait des dépendances, un tel découpage séquentialise le code. La solution consiste donc à faire faire, à chaque processeur, des calculs redondants : $w(w-1)$ tâches correspondant à deux triangles de part et d'autre du rectangle.

Lorsque w augmente, le surcoût de communications diminue, la localité des accès aux données augmente, donc le temps de calcul élémentaire diminue. En contrepartie, quand w augmente, la quantité de calcul redondant augmente aussi. Tous ces paramètres doivent donc intervenir dans le choix de la meilleure taille de tuile.

5.4 Mise en pratique

Le support parallèle utilisé est constitué de deux piles de PC interconnectés par un réseau Myrinet.
Le premier nommé PopC a les caractéristiques suivantes :

- 12 noeuds identiques
  - Processeurs : Pentium Pro 200Mhz (cache L2 : 256Ko),
  - Mémoire : 64Mo par noeud,
  - Carte Myrinet LANAI4.1 256Kbytes,
- 2 x switch Myrinet double M2M-DUAL-SW8/SAN (i.e. 4 crossbars 8x8),
- Hub Ethernet 10Mb/s 16 ports.
- Compilateur gcc (egcs-1.1.2) option -O3.

Le second nommé Pom a les caractéristiques suivantes :

- 16 noeuds identiques
  - Processeurs : PowerPC 604e 200Mhz (cache L1 : 32Ko+32Ko, cache L2 : 256Ko),

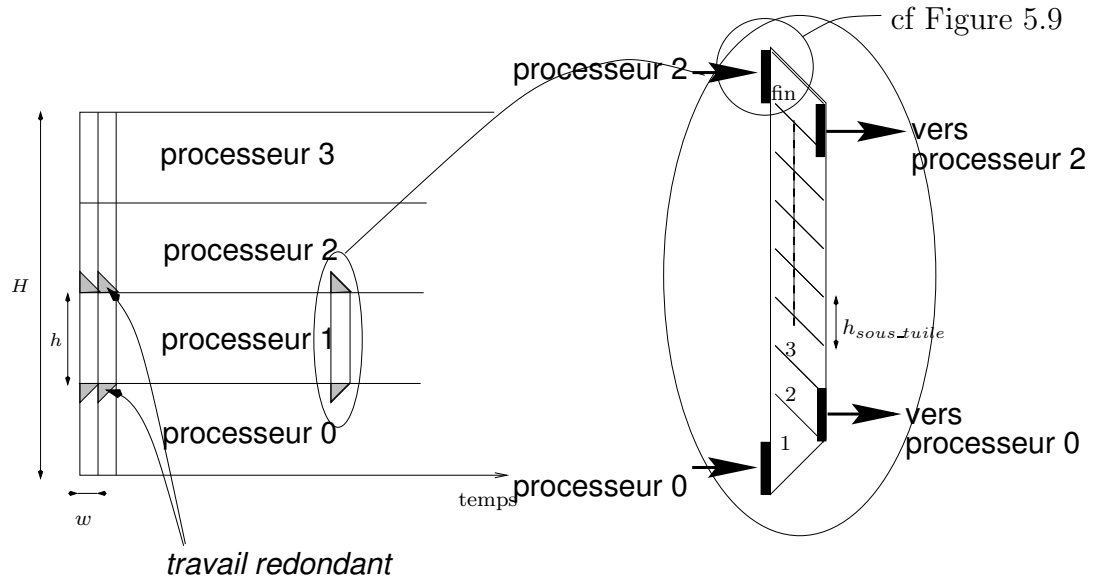


FIG. 5.8: *Solution rectangulaire : du fait des dépendances, le calcul de chaque rectangle nécessite le calcul de deux triangles situés de part et d'autre du rectangle. Le choix fait ici est de faire faire ce calcul par le processeur exécutant le rectangle. Cela correspond à un calcul redondant, car il est également effectué par chaque processeur voisin. Ainsi, avant chaque étape de calcul, chaque processeur communique à ses voisins les données utiles au calcul des parties redondantes. Si le domaine d'itérations est grand, alors le parallélépipède peut être lui-même partitionné en un triangle et en parallélogrammes de hauteur h_{sous_tuile} ; ces sous-tuiles étant exécutées du bas vers le haut.*

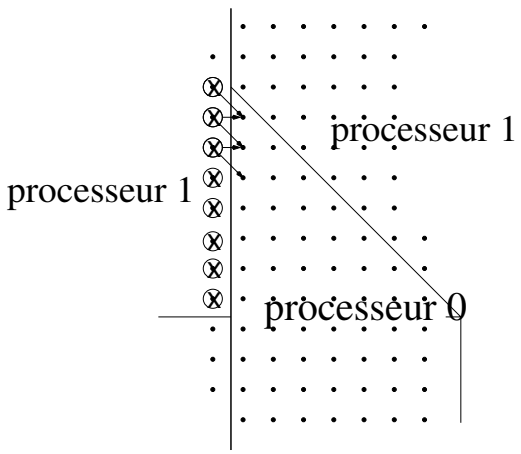


FIG. 5.9: *Partie du graphe de la Figure 5.8. L'ensemble des données communiquées sont matérialisées par un cercle pour x_i et v_i, et par une croix pour les perturbations.*

  - Mémoire : 64Mo par noeud,
  - Carte Myrinet LANAI4.1 512Kbytes,
- 2 x switch Myrinet octal M2M-OCT-SW8,
- Hub Ethernet 10Mb/s 24 ports.
- Compilateur gcc, version v2.7, option -O3.

Ces ressources parallèles sont disponibles parmi d'autres plateformes au Laboratoire de Calculs aux Hautes Performances (cf `http ://www.ens-lyon.fr/LHPC/FRANCAIS/choix.html`). L'interface de communication est développée par le LHPC : BIP, IP-BIP et MPI-BIP y sont disponibles [80].

Les mesures présentées dans ce paragraphe ont été effectuées sur la Pom.

5.4.1 Temps d'accès mémoire

Le but de ce paragraphe est de montrer l'importance du pavage sur monoprocesseur, et surtout de déduire un intervalle acceptable de hauteur de sous-tuile assurant des performances quasi-optimales. Nous avons pour cela effectué des mesures à l'aide d'un espace d'itération rectangulaire de taille $100 \times H$, exécuté en séquentiel, sans phase d'orthonormalisation.

Comme on peut le remarquer sur la Figure 5.10, l'exécution du programme non pavé met en évidence les deux niveaux de cache : trois marches distinctes sont visibles, $H < H_{L1} = 300$, $500 < H < 2000$ et $H > 5000$ correspondant respectivement au cache L1, L2 et à la mémoire. Du fait de la forme parallélogramme d'une sous-tuile, sa largeur w entre en jeu dans l'évaluation de sa hauteur h_{sous_tuile}. Ainsi, on veut que $w + h_{sous\_tuile} < H_{L1} = 300$. Comme $w \leq 4n_{ortho} \lesssim 100$, nous avons choisi $h_{sous\_tuile} = 100$.

Nous avons alors pavé notre programme séquentiel en tuiles de forme parallélogramme et de taille w. On remarque alors sur la Figure 5.10 que l'exécution du programme pavé est sensible aux accès mémoire pour $H > 3000$. En fait, il faudrait effectuer un pavage hiérarchique avec des "super-tuiles" de taille 1000×1000, ce qui n'est pas réalisable du fait des orthonormalisations tous les $4n_{ortho} \lesssim 100$ itérations.

Ainsi, à partir de maintenant, les tuiles de taille $w \times h$ sont pavées à l'aide de sous-tuiles de taille $w \times h_{sous\_tuile} = w \times 100$.

5.4.2 Taille de tuiles optimale sur monoprocesseur

L'expérience du paragraphe précédent nous a permis de fixer la hauteur des sous-tuiles à $h_{sous\_tuile} = 100$. Reste à fixer la largeur w des tuiles. Les expériences présentées dans ce paragraphe montrent qu'un modèle simple (temps de calcul = temps de calcul de base+temps d'accès mémoire) fournit une approximation correcte de la réalité, et qu'il est donc aisé (si on ne tient pas compte des communications) de trouver la largeur de tuile optimale. En fait, on verra dans le paragraphe suivant, qu'en moyenne, le temps de communication par colonne est inférieur à 200 fois le temps de calcul d'une tâche. Ainsi l'impact des communications vaut

$$\frac{\text{temps de communications par colonne}}{\text{temps de calcul d'une colonne}} < \frac{200}{h} = \frac{200P}{H}$$

Pour des tailles de domaine de l'ordre de $H \geq 4000P$, on pourra donc négliger l'impact des communications (5 %).

La largeur des tuiles étant limitée à $4n_{ortho} < 100$, plus les tuiles sont larges, plus les accès aux données sont locaux. En fait, si on note τ_{L2} (respectivement τ_{Mem}) le temps de rapatriement moyen

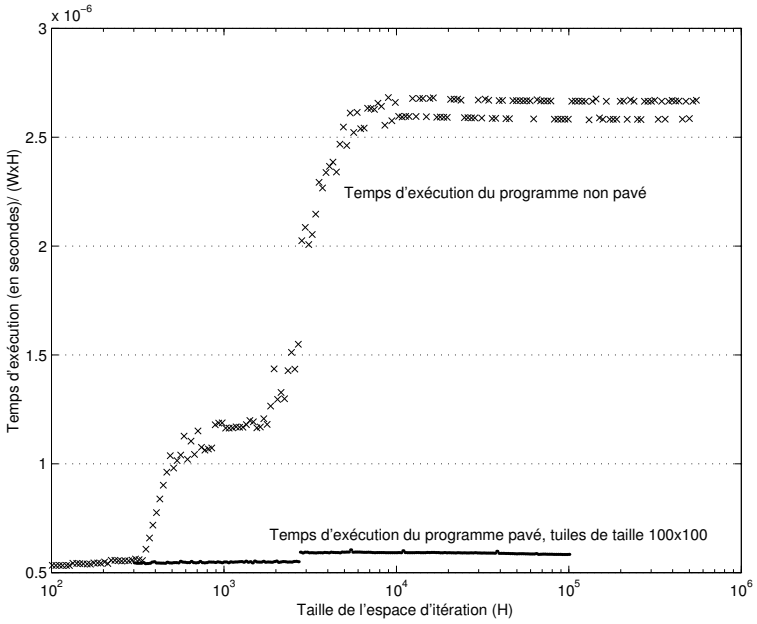


FIG. 5.10: *Temps en secondes pour exécuter une unité de calcul. Le nombre d'itérations est fixé à* 100. *Le nombre de particules varie entre* 10^2 *et* 10^6. *Pour* $500 < H < 3000$, *le programme pavé va plus de 2 fois plus vite que le programme non pavé, et pour* $H > 5000$, *il est plus de 5 fois plus rapide.*

d'une donnée du cache L2 vers le cache L1 (respectivement de la mémoire vers le cache L2) ; que l'on note τ_{calc} le temps de calcul élémentaire, alors le temps d'exécution d'un rectangle de taille $w \times h$ est donné par la formule (où $\delta_{h>3000}$ vaut 1 si $h > 3000$ et 0 sinon)

$$T_{calc}(h, w) \simeq hw \times \left((\tau_{L2} + \delta_{h>3000}\tau_{Mem}) \times \frac{1}{w} + \tau_{calc} \right).$$

C'est ce que l'on observe expérimentalement sur la Figure 5.11. Ainsi, pour favoriser la localité des accès aux données, la largeur des tuiles doit être la plus grande possible, mais en contrepartie la quantité de travail est augmentée d'un rapport $1 + (w - 1)/h$.

En fait, la formule du temps de calcul d'une tuile parallélépipède (rectangle de taille $w \times h$ plus deux triangles de hauteur w) est

$$T_{calc}(h, w) \simeq hw \times \left((\tau_{L2} + \delta_{h>3000}\tau_{Mem}) \times \left(\frac{1}{w} + \frac{2}{h} \right) + \left(1 + \frac{1}{h}(w - 1) \right) \tau_{calc} \right)$$

C'est ce que l'on peut observer sur la Figure 5.12 pour des valeurs de $w \gtrsim 15$. Pour des valeurs de $w \lesssim 15$, il faudrait, pour évaluer le surcoût d'accès mémoire, un modèle un peu plus précis que la simple formule $(\tau_{L2}+\delta_{h>3000}\tau_{Mem})$ (cf Figure 5.10). Mais comme nous le verrons dans le paragraphe suivant, pour les valeurs de h pour lesquelles w doit être petit, le surcoût des communications n'est plus du tout négligeable. Il devient donc, de toute manière, hasardeux de formuler analytiquement le temps d'exécution moyen dans ce cas de figure.

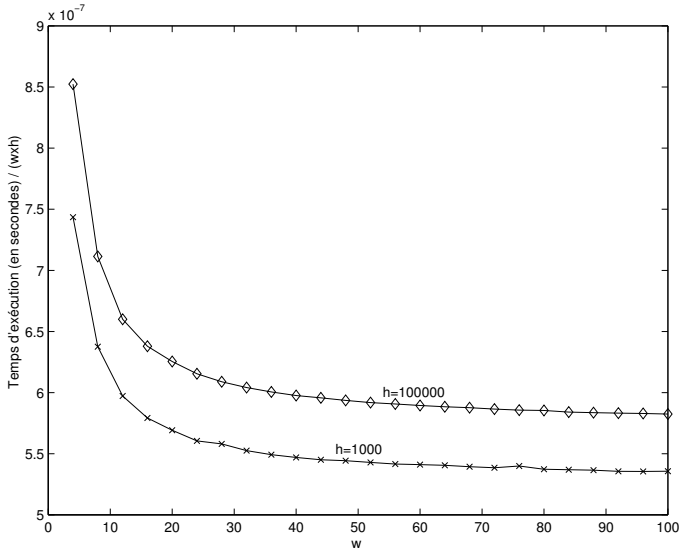


FIG. 5.11: *Influence de la largeur w sur la localité des accès aux données, pour un programme pavé séquentiel tel que : l'espace d'itérations est de taille $w \times h$; les tuiles sont de taille $w \times h_{sous\_tuile} = w \times 100$. Pour $h = 10^3$, toutes les données peuvent être stockées dans le cache L1 et dans le cache L2. En revanche, pour $h = 10^5$, l'impact du pavage est beaucoup plus important car les données doivent être rapatriées de la mémoire vers, consécutivement, le cache L2 puis le cache L1.*

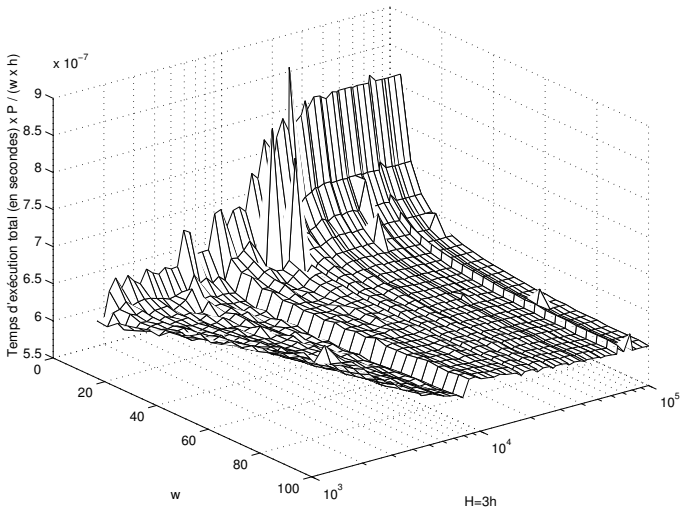


FIG. 5.12: *Influence de la largeur des tuiles (w) sur le temps d'exécution total (sans communication) en fonction de la taille du domaine (H) du programme parallèle pavé exécuté sur 3 processeurs.*

5.4.3 Impact des communications

Dans ce paragraphe, nous évaluons l'impact des communications sur le temps d'exécution global de l'espace d'itérations. Comme expliqué dans le Chapitre 2, on utilise usuellement un modèle formé de deux composantes pour décrire le temps de communication : le temps d'initialisation de la communication β, et le temps de communication propre $l\tau$, proportionnel au nombre de données communiquées l. Le plus souvent $\beta \gg \tau$, ce qui motive le pavage du domaine d'itérations afin d'augmenter la granularité du calcul. Ainsi, pour un domaine de largeur $W = 4\frac{T}{dt}$, pavé en tuiles de taille $w \times h = w \times \frac{H}{P}$, il y a $\frac{W}{w}$ étapes de communication de taille proportionnelle à w. En normalisant par W, on obtient un coût de communication par colonne valant $\frac{\beta}{w} + \tau$. En fait, comme le montre la Figure 5.13, il en est tout autrement :

- On observe deux phases consécutives : lorsque le message est plus grand que 960 bytes ($w \geq 20$), le message est décomposé en plusieurs sous-messages par l'interface de communication, ce qui diminue les performances. Cela se traduit, sur les courbes, par un palier entre $w = 16$ et $w = 20$.
- Le surcoût de communications est très sensible à la quantité de données utilisées par le programme (cf Figure 5.14) :
  - Il semble que le contexte des communications soit enlevé du cache par les calculs entre chaque phase de communication. Ceci expliquerait pourquoi le surcoût de communication (temps de calcul avec communications moins temps de calcul sans communication) est plus important que le temps de communication pur sans calcul intermédiaire.
  - Ensuite, il peut se passer aussi l'inverse, c'est à dire que la présence de communications augmente le temps de calcul élémentaire. Ainsi, la bosse de la Figure 5.14, aux alentours de $h = \frac{H}{P} \approx 2000$, correspond au cas où les données sans communication tiendraient dans le cache L2 mais de manière limite.

5.4.4 Paramètres optimaux

A la lumière des remarques précédentes, nous pouvons séparer trois cas :

- Pour de petites hauteurs de domaine $\frac{H}{P}$, le surcoût dû à la redondance de travail est significatif : w doit être inférieur à 20 dans ce cas. Dans ce contexte, plus les tuiles sont larges, moins le surcoût de communication est important, plus le temps de calcul de base est rapide (à cause de la localité), mais plus on effectue, en contrepartie, de travail redondant. Dans ce cadre, coût de communication et temps de calcul élémentaire l'emportent. La meilleure largeur de tuile est donc $w = 16$ (cf Figure 5.15).
- Pour des domaines d'itérations larges, ni le temps de communication, ni le travail redondant n'ont un impact significatif sur le temps d'exécution total. Ainsi, dans ce cas, la meilleure largeur de tuile est $w = 4n_{ortho} = 100$ (cf Figure 5.15).
- Entre ces deux cas extrêmes, du fait de l'aspect chaotique du surcoût de communication, il est difficile de trouver une solution analytique. Notre solution a consisté à tester toutes les possibilités (seulement 25 cas possibles pour w) en prenant $W = 4\frac{T}{dt} = 500$ (cf Figure 5.16).

5.4.5 Mesures de performances

Dans ce paragraphe, nous présentons les performances obtenues sur la pile de PC décrite au Paragraphe 5.4. Malheureusement, du fait de la forte utilisation de ces ressources de calcul, il est assez difficile d'obtenir une réservation simultanée de l'ensemble des processeurs. Il a donc fallu, rajouter une routine d'équilibrage de charge à notre programme : ainsi, la hauteur d'une tuile h_j

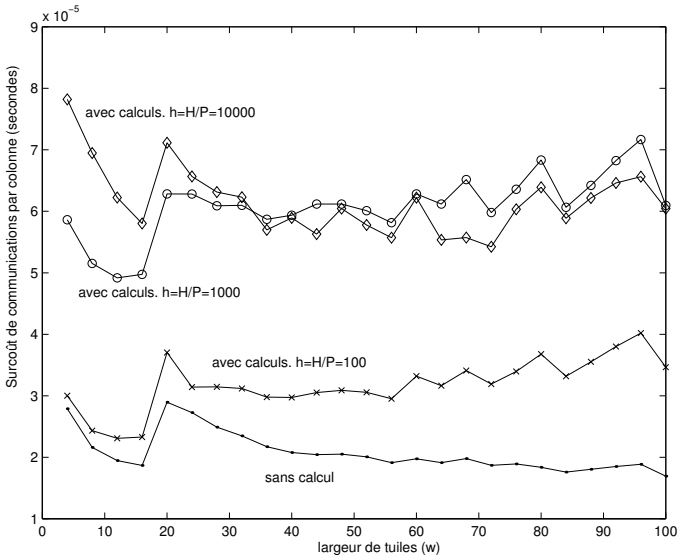


FIG. 5.13: *Evaluation du surcoût de communication. Plusieurs tests sont effectués. Le premier ("sans calcul") correspond a mesurer le temps d'exécution du programme dans lequel les calculs ont été commentés. Le second ("avec calculs") correspond à faire la différence entre le temps d'exécution du programme normal et le temps d'exécution du programme dans lequel les communications ont été commentées. Nous avons effectué ce second test pour différentes valeurs de* $h = \frac{H}{P} \in \{10^2, 10^3, 10^4\}$*, car comme on peut le voir Figure 5.14 le surcoût de communication dépend de la taille du domaine.*

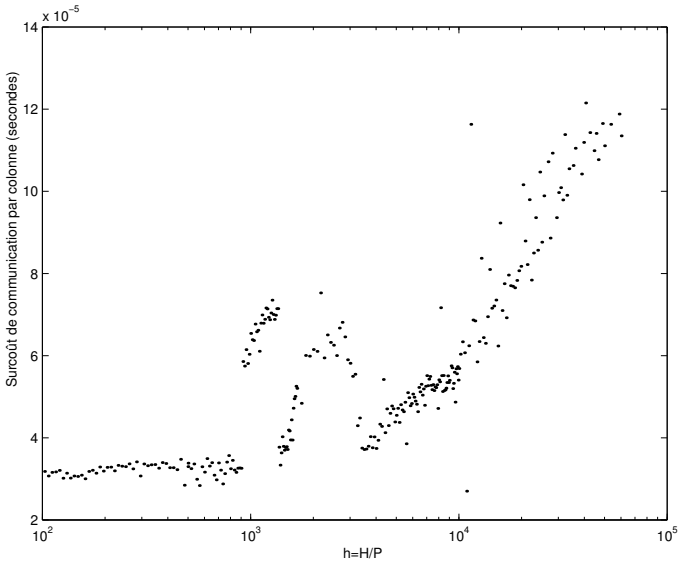


FIG. 5.14: *Influence du nombre de données par processeur sur le surcoût de communication.* $h = \frac{H}{P}$ *varie de* 10^2 *à* 10^5*, pour* $P = 6$*, et* $w = 32$*.*

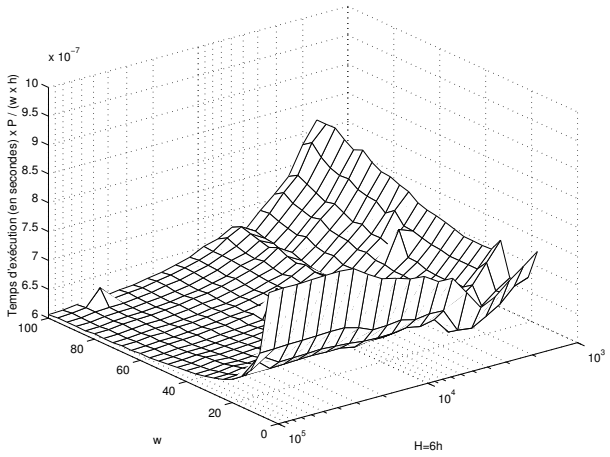


FIG. 5.15: *Influence de la largeur des tuiles (w) sur le temps total d'exécution, en fonction de la hauteur du domaine H. La taille du domaine d'itération est de $w \times H$; $h = \frac{H}{P}$; $P = 6$.*

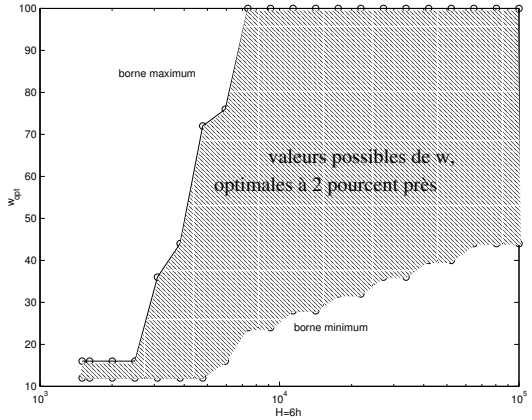


FIG. 5.16: *Valeurs optimales de w en fonction de H sur 6 processeurs. Si l'on note $t_{calc}(H, w)$ le temps d'exécution moyen d'une tâche ; pour une hauteur de domaine donné H, on note w_{opt} la largeur de tuile optimale : $t_{calc}(H, w_{opt}) = \min_w t_{calc}(H, w)$. Alors en autorisant une erreur de 2 %, la borne inférieure correspond à $\min\{w,\ t_{calc}(H, w) < t_{calc}(H, w_{opt}) \times 1.02\}$ et la borne supérieure à $\max\{w,\ t_{calc}(H, w) < t_{calc}(H, w_{opt}) \times 1.02\}$.*

exécutée par le processeur P_j est calculée de telle manière que $\forall j, T_{calc}(h_j, w, P_j) \simeq Constante$. L'imperfection de la procédure (manque de réactivité) explique l'irrégularité des courbes présentées ici. De plus, afin de pouvoir générer simplement les expériences, *nous nous sommes restreint à une largeur de tuile de* $w = 100$, alors que de plus petites valeurs auraient fourni de meilleures performances pour de petits espaces d'itération (cf 5.16). Nous verrons que les résultats dans ce cas sont néanmoins corrects. Deux expériences ont été effectuées :

- l'une sur 11 processeurs homogènes de la pom.
- l'autre sur 25 processeurs hétérogènes : 14 processeurs de la pom de temps de cycle moyen 3.93×10^{-7} et 11 processeurs de la popc de temps de cycle moyen 7.09×10^{-7}.

Pour chacune de ces expériences, nous avons fait varier la taille du problème H entre 4000 et 10^6. Le domaine d'itération a alors pour taille $W \times H = 4\frac{T}{dt} \times H = 500 \times H$. Nous avons reporté,

1. le temps d'exécution moyen d'une tâche $t_{calc} = \frac{T_{exe\_para}}{WH}$.
2. le facteur d'accélération[4] de l'exécution parallèle.

Expérience sur 11 processeurs. Les résultats sont reportés dans les figures 5.17 et 5.18.

La Figure 5.17 représente deux courbes :

- le *temps d'exécution moyen d'une tâche*, correspond au temps total d'exécution divisé par la taille du domaine d'itération $W \times H$.
- le *temps d'exécution optimal* (droite horizontale) est calculé comme suit : pour toute taille de domaine, le temps d'exécution séquentiel moyen d'une tâche est mesuré. Nous avons pris la valeur minimum de ces temps divisée par 11.

La Figure 5.18 représente le facteur d'accélération de notre exécution parallèle : pour une taille de problème donnée, le programme est exécuté en séquentiel et en parallèle sur 11 processeurs. Le facteur d'accélération est le rapport de ces deux valeurs. Pour cet exemple, un bon facteur doit être situé aux alentours de 11.

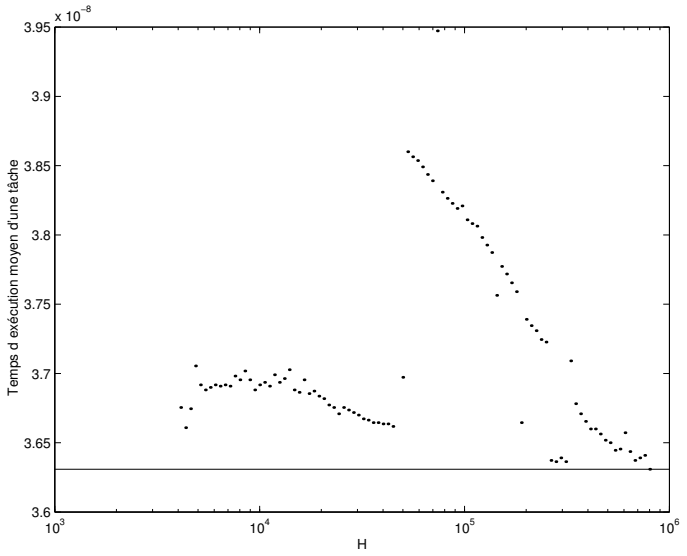


FIG. 5.17: *Temps d'exécution moyen d'une tâche sur* 11 *processeurs de la pom pour une taille de problème variant entre* $H = 4000$ *et* $H = 10^6$.

[4] *speedup* en anglais

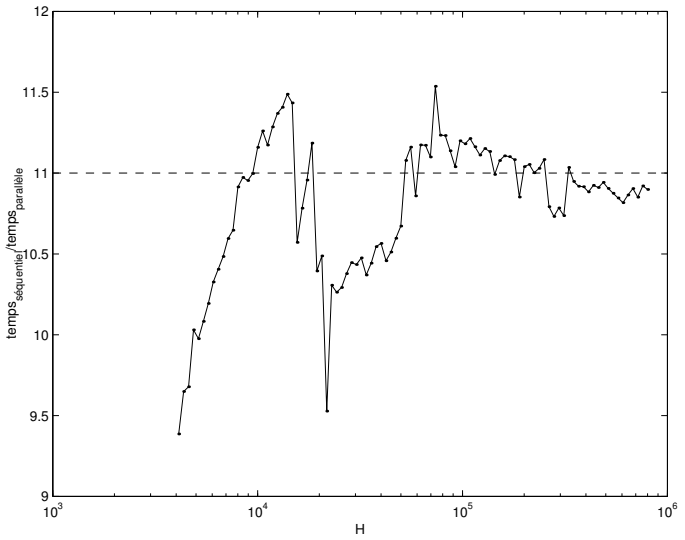


FIG. 5.18: *Facteur d'accélération de l'exécution parallèle sur* 11 *processeurs de la pom.*

La courbe de temps d'exécution (Figure 5.17) présente un palier correspondant à la limite de la mémoire cache. La courbe de temps d'exécution ayant la même allure mais décalée vers la gauche, le rapport du temps séquentiel sur le temps parallèle représenté Figure 5.18 contient deux bosses (dépassant ici la valeur "optimale" 11).

Expérience sur 25 processeurs. Les résultats sont reportés dans les figures 5.19 et 5.20. La Figure 5.19 représente deux courbes :

- le temps d'exécution moyen, correspond au temps total d'exécution divisé par la taille du domaine d'itération $W \times H = 500 \times H$.
- le *temps d'exécution extrapolé optimal* (droite horizontale) est calculé comme suit : le temps d'exécution séquentiel moyen d'une tâche minimal (sur l'ensemble des tailles de problèmes) est mesuré sur un noeud de la pom (noté $t_{seq}(pom)$), ainsi que sur un noeud de la popc (noté $t_{seq}(popc)$). Le temps séquentiel est alors extrapolé par la formule

$$\frac{1}{\frac{11}{t_{seq}(popc)} + \frac{15}{t_{seq}(pom)}} \simeq 1.95 \times 10^{-8}$$

.

La Figure 5.20 représente le facteur d'accélération de notre exécution parallèle.

De même que pour la Figure 5.17, la courbe de temps d'exécution (Figure 5.19) présente un palier correspondant au dépassement de la taille de mémoire cache. Par contre, on remarque un temps d'exécution important pour de faibles tailles de domaine. Cela est dû aux temps de communication entre la popc et la pom, beaucoup plus lent qu'entre deux noeuds de la pom, et dont l'impact est non négligeable.

5.5 Conclusion

Dans ce chapitre, nous avons choisi une technique de parallélisation d'un code de type SOR. Nous avons proposé une solution à la fois esthétique et efficace, basée sur l'utilisation de calculs

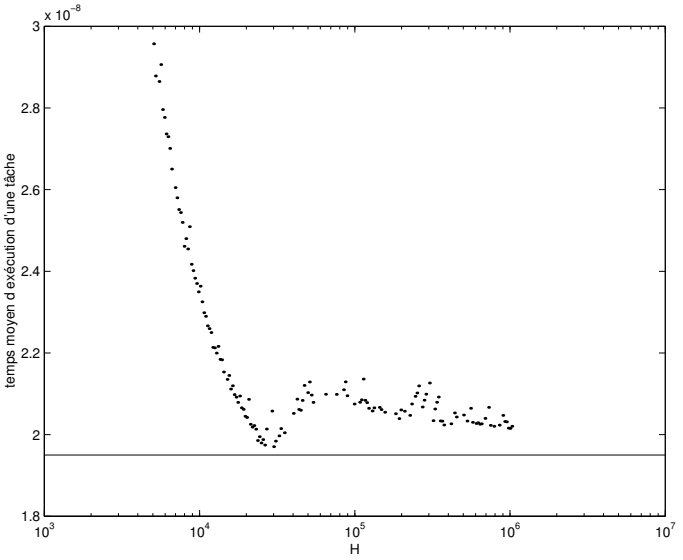


FIG. 5.19: *Temps d'exécution du programme sur* 25 *processeurs,* 11 *processeurs de la popc et* 14 *processeurs de la pom, pour une taille de problème variant entre* $H = 5000$ *et* $H = 10^6$.

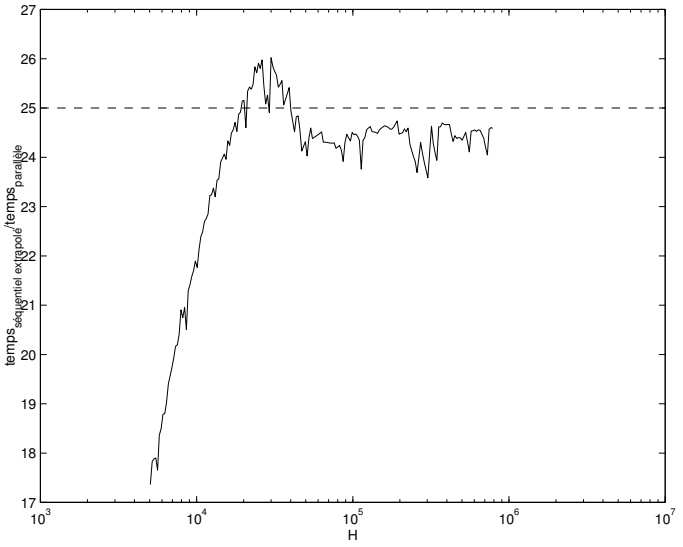


FIG. 5.20: *Facteur d'accélération de l'exécution parallèle sur* 25 *processeurs.*

redondants, qui préserve le parallélisme, la localité des accès aux données, et la granularité des calculs. Classiquement, la théorie du pavage préfère considérer des tuiles tordues, et ne pense pas à la redondance. Les résultats présentés ici ont donc montré qu'une telle méthode, pouvant aboutir à un code simple et très performant, se doit d'être considérée.

D'autre part, ce travail a été l'occasion d'évaluer les modèles usuellement utilisés en techniques de pavage pour la recherche de la taille et de la forme optimale de tuile. Ces expériences ont notamment montré qu'il est illusoire de vouloir utiliser une formule analytique pour caractériser l'impact des communications. Ainsi, au lieu de chercher une formule, exprimant en fonction de *tous les paramètres*, le temps total d'exécution, puis de déduire *la* configuration optimale minimisant cette valeur, il est plus judicieux de traiter les différents paramètres séparément, et en fonction d'une marge d'erreur raisonnablement choisie (pour notre application, nous nous sommes autorisés une erreur de 2% par rapport à l'optimal), de déterminer les contraintes associées. Ces contraintes se présentent généralement sous forme d'inégalités (granularité des calculs, localité des données, etc.), de différents choix possibles sur la forme des tuiles (dépendances, parallélisme, chemin critique), etc. Elles peuvent être évaluées indépendamment ou bien l'une après l'autre, dans leur ordre d'importance relative, par exemple. S'il existe finalement plusieurs solutions, c'est la plus simple qui doit être choisie. Notons que cette méthode est applicable à la technique d'optimisation de code qu'est le pavage, du fait du caractère non combinatoire des problèmes intermédiaires qui s'y posent.

Chapitre 6

Ressources hétérogènes : pavage, distribution 1D et équilibrage de charge

6.1 Introduction

Dans ce chapitre, nous abordons différents problèmes liés à l'implémentation de programmes sur un réseau de stations hétérogène[1]. Un tel travail est motivé par l'existence de très nombreuses machines de ce type, autant dans les universités que dans les entreprises. Un réseau hétérogène est la machine parallèle "du pauvre". L'idée est d'utiliser conjointement, pour une application MPI ou PVM, le maximum de ressources de calcul de la machine la plus lente au super-calculateur, mais aussi de partager les ressources d'une machine parallèle avec d'autres utilisateurs.

Tous les problèmes habituels liés à l'implémentation de programmes sur machines parallèles (équilibrage de charge, ordonnancement des tâches et des communications, algorithmique parallèle, etc.) se posent, de manière encore plus complexe, dans le contexte de ressources hétérogènes. Par exemple, nous verrons que si, pour une application à gros grains de calculs, un algorithme glouton est souvent quasi-optimal dans le cadre de ressources homogènes, il n'en est rien ici. Notons que les résultats présentés dans ce chapitre constituent une première approche du problème. Ainsi, nous restreignons notre recherche aux allocations statiques, nous supposons donc les ressources dédiées, c'est-à-dire que leur puissance ne varie pas au cours du temps. D'autre part, nous ne considérons ici que des allocations unidimensionnelles, les chapitres suivants étant consacrés aux problèmes liés à l'allocation bidimensionnelle des données.

Afin d'étayer notre étude, nous aborderons consécutivement plusieurs problèmes classiques de parallélisation. Dans un premier temps, nous nous restreindrons au cas simple d'équilibrage de charge : étant données M tâches à gros grains, indépendantes, et p ressources de puissance de calcul respectivement $\frac{1}{t_1}, \frac{1}{t_2}, \ldots, \frac{1}{t_p}$. Nous devons résoudre les questions suivantes : doit-on utiliser toutes les ressources ? Quelle tâche doit-on allouer à chaque processeur afin que le temps d'exécution global soit le plus petit possible ? Ces questions feront l'objet du Paragraphe 6.2 que nous illustrerons à l'aide du programme de multiplication de matrices. Ensuite, nous aborderons le problème sous une approche plus algorithmique : le programme de décomposition LU nécessite que l'équilibrage de charge soit à la fois global mais aussi local, c'est à dire qu'un sous-groupe de tâches, considéré indépendamment, soit lui aussi correctement équilibré sur les différentes ressources. Nous proposons

[1] En anglais *HNOWs* de "Heterogeneous Network of Workstations."

à cet effet un algorithme nommé *incrémental*. C'est ce qui fait l'objet du Paragraphe 6.3 de ce chapitre. Notons que le programme de décomposition LU ne constitue ici qu'une illustration, une étude algorithmique plus approfondie de cet exemple faisant l'objet du Chapitre 7 de cette thèse. Finalement, le Paragraphe 6.4 fournit une extension des résultats présentés dans le Chapitre 2, au cas hétérogène.

6.2 Equilibrage de charge.

L'utilisation efficace de ressources hétérogènes pour l'exécution d'une application, a fait l'objet d'une attention importante ces dernières années. Nous fournissons dans ce paragraphe quelques références à la littérature existante.

Lorsque l'ensemble des ressources n'est pas dédié, que la charge de chaque processeur n'est pas prévisible, ou même qu'une panne de l'une des ressources est envisageable, une allocation dynamique des tâches semble incontournable. Les stratégies dynamiques vont du simple paradigme maître-esclave (où chaque processeur choisit dans une file d'attente une nouvelle tâche à exécuter dès que son travail courant est terminé) à des stratégies plus sophistiquées qui, le plus souvent, décident d'une nouvelle distribution des tâches en fonction du travail effectué par chaque processeur durant un passé proche. Pour plus de détails, nous référons le lecteur à l'article de synthèse de Berman [13] et aux articles plus spécialisés [4, 31, 32, 43, 52, 59]. Les modèles de prédiction de performances sont entre autres traités dans [40, 95].

Dans le cas de ressources dédiées, la flexibilité de *l'équilibrage de charge dynamique* rend cette approche attrayante : la charge de chaque machine peut s'auto-réguler et donc s'auto-équilibrer malgré l'hétérogénéité des ressources de calcul. Mais, comme nous pourrons le voir dans ce chapitre, la présence de dépendances de données, risque de réduire l'activité de *tous* les processeurs à celle du processeur le plus lent. Ceci motive l'approche statique choisie dans ce chapitre, ainsi que le choix d'une approche semi-statique dans le cas de ressources non dédiées. De plus, les applications qui nous concernent ici (algèbre linéaire, programme de relaxation) étant très régulières, la prédiction de performance ne nécessite pas l'utilisation d'une stratégie sophistiquée, et nous pouvons utiliser une approximation, *a priori*, de la vitesse de chaque processeur. Nous pouvons donc chercher à allouer *statiquement* les tâches, *avant* le début de l'exécution.

6.2.1 Allocation statique de tâches

Formulation du problème. Dans le cas d'une allocation statique de noyaux de calcul numérique, la vitesse d'un processeur est évaluée grâce à une mesure préalable du temps d'exécution d'un noyau. Ce noyau devant être, bien entendu, représentatif des calculs effectués par l'application. Notons $t_1, t_2, \ldots, t_p$, les temps mesurés sur chaque processeur, que l'on nomme "*temps de cycle des processeurs*".

Supposons que les tâches à distribuer sur les processeurs correspondent à une quantité de travail identique (*homogénéité des tâches*) et qu'elles soient *atomiques* et indépendantes. Nous nommons désormais cette entité de base *un atome*. Si le nombre d'atomes est petit, l'équilibrage de charge n'est pas trivial : intuitivement, le nombre de tâches à distribuer par processeur (notons le $c_1, c_2, \ldots, c_p$) doit être inversement proportionnel à t_i. Formellement, il faut que

$$c_i \times t_i = Constante = K$$

En d'autres termes, s'il y a B atomes à distribuer,

$$c_i = \frac{\frac{1}{t_i}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times B$$

Bien sûr, si M est un multiple de $C = \text{ppcm}(t_1, t_2, \ldots, t_p) \sum_{i=1}^{p} \frac{1}{t_i}$, l'équilibrage de charge est parfait. En contrepartie, si B est petit, et que B n'est pas multiple de C, on voit bien que la formule ci-dessus ne fournit pas un renseignement assez précis (car c_i doit être un entier). En particulier, il n'est pas clair, pour un processeur P_i tel que $c_i < 1$, que l'on doive lui allouer un atome ou non.
En fait, l'allocation optimale est obtenue à l'aide de l'Algorithme 7. Le principe de cet algorithme a déjà été utilisé dans la littérature notamment pour résoudre le problème dual d'allocation de tâches hétérogènes sur un ensemble homogène de processeurs [76].

Algorithme 7. *Allocation statique optimale de B atomes indépendants sur p processeurs de temps de cycle respectifs $t_1, t_2, \ldots, t_p$.*

Distribue$(B, t_1, t_2, \ldots, t_p)$
{ Initialisation : calcule c_i tels que $c_i \times t_i \approx Constante$ et $c_1 + c_2 + \ldots + c_p \leq B$ }
do $i = 1, p$
    $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times B \right\rfloor$
{ Incrémente itérativement les c_i qui minimisent le temps d'exécution tant que $\sum_{k=1}^{p} c_i < B$}
while $\sum_{i=1}^{p} c_i < B$ do
    trouve $k \in \{1, \ldots, p\}$ tel que $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1)\}$
    $c_k = c_k + 1$
Retourne$(c_1, c_2, \ldots, c_k)$

Proposition 1 *L'Algorithme 7 fournit l'allocation optimale.*

Preuve. Considérons une allocation optimale notée $o_1, o_2, \ldots, o_p$.
Soit j tel que $\forall i \in \{1, \ldots, p\}$, $T_{exe} = o_j t_j \geq o_i t_i$. Afin de prouver que l'algorithme est correct, nous prouvons l'invariant

$$(C) : \forall i \in \{1, \ldots, p\},\ c_i t_i \leq o_j t_j$$

Après l'étape d'initialisation, $c_i \leq \frac{\frac{1}{t_k}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times B$. Comme, $B = \sum_{k=1}^{p} o_k \leq o_j t_j \times \sum_{k=1}^{p} \frac{1}{t_k}$, on a $c_i t_i \leq \frac{B}{\sum_{k=1}^{p} \frac{1}{t_k}} \leq o_j t_j$, et la condition (C) est vérifiée.
Le fait que (C) soit vérifiée *après chaque incrémentation* est prouvé par récurrence : supposons pour cela, qu'à une étape donnée, c_k soit incrémenté. Avant cette étape, $\sum_{i=1}^{p} c_i < B$, ainsi il existe $k' \in \{1, \ldots, p\}$ tel que $c_{k'} < o_{k'}$. On a $t_{k'}(c_{k'} + 1) \leq t_{k'} o_{k'} \leq t_j o_j$, et le choix de k implique que $t_k(c_k + 1) \leq t_{k'}(c_{k'} + 1)$. La condition (C) est donc vérifiée après cette étape.
Finalement, l'allocation ainsi obtenue $(c_1, c_2, \ldots, c_p)$ est optimale car $\max\{c_i \times t_i\} \leq o_j t_j = \max(o_i t_i)$. ■

Complexité. Comme après la phase d'initialisation $c_1 + c_2 + \ldots + c_p \geq B - p$, l'algorithme est composé d'au plus p étapes, sa complexité est donc $O(p^2)$. La complexité peut être réduite à $O(p \log(p))$ en utilisant une structure de données *ad-hoc*.

Elimination des processeurs lents. De même que dans le cas homogène, la question de l'utilisation ou pas de *toutes* les ressources se pose : même si les tâches distribuées sont indépendantes, la diffusion des données initiales ainsi que le rapatriement des données finales peuvent impliquer un coût supérieur au gain de la distribution des calculs. Comme nous pourrons le voir dans le paragraphe suivant, c'est le cas pour la multiplication de matrices : l'algorithme est constitué de phases successives de calculs indépendants, séparés par des phases de redistribution des données.

Une réponse simple à cette question consiste à s'assurer qu'à chaque processeur est alloué une quantité de travail suffisante pour justifier de son utilisation. Cela peut être obtenu de différentes manières :

1. [**Pavage puis équilibrage**] L'espace des tâches est *initialement pavé* afin de fournir une granularité de calcul suffisante : la taille d'une tuile doit être suffisante pour justifier du temps de communication induit par son allocation. Les tuiles sont alors allouées aux différents processeurs, à l'aide de l'Algorithme 7.
2. [**Equilibrage puis pavage**] L'espace des tâches initialement non pavé est distribué sur les différents processeurs en tenant compte à la fois de l'équilibrage de charge, et en imposant une quantité minimale de travail aux différents processeurs utilisés. Cela peut être obtenu à l'aide de l'Algorithme 9 décrit Page 119. L'ensemble des tâches allouées à un processeur est alors pavé.

Dans le cas de calculs d'algèbre linéaire, les librairies étant optimisées pour certaines tailles de tuiles, la première solution est a priori plus adaptée. En contre-partie, la seconde solution favorise un équilibrage de charge plus précis utile lorsque la quantité de tâches à allouer est faible.

6.2.2 Application au produit "Matrice-Matrice".

L'algorithme du produit de deux matrices ($C = A \times B$), s'applique bien à notre problème d'équilibrage de charge statique. Nous analysons dans ce paragraphe comment implémenter le produit de matrices avec une distribution unidimensionnelle des données. Nous expliquons consécutivement comment la matrice est pavée (1), comment les colonnes de tuiles sont allouées au processeurs (2), puis finalement nous décrivons les grandes lignes de l'algorithme dans le cas d'une telle allocation des données (3) :

1. [**Pavage de la matrice**] Afin de pouvoir utiliser des librairies optimisées d'algèbre linéaire telles que LAPACK, les librairies d'algèbre linéaire distribuées telles que ScaLAPACK, utilisent classiquement une distribution "bloc-cyclique" des données (cf Chapitre 7 pour une description plus précise). En d'autres termes la brique de calcul de base est non pas un scalaire, mais une sous-matrice de taille $b \times b$. Ainsi, si les matrices initiales à multiplier sont de taille $n \times n$, le calcul devient un produit de deux matrices (constituées de blocs) de taille $\frac{n}{b} \times \frac{n}{b}$. Typiquement, b vaut 32 ou 64. On est donc en présence d'un calcul à gros grains dont la taille est fixée en fonction des différentes caractéristiques des processeurs (pipeline, registres, taille de cache etc.).
2. [**Allocation des tuiles**] Dans le cas d'une distribution unidimensionnelle des données, cas auquel nous nous restreignons ici, les colonnes des trois matrices sont distribuées identiquement sur les processeurs. Ainsi pour une distribution cyclique, les colonnes $A_{*,k}$, $B_{*,k}$ et $C_{*,k}$

sont alloués au processeur (k modulo p). Dans le cas de notre allocation, les colonnes $A_{*,k}$, $B_{*,k}$ et $C_{*,k}$ sont alloués au processeur P_1 si $k \leq c_1$, au processeur P_2 si $c_1 < k \leq c_1 + c_2$, etc.

3. **[Phases de l'algorithme]** Avec une telle distribution, l'algorithme de produit de matrices peut être décomposé en différentes étapes successives à l'intérieur desquelles les tâches exécutées sont indépendantes. Les communications entre chaque étape sont réduites à un simple décalage horizontal de la matrice A au travers de l'anneau de processeurs : ainsi, à l'étape $0 \leq t < \frac{n}{b}$, le processeur P_1 possède les colonnes $A_{*,1+t:c_1+t}$, le processeur P_2 les colonnes $A_{*,c_1+1+t:c_1+c_2+t}$, etc. Le processeur P_1 rajoute alors au bloc $C_{*,1:c_1}$ le produit des blocs $A_{*,1+t:c_1+t}$ par $B_{1+t:c_1+t,1:c_1}$, le processeur P_2 rajoute au bloc $C_{*,1+c_1:c_1+c_2}$ le produit des blocs $A_{*,1+c_1+t:c_1+c_2+t}$ par
$B_{1+c_1+t:c_1+c_2+t,1:c_1}$, etc (cf Figure 6.1 pour un exemple).

Considérons l'exemple suivant où l'on a 3 processeurs de temps de cycle respectifs $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$. Supposons que l'on souhaite calculer le produit $C = A \times B$ où A et B sont de taille 2496×2496. Les matrices peuvent être décomposées en blocs de taille 32×32 (32 est une taille de bloc typique pour une station de travail avec une mémoire cache [15]). Ainsi, $B = 78$ blocs de colonnes doivent être distribués sur les processeurs, correspondant aux $B = 78$ tâches indépendantes exécutées à chaque étape. Le Tableau 6.1 fournit les valeurs prises par les différentes variables lors de l'exécution de l'Algorithme 7 appliqué à notre problème. La Figure 6.1 décrit quelques étapes de l'algorithme de multiplication de matrices pour notre exemple.

Etape	c_1	c_2	c_3	$\max_i(c_i t_i)$
$\sum c_i = 76$ (initialisation)	39	23	14	117
$\sum c_i = 77$	40	23	14	120
$\sum c_i = 78 = B$	40	24	14	120

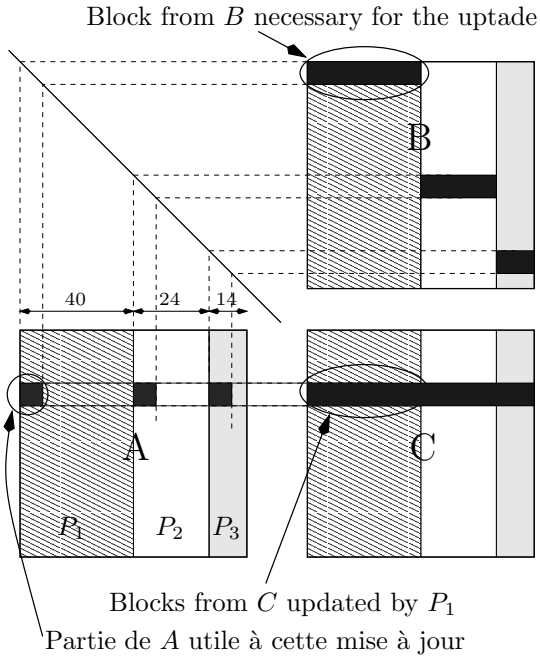
TAB. 6.1: *Etapes de l'Algorithme 7 pour 3 processeurs de temps de cycle $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$, avec $B = 78$.*

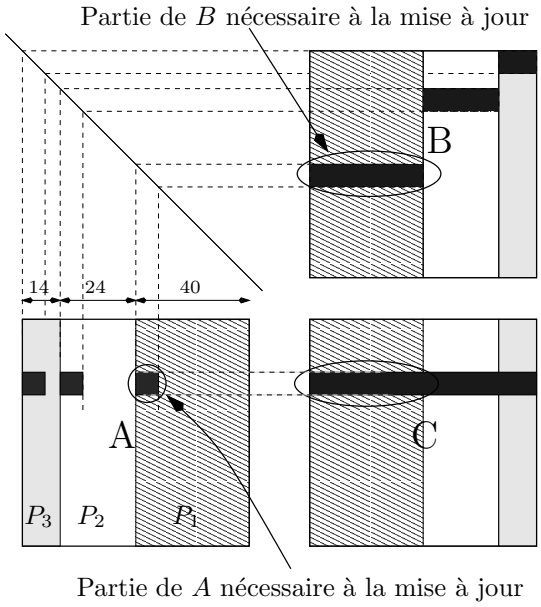
6.3 Equilibrage de charge incrémental, application à la résolution de systèmes linéaires

Si l'approche précédente est tout à fait adaptée à l'algorithme de multiplication de matrices, elle n'est, en contrepartie par contre, pas très efficace pour l'implémentation d'un noyau de décomposition LU ou QR. En effet, comme nous le verrons ci-dessous, alors que la distribution obtenue à l'aide de l'Algorithme 7 était valable pour toutes les étapes du produit de matrices, dans le cas de la décomposition LU, à chaque étape, l'ensemble des données traitées diminue, l'allocation n'est donc plus valide et une redistribution coûteuse des données est nécessaire.

6.3.1 Algorithme de décomposition LU

Tout comme dans le paragraphe précédent, nous nous restreignons ici à l'étude d'une distribution unidimensionnelle statique des données sur les processeurs. Une étude algorithmique plus précise, traitant en particulier le cas d'une distribution 2D, est décrite dans le Chapitre 7 de cette thèse. Considérons ainsi une allocation unidimensionnelle des données en blocs de colonnes. L'algorithme de décomposition LU fonctionne approximativement de la manière suivante [28] : à chaque étape,





Première étape :
(P_1) $C[*, 1:40] += A[*, 1] \times B[1, 1:40]$
(P_2) $C[*, 41:64] += A[*, 41] \times B[41, 41:64]$
(P_3) $C[*, 65:78] += A[*, 65] \times B[65, 65:78]$

$39^{\text{ième}}$ étape :
(P_1) $C[*, 1:40] += A[*, 39] \times B[39, 1:40]$
(P_2) ...

FIG. 6.1: *Différentes étapes de l'algorithme de multiplication de matrices sur 3 processeurs de temps de cycle respectifs $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$. Tous les indices de la figure sont des indices de blocs. Seul le calcul d'une ligne de C est représenté dans le schéma. On remarque qu'entre la première étape et la 39-ième, la distribution de A est décalée horizontalement.*

le processeur qui possède le bloc *pivot*, le *factorise* et le diffuse[2] à tous les autres processeurs qui *mettent* alors *à jour*[3] les colonnes restantes. A l'étape suivante, c'est le bloc des b colonnes suivantes qui devient à son tour le pivot, et le calcul progresse ainsi. La Figure 6.2 fournit le schéma d'une étape de l'algorithme de décomposition LU.

Du fait que la plus grande partie du temps de calcul corresponde à la mise à jour, l'idée est de trouver une allocation statique qui équilibre au mieux les tâches lors de chaque étape de mise à jour. Considérons la première étape. Après la factorisation du premier bloc, chacune des mises à jour d'un bloc, effectuée par un processeur, est un noyau de calcul indépendant : si la matrice est de taille $(nb) \times (nb)$, il y a $B = (n-1)$ noyaux de calcul à allouer. Nous pourrions utiliser l'Algorithme 7 pour équilibrer les charges, mais la taille du domaine de travail diminue au fur et à mesure que la décomposition progresse. A la seconde étape, le nombre de blocs à mettre à jour est $(n-2)$. Si l'on souhaite à nouveau équilibrer ces charges, les données doivent être redistribuées, ce qui implique ainsi, entre chaque étape, un nombre important de communications. De plus, dans l'optique du développement d'une librairie d'algèbre linéaire distribuée hétérogène, l'allocation doit a priori être identique au début et à la fin de la décomposition.

Nous cherchons donc une distribution statique des données fournissant un équilibrage des charge de mise à jour, commun à toutes les étapes. En d'autres termes, nous cherchons une distribution de B tâches sur p processeurs de temps de cycle respectifs $t_1, t_2, \dots t_p$ telle que pour tout $i \in \{2, \dots, B\}$ le nombre de blocs de $\{i, \dots, B\}$ que possède chaque processeur P_j, soit (approximativement) inversement proportionnel à son temps de cycle t_j. Le *programme incrémental* de Robert et al. initialement présenté dans [21] pour fournir une solution au problème décrit dans le Paragraphe 6.2.1, permet d'obtenir une solution optimale à ce problème.

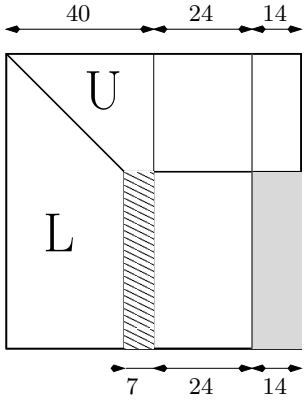


FIG. 6.2: $33^{ième}$ *étape de l'algorithme de décomposition LU (les indices correspondent à des indices de bloc) : avec l'ancienne allocation, l'exécution devient moins équilibrée. Ici, après la factorisation du bloc de colonnes* 33*, le processeur* 1 *a* 7 *factorisations à effectuer et travaille durant* $7 \times 3 = 21$ *unités de temps, alors que le processeur* 2 *travaille durant* $24 \times 5 = 120$ *unités de temps.*

6.3.2 Algorithme incrémental de Robert et al.

Afin d'illustrer le principe de l'algorithme, considérons un exemple avec 3 processeurs de temps de cycle respectifs $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$. Le Tableau 6.2 reporte l'allocation fournie par

[2] *broadcast* en anglais
[3] *update* en anglais

l'algorithme jusqu' à $B = 10$. Dans ce tableau, l'intitulé "processeur sélectionné" représente l'indice du processeur choisi pour détenir l'atome suivant. A chaque étape, le processeur sélectionné est choisi afin que le coût de l'allocation globale obtenue soit minimal. Notons c_i le nombre d'atomes alloués au processeur i. Dans ce cas, le coût de l'allocation $\mathcal{C} = (c_1, c_2, \ldots, c_p)$ vaut $\max_{1 \leq i \leq p} c_i t_i$. Ainsi le coût moyen d'exécution d'un atome vaut $t_{para} = \frac{\max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^{p} c_i}$.

Nombre d'atomes	c_1	c_2	c_3	Coût	Processeur sélectionné	σ
0	0	0	0		1	
1	1	0	0	3	2	$\sigma[1] = 1$
2	1	1	0	2.5	1	$\sigma[2] = 2$
3	2	1	0	2	3	$\sigma[3] = 1$
4	2	1	1	2	1	$\sigma[4] = 3$
5	3	1	1	1.8	2	$\sigma[5] = 1$
6	3	2	1	1.67	1	$\sigma[6] = 2$
7	4	2	1	1.71	1	$\sigma[7] = 1$
8	5	2	1	1.87	2	$\sigma[8] = 1$
9	5	3	1	1.67	3	$\sigma[9] = 2$
10	5	3	2	1.6		$\sigma[10] = 3$

TAB. 6.2: *Différentes étapes de l'algorithme incrémental avec* 3 *processeurs de temps de cycle respectifs* $t_1 = 3$, $t_2 = 5$, *et* $t_3 = 8$.

Par exemple, à la quatrième étape, c'est à dire lors de l'allocation du quatrième noyau, l'algorithme démarre par l'allocation obtenue pour 3 atomes, c'est à dire $(c_1, c_2, c_3) = (2, 1, 0)$. A quel processeur P_i doit être assigné le quatrième noyau, en d'autres termes, quel c_i doit être incrémenté ? Il y a trois possibilités $(c_1 + 1, c_2, c_3) = (3, 1, 0)$, $(c_1, c_2 + 1, c_3) = (2, 2, 0)$ et $(c_1, c_2, c_3 + 1) = (2, 1, 1)$ de coût respectif $\frac{9}{4}$ (P_1 termine en dernier), $\frac{10}{4}$ (P_2 termine en dernier), et $\frac{8}{4}$ (P_3 termine en dernier). Ainsi, la troisième solution est sélectionnée, correspondant au résultat $(c_1, c_2, c_3) = (2, 1, 1)$.

De manière plus formelle, l'algorithme s'écrit :

Algorithme 8. *Distribution incrémentale optimale de* B *atomes sur* p *processeurs de temps de cycle respectifs* $t_1, t_2, \ldots t_p$ *pour tout sous-ensemble d'atomes* $[1, m]$, $m \leq B$.

Distribue$(B, t_1, t_2, \ldots, t_p)$
{ Initialisation : aucune tâche à distribuer. $m = 0$ }
do $i = 1, p$
    $c_i = 0$
{ Construit itérativement l'allocation σ}
do $m = 1, B$
    trouve $k \in \{1, \ldots, p\}$ tel que $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1)\}$
    $c_k = c_k + 1$
    $\sigma[m] = k$
Retourne$(\sigma, c_1, c_2, \ldots, c_k)$

Bien sûr, si nous souhaitions allouer 10 atomes indépendants, nous pourrions utiliser l'Algorithme 7 pour trouver que 5 atomes devraient être alloués au processeur P_1, 3 au processeur P_2 et 2 au processeur P_3. Mais l'algorithme incrémental retourne la solution optimale pour tout nombre

d'atomes de 1 à B, et retourne la distribution associée.

Proposition 2 *L'algorithme incrémental retourne la distribution optimale pour tout sous-ensemble d'atomes* $[1, m]$ *où* $m \leq B$.

Preuve. La preuve peut être calquée sur celle de la Proposition 1 (Page 105). Il suffit de modifier légèrement l'invariant à prouver :

$$(C_m) : \forall i \in \{1, \ldots, p\},\ c_i t_i \leq o_{j_m}^{(m)} t_{j_m} = \max_{1 \leq j \leq m} o_j^{(m)} t_j$$

où $\mathcal{O}^{(m)} = (o_1^{(m)}, o_2^{(m)}, \ldots, o_p^{(m)})$ est une solution optimale au problème avec m atomes, et j_m l'indice du processeur qui, pour une telle allocation, termine son travail en dernier. Par récurrence, à l'étape m, on a (C_m) et comme trivialement (le temps total d'exécution ne peut qu'augmenter si l'on rajoute un atome à exécuter) $o_{j_m}^{(m)} t_{j_m} \leq o_{j_{m+1}}^{(m+1)} t_{j_{m+1}}$, avant l'étape $m+1$ on a donc (C_{m+1}). L'étape $m+1$ étant identique à toute étape de l'Algorithme 7, la preuve de la Proposition 1 montre que (C_{m+1}) est maintenu après l'étape $m+1$. ■

La complexité de l'algorithme incrémental est $O(pB)$ où p est le nombre de processeurs et B le nombre d'atomes à distribuer. De même que pour l'Algorithme 7, elle peut être aisément améliorée en $O(\log(p)B)$. Rappelons que le résultat s'exprime en $O(B)$, et qu'en conséquence cette complexité est relativement bonne.

6.3.3 Application à la décomposition LU

Pour la décomposition LU, l'idée est d'allouer périodiquement, sous forme d'un motif de largeur B, les blocs de colonnes aux processeurs (cf Figure 6.3). B est un paramètre pouvant valoir n, pour une matrice de taille $(nb) \times (nb)$ découpée en blocs de taille $b \times b$, mais pouvant aussi être plus petit si l'on souhaite l'allocation plus régulière. En effet, le recouvrement des communications avec les calculs n'est possible que grâce à un processus de pipeline entre les colonnes, processus brisé lorsque l'allocation est trop irrégulière. On verra de plus dans le Paragraphe 6.4, qu'en pratique il est inutile que B soit très grand. Supposons ici, sans perte de généralité, que B vaille n, c'est à dire que le motif ait la même largeur que la matrice et ne soit donc pas répété.

Pour définir le motif, nous utilisons le résultat de l'algorithme incrémental *en sens inverse*, c'est à dire que le bloc de colonnes d'indice $1 \leq k \leq B$ sera alloué au processeur $\sigma(B - k + 1)$. En effet, l'algorithme incrémental retourne l'allocation optimale pour tout sous-ensemble $[1, i]$ de $[1, B]$ où $i \in [1, B]$, alors que la distribution recherchée doit équilibrer tout sous-ensemble $[i, B]$ de $[1, B]$ où $i \in [1, B]$. Le Tableau 6.2 donne l'allocation obtenue par notre algorithme pour $B = 10$ avec 3 processeurs de temps de cycle $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$. Ainsi, le motif obtenu vaut $(P_3 P_2 P_1 P_1 P_2 P_1 P_3 P_1 P_2 P_1)$.

6.4 Système à différences finies. Equilibrage et ordonnancement.

Le but de ce paragraphe est d'élargir les résultats présentés dans le Chapitre 2, relatifs à un espace d'itération de forme parallélogramme (r pas nécessairement égal à 0), à un ensemble hétérogène de processeurs.

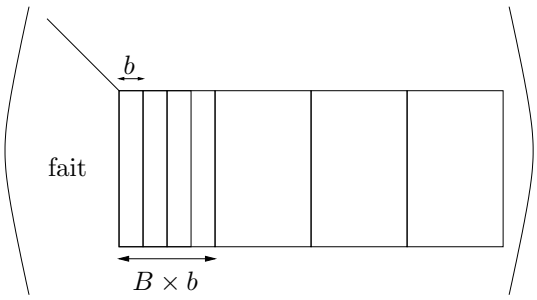


FIG. 6.3: *Allocation périodique utilisant un motif de largeur B.*

L'évolution d'un système à différences finies, ou tout problème semblable à celui présenté dans le Chapitre 5, peuvent se traiter comme suit : une fois la taille et la forme des tuiles fixées, il suffit d'allouer les tuiles aux processeurs et de trouver un ordonnancement minimisant le temps total d'exécution. La présence de dépendances et la nécessité de minimiser l'impact des communications, rendent le problème compliqué. Dans le cas simple d'un espace d'itérations rectangulaire ($r = 0$), Calland et al. [23] ont montré que l'allocation optimale était par colonnes : pour une telle allocation, chaque colonne de tuiles est allouée à un même processeur, et les processeurs sont distribués cycliquement sur les colonnes. Dans le cas où l'espace d'itérations n'est plus de forme rectangulaire, ou lorsque l'ensemble des ressources de calcul n'est plus homogène, ce résultat n'est plus vérifié. Malheureusement nous ne connaissons pas d'allocation optimale dans de telles configurations. Le but de ce paragraphe est de proposer une solution heuristique. Avec les mêmes notations que celles utilisées dans le Chapitre 2, nous évaluons le temps total d'exécution pour notre allocation, et en déduisons, sous certaines contraintes, son optimalité asymptotique.

Hypothèses de travail. Les conditions (H2), (H3), (H5) et (H6) du Paragraphe 2.4 décrites Page 20 sont maintenues. Mais l'ensemble de processeurs étant ici hétérogène, l'allocation considérée est différente, et les notation T_{calc} et c n'ont plus court.

- L'espace d'itération est de forme parallélogramme de taille $wW \times hM$, pavé par des tuiles de forme rectangulaire de taille $w \times h$. Les dépendances entre tuiles sont $(1,0)^t$ et $(0,1)^t$. Rappelons que la définition de r est dépendante de h et de w. En fait, pour un domaine d'itération fixé, r est proportionnel à $\frac{w}{h}$. On peut aussi influer sur la valeur de r en tordant l'espace d'itérations dans les limites imposées par le cône de dépendances.
- On considère p processeurs. Pour un processeur P_i ($1 \leq i \leq p$), t_i correspond au temps de calcul d'une tuile. Le temps de communication, est supposé indépendant du couple de processeurs et vaut τ_{comm}.
- L'allocation des tuiles, unidimensionnelle, est effectuée par colonnes de tuiles.

6.4.1 Allocation heuristique

L'allocation est construite de la manière suivante :

- Les tuiles sont allouées de manière unidimensionnelle par colonnes sur l'ensemble des processeurs.
- On introduit un paramètre B à fixer. On détermine un motif de largeur B représentant l'allocation des B premières colonnes qui sera reproduit périodiquement. Ainsi, l'allocation

obtenue est périodique de période B.
- Le motif est déterminé par l'Algorithme 7 présenté Page 105 : cet algorithme fournit le nombre de colonnes de tuiles (c_i) par processeur. Ensuite, les processeurs sont réindexés de telle manière que $c_1t_1 \leq c_2t_2 \leq \ldots c_pt_p$. Alors le processeur P_1 possède les c_1 premières colonnes de tuiles , P_2 les c_2 colonnes de tuiles suivantes, etc. Le processeur P_p possède les c_p dernières colonnes de tuiles.
- A l'intérieur d'un bloc de c_i colonnes de tuiles, les tuiles sont exécutées par le processeur P_i dans l'ordre lexicographique qui favorise la direction horizontale.

Remarque 5. Du fait des dépendances horizontales, avec une allocation cyclique des colonnes où $B = P$ et $c_i = 1$, les tuiles seraient exécutées au rythme du processeur le plus lent, c'est à dire (asymptotiquement) comme si tous les processeurs avaient un temps de cycle égal à $\max(t_i)$.

Remarque 6. Un algorithme glouton naïf (dès qu'un processeur a terminé son travail, la colonne à exécuter suivante lui est attribuée), allouerait les colonnes cycliquement sur les processeurs.

Remarque 7. Une allocation par colonnes utilisant l'Algorithme incrémental ne fonctionne pas. En effet, de même que pour une allocation cyclique des colonnes, bien que l'équilibrage global des charges soit bon, la présence des dépendances horizontales ralentit le processus au rythme du processeur le plus lent.

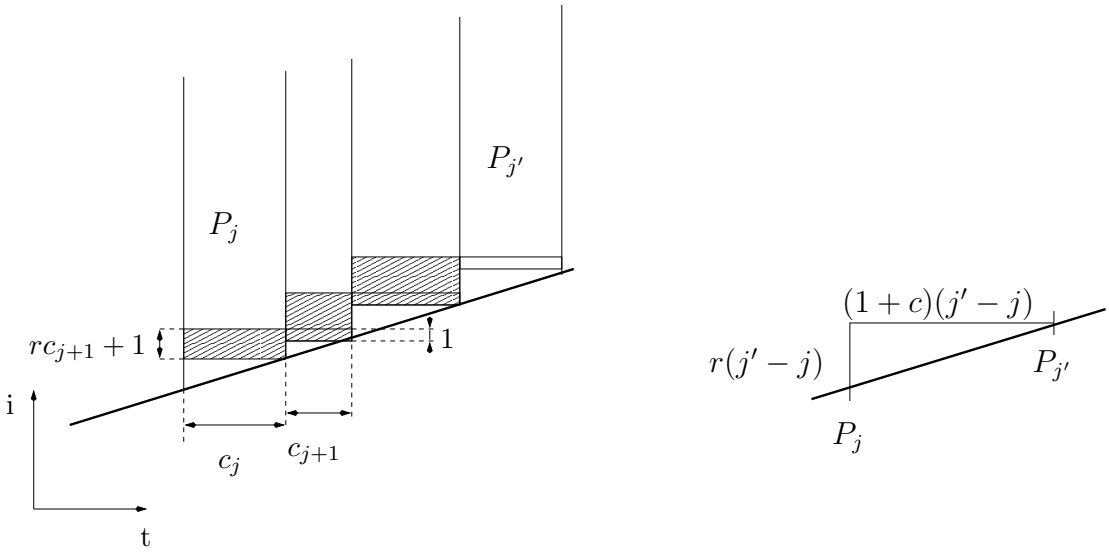
Remarque 8. Intuitivement, le réordonnancement des processeurs par leur temps d'exécution c_it_i tels que $c_1t_1 \leq c_2t_2 \leq \ldots \leq c_pt_p$ est effectué afin que les processeurs ne s'attendent pas mutuellement.

Tout comme dans le Chapitre 2, l'idée pour trouver le temps total d'exécution est de chercher le chemin critique du graphe des tâches. Les processeurs ayant une vitesse différente, et l'allocation n'étant plus cyclique, la longueur des différents chemins doit être réévaluée. En particulier,
- dans le cas où $r > 0$, la longueur du chemin le long de la bordure inférieure évolue quadratiquement avec B (cf Figure 6.4) ;
- la condition $1+r+c \leq 0$, qui impliquait l'indépendance des colonnes dans le cas homogène, se traduit dans le cas hétérogène par l'inégalité $\forall(j, j')$, $c_jt_j+\tau_{comm}+rc_jt_{j'}c_{j'} \leq 0$ (cf Figure 6.5).

Intuitivement, notre ordonnancement est asymptotiquement optimal, relativement à l'équilibrage de charge, si le processeur P_p travaillant le plus longtemps n'est jamais inactif, sauf au début et à la fin de l'exécution globale. Une telle condition se traduit par le fait que le chemin horizontal entre deux bandes de colonnes de tuiles consécutives, possédées par le même processeur, est plus petit que le temps passé par ce même processeur pour exécuter toute une bande de colonnes de tuiles (Mc_it_i unités de temps). La Figure 6.6 illustre cette condition dans le cas où $r > 0$.

Analysons le problème de manière plus formelle. Pour cela, et afin de simplifier les notations par la suite, considérons à présent que $c_0 = c_p$ et que $\forall i \geq 1$, $c_i = c_i \text{ modulo } _p$.

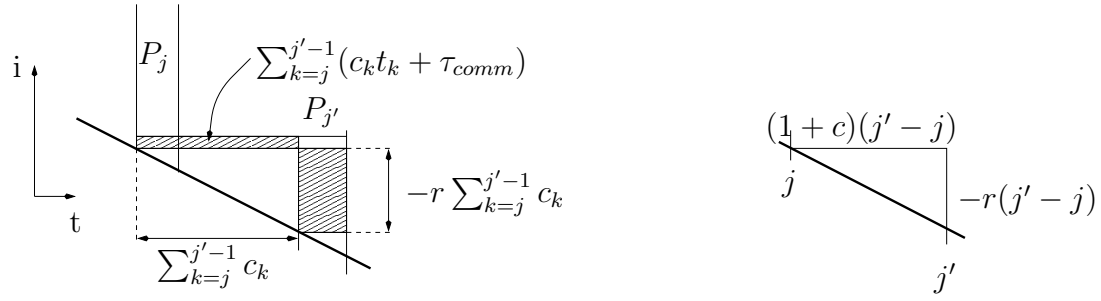
Alors, le processeur P_p n'est jamais inactif (sauf au début et à la fin) si le temps d'exécution d'un bloc de c_p colonnes de tuiles est plus long que le temps de latence de démarrage. Cette condition



$\sum_{k=j}^{j'-1} c_k(rc_{k+1}+1)t_k + \tau_{comm}$ **Cas hétérogène**

$\sum_{k=j}^{j'-1}(1+r+c)$ **Cas homogène**

FIG. 6.4: *Longueur de latence horizontale quand $r > 0$: dans le cas hétérogène le chemin critique est situé le long de la bordure inférieure.*



$\max\left(\sum_{k=j}^{j'-1} c_k(-rc_{j'}t_{j'}), \sum_{k=j}^{j'-1}(c_k t_k + \tau_{comm})\right)$ **Cas hétérogène**

$\sum_{k=j}^{j'-1} \max(-r, 1+c)$ **Cas homogène**

FIG. 6.5: *Latence horizontale quand $r \leq 0$: la condition $1+r+c \leq 0$ du cas homogène, se traduit dans le cas hétérogène par l'inégalité $\forall(k,j')$, $c_k t_k + \tau_{comm} + rc_k t_{j'} c_{j'} \leq 0$.*

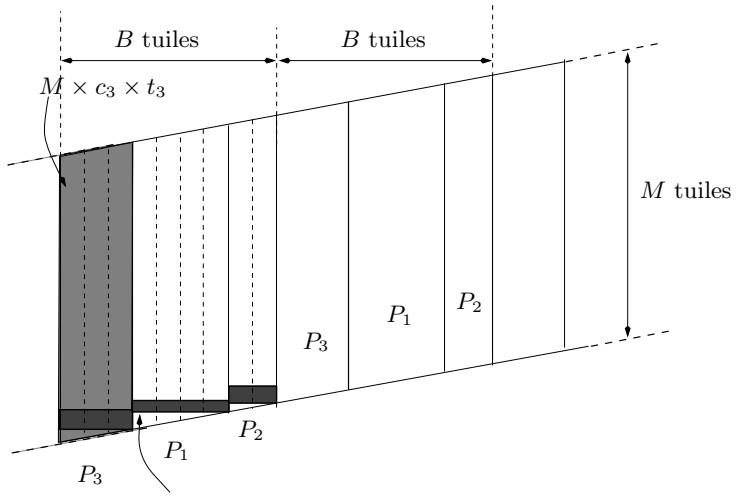


FIG. 6.6: *Allocation lorsque $B = 9$ pour $p = 3$ processeurs de temps de cycle respectifs $(t_1, t_2, t_3) = (3, 7, 5)t_{calc}$: on a $(c_1, c_2, c_3) = (4, 2, 3)$. Le processeur P_3 n'est jamais inactif (sauf au début et à la fin de l'exécution globale) si le temps d'exécution d'un bloc de colonnes (partie grisée en clair) est supérieur au temps de latence horizontal (partie grisée en foncé).*

peut ainsi être approximée par l'inégalité suivante :

Condition (C)

$$\left|\begin{array}{l} \text{Soit } r \geq 0 \text{ et } Mt_pc_p \geq \sum_{j=1}^{p} [c_j(rc_{j+1}+1)t_j + \tau_{comm}] \\ \text{Soit } r < 0 \text{ et } \left|\begin{array}{l} \forall (j,j'),\ c_jt_j + \tau_{comm} + rc_jt_{j'}c_{j'} \leq 0 \text{ (latence nulle)} \\ \text{ou bien si } M + (B - c_p)r \geq 0,\ Mt_pc_p \geq Brt_pc_p + \sum_{j=1}^{p}(c_jt_j + \tau_{comm}) \end{array}\right. \end{array}\right.$$

On peut déjà faire les remarques suivantes :

1. Tout comme pour le cas homogène, r à tout intérêt à être le plus petit possible. Rappelons que l'intervalle de liberté de r est limité par le cône de dépendances.
2. Lorsque $r \geq 0$ la valeur limite de M augmente avec B^2.
3. Si la condition (C) n'est pas vérifiée, les processeurs sont fortement inactifs. Il faut donc, en pratique, chercher à atteindre cette condition (cf Paragraphe 6.4.2 pour une discussion sur le choix de B).

Théorème 11 *Si la condition (C) est vérifiée, alors le temps total d'exécution peut être majoré par l'expression*

– *Si $r \geq 0$,*

$$T_{exe} \simeq \sum_{j<p} [c_j(rc_{j+1}+1)t_j + \tau_{comm}] + Mt_pc_p \left\lfloor \frac{W}{B} \right\rfloor + \sum_{j \leq f} [c_j(rc_{j+1}+1)t_j + \tau_{comm}]$$

– *Si $r < 0$,*

$$T_{exe} \simeq \left(\sum_{j<p} c_j(t_j + rc_pt_p) + \tau_{comm} \right)^+ + Mt_pc_p \left\lfloor \frac{W}{B} \right\rfloor + \left(\sum_{j \leq f} c_j(t_j + rc_pt_p) + \tau_{comm} \right)^+$$

où f est l'indice, modulo p, du processeur exécutant la dernière colonne de l'espace d'itérations (si l'indice de ce dernier processeur vaut p, alors $f = 0$).

Preuve. Cette formule correspond à la longueur du chemin décrit Figure 6.7 : selon que $r > 0$ ou pas, la latence horizontale s'exprime différemment, comme décrit figures 6.4 et 6.5. Ainsi, la preuve peut être calquée sur celle du cas "Forme parallélogramme-distribution cyclique" de la Page 28.

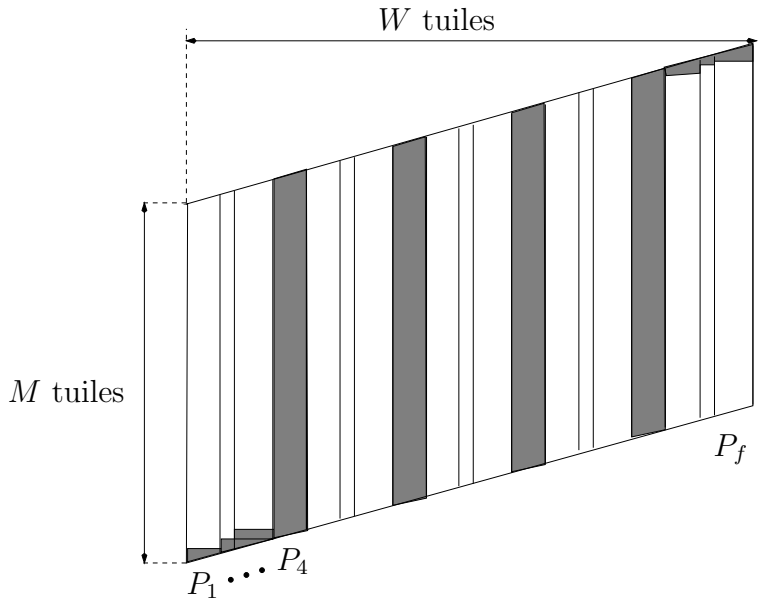


FIG. 6.7: *Lorsque la condition (C) est vérifiée, le temps d'exécution total (chemin critique grisé) correspond à la somme du temps de latence initial, du temps d'exécution du processeur P_p et du temps de latence finale. Ici, $r > 0$, $p = 4$ et $f = 3$.*

■

6.4.2 Equilibrage de charge vs latence d'exécution.

Dans ce paragraphe, nous abordons dans un premier temps le problème lié au choix de B. En effet, B a une influence à la fois sur l'équilibrage de charge, mais aussi sur la latence entre chaque période. Lorsque la taille du domaine d'itération est petite, le fait de paver celui-ci *avant* d'effectuer l'équilibrage de charge rend le problème encore plus critique. Nous proposons donc dans un deuxième temps une approche différente dans laquelle le pavage est effectué *après* l'équilibrage de charge : l'Algorithme 7 est adapté afin d'assurer durant la phase d'allocation des tâches une granularité suffisante.

Choix de B. Les caractéristiques de l'application et du système jouant un rôle important dans la détermination analytique d'un B "optimal", nous nous restreindrons ici à quelques remarques assez générales.

Remarque 9. Reprenons la définition du coût moyen d'exécution d'un atome introduite dans le Paragraphe 6.3.2 Page 110 : pour un équilibrage optimal $(o_1, \ldots, o_p)$ avec un motif de largeur B, $t_{para}(B) = \frac{\max o_i t_i}{B}$. Clairement, $\lim_{B\to\infty} t_{para}(B) = \frac{1}{\sum_{1\leq i\leq p} \frac{1}{t_i}}$. C'est le coût d'un équilibrage "parfait". Notons le $t_{para}(\infty)$. A priori, B doit être le plus grand possible. En fait, la vitesse de chaque processeur n'a pas de valeur très précise, même pour une séquence d'instructions fixes, et il est illusoire de vouloir prendre une valeur de B très grande.

Il faudrait donc exprimer analytiquement pour chaque B, l'erreur commise par un équilibrage donné. Mais, comme en témoigne l'exemple présenté Tableau 6.2, $t_{para} : B \longmapsto t_{para}(B)$ n'est pas une fonction décroissante de B. Faute d'avoir une expression analytique simple du degré de déséquilibre obtenu pour un B fixé, nous nous restreignons à borner cette valeur. C'est le résultat que fournit le théorème suivant :

Théorème 12

$$\frac{t_{para}(B) - t_{para}(\infty)}{t_{para}(\infty)} \leq \frac{p-1}{B}$$

où $t_{para}(B)$ est, pour un équilibrage optimal avec un motif de largeur B sur p processeurs, le coût moyen d'exécution d'un atome.

Preuve. Le déséquilibre d'une allocation est lié à la contrainte d'insécabilité des atomes distribués. Si tel n'était pas le cas, le temps nécessaire à p processeurs de temps de cycle $t_1, t_2, \ldots, t_p$ pour exécuter B atomes serait de

$$B \times t_{para}(\infty) = \frac{B}{\sum_{1\leq i\leq p} \frac{1}{t_i}}.$$

On sait que pour une allocation $\mathcal{C} = (o_1, o_2, \ldots, o_p)$ optimale de B atomes obtenue à l'aide de l'Algorithme 7, si P_f est le processeur qui termine son travail en dernier, alors $\forall i$, $(o_i + 1)t_i \geq o_f t_f = Bt_{para}(B)$. Le pire des cas est celui où tous les autres processeurs pourraient, durant leur temps d'inactivité, exécuter un atome de plus. C'est à dire le cas où $\forall i \neq f$, $(o_i + 1)t_i = o_f t_f$. Cela se traduit par $\frac{o_f t_f}{B+(p-1)} = t_{para}(B + (p-1)) = t_{para}(\infty)$. D'où le résultat. ■

Comme nous l'avons remarqué précédemment, lorsque r est positif, plus B est grand, plus M doit être grand pour que la condition (C) présentée Page 115 soit vérifiée. Si cette condition n'est pas vérifiée, la latence entre chaque période est trop importante et les processeurs sont fortement inactifs entre chaque bande de colonnes. Si M est grand, ou que r est négatif, il suffit de choisir une valeur de B assurant un équilibrage à 1 pour cent près par exemple, et l'allocation obtenue est asymptotiquement optimale (à un pour cent près).

En revanche, si M n'est pas assez grand, l'approche consistant à paver d'abord l'espace puis à allouer ensuite les tuiles, n'est pas bonne. *Il faut faire l'inverse* : pour une valeur de B fixée, allouer les colonnes aux processeurs puis paver les blocs de colonnes pour favoriser la localité des accès aux données et la granularité des calculs.

Assurer la granularité durant la phase d'équilibrage. Le pavage des calculs venant en amont de l'équilibrage de charge, il faut veiller, durant cette étape, à ce qu'une certaine granularité des calculs soit assurée. L'idée est alors d'imposer pour tous les processeurs sauf 1 (notons le P_1) un nombre minimum de tâches. Cela peut se traduire par une borne inférieure w_i sur le nombre de colonnes c_i détenues par chaque processeur P_i ($i > 1$) dans un motif. Le problème se pose donc maintenant de la manière suivante :

- Soient p processeurs de temps de cycle respectifs $t_1, \ldots, t_p$.
- Soient $w_2, \ldots, w_p$ la granularité minimum requise pour chaque processeur $P_2, \ldots, P_p$. Ainsi, une allocation $(c_1, \ldots, c_p)$ est considérée comme valide si $\forall i > 1$, $[c_i \geq w_i \vee c_i = 0]$.
- Soit une largeur de motif B, il faut déterminer l'allocation valide $(c_1, \ldots, c_p)$ telle que $\sum_{i=1}^{p} c_i = B$ qui minimise la grandeur $\max_{1 \leq i \leq p} c_i t_i$.

La réponse est une conséquence du Théorème 13 qui nécessite la définition suivante :

Définition 7 (Allocation optimale constructible) *Soit $c_1, \ldots, c_q$ une allocation de $\sum_{i=1}^{q} c_i$ atomes sur q processeurs de temps de cycles $c_1, \ldots, c_q$. Cette allocation est dite constructible si elle vérifie la condition suivante $[\forall i,\ (c_i + 1)t_i \geq T_{max} = \max_{1 \leq k \leq q} c_k t_k]$.*

Bien sûr, une allocation constructible est optimale, mais il existe certains cas particuliers d'allocations non constructibles et pourtant optimales.

Théorème 13 *Soit un ensemble de processeurs de temps de cycle $t_1, t_2, \ldots, t_q$. Soient B atomes à allouer avec la contrainte $[\forall i > 1,\ c_i \geq w_i]$, tel que $[\forall i, w_i t_i \leq w_q t_q]$. Alors, il existe une allocation valide optimale constructible de ces atomes sur les q processeurs, si et seulement si*

$$B_{min} = \sum_{i=1}^{q} \max\left(w_i, \left\lceil \frac{w_q \times t_q}{t_i} \right\rceil - 1\right) \leq B$$

Preuve. La démonstration se décompose en trois étapes :

1. dans un premier temps nous construisons une allocation constructible optimale de B_{min} atomes sur les q processeurs.
2. Ensuite, nous montrons que cette allocation est valide.
3. Finalement, nous montrons qu'aucune allocation constructible optimale de $B_{min} - 1$ atomes sur q processeurs n'est valide.

[Construction d'une allocation optimale] Montrons tout d'abord que l'allocation $\mathcal{C} = (c_1, \ldots, c_q)$ définie par

$$\forall i, c_i = \max\left(w_i, \left\lceil \frac{w_q \times t_q}{t_i} \right\rceil - 1\right) \tag{6.1}$$

est constructible optimale pour B_{min} atomes. En effet, d'une part $c_q = w_q$. Ensuite, prenons $1 \leq i \leq q$, d'après 6.1 on a

$$\text{si } c_i = w_i, \text{ alors} \qquad c_i t_i = w_i t_i \leq w_q t_q = c_q t_q \tag{6.2}$$

$$\text{sinon, on a} \quad c_i t_i = \left(\left\lceil \frac{w_q \times t_q}{t_i} \right\rceil - 1\right) t_i < \frac{w_q t_q}{t_i} t_i = c_q t_q \tag{6.3}$$

Ainsi $T_{max} = c_q t_q$. Maintenant, d'après 6.1 à nouveau, on a $(c_i + 1)t_i = \left\lceil \frac{w_q \times t_q}{t_i} \right\rceil t_i \geq w_q t_q = c_q t_q$.
[Validité] L'allocation est valide par construction.

[Réciproque] Une allocation constructible à $B_{min} - 1$ atomes est obtenue à partir de l'allocation à B_{min} atomes en ôtant un atome à l'un des q processeurs : P_i. Pour que cette allocation $(c_1, \ldots, c_i - 1, \ldots, c_q)$ soit valide, nécessairement $c_i - 1 \geq w_i$, d'où $c_i \neq w_i$ et $c_i = \left\lceil \frac{w_q \times t_q}{t_i} \right\rceil - 1$. D'après 6.3, l'allocation ainsi construite n'est pas constructible. ■

L'algorithme de complexité $O(p \log(p))$ est alors le suivant :

Algorithme 9. *Allocation statique optimale de B atomes indépendants sur p processeurs de temps de cycle respectifs $t_1 \leq t_2 \leq \ldots \leq t_p$ à l'aide d'une granularité minimale respective $w_2, \ldots, w_p$.*

Distribue$(B, t_1, t_2, \ldots, t_p, w_2, \ldots, w_p)$
{ Détermine par dichotomie l'ensemble $\{P_1, \ldots, P_q\}$ des processeurs utilisés}
$q = \max \left\{ i > 1, \sum_{k=1}^{i} \max \left(w_k, \left\lceil \frac{w_i \times t_i}{t_k} \right\rceil - 1 \right) \leq B \right\} \cup \{1\}$
{ Initialisation : calcule c_i $(i \leq q)$ tels que $c_i \times t_i \approx Constante$ et $c_1 + c_2 + \ldots + c_q \leq B$ }
do $i = 1, q$
$\quad c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{q} \frac{1}{t_i}} \times B \right\rfloor$
do $i = q + 1, p$
$\quad c_i = 0$
{ Incrémente itérativement les c_i qui minimisent le temps d'exécution tant que $\sum_{k=1}^{q} c_i < B$}
while $\sum_{i=1}^{q} c_i < B$ do
$\quad$ trouve $k \in \{1, \ldots, q\}$ tel que $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1)\}$
$\quad c_k = c_k + 1$
Retourne$(c_1, c_2, \ldots, c_p)$

6.5 Conclusion

Dans ce chapitre, nous avons abordé différents problèmes liés à l'implémentation de programmes sur un ensemble de ressources de calcul hétérogène. Nous nous sommes restreint pour cela à la recherche d'une allocation statique unidimensionnelle des données. Nous avons traité le problème de l'équilibrage de gros grains de calcul indépendants. Puis nous avons adapté l'algorithme dynamique de Robert et al. [21] fournissant une solution au problème d'équilibrage statique de charges indépendantes, pour construire une distribution adaptée à la décomposition LU. Finalement, nous avons rediscuté le problème de pavage, abordé dans le Chapitre 2, dans le cadre de ressources hétérogènes.

Les modèles de machines utilisés ici sont simples. Remarquons en particulier que généralement l'hétérogénéité des ressources n'est pas réduite à la simple hétérogénéité des vitesses de calcul. Par exemple, il est nécessaire de prendre en compte la taille mémoire, car elle est rarement proportionnelle à la vitesse des processeurs. En fait, il est facile d'adapter l'algorithme incrémental présenté dans ce chapitre à des modèles plus complexes. Par exemple, il peut être généralisable au cas où le coût C est défini comme suit : $C(c_1, \ldots, c_p) = \max_{1 \leq i \leq p} C_i(c_i, B)$ où C_i le coût du processeur P_i est une fonction croissante de (c_i, B).

La méthode consistant à paver l'espace d'itérations, puis à équilibrer les charges et ordonnancer

les calculs, est discutable. Lorsque l'espace d'itérations est très grand, cette méthode est simple et nous assure un résultat optimal. De même, dans le cadre de librairies d'algèbre linéaire où les codes monoprocesseurs optimisés à chaque architecture fournissent des performances inégalables, cette approche est incontournable. En revanche, dans le cadre de la parallélisation automatique de petits espaces d'itérations, cette méthode n'est sans doute pas acceptable. L'idée est alors de forcer la localité et la granularité des calculs, à priori, en introduisant des contraintes (bornes sur les différentes valeurs à déterminer) lors des étapes d'allocation et d'ordonnancement, puis de paver les différentes bandes allouées aux processeurs. Nous avons donné quelques éléments illustrant cette approche dans le cadre du problème de systèmes à différences finies. Ces résultats restent bien évidemment applicables au cas de ressources homogènes.

Chapitre 7

Ressources hétérogènes : pavage 2D et algorithmique.

7.1 Introduction

Le but de ce chapitre est de fournir les éléments nécessaires à l'extension de la librairie ScaLAPACK [15] à un réseau de stations hétérogène. Plus précisément nous concentrons notre étude sur des noyaux d'algèbre linéaire dense tels que la multiplication de matrices et les décompositions LU et QR. Notre but n'est pas de réécrire entièrement la librairie ScaLAPACK, mais de profiter de sa structure hiérarchique afin de n'en modifier que la couche supérieure.

Lorsque les machines ne sont pas homogènes, une distribution bloc-cyclique n'est plus adaptée ; il faut donc trouver un nouveau format (générique) de distribution de données. A la lumière de l'étude menée dans le chapitre précédent, nous proposons un schéma d'allocation efficace sur un anneau de processeurs (distribution unidimensionnelle). De manière générale, une distribution bidimensionnelle des données est plus efficace qu'une distribution unidimensionnelle, aussi bien théoriquement [41, 28] qu'expérimentalement [15]. En particulier, une distribution 2D est plus scalable [27]. Par exemple une pile de PCs reliés par un réseau Myrinet [35] peut virtuellement prendre l'une ou l'autre de ces configurations. Mais configurer un ensemble de P processeurs hétérogène en une grille de taille $p \times q$ où $pq \leq P$, s'avère un problème difficile : il faut choisir le meilleur agencement parmi toutes les permutations possibles, et ce problème est NP-complet. Nous proposons à cet effet une heuristique polynomiale efficace mais sans garantie.

Ainsi, dans un premier temps (Paragraphe 7.2), après avoir brièvement rappelé le principe des algorithmes de multiplication de matrices et de décomposition LU et QR implémentés dans ScaLAPACK, nous présentons notre stratégie d'allocation de données fondée sur la définition d'une grille parfaite. L'agencement d'un ensemble de processeurs en une grille parfaite non singulière n'existant pas nécessairement, nous introduisons alors dans le Paragraphe 7.3.1 la problématique liée à la construction d'une grille optimale et résumons quelques résultats connexes antérieurs. Nous prouvons alors la NP-complétude de notre problème (Paragraphe 7.3.3), fournissons quelques réductions (Paragraphe 7.3.4) et proposons une heuristique (Paragraphe 7.3.5). Ces algorithmes ayant été implémentés, nous avons effectué des mesures de performances que nous décrivons dans le Paragraphe 7.4. Nous concluons ce chapitre par la présentation d'extensions possibles et de travaux futurs.

7.2 Description des algorithmes, sur une grille homogène et sur une grille hétérogène parfaite

Dans ce paragraphe, nous rappelons dans un premier temps le principe des algorithmes, implantés dans ScaLAPACK [15], pour la multiplication de matrices et les décompositions LU et QR sur une grille bidimensionnelle de processeurs (le cas 1D peut être vu comme un cas particulier du cas 2D). Ensuite après avoir donné la définition d'une grille hétérogène parfaite, nous présentons notre stratégie d'allocation et décrivons l'algorithme associé.

7.2.1 Algèbre linéaire sur une grille 2D

7.2.1.1 Multiplication de matrices sur une grille homogène

Considérons le produit de matrices carrées $C = A \times B$ de taille $n \times n$. L'algorithme utilisé par ScaLAPACK est décrit dans [2, 45, 68]. Le principe est le suivant : considérons une grille de processeurs de taille $p \times q$. Supposons dans un premier temps que $p = q = n$. Dans ce cas, les trois matrices partagent le même schéma d'allocation sur la grille 2D : le processeur $P_{i,j}$ stocke les scalaires $a_{i,j}$, $b_{i,j}$ et $c_{i,j}$. Ainsi à chaque étape k,

- chaque processeur $P_{i,k}$ (pour tout $i \in \{1, .., p\}$) diffuse horizontalement $a_{i,k}$ aux processeurs $P_{i,1:q}$.
- chaque processeur $P_{k,j}$ (pour tout $j \in \{1, .., q\}$) diffuse verticalement $b_{k,j}$ aux processeurs $P_{1:p,j}$.
- de telle manière que chaque processeur $P_{i,j}$ peut alors indépendamment exécuter $c_{i,j} = a_{i,k} \times b_{k,j} + c_{i,j}$.

Cet algorithme est utilisé dans ScaLAPACK pour sa scalabilité, et parce qu'il ne nécessite aucune redistribution initiale des données (contrairement à l'algorithme de Cannon [68]). De plus, les diffusions étant effectuées indépendamment et de manière régulière, elles peuvent être pipelinées.

Comme noté dans le Chapitre 6, ScaLAPACK utilise une version pavée de cet algorithme : chaque scalaire dans la description ci-dessus doit être remplacé par un bloc de taille $b \times b$. Une matrice de taille $nb \times nb$ sera donc à partir de maintenant considérée virtuellement comme une matrice de taille $n \times n$ où un élément atomique est un bloc de taille $b \times b$. De plus, généralement, le nombre de blocs est bien supérieur à pq, et ceux-ci sont distribués cycliquement sur la grille de processeurs, de telle manière qu'à chaque étape un processeur soit responsable de la mise à jour de plusieurs blocs. En d'autres termes, ScaLAPACK utilise une distribution $CYCLIC(b)$ dans chacune des directions, identique pour chacune des trois matrices.

7.2.1.2 Décomposition LU et QR sur une grille homogène

Considérons une matrice A constituée de n^2 blocs de taille $b \times b$ que l'on souhaite décomposer en LU. Supposons que les $k \in \{1, \ldots, n\}$ premières colonnes de blocs aient déjà été mises à jour. A l'étape suivante,

1. la colonne de blocs suivante est factorisée et permutée si nécessaire (en fonction du pivot). C'est ce que l'on note par "F". Cela correspond au calcul de la partie L de la décomposition LU : calcul d'un facteur et d'une permutation. La factorisation d'une colonne nécessite des communications verticales au sein de l'ensemble des processeurs détenant cette colonne.
2. Le facteur pivot inférieur est alors diffusé aux processeurs de droite en utilisant une topologie d'anneau, c'est à dire communiqué à son voisin de droite qui lui même le communique à son

voisin de droite, etc. Cette forme pipelinée permet un recouvrement de la diffusion par les calculs.

3. Chaque facteur supérieur peut alors être mis à jour. Phase que l'on note "U" comme *update*. Chacun de ces facteurs est diffusé aux processeurs de la même colonne en utilisant une topologie d'arbre couvrant minimum, et les blocs restants peuvent ainsi être mis à jour.

Cette version appelée *"outer-product algorithm"*, choisie pour l'implémentation de ScaLAPACK [27], est scalable et fournit de bonnes performances [41, 27, 14]. La parallélisation de la décomposition QR est analogue [30, 28].

La Figure 7.1 représente le graphe des tâches de cette version. En pratique, la factorisation du pivot ("F" sur le schéma) est effectuée juste après sa mise à jour ("U" sur le schéma) afin de pouvoir débuter sa diffusion le plus tôt possible. Ainsi, si la machine cible le permet, les communications liées à la diffusion sont recouvertes par les calculs de mise à jour.

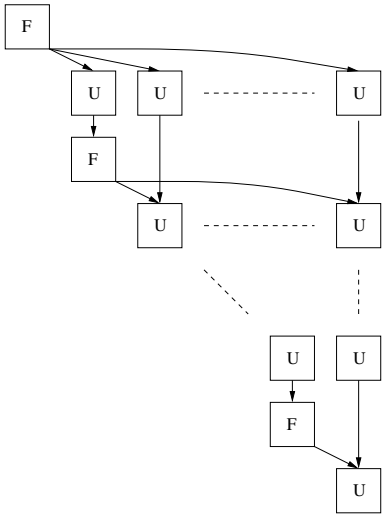


FIG. 7.1: *Graphe de tâches des décompositions LU et QR. L'indice "F" correspond à la factorisation du pivot et "U" à la mise à jour.*

7.2.2 Grille 2D hétérogène de processeurs parfaite

Définition d'une grille parfaite. Considérons $p \times q$ processeurs $P_{i,j}$ de temps de cycle $t_{i,j}$ où $1 \leq i \leq p$ et $1 \leq j \leq q$. Une telle grille est dite *parfaite* si la matrice des temps de cycle $T = (t_{i,j})$ (ou la matrice des vitesses $S = (s_{i,j}) = (\frac{1}{t_{i,j}})$) est de rang 1.
Notons $r_1 = 1$ et $r_i = \frac{s_{i,1:q}}{s_{1,1:q}}$, les deux vecteurs $s_{i,1:q}$ et $s_{1,1:q}$ étant colinéaires. De même, notons $c_1 = s_{1,1}$ et $c_j = \frac{s_{1:p,j}}{s_{1:p,1}} \times c_1$. Avec ces notations, on définit les vecteurs colonne $r = (r_1, r_2, \ldots, r_p)^t$ et $c = (c_1, c_2, \ldots, c_q)^t$, et l'on a $S = rc^t$. Remarquons que la matrice $T_{exe} = rTc^t$ a tous ses coefficients égaux à 1. En fait, la matrice T_{exe} fournit le temps d'exécution de chacun des processeurs. Le fait que tous les coefficients vaillent 1 signifie que tous les processeurs terminent en même temps (équilibrage parfait).

La colonne de processeurs $P_{1:p,j}$ peut alors être virtuellement considérée comme un processeur de vitesse c_j (il faut donc lui assigner un nombre de colonnes proportionnel à c_j). De même, la ligne

de processeurs $P_{i,1:q}$ peut être virtuellement considérée comme un processeur de vitesse r_i (on doit lui assigner un nombre de lignes proportionnel à r_i).

Ainsi, de manière statique, *à une permutation des lignes et des colonnes près*, l'allocation des données d'une matrice sur une grille de processeurs parfaite se présente comme schématisé dans la Figure 7.2.

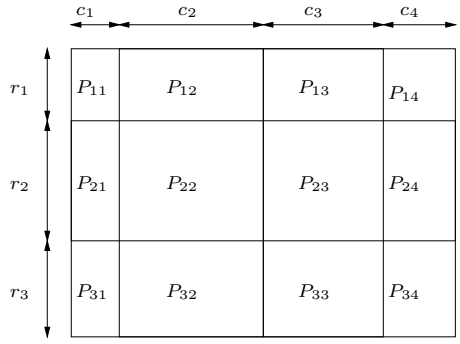


FIG. 7.2: *Allocation des matrices sur une grille de processeurs de taille* 3×4.

En d'autres termes, l'allocation des données sur la grille de processeurs parfaite peut être décomposée en deux temps : une allocation horizontale et une allocation verticale. Il suffit donc d'appliquer les résultats du Chapitre 6 à chacune de ces deux directions pour obtenir l'allocation souhaitée. Le paragraphe suivant est voué à la description précise de cette allocation, légèrement améliorée dans la direction horizontale afin de tenir compte du pivot dans la décomposition LU (cf Paragraphe 6.3.1).

7.2.3 Allocation sur une grille parfaite

Rappelons que les matrices de taille $nb \times nb$ sont initialement pavées en tuiles de taille $b \times b$. Ce sont ces tuiles qui sont distribuées sur les processeurs. Un degré de virtualisation est donc ajouté, et on parlera à partir de maintenant de matrices de taille $n \times n$.

La construction de l'allocation se décompose en plusieurs étapes. Afin d'illustrer son principe considérons l'exemple suivant :

- On dispose d'une grille de 2×3 processeurs de temps de cycle $T = \begin{pmatrix} 3 & 5 & 7 \\ 6 & 10 & 14 \end{pmatrix}$
- la matrice est de taille 12×12.

Allocation cyclique d'un motif. Dans un premier temps, on fixe une taille de motif $B_p \times B_q$ (similaire à la largeur de motif (B), utilisée dans les paragraphes 6.3.1 et 6.4.2). Le motif, constitué d'indices de processeurs, est à déterminer. Il est reproduit périodiquement dans chacune des deux directions, avec une période B_q dans la direction horizontale et B_p dans la direction verticale. Bien sûr, B_p et B_q peuvent être choisis égaux à n. Plus ils sont grands, plus l'équilibrage de charge statique est bon, mais prendre des valeurs trop grandes est inutile (cf Paragraphe 6.4.2). D'autre part, il peut être intéressant de prendre de petites valeurs pour B_p et B_q afin que l'allocation soit la plus régulière possible.

Pour notre exemple, nous pouvons prendre $B_p = 3$ et $B_q = 6$.

Détermination verticale du motif. Nous cherchons ici une allocation des matrices commune aux différents algorithmes d'algèbre linéaire considérés dans ce chapitre.

- pour la multiplication de matrices, l'ensemble des lignes doit être bien équilibré.
- pour les décompositions LU ou QR, à toute étape k, l'ensemble des lignes $\{k, k+1, \ldots, n\}$ doit être bien équilibré.

On en conclut que pour tout m, l'ensemble $\{m, m+1, \ldots, B_p\}$ des lignes du motif doit être équilibré.

Il suffit donc d'appliquer le principe de l'algorithme incrémental présenté dans le Chapitre 6, ce qui donne :

Algorithme 10. *Allocation statique verticale pour la décomposition LU sur p lignes de processeurs de temps de cycle $t_{1,1}, t_{2,1}, t_{3,1}, \ldots, t_{p,1}$.*

Distribue_lignes$(B_p, t_{1,1}, t_{2,1}, t_{3,1}, \ldots, t_{p,1})$
{ Initialisation }
do $i = 1, p$
    $r_i = 0$
{ Construit itérativement l'allocation σ du bas vers le haut}
do $m = B_p, 1 : -1$
    trouve $k \in \{1, \ldots, p\}$ tel que $t_{k,1} \times (r_k + 1) = \min\{t_{i,1} \times (r_i + 1)\}$
    $r_k = r_k + 1$
    $\sigma_v[m] = k$
Retourne$(\sigma_v, r_1, r_2, \ldots, r_p)$

Ainsi, pour notre exemple, l'Algorithme 10 donne verticalement le motif $(P_{2,*}P_{1,*}P_{1,*})$ correspondant au résultat $\sigma_v = (2, 1, 1)$.

Détermination horizontale du motif. Cette étape est plus complexe, car il faut tenir compte du pivot de la décomposition LU. Il faut donc trouver un équilibrage statique global, tel qu'à chaque étape la factorisation du pivot ainsi que les mises à jour soient globalement équilibrées. La solution est basée sur les idées suivantes :

- allouer systématiquement la colonne du pivot à la colonne de processeurs la plus rapide. Ainsi, à l'étape k, *la mise à jour et la factorisation* de la k-ième colonne de blocs sont effectuées par la colonne de processeurs la plus rapide (notée $P_{1:p,1}$).
- l'allocation statique est déterminée en utilisant une version légèrement modifiée de l'algorithme incrémental : les deux premiers atomes sont automatiquement alloués à la colonne de processeurs la plus rapide.

L'algorithme d'allocation est le suivant :

Algorithme 11. *Allocation statique horizontale pour la décomposition LU sur q colonnes de processeurs de temps de cycle* $t_{1,1} \leq t_{1,2}, t_{1,3}, \ldots, t_{1,q}$.

> **Distribue_colonnes**$(B_q, t_{1,1}, t_{1,2}, t_{1,3}, \ldots, t_{1,q})$
> { Initialisation : allocation des 2 premières colonnes }
> $c_1 = 2$
> $\sigma_h[1] = \sigma_h[2] = 1$
> do $j = 2, q$
> $c_j = 0$
> { Construit itérativement l'allocation σ_h de la droite vers la gauche}
> do $m = B_q, 3 : -1$
>   trouve $k \in \{1, \ldots, q\}$ tel que $t_{1,k} \times (c_k + 1) = \min\{t_{1,j} \times (c_j + 1)\}$
> $c_k = c_k + 1$
> $\sigma_h[m] = k$
> **Retourne**$(\sigma_h, c_1, c_2, \ldots, c_q)$

Sur notre exemple, on obtient donc comme motif horizontal $(P_{*,1}P_{*,1}P_{*,2}P_{*,1}P_{*,3}P_{*,2})$ correspondant au résultat $\sigma_h = (1, 1, 2, 1, 3, 2)$.

En conséquence, le motif vaut

$$\begin{bmatrix} P_{2,1} & P_{2,1} & P_{2,2} & P_{2,1} & P_{2,3} & P_{2,2} \\ P_{1,1} & P_{1,1} & P_{1,2} & P_{1,1} & P_{1,3} & P_{1,2} \\ P_{1,1} & P_{1,1} & P_{1,2} & P_{1,1} & P_{1,3} & P_{1,2} \end{bmatrix},$$

et l'allocation est donnée par la table suivante :

$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$
$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$
$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$
$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,3}$	$P_{2,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,3}$	$P_{1,2}$

Une fois l'allocation déterminée, l'algorithme de factorisation LU se décompose de la manière suivante :

- **Etape 1** : factorisation par la première colonne de processeurs $P_{1:p,1}$ de la première colonne de blocs. Diffusion horizontale du facteur vers les autres colonnes de processeurs.
- **Etape** $k > 1$**, première colonne de processeurs** $(P_{1:p,1})$:
  - mise à jour puis factorisation de la k-ième colonne de blocs.
  - diffusion de la k-ième colonne vers les autres colonnes de processeurs. C'est à dire chaque processeur $P_{i,1}$ diffuse aux processeurs $P_{i,2:q}$ le bloc $A_{i,k}$.

- réception de la $(k+1)$-ième colonne de blocs provenant de la $\sigma_h[k+1]$-ième colonne de processeurs ($P_{1:p,\sigma_h[k]}$).
- mise à jour des blocs restants.

– **Etape $k > 1$, $\sigma_h[k+1]$-ième colonne de processeurs ($P_{1:p,\sigma_h[k+1]}$)** :
- mise à jour de la $(k+1)$-ième colonne de blocs.
- envoi de cette colonne à la première colonne de processeurs. C'est à dire, chaque processeur $P_{i,\sigma_h[k+1]}$ envoie le bloc $A_{i,k+1}$ au processeur $P_{i,1}$.
- mise à jour des colonnes de blocs restantes

– **Etape $k > 1$, processeurs restants** : mise à jour des blocs de colonnes restantes.

Le schéma de la Figure 7.3 représente les tâches exécutées par les différentes colonnes de processeurs (les communications verticales ne sont pas représentées), pour l'ensemble de processeurs de notre exemple et sur une matrice de taille 6×6.

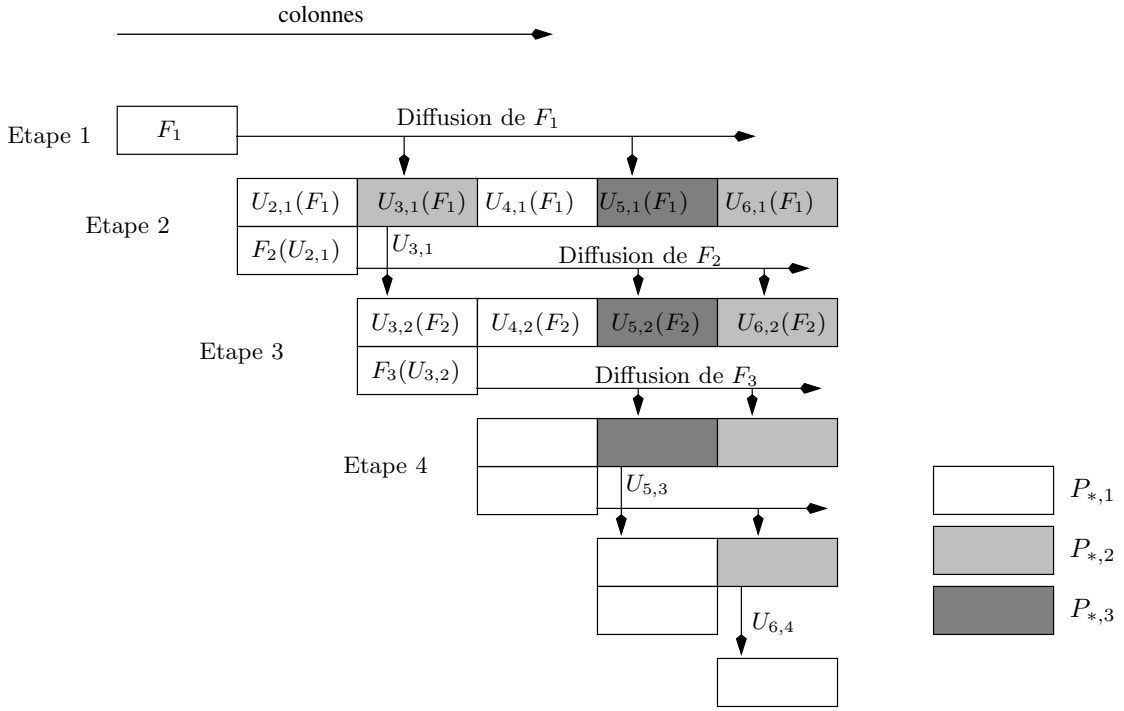


FIG. 7.3: *Allocation et graphe des tâches de la décomposition LU sur 6 colonnes et 3 colonnes de processeurs de temps de cycles respectifs 3,5 et 7. F_k représente la factorisation de la colonne k et $U_{j,k}$ représente la mise à jour de la colonne j à l'aide du facteur F_k.*

7.3 Allocation d'un ensemble de processeurs hétérogène sur une grille 2D

Dans le paragraphe précédant, nous avons présenté la stratégie d'allocation sur une grille hétérogène parfaite de processeurs, et décrit dans ce cas le principe de l'algorithme de factorisation LU. Dans ce paragraphe, nous traitons le cas général d'un ensemble hétérogène quelconque de processeurs en essayant de se rattacher au cas particulier d'une grille parfaite. Ainsi, ce paragraphe peut être décomposé de la manière suivante : dans un premier temps, nous introduisons la

problématique puis formalisons le problème qui ainsi exprimé est NP-complet. C'est ce que nous prouvons alors. Nous proposons alors une méthode exhaustive ainsi qu'une solution heuristique.

7.3.1 Problématique

En pratique, il est rare qu'il soit possible d'agencer un ensemble de processeurs hétérogène donné, en une grille parfaite non singulière (i.e. non réduite à un anneau). Prenons par exemple quatre processeurs de temps de cycle $1, 2, 3$ et 5 : il n'existe pas de matrice de taille 2×2 et de rang 1, constituée des nombres $1, 2, 3$ et 5. Un temps de cycle ne pouvant, bien entendu, pas être utilisé deux fois.

Il existe deux solutions pour pallier à ce problème. Soit on agence les processeurs en une grille, par exemple $T = \begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}$, et on approche cette grille par une grille parfaite, par exemple $T^{opt} = \begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix}$. C'est la solution choisie dans ce chapitre. On obtient alors un découpage des données semblable à celui schématisé Figure 7.2. Soit on cherche un découpage des données autre qu'un découpage en grille (cf Figure 7.4). C'est la solution proposée par Kalinov et Lastovetky [61].

Solution de Kalinov et Lastovetsky. L'article de Kalinov et Lastovetky est le seul à notre connaissance traitant du problème d'allocation de données pour l'implémentation de l'algorithme de décomposition LU sur une grille 2D. Ils proposent une distribution "bloc-cyclique hétérogène", et utilisent l'outil de programmation mPC [8]. Ils considèrent un agencement des processeurs fixe, dont ils ne donnent pas de méthode de construction, et allouent les données "par colonne".

L'idée est la suivante : considérons un ensemble de processeurs de temps de cycle $T = (t_{ij})$ où T n'est pas nécessairement de rang 1 et peut contenir des noeuds vides ($t_{ij} = \infty$).

1. dans un premier temps on définit, tout comme pour l'allocation sur une grille parfaite, un motif de taille $B_p \times B_q$ qui sera reproduit cycliquement dans chacune des deux directions.
2. ensuite, on équilibre chaque colonne de processeurs indépendamment. Ainsi, le processeur P_{ij} a approximativement $r_{ij} \approx \frac{B_p}{t_{ij}} \times \frac{1}{\sum_{1 \leq k \leq p} \frac{1}{t_{kj}}}$ lignes qui lui sont attribuées.
3. puis on équilibre les colonnes en prenant pour chaque colonne de processeurs $P_{1:p,j}$ comme vitesse $v_j = \sum_{1 \leq i \leq p} \frac{1}{t_{ij}}$. Ainsi, la colonne de processeurs $P_{1:p,j}$ a approximativement $c_j \approx B_q \times \frac{v_j}{\sum_{1 \leq k \leq q} v_k}$ colonnes qui lui sont attribuées.

La Figure 7.4 représente le type de motif que l'on obtiendrait pour un ensemble de 5 processeurs agencés selon $T = \begin{pmatrix} t_{1,1} & t_{1,2} \\ t_{2,1} & t_{2,2} \\ \infty & t_{3,2} \end{pmatrix}$.

Comme on peut le remarquer sur cette figure, un processeur peut posséder plus d'un seul voisin à sa droite. Cela rend l'implémentation de la diffusion difficile, et c'est pourquoi nous avons choisi dans ce chapitre de nous restreindre au cas d'une allocation "en grille". La recherche d'une allocation optimale par colonnes semble néanmoins nécessaire dans le cas où la vitesse des différentes ressources varie au cours du temps. Le problème est alors dual à celui traité ici : sous la contrainte d'un équilibrage de charge parfait, on cherche l'allocation qui minimise le surcoût des communications. Une solution heuristique est proposée dans [60] et nous fournissons une solution optimale en un temps polynomial dans le Chapitre 8 de cette thèse.

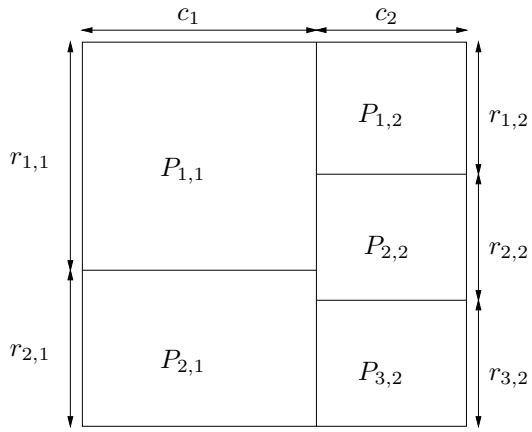


FIG. 7.4: *Schéma d'une allocation par colonnes sur un ensemble de 5 processeurs.*

Découpage en grille. Le problème de découpage d'un carré en grille à été fortement étudié dans un contexte dual : il s'agit d'allouer un ensemble de calculs *hétérogène* sur une grille de processeurs *homogène*. En d'autres termes, étant donnée une matrice de nombres entiers positifs, il faut trouver un partitionnement de cette matrice en grille minimisant le maximum des poids de chaque partie, le poids d'une partie (un rectangle) étant la somme des entiers qu'elle contient. Ce problème est NP-complet [51], et des solutions heuristiques sont proposées dans [65, 9].

Problèmes géométriques connexes. Le problème d'optimisation que nous traitons dans ce chapitre est semblable à un certain nombre d'autres problèmes géométriques existants. Une synthèse est effectuée dans le chapitre suivant. Notons néanmoins que de nombreux problèmes d'optimisation géométrique NP-complets sont décrits dans le "NP-Compendium" [34, 10].

7.3.2 Formalisation du problème

Considérons P processeurs $P_1, P_2, \ldots, P_P$ de temps de cycle $t_1, t_2, \ldots, t_P$. Le problème est d'agencer ces processeurs sur une grille bidimensionnelle de taille $p \times q \leq P$ la plus "proche" possible d'une grille parfaite. En d'autres termes, on veut trouver un agencement associé à une grille parfaite, de telle manière que l'allocation ainsi construite permette une exécution du produit de matrices en un temps minimal. L'approximation s'applique aux deux autres algorithmes LU et QR.

Considérons un tel arrangement des $pq \leq P$ processeurs en une grille de taille $p \times q$. Réindexons les processeurs par P_{ij} et leur temps de cycle par t_{ij}. Supposons que soit assigné au processeur P_{ij} un rectangle de taille $r_i \times c_j$. Il y a alors plusieurs manières équivalentes d'évaluer la valeur de cette allocation :

– Chaque processeur $P_{i,j}$ travaillant durant $r_i \times c_j \times t_{ij}$ unités de temps, le temps total d'exécution d'un motif (de taille $B_p \times B_q = (\sum_{i=1}^{p} r_i \times \sum_{j=1}^{q} c_j)$ vaut donc

$$t_{exe} = \max_{i,j}\{r_i \times t_{ij} \times c_j\}.$$

Cette fonction de $r = (r_1, \ldots, r_p)$ et $c = (c_1, \ldots, c_q)$ étant bilinéaire, le coût d'une allocation peut être normalisé à un motif de taille 1×1. On obtient le problème d'optimisation suivant :

$$\textbf{Objectif } Obj_1 : \min_{(\sum_i r_i = 1; \sum_j c_j = 1)} \max_{i,j}\{r_i \times t_{ij} \times c_j\}$$

A partir d'une solution au problème Obj_1, on déduit le motif recherché en multipliant les r_i par B_p et les c_j par B_q et en ajustant à des entiers proches.

– L'expression duale de ce problème d'optimisation est la suivante : quelle est la taille de motif maximum pouvant être traité en une unité de temps par l'ensemble des processeurs ? On aboutit au problème d'optimisation équivalent suivant :

$$\textbf{Objectif } Obj_2 : \max_{r_i \times t_{ij} \times c_j \leq 1} \{(\sum_i r_i) \times (\sum_j c_j)\}$$

De même que dans le cas précédent, il suffit de multiplier chacun des r_i par $\frac{B_p}{\sum r_i}$ et chacun des c_j par $\frac{B_q}{\sum c_j}$ pour obtenir le motif recherché initialement.

– Dans cette expression il y a $p+q$ variables r_i et c_j alors qu'il n'y a que $p+q-1$ degrés de liberté : si on multiplie les r_i par un facteur λ et que l'on divise les c_j par ce même facteur, on obtient une solution équivalente. Ainsi, on peut imposer, sans perte de généralité, $r_1 = 1$. On peut alors, dans ce cas, manipuler l'équation ci-dessus et obtenir

$$\begin{aligned}
\max_{r_i \times t_{ij} \times c_j \leq 1} \{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} c_j)\} &= \max_{r_i} \{ \max_{c_j \text{ avec } r_i \times t_{ij} \times c_j \leq 1} \{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} c_j)\}\} \\
&= \max_{r_i} \{(\sum_{i=1}^{p} r_i) \times \max_{c_j \text{ avec } r_i \times t_{ij} \times c_j \leq 1} \{(\sum_{j=1}^{q} c_j)\}\} \\
&= \max_{r_i} \{(\sum_{i=1}^{p} r_i) \times \max_{\forall i,\, c_j \leq \frac{1}{r_i \times t_{ij}}} \{(\sum_{j=1}^{q} c_j)\}\} \\
&= \max_{r_i} \{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} \min_i \{\frac{1}{r_i \times t_{ij}}\})\} \\
&= \max_{r_i} \{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} \frac{1}{\max_i \{r_i \times t_{ij}\}})\}
\end{aligned}$$

On a ainsi une expression avec p variables et $(p-1)$ degrés de liberté. Mais cette expression est difficilement utilisable. Nous avons essayé de réduire cette équation en vain (ceci nous a conduit à une solution analytique pour le cas $p = q = 2$ [17]). De plus, le problème qui nous concerne est plus complexe. En effet, il nous faut minimiser l'une de ces expressions *pour tout agencement* de pq processeurs en une grille de taille $p \times q$. En fait, ce problème est NP-complet. C'est ce que nous nous proposons de montrer dans le paragraphe suivant.

7.3.3 NP-Complétude

Dans ce paragraphe, nous formalisons le problème d'optimisation précédemment décrit et restreint au cas particulier $p = q$ et $P = p^2$, et prouvons sa NP-Complétude. Nous utilisons à cet effet, la notion (plus pratique) de vitesse d'un processeur définie comme $s_i = \frac{1}{t_i}$. Nous débutons avec la définition du problème d'équilibrage de charge 2D où σ représente l'arrangement des p^2 processeurs sur la grille de taille $p \times p$, et r_i et c_j les variables utilisées dans Obj_1.

Définition 8 *MAX-GRID(s) : soient p^2 nombres réels positifs $s_1, \ldots, s_{p^2}$, trouver*

$$(r_1, \ldots, r_p, c_1, \ldots, c_p) \textit{ et un arrangement (bijection) } \sigma \textit{ de } [1,p] \times [1,p] \textit{ vers } [1,p^2]$$

tels que

$$\forall (i,j) \in [1,p] \times [1,p], \quad r_i c_j \leq s_{\sigma(i,j)} \text{ et } (\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \text{ soit maximal.}$$

Le problème de décision associé au problème d'optimisation MAX-GRID est le suivant :

Définition 9 *MAX-GRID(s,K) : soient p^2 nombres réels positifs $s_1, \ldots, s_{p^2}$ et un nombre réel positif K, existe-t-il*

$$(r_1, \ldots, r_p, c_1, \ldots, c_p) \text{ ainsi qu'un arrangement (bijection) } \sigma \text{ de } [1,p] \times [1,p] \text{ vers } [1,p^2]$$

tels que

$$\forall (i,j) \in [1,p] \times [1,p], \quad r_i c_j \leq s_{\sigma(i,j)} \text{ et } (\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \geq K \text{ ?}$$

Théorème 14 *MAX-GRID(s,K) est NP-complet.*

Preuve. La preuve est longue et assez technique. Elle nécessite plusieurs lemmes. L'idée est de réduire le problème à celui de 2-Partition :

Lemme 11

$$\textit{2P-eq} \leq_P \textit{MAX-GRID},$$

où 2P-eq est défini comme suit :

Définition 10 *2-Partition-Equal (2P-eq)*
Soit un ensemble de m entiers $\mathcal{A} = \{a_1, \ldots, a_m\}$, existe-t-il une partition de $\{1, \ldots, m\}$ en deux ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ tels que

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i \text{ et } card(\mathcal{A}_1) = card(\mathcal{A}_2) \text{ ?}$$

Comme 2P-eq est NP-complet [47], le Lemme 11 prouve le Théorème 14.

(a) Réduction : 2P-eq $\leq_P$ MAX-GRID(s,K). Considérons une instance arbitraire du problème de 2-Partition-Equal, i.e. la donnée d'un ensemble $\mathcal{A} = \{a_1, \ldots, a_{2n}\}$ composé de $2n$ entiers. Il nous faut transformer polynomialement cette instance en une instance du problème MAX-GRID ayant une solution si et seulement si l'instance initiale de 2-Partition-Equal admet une solution.

Définissons pour cela $\{b_1, \ldots, b_{2n}\}$ par $\forall i, b_i = (a_i + 2n \max_k a_k)$. Ainsi, $2n \max a_i \leq b_i \leq (2n+1) \max a_i$. On considère alors l'instance du problème MAX-GRID (notée abusivement MAX-GRID$(b_1, \ldots, b_{2n}, K)$) où

$$\begin{cases} K &= (4n \max a_i)^2 + \sum_{i=1}^{2n} b_i + \frac{(\sum_{i=1}^{2n} b_i)^2}{4(4n \max a_i)^2}, \\ s_1 &= (4n \max a_i)^2, \\ s_{i+1} &= b_i, \quad \forall i,\ 1 \leq i \leq 2n, \\ s_i &= 1, \quad \forall i,\ 2n+2 \leq i \leq (n+1)^2. \end{cases}$$

Par la suite, nous allons montrer qu'une solution à ce problème est nécessairement de la forme schématisée dans la Figure 7.5, où σ, restreint à l'ensemble $([2, n+1], 1) \bigcup (1, [2, n+1])$, définit une bijection avec $[2, 2n+1]$ et

$$\begin{cases} & r_1 c_1 &=& (4n \max a_i)^2, \\ \forall i, \quad 2 \leq i \leq n+1, & r_i &=& \frac{b_{\sigma(i,1)}}{c_1}, \\ \forall j, \quad 2 \leq j \leq n+1, & c_j &=& \frac{b_{\sigma(1,j)}}{r_1}, \end{cases}$$

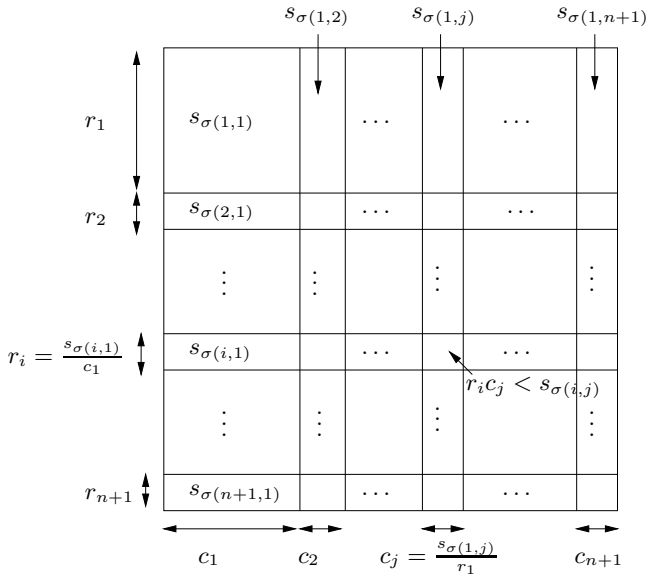


FIG. 7.5: *Solution de MAX-GRID($b_1, b_2, \ldots, b_{2n}, K$).*

Ainsi, nous pouvons vérifier que

$$(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \geq K \Longleftrightarrow \sum_{2}^{n+1} b_{\sigma(i,1)} = \sum_{2}^{n+1} b_{\sigma(1,j)}$$

De manière intuitive, comme s_1 est largement plus grand que les autres aires et que $b_i \gg 1$, il faut (à une permutation près sur les lignes et sur les colonnes) positionner s_1 dans le coin en haut à gauche et remplir la première ligne et la première colonne avec les b_i. Il est donc possible de saturer chacun de ces processeurs en prenant $r_1 c_1 = s_1$, $r_i = \frac{s_{\sigma(i,1)}}{c_1}$ et $c_j = \frac{s_{\sigma(1,j)}}{r_1}$. Ensuite, l'aire de la grille est maximale lorsqu'elle est équilibrée, c'est à dire si les b_i vérifient 2P-eq. Ces différents points constituent les étapes successives de la preuve ci-dessous.

Lemme 12 *Si $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j) \geq K$, alors l'aire assignée au processeur de vitesse s_1 a pour valeur minimum $11n^2 \max a_i{}^2$.*

Preuve. A une permutation près, supposons que $\sigma^{-1}(1) = (1,1)$, c'est à dire que l'aire assignée au processeur de vitesse s_1 vaille $r_1 c_1$. Comme $\sum_{i \geq 2} s_i \leq 5n^2 \max a_i{}^2$, et que

$$(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \leq r_1 c_1 + \sum_{i \geq 2} s_i,$$

on a

$$r_1 c_1 \geq 11 n^2 \max {a_i}^2.$$

■

Lemme 13 *Soit σ une bijection de $[1, n+1] \times [1, n+1]$ vers $[1, (n+1)^2]$ telle que $\sigma(1,1) = 1$. Supposons que $r_1 c_1 \geq 11 n^2 \max {a_i}^2$ (Lemme 12), alors $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j)$ est maximal si et seulement si*

$$\begin{cases} \forall i \geq 2, & r_i c_1 = s_{\sigma(i,1)}, \\ \forall j \geq 2, & c_j r_1 = s_{\sigma(1,j)}. \end{cases}$$

Preuve. Par définition,

$$\begin{cases} \forall i \geq 2, & r_i c_1 \leq s_{\sigma(i,1)} \\ \forall j \geq 2, & c_j r_1 \leq s_{\sigma(1,j)}, \end{cases}$$

et $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j)$ est maximal lorsque chaque r_i et chaque c_j sont maximals.

De plus, si

$$\begin{cases} \forall i \geq 2, & r_i c_1 = s_{\sigma(i,1)} \\ \forall j \geq 2, & c_j r_1 = s_{\sigma(1,j)} \end{cases}$$

alors toutes les autres conditions $r_i c_j \leq s_{\sigma(i,j)}$ sont automatiquement vérifiées. En effet,

$$\forall (i,j) \neq (1,1), \quad 1 \leq s_{\sigma(i,j)} \leq (2n+1) \max a_i$$

d'où

$$\begin{aligned} r_i c_j &= \frac{s_{\sigma(i,1)} s_{\sigma(1,j)}}{r_1 c_1}, \\ &\leq \frac{(2n+1)^2 \max {a_i}^2}{11 n^2 \max {a_i}^2} \\ &\leq 1 \\ &\leq s_{\sigma(i,j)}. \end{aligned}$$

■

Pour un σ donné, la valeur maximale de $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j)$ vaut

$$S(\sigma) = (r_1 + \frac{\sum_2^{n+1} s_{\sigma(i,1)}}{c_1})(c_1 + \frac{\sum_2^{n+1} s_{\sigma(1,j)}}{r_1}).$$

Notons

$$s = r_1 c_1, \quad S_1 = \sum_2^{n+1} s_{\sigma(i,1)} \text{ et } S_2 = \sum_2^{n+1} s_{\sigma(1,j)}.$$

Alors,

$$S(\sigma) = s + (S_1 + S_2) + \frac{S_1 S_2}{s}.$$

Lemme 14

$$S(\sigma) \geq K \Longleftrightarrow (s = (4n \max a_i)^2) \text{ et } (S_1 = S_2 = \frac{\sum b_i}{2})$$

et alors $S(\sigma) \geq K$ si et seulement si il existe une solution au problème de 2P-eq défini ci-dessus.

Preuve. Par construction, $S_1 + S_2 \leq \sum b_i$ et donc $S_1 S_2 \leq \frac{(\sum b_i)^2}{4}$. De plus, $S_1 S_2 = \frac{(\sum b_i)^2}{4}$ si et seulement si $S_1 = S_2 = \frac{\sum b_i}{2}$. Ainsi

$$S(\sigma) \leq s + \sum b_i + \frac{(\sum b_i)^2}{4s}.$$

Considérons la fonction g définie par

$$g(s) = s + \frac{(\sum b_i)^2}{4s}.$$

Cette fonction est décroissante puis croissante et admet un minimum en $\frac{\sum b_i}{2}$. Comme

$$\frac{\sum b_i}{2} \ll 11n^2 \max a_i{}^2 \leq s \leq (4n \max a_i)^2,$$

g est maximal quand $s = (4n \max a_i)^2$. En d'autres termes,

$$\begin{cases} S(\sigma) \leq (4n \max a_i)^2 + \sum b_i + \frac{(\sum b_i)^2}{4(4n \max a_i)^2} = K \\ S(\sigma) = K \text{ si et seulement si } (s = (4n \max a_i)^2) \text{ et } (S_1 = S_2 = \frac{\sum b_i}{2}). \end{cases}$$

■

D'ou le résultat : MAX-GRID$(b_1, \ldots, b_{2n}, K)$ admet une solution si et seulement si 2P-eq$(b_1, \ldots, b_{2n})$ admet une solution et donc si et seulement si l'instance initiale de 2P-eq$(a_1, \ldots, a_{2n})$ admet une solution.

(b) Consistance de la transformation effectuée. La dernière partie de la preuve consiste à vérifier que l'instance du problème de MAX-GRID s'exprime à l'aide d'un énoncé de taille polynomiale en la taille de l'énoncé de l'instance initiale du problème de 2P-eq.

Lemme 15 *Notons* $MAX = \max_k a_k$. *Représentons le codage des données a et b par* $c(a)$ *et* $c(b)$. *Alors,*

$$\mathit{Taille}(c(b)) = O(\mathit{Taille}(c(a))^2).$$

Preuve.

$$\text{Taille}(c(a)) = \sum_k \log(a_k) \geq \log(\text{MAX}) + (n-1)\log(\min_k a_k) \geq (n-1)\log 2 + \log \text{MAX}.$$

$$\text{Taille}(c(b)) = \sum_k \log(b_k) = \sum_k \log(2n\ \text{MAX}(1 + \frac{a_k}{n\text{MAX}})) \leq 1 + n(\log n + \log 2) + n \log \text{MAX}.$$

D'où,

$$\text{Taille}(c(b)) = O(\text{Taille}(c(a))^2).$$

■

Ceci achève la preuve de NP-complétude du problème MAX-GRID. ■

7.3.4 Solution exacte

Dans cette partie, nous donnons un algorithme permettant de résoudre de manière exacte le problème d'optimisation pour *un petit nombre de processeurs* $P = pq$ sur une grille de taille fixée $p \times q$. Dans un premier temps, nous réduisons le nombre d'arrangements à générer. Ensuite, nous proposons un algorithme résolvant le problème d'optimisation Obj_1 (et son équivalent Obj_2) pour un arrangement donné. Malheureusement, chacune des ces deux étapes fournit un nombre exponentiel de cas à considérer. Ainsi, le paragraphe suivant sera-t-il consacré à la construction d'une solution heuristique.

7.3.4.1 Réduction de la recherche à l'ensemble des arrangements croissants

Dans ce paragraphe, nous montrons qu'il est inutile de générer tous les arrangements possibles ($(pq)!$ possibilités), et nous réduisons cette recherche à l'ensemble des arrangements croissants ($\frac{(pq)!\left(\Pi_{i=1}^{p-1} i!\right)\left(\Pi_{i=1}^{q-1} i!\right)}{\Pi_{i=1}^{p+q-1} i!}$ possibilités) dont la définition est donnée ci-dessous.

Définition 11 (Arrangement) *Un arrangement est une bijection* σ *de* $[1, pq]$ *vers* $[1, p] \times [1, q]$ *qui affecte une position à chaque processeur dans la grille.*

Définition 12 (Arrangement croissant) *Un arrangement* σ *est croissant si pour tout* $i \leq i'$ *et* $j \leq j'$ *on a* $t_{\sigma^{-1}(i,j)} \leq t_{\sigma^{-1}(i',j')}$.

Définition 13 (préordre sur la grille de processeur) *Soient* (i, j) *et* (i', j') *dans* $[1, p] \times [1, q]$. *On dira que* $(i, j) \leq (i', j')$ *si et seulement si* $i \leq i'$ *et* $j \leq j'$.

Proposition 3 *Il existe une solution optimale minimisant l'expression du temps d'exécution de* Obj_1 *construite sur un arrangement croissant.*

Preuve. La preuve est construite sur les points suivants :

1. Soient $(t_1, t_2, \ldots, t_{pq})$, pq temps de cycle.
2. Soit un arrangement optimal sur la grille de taille $p \times q$ (notons que cet arrangement n'est pas nécessairement croissant).
3. On montre alors que l'on peut appliquer un certain nombre de transpositions "correctes" sur l'arrangement tout en préservant son optimalité, et se "rapprocher" ainsi d'un arrangement croissant.
4. On montre que le nombre d'étapes, décrites ci-dessus, nécessaires pour atteindre un arrangement croissant, est fini.

On a besoin pour cela de la définition suivante :

Définition 14 (Transposition correcte) *Soit* σ *un arrangement. Si* $\sigma(k) < \sigma(l)$ *et* $t_k > t_l$, *alors la transposition* $\tau(k, l)$ *échangeant* $\sigma(k)$ *et* $\sigma(l)$ *est dite correcte.*

Soit un arrangement σ, alors il existe une famille finie de transpositions correctes qui transforme σ en un arrangement croissant. Pour prouver cela, nous introduisons une *fonction poids* W reflétant le degré de non-croissance d'une transposition. Nous allons alors montrer qu'une transposition correcte diminue le poids de l'arrangement sur laquelle elle est appliquée.

Prenons,

$$W(\sigma) = \sum_{i,j} t_{\sigma^{-1}(i,j)} \times (p + q - i - j)$$

Montrons que pour toute transposition correcte τ, $W(\tau(\sigma)) < W(\sigma)$. Considérons pour cela un arrangement non croissant σ et les couples $(i,j) \leq (i',j')$ tels que $t_{\sigma^{-1}(i,j)} > t_{\sigma^{-1}(i',j')}$. En posant $k = \sigma^{-1}(i,j)$ et $l = \sigma^{-1}(i',j')$ la transposition $\tau = \tau(k,l)$ est alors correcte. Notons $\sigma' = \tau \circ \sigma$. On a,

$$\begin{aligned} W(\sigma') &= W(\sigma) + (t_{\sigma^{-1}(i',j')} - t_{\sigma^{-1}(i,j)})(p + q - i - j) \\ &\quad +(t_{\sigma^{-1}(i,j)} - t_{\sigma^{-1}(i',j')})(p + q - i' - j') \\ &= W(\sigma) - ((i' - i) + (j' - j))(t_{\sigma^{-1}(i,j)} - t_{\sigma^{-1}(i',j')}) \\ &< W(\sigma) \end{aligned}$$

Considérons un arrangement optimal σ. Soit $r_1, \ldots, r_p$ et $c_1, \ldots, c_q$ une solution au problème d'optimisation Obj_2. Comme une solution équivalente peut être obtenue en transposant deux lignes ou deux colonnes, on peut supposer sans perte de généralité que $r_1 \geq r_2 \geq \ldots \geq r_p$ et $c_1 \geq c_2 \geq \ldots \geq c_q$. Supposons que σ ne soit pas croissante. Alors il existe α et β tels que $(1,1) \leq (\alpha,\beta) < (p,q)$ et que l'on ait soit $t_{\sigma^{-1}(\alpha+1,\beta)} < t_{\sigma^{-1}(\alpha,\beta)}$, soit $t_{\sigma^{-1}(\alpha,\beta+1)} < t_{\sigma^{-1}(\alpha,\beta)}$. Supposons sans perte de généralité que l'on ait $t_{\sigma^{-1}(\alpha+1,\beta)} < t_{\sigma^{-1}(\alpha,\beta)}$. En effet, le problème étant symétrique, l'autre cas se traite de manière analogue.

Soit $k = \sigma^{-1}(\alpha,\beta)$ et $l = \sigma^{-1}(\alpha+1,\beta)$. Par construction, la transposition $\tau = \tau(k,l)$ est correcte. Notons $\sigma' = \tau \circ \sigma$. Pour montrer l'optimalité de σ' (au sens de Obj_2), les r_i et c_j étant inchangés, il faut montrer que pour tout i et j, $r_i c_j t_{\sigma'^{-1}(i,j)} \leq 1$.

En effet, comme $r_\alpha \geq r_{\alpha+1}$ et $t_{\sigma^{-1}(\alpha+1,\beta)} < t_{\sigma^{-1}(\alpha,\beta)}$,

$$\begin{cases} r_\alpha c_\beta t_{(\tau\circ\sigma)^{-1}(\alpha,\beta)} = r_\alpha c_\beta t_{\sigma^{-1}(\alpha+1,\beta)} \leq r_\alpha c_\beta t_{\sigma^{-1}(\alpha,\beta)} \leq 1 \\ r_{\alpha+1} c_\beta t_{(\tau\circ\sigma)^{-1}(\alpha+1,\beta)} = r_{\alpha+1} c_\beta t_{\sigma^{-1}(\alpha,\beta)} \leq r_\alpha c_\beta t_{\sigma^{-1}(\alpha,\beta)} \leq 1 \end{cases}$$

Ainsi, σ' est optimale. De plus, $W(\sigma') < W(\sigma)$. Maintenant, W ne pouvant prendre qu'un nombre fini de valeurs, la réitération de ce processus aboutit nécessairement, en un nombre fini de pas, à un arrangement croissant. ■

7.3.4.2 Résolution du problème d'optimisation pour un arrangement donné

Dans ce paragraphe, nous donnons une méthode pour résoudre le problème d'optimisation Obj_1 (ou Obj_2) pour un arrangement donné. Soit σ un arrangement sur une grille de taille $p \times q$, et $(r_1, \ldots, r_p, c_1, \ldots, c_q)$ une solution au problème d'optimisation Obj_2. Cette solution maximise l'expression quadratique $(\sum_{1 \leq i \leq p} r_i)(\sum_{1 \leq j \leq q} c_j)$ et vérifie les pq inégalités $r_i t_{ij} c_j \leq 1$. Une des conséquences de la méthode présentée dans ce paragraphe est qu'au moins $p+q-1$ de ces inégalités sont atteintes. Nous effectuons pour cela une analogie avec la couverture d'un graphe biparti complet, et nous en déduisons une méthode de parcours permettant de tester toutes les possibilités susceptibles d'être optimales.

Arbre couvrant pour un arrangement donné. Considérons pour cela le graphe biparti complet $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ composé, d'un coté, de p sommets étiquetés par r_i, et de l'autre, de q sommets étiquetés par c_j. Une arête entre deux de ces sommets (r_i, c_j) est pondérée par t_{ij}.

Considérons un arbre couvrant $\mathcal{T} = (\mathcal{V}, \mathcal{E}')$ de ce graphe. Sachant que $r_1 = 1$, au moyen d'un parcours de cet arbre démarrant en r_1, on peut ainsi construire une solution au problème, en forçant la condition

$$\forall (r_i, c_j) \in \mathcal{E}', \;\; r_i t_{i,j} c_j = 1.$$

Une telle solution n'est *acceptable* que si toutes les autres inégalités sont vérifiées ($\forall (r_i, c_j) \in \mathcal{E}, \;\; r_i t_{i,j} c_j \leq 1$). Dans ce cas, on dit que *l'arbre couvrant est acceptable*. La valeur d'un arbre couvrant acceptable est $(\sum r_i)(\sum c_j)$.

arbre couvrant associé à une solution optimale. Le but ici, est de montrer qu'un arbre couvrant de valeur maximale, fournit une solution au problème d'optimisation Obj_2. En d'autres termes, pour toute solution au problème d'optimisation Obj_2 est associé un arbre couvrant. En conséquence, un arbre couvrant étant constitué de $p + q - 1$ arêtes, pour toute solution optimale au moins $p + q - 1$ des inégalités $r_i c_i t_{i,j} \leq 1$ sont atteintes.

Considérons une solution optimale au problème Obj_2. Considérons le sous-graphe biparti associé : $\mathcal{U} = (\mathcal{V}, \mathcal{E}')$ contenant une arête entre r_i et c_j ($(r_i, c_j) \in \mathcal{E}'$) si et seulement si $r_i c_j t_{i,j} = 1$. Montrons que ce sous-graphe est connexe. Dans ce cas, tout arbre couvrant de ce sous-graphe sera acceptable, ce qui prouve alors que la construction de l'ensemble des arbres couvrants acceptables permet d'atteindre toute solution optimale.

Raisonnons par l'absurde, supposons ce sous-graphe non connexe.

Considérons donc (sans perte de généralité) que $\mathcal{V}' = \{r_1, \ldots, r_{p'}, c_1, \ldots, c_{q'}\}$ soit une partie connexe de $\mathcal{U}$ et que $p' < p$.

Supposons, dans un premier temps, que $q' = q$. Dans ce cas, pour tout j, on a $r_{p'+1} c_j t_{p'+1,j} < 1$. On pourrait donc augmenter $r_{p'+1}$ d'un facteur $\alpha = \min_{1 \leq j \leq q} \frac{1}{r_{p'+1} c_j t_{p'+1,j}} > 1$ et ainsi construire une solution strictement meilleure, puisque le produit $\sum_{1 \leq i \leq p} r_i \sum_{1 \leq j \leq q} c_j$ est augmenté. Ce qui est absurde, la solution initiale étant supposée optimale.

Ainsi $q' < q$ et $p' < p$. On a donc pour tout $i > p'$ et $j \leq q'$, $r_i c_j t_{i,j} < 1$. De même pour tout $i \leq p'$ et $j > q'$. On pose alors :

$$\left\{ \begin{array}{lcl} \alpha_r & = & \min_{(i > p' \text{ et } j \leq q')} \frac{1}{r_i c_j t_{i,j}} \\ \alpha_c & = & \min_{(i \leq p' \text{ et } j > q')} \frac{1}{r_i c_j t_{i,j}} \end{array} \right.$$

et

$$\left\{ \begin{array}{lcl} R_a & = & \sum_{i \leq p'} r_i \\ R_b & = & \sum_{i > p'} r_i \\ C_a & = & \sum_{j \leq q'} c_j \\ C_b & = & \sum_{j > q'} c_j \end{array} \right.$$

L'idée est soit d'augmenter les éléments correspondant à R_b d'un facteur α_r tout en diminuant en contrepartie les éléments correspondant à C_b pour maintenir la solution acceptable, soit d'augmenter les éléments correspondant à C_b d'un facteur α_c en diminuant les éléments correspondant à R_b d'un facteur $\frac{1}{\alpha_c}$. Comme nous allons le voir, l'une au moins de ces deux solutions augmente le produit $\sum_{1 \leq i \leq p} r_i \sum_{1 \leq j \leq q} c_j$, ce qui contredit l'hypothèse d'optimalité de la solution initiale.

Considérons pour cela la fonction f de $\mathbb{R}_+^*$ dans $\mathbb{R}_+^*$ définie par $f(\lambda) = (\lambda R_b + R_a)(\frac{C_b}{\lambda} + C_a)$. Cette fonction est continue, strictement décroissante puis strictement croissante. Ainsi,

– si $f'(1) \geq 0$, alors pour tout $\lambda > 1$, $f(\lambda) > f(1)$. En particulier, $f(\alpha_r) > f(1)$.
– si $f'(1) \leq 0$, alors pour tout $\lambda \leq 1$, $f(\lambda) > f(1)$. En particulier, $f(\frac{1}{\alpha_c}) > f(1)$.

■

Algorithme de construction des arbres couvrants. Le principe de l'algorithme est le suivant :
– on construit récursivement l'ensemble des arbres couvrants de racine r_1.
– au fur et à mesure de la construction, on évalue les valeurs des noeuds r_i et c_j recouverts par l'arbre (à l'aide de la contrainte $r_i c_j t_{ij} = 1$).
– on interrompt la construction d'un arbre si le sous-arbre déjà construit n'est pas acceptable (on peut fortement réduire le coût de l'algorithme en essayant de prévoir si un sous-arbre peut aboutir à un arbre couvrant acceptable ou pas).
– lorsqu'un arbre couvrant acceptable est construit, on l'évalue. On garde celui de valeur maximale.

7.3.5 Solution heuristique pour pq processeurs sur une grille $p \times q$.

Dans ce paragraphe, nous présentons une heuristique polynomiale permettant de construire un arrangement "proche" d'une grille parfaite, dont les grandes étapes sont les suivantes :
– on démarre par un arrangement croissant, et on construit une grille parfaite proche de cet arrangement.
– à cette grille parfaite correspond une allocation de données. On construit alors un nouvel arrangement plus adapté à cette allocation.
– on réitère le processus jusqu'à convergence, c'est à dire jusqu' à ce qu'une itération ne modifie plus l'arrangement.

Le nombre d'itérations nécessaires à la convergence du processus n'est pas prouvé. En ce sens, nous n'avons aucune garantie théorique sur la complexité de cette solution, si ce n'est que l'on peut toujours arrêter le processus quand on veut. Le paragraphe suivant fournit une pseudo justification à l'aide d'une génération aléatoire d'ensembles de processeurs.

Arrangement croissant initial. L'idée est de partir d'un arrangement assez régulier. Nous choisissons pour cela un ordre lexicographique. En d'autres termes,

$$t_{i,j} \leq t_{i',j'} \Leftrightarrow (i,j) <_{lex} (i',j').$$

Par exemple, dans le cas de 9 processeurs de temps de cycle $(1, 2, \ldots, 9)$, l'arrangement initial considéré est

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \text{ de manière équivalente } S = \begin{pmatrix} 1 & 0.5 & 0.333 \\ 0.25 & 0.2 & 0.167 \\ 0.143 & 0.125 & 0.111 \end{pmatrix} \quad (7.1)$$

Recherche d'une grille parfaite proche de T. La décomposition SVD [48] fournit la meilleure approximation (pour la norme 2) d'une matrice par une matrice de rang 1. Ainsi, la matrice des vitesses $S = (\frac{1}{t_{i,j}})_{i,j}$ se décompose en valeurs singulières par $U\Sigma V^t = S$. Notons par s la plus grande valeur singulière, et par a et b les vecteurs singuliers associés : $S^{opt} = sab^t$ est alors la matrice de rang 1 la plus proche de S au sens de la norme 2 : c'est la matrice S plutôt que la matrice T qui est considérée, car cette approximation favorisant les grandes composantes, on

favorise ainsi les processeurs les plus rapides. Ainsi, si l'on pose $r = sa$ et $c = b$, on peut espérer que $\forall i, j,\ r_i c_j \simeq s_{i,j}$ c'est à dire $\forall i, j,\ r_i t_{i,,j} c_j \simeq 1$. Comme il faut que $\forall i, j,\ r_i t_i i, j c_j \leq 1$, on divise alors c par $\max_{i,j} r_i t_{i,j} c_j$.

Considérons à nouveau l'exemple 7.1. L'allocation obtenue au terme de cette première étape est donnée par $r = \begin{pmatrix} 1.1661 \\ 0.3675 \\ 0.2100 \end{pmatrix}$ et $c = \begin{pmatrix} 0.6803 \\ 0.4288 \\ 0.2859 \end{pmatrix}$. Dans ce cas, $T_{exe} = rTc^t = (r_i t_{i,j} c_j)_{i,j} = \begin{pmatrix} 0.7933 & 1 & 1 \\ 1 & 0.7879 & 0.6303 \\ 1 & 0.7203 & 0.5402 \end{pmatrix}$.
La valeur de cette allocation est $(\sum r_i)(\sum c_j) = 2.4322$.

Notons que cette allocation peut être aisément améliorée en rendant connexe le sous-graphe $\mathcal{U}$ de cette solution comme défini dans le Paragraphe 7.3.4.

Réarrangement des processeurs. Dans ce paragraphe, nous proposons une méthode permettant de construire un nouvel arrangement des processeurs, de telle manière que la grille ainsi construite soit plus proche d'une grille parfaite que la grille initiale. A partir de la matrice correspondant à la grille parfaite précédemment définie $T^{opt} = (\frac{1}{r_i\ c_j})_{i,j}$, on construit le nouvel arrangement de la manière suivante :

$$\forall i, j, k, l, \quad t_{i,j} \leq t_{k,l} \Longleftrightarrow t^{opt}_{i,j} \leq t^{opt}_{k,l}.$$

En d'autres termes, T^{opt} correspond à la grille de processeurs optimale, c'est à dire équilibrant parfaitement la charge, pour l'allocation (r, c) donnée. Etant donné notre ensemble de processeurs de temps de cycle fixés, on a intuitivement envie de les arranger dans le même ordre que cette grille optimale de processeurs.

Dans le cadre de notre exemple, avec les valeurs de r et c précédemment établies, on obtient

$$T^{opt} = \begin{pmatrix} 1.2606 & 2.0000 & 3.0000 \\ 4.0000 & 6.3464 & 9.5195 \\ 7.0000 & 11.1061 & 16.6592 \end{pmatrix}.$$

On trie alors en conséquence les éléments de la matrice T qui devient

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 7 \\ 6 & 8 & 9 \end{pmatrix}, \text{ de manière équivalente } S = \begin{pmatrix} 1 & 0.5 & 0.333 \\ 0.25 & 0.167 & 0.125 \\ 0.2 & 0.143 & 0.111 \end{pmatrix}$$

et la valeur de l'allocation associée est $(\sum r_i)(\sum c_j) = 2.5065$ (au lieu de 2.4322 initialement).

Réitération du processus. Puis on réitère le procédé décrit ci-dessus :

1. Approximation de S par $S^{opt} = sab^t$.
2. On pose $r = sa$ et $c = b$.
3. On pose $T^{opt} = (\frac{1}{r_i\ c_j})_{i,j}$ la grille de processeurs parfaite optimale associée à cette allocation.
4. On réordonne la grille de processeurs dans le même ordre que T^{opt}.

On considère qu'il y a convergence lorsque la matrice T n'est plus modifiée par le processus. Encore une fois, nous n'avons aucune garantie sur la convergence du processus.

Pour notre exemple, la convergence est atteinte au bout de trois étapes. La valeur de l'allocation obtenue est alors 2.5889 et l'arrangement correspondant

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 6 & 8 \\ 5 & 7 & 9 \end{pmatrix}.$$

7.3.6 Cas général : P quelconque

Jusqu'à présent, nous n'avons considéré que le cas particulier où p et q étaient fixés, et le nombre de processeurs valait $P = pq$.

En fait, ces solutions permettent de résoudre le problème de manière générale. Il suffit pour cela de considérer l'existence de processeurs de vitesse nulle ou de temps de cycle infini. Illustrons cela par deux exemples :

Premier exemple. Soit l'ensemble de $P = 9$ processeurs de vitesses données par la liste

$$(362, 357, 357, 305, 250, 134, 287, 284, 128)$$

Ces vitesses correspondent à l'ensemble de processeurs utilisés pour les mesures de performances présentées dans le Paragraphe 7.4.2 Page 142. Supposons que l'on souhaite agencer ces processeurs en une grille bidimensionnelle de hauteur $2 \leq p' \leq 3$ et de largeur $3 \leq q' \leq 4$, car pour des raisons de scalabilité, la grille ne doit pas être trop allongée. Notons la symétrie horizontal/vertical du problème théorique ; en pratique, les communications verticales étant légèrement plus coûteuses que les communications horizontales, on préfère une grille aplatie (2×4) à élancée (4×2). Posons donc $p = 3$, $q = 4$ et $P = 12$, et considérons la liste de vitesses suivante

$$(362, 357, 357, 305, 250, 134, 287, 284, 128, 0, 0, 0)$$

L'heuristique nous fournit alors la solution :

$$S = \begin{pmatrix} 362 & 357 & 250 & 0 \\ 357 & 305 & 134 & 0 \\ 287 & 283 & 128 & 0 \end{pmatrix} \text{ et } \begin{cases} r = (18.2, 15.5, 14.4)^t \\ c = (19.9, 19.6, 8.6, 0)^t \end{cases}$$

En d'autres termes, l'ensemble des 9 processeurs initiaux doivent être agencés en une grille de taille 3×3.

Second exemple. Supposons à présent que le processeur le plus lent de l'exemple précédent ait une vitesse non plus de 128, mais de 50. Dans ce cas, on obtient la solution

$$S = \begin{pmatrix} 362 & 357 & 357 & 250 \\ 305 & 287 & 284 & 134 \\ 50 & 0 & 0 & 0 \end{pmatrix} \text{ et } \begin{cases} r = (26.2, 21.1, 0)^t \\ c = (13.8, 13.6, 13.5, 6.4)^t \end{cases}$$

Le processeur le plus lent, ne doit pas être utilisé, et l'ensemble des 8 processeurs restants doivent être agencés en une grille de taille 2×4.

7.4 Evaluation de performances de la solution heuristique

Dans cette partie, nous évaluons expérimentalement la complexité de notre heuristique (présentée dans le Paragraphe 7.3.5) ainsi que l'équilibrage de charge qu'elle permet d'atteindre. Finalement, nous présentons des courbes de performance des algorithmes de multiplication de matrices et de décomposition LU et QR, obtenues sur un réseau de stations de travail.

7.4.1 Evaluation de l'heuristique

Nous n'avons aucune garantie théorique sur l'heuristique proposée dans le Paragraphe 7.3.5. En particulier, il nous faut évaluer à la fois sa complexité, ainsi que la valeur de l'allocation fournie. Pour cela, nous générons aléatoirement des ensembles de p^2 processeurs de temps de cycle compris entre 0 et 1 que nous cherchons à arranger sur une grille de taille $p \times p$.

Evaluation de la solution. L'évaluation d'une solution donnée au problème d'optimisation Obj_2 peut être donnée par la fonction

$$C(r_1, r_2, \ldots, r_p, c_1, \ldots, c_p) = \frac{(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j)}{\sum_{i,j} \frac{1}{t_{ij}}}.$$

En effet, durant une unité de temps,
- la quantité totale de travail effectué vaut $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j)$.
- la quantité de travail maximale que chaque processeur P_{ij} peut effectuer (cas optimal pas nécessairement atteignable) vaut $\frac{1}{t_{ij}}$.

Ainsi, la Figure 7.6 représente la valeur moyenne de C, après convergence de l'heuristique, pour une taille de problème (p) donnée.

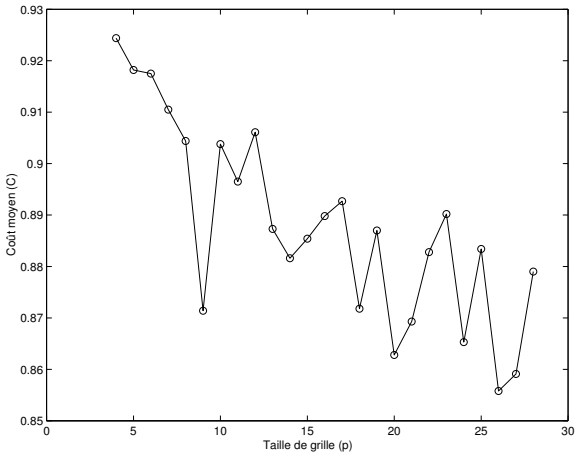


FIG. 7.6: *Evaluation du résultat. Comparaison avec la borne absolue.*

Comparaison avec une solution triviale. Il est intéressant d'évaluer le gain de notre heuristique par rapport à une solution triviale. Pour cela, nous comparons la solution obtenue après convergence de l'algorithme avec la solution prise initialement par l'algorithme (agencement lexicographique). On obtient la courbe représentée Figure 7.7 qui trace, en fonction de p, l'évolution

moyenne du gain

$$\tau = \frac{((\sum r_i)(\sum c_j))_{\text{après convergence}}}{((\sum r_i)(\sum c_j))_{\text{après la première étape}}} - 1.$$

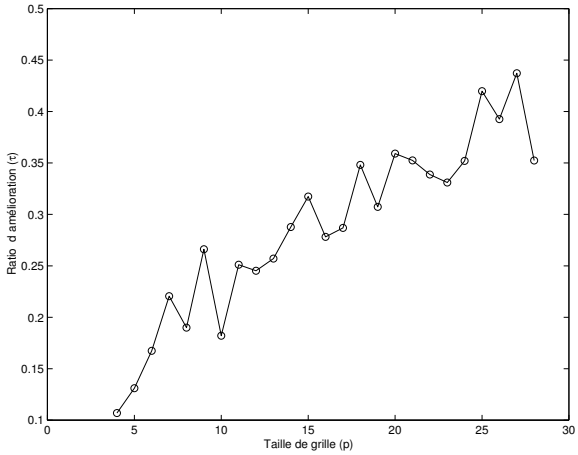


FIG. 7.7: *Evolution de τ.*

Complexité de l'heuristique. La complexité de l'heuristique dépend du nombre d'étapes nécessaires à la convergence. Ainsi, la Figure 7.8 schématise le nombre moyen d'étapes requises pour atteindre la convergence.

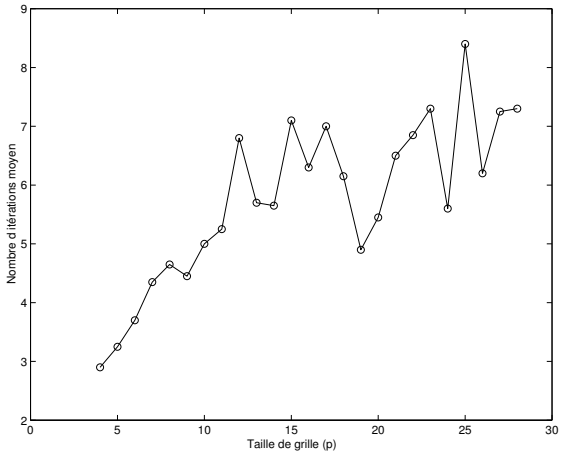


FIG. 7.8: *Nombre d'itérations nécessaire à la convergence.*

7.4.2 Expériences MPI

7.4.2.1 Evaluation de la vitesse des processeurs

Il existe trop de paramètres pour exprimer de manière précise la vitesse d'exécution d'un processeur sur un programme donné, même si sa charge ne varie pas au cours du temps. L'appellation "temps de cycle", doit être comprise comme *temps de cycle normalisé* [32], c'est à dire le temps

Nom	Mhz	Multiplication de matrices 500x500 (Mflops)
texas	507	362
alabama	507	357
mississipi	498	357
onyx	431	305
cotnari	351	250
cuervo	200	134
muddy-waters	402	287
pink-floyd	402	284
petit-gris	200	128

TAB. 7.1: *Description des 9 processeurs utilisés pour les mesures de performance.*

d'exécution élémentaire pour une application donnée, évalué à l'aide de mesures effectuées sur des problèmes de petite taille (répétés plusieurs fois, en prenant une valeur moyenne).

Nous avons ainsi mesuré la vitesse d'exécution d'un produit de matrices de taille 500×500 en arithmétique double précision, sur chaque processeur. Bien qu'abusif, nous avons gardé ce rapport de vitesses pour les décompositions LU et QR. En fait, nous avons tenu à garder une allocation identique des données pour les différents algorithmes. En effet, dans le cadre d'une librairie, il peut être souhaitable que la distribution des données soit identique tout au long du programme. Les processeurs sont des Intel Pentium, sur lesquels nous avons utilisé la librairie ATLAS [90] qui permet de générer automatiquement un appel efficace aux BLAS pour chaque architecture particulière. Par exemple, sur un Pentium III 500 Mhz, ATLAS 3.0 atteint une performance de 362 Mflops pour la multiplication de matrices de taille 500. La mémoire cache de haut niveau est vidée entre chaque mesure de temps, et nous avons utilisé la routine wallclock du système d'exploitation. La Table 7.1 décrit les processeurs utilisés dans nos expériences, leurs performances maximales ainsi que leurs performances mesurées à l'aide de notre routine de test.

7.4.2.2 Résultats numériques

Dans ce paragraphe, nous comparons les performances obtenues à l'aide de la distribution bloc-cyclique de ScaLAPACK avec celles obtenues à l'aide de l'allocation 2D que nous proposons. Nous présentons les résultats pour les algorithmes de produit de matrices et de décomposition QR, exécutés sur les 9 processeurs décrits Table 7.1. La vitesse du processeur le plus lent est de 128 Mflops, alors que celle du processeur le plus rapide est de 362 Mflops. Le rapport $\frac{362}{128} = 2.8$ montre que l'ensemble de processeurs choisi est raisonnablement hétérogène : en fait cela correspond à des machines dont l'age diffère d'au plus deux ans.

Pour cet ensemble de processeurs, l'heuristique décrite dans le Paragraphe 7.3.5 fournit la distribution de la Figure 7.9. De manière théorique, cette allocation nous permet d'exécuter $(\sum_i r_i) \times (\sum_j c_j) = 2318.44$ éléments en une unité de temps. Avec une distribution cyclique, chaque processeur étant ralenti à la vitesse du plus lent, nous pourrions atteindre seulement une vitesse de $9 \times 128 = 1152$ éléments par unité de temps. Le facteur d'amélioration théorique (sans tenir compte des communications) est donc de $\frac{2318.44}{1152} = 2.01$.

Les mesures effectuées pour différentes tailles de matrices, variant de 500 à 4000, sont rapportées figures 7.11 et 7.12. Dans chacune d'elles, quatre courbes sont représentées, correspondant respectivement (i) à une distribution hétérogène unidimensionnelle (ii) à une distribution hétérogène

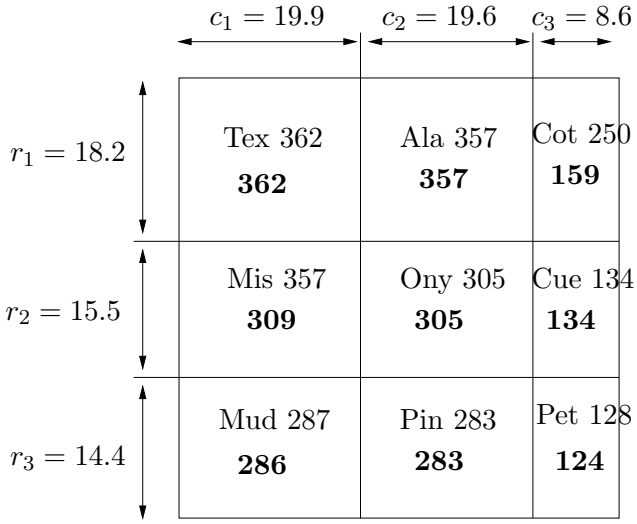


FIG. 7.9: *Distribution hétérogène. Les valeurs écrites en caractères gras correspondent à la quantité de travail alloué effectivement aux processeurs.*

bidimensionnelle (iii) à une distribution bloc-cyclique bidimensionnelle de ScaLAPACK et (iv) à une estimation théorique de notre distribution hétérogène bidimensionnelle. Cette estimation est obtenue à l'aide des résultats de la distribution bloc-cyclique multipliés par le rapport d'amélioration théorique 2.01. Pour chacune des deux expériences, la distribution bidimensionnelle hétérogène est meilleure que la distribution unidimensionnelle hétérogène, elle-même meilleure que la distribution bidimensionnelle bloc-cyclique.

En contrepartie, nous pouvons remarquer que la courbe théorique n'est pas atteinte. Cela est dû au fait, que les communications qui ont ici (réseau interne au laboratoire) un impact important sur le temps d'exécution, ne sont pas prises en compte dans notre approximation. Bien sûr, le volume de communication évoluant en $O(n^2)$ (pour une taille de matrice de $n \times n$) et le volume de calcul évoluant en $O(n^3)$, cet impact diminue lorsque la taille des matrices augmente. C'est en effet ce que l'on observe Figure 7.10.

Pour finir, remarquons qu'une allocation unidimensionnelle donne un temps d'exécution supérieur à une distribution bidimensionnelle. En effet, malgré le fait qu'une allocation unidimensionnelle permette un meilleur équilibrage de charge, son manque de scalabilité rend avec 9 processeurs, l'impact des communications prédominant.

7.5 Conclusion

Dans ce chapitre, nous avons proposé une allocation statique des données pour l'implémentation de noyaux de calcul d'algèbre linéaire tels que la multiplication de matrices et les décomposition LU et QR. Pour des raisons de scalabilité, nous avons traité le cas d'allocation 2D. Ceci nous a conduit à considérer le problème NP-complet d'arrangement en une grille non singulière parfaite, d'un ensemble de processeurs hétérogène. Nous avons proposé une solution heuristique dont nous avons montré l'efficacité à l'aide d'expériences et de mesures de performances.

Ces résultats peuvent être généralisés au cas d'un réseau hétérogène de profondeur supérieure à 1 [18] : l'hétérogénéité des communications dans le cas d'un cluster de clusters (etc.) peut être prise

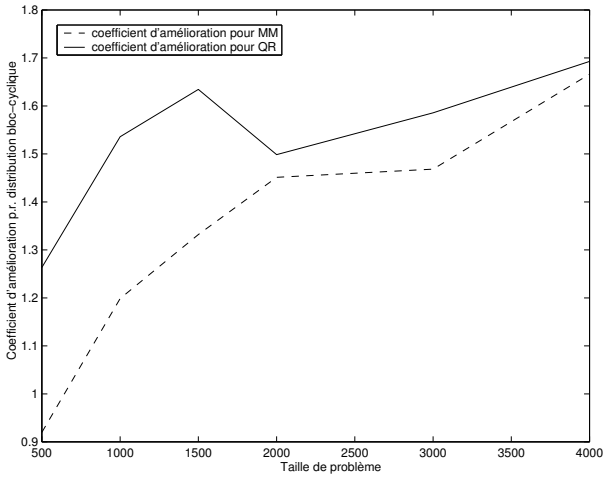


FIG. 7.10: *Evolution du coefficient d'amélioration par rapport à une distribution cyclique pour la multiplication de matrices ainsi que la décomposition QR. La borne théorique (sans communication) vaux* 2.01.

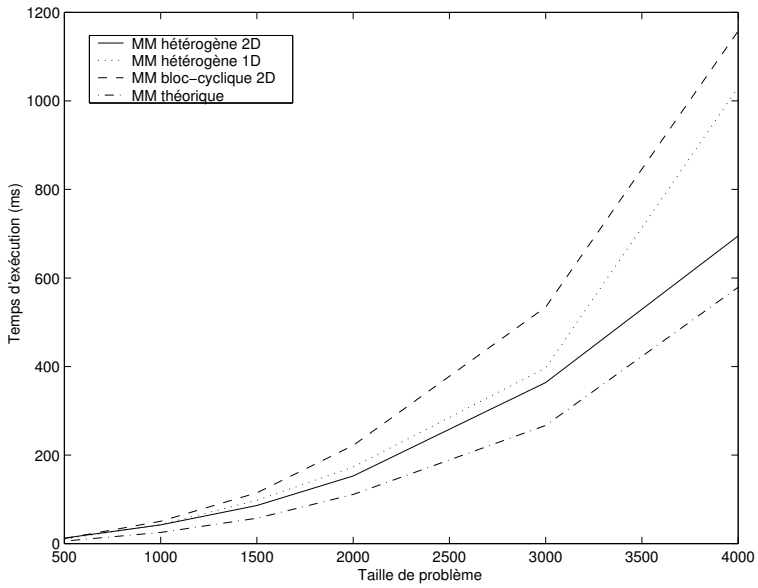


FIG. 7.11: *Mesures du temps d'exécution du produit de matrices pour différentes allocations.*

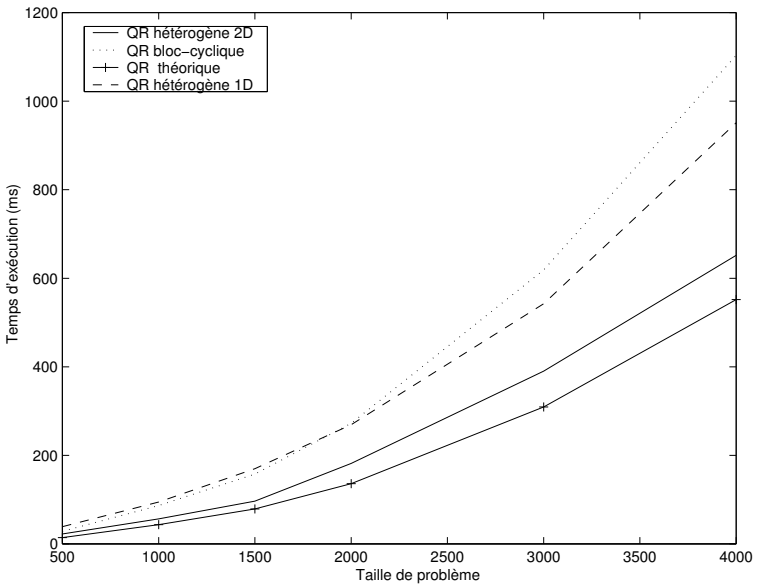


FIG. 7.12: *Mesures du temps d'exécution de la décomposition QR pour différentes allocations.*

en compte en considérant le système d'un point de vue hiérarchique. La construction du motif est alors légèrement plus complexe [16], et plusieurs choix se posent. Il faudrait dans ce cas comparer expérimentalement les différentes approches, et proposer une solution générale à notre problème statique.

Rappelons que l'étude menée dans ce chapitre est restreinte à la recherche d'une allocation statique des données. Mais comme nous avons pu le noter dans le chapitre précédant, une approche purement dynamique souffre, pour les applications présentées ici, de la présence de dépendances de données : les processeurs risquent d'être ralentis au rythme du processeur le plus lent. En contrepartie, dans le contexte de l'utilisation de ressources hétérogènes, l'ensemble des processeurs peut ne pas être dédié à un seul utilisateur. D'autant plus qu'une telle situation est favorable à une exécution distribuée, car elle fournit un rapport intéressant temps de calcul/temps de communication. En fait, nous pensons que dans un tel contexte, une approche semi-statique est louable : l'idée est de redistribuer les données et les calculs, de temps en temps, entre chaque phase statique correctement identifiée. En revanche, nous ne sommes pas persuadés qu'une allocation aussi contrainte (en grille) soit adaptée à une stratégie mixte. C'est pourquoi nous proposons, dans le chapitre suivant, une solution à notre problème d'équilibrage de charge à l'aide d'une allocation plus libre. Il sera intéressant dans l'avenir de comparer expérimentalement ces deux approches.

Chapitre 8

Partitionnement libre d'une matrice : équilibrage de charge et minimisation des communications.

8.1 Introduction

Dans le cadre d'implémentation de noyaux d'algèbre linéaire sur plateforme hétérogène, nous avons, dans le chapitre précédent, cherché à agencer un ensemble de processeurs en une grille parfaite. Une telle configuration, par sa régularité, permet une implémentation très efficace des algorithmes. En particulier, les communications peuvent être optimisées, et du recouvrement calculs/communications est possible. Dans ce cadre, le but est d'équilibrer les charges *au mieux*. Dans cette partie, nous autorisant plus de libertés sur l'allocation des données, nous pouvons équilibrer *parfaitement* les charges de calcul (allouer une surface s_i proportionnelle à la vitesse du processeur P_i). Nous cherchons donc, pour un équilibrage de charge parfait, à *minimiser le coût des communications*.

Selon le type de réseau interconnectant les différentes ressources de calcul, l'ensemble des communications pourront être séquentielles (réseau Ethernet), ou parallèles (réseau Myrinet). Considérons par exemple l'algorithme de multiplication de matrices. Comme dans de nombreux autres noyaux de calcul d'algèbre linéaire ou dans certaines applications numériques, la majeure partie des communications sont des diffusions horizontales et verticales (cf Chapitre 7 Page 122). Supposons que chaque matrice soit partitionnée en p rectangles correspondant à ce que chaque processeur doit mettre à jour durant les différentes étapes de l'algorithme. Alors, chaque processeur, reçoit à chaque diffusion verticale, une quantité de données proportionnelle à la largeur du rectangle qui lui est alloué. De même, à chaque diffusion horizontale, il reçoit une quantité de données proportionnelle à la hauteur de ce même rectangle. Les nombres respectifs de diffusions horizontales et verticales étant identiques, le volume de communications par rectangle (processeur) est proportionnel à son demi-périmètre. Ainsi, si les communications s'effectuent en parallèle, on cherchera à partitionner la matrice en minimisant le *maximum* des demi-périmètres des rectangles. Si les communications sont séquentielles, on cherchera à minimiser *la somme* des demi-périmètres.

Considérons, par exemple, l'algorithme de multiplication de matrice décrit Page 122 du Chapitre 7. Supposons que l'on détienne p processeurs de temps de cycle $t_1, \ldots, t_p$. Il s'agit alors de partitionner la matrice de taille $nb \times nb$ en p rectangles d'aires respectives $s_1, \ldots, s_p$ où $s_i t_i = \frac{(nb)^2}{\sum_{k=1}^{p} \frac{1}{t_k}} = C$. La Figure 8.1 décrit une phase de communication une fois le partitionnement ef-

fectué. Dans l'absolu, il faudrait, bien entendu, que pour chaque rectangle de forme $bh_i \times bv_i$, h_i et v_i soient entiers, ce qui en général n'est pas compatible avec la vérification de manière exacte de l'égalité $bh_i \times bv_i = \frac{C}{t_i}$. Mais comme nous l'avons déjà souligné dans les chapitres précédents, le temps de cycle d'un processeur n'est pas une mesure précise, et plutôt que de chercher à résoudre le problème pour une taille $nb \times nb$ de matrice donnée, nous préférons trouver une solution générique pour une matrice de taille 1×1 à l'aide de rectangles de longueurs de coté h et v définis dans $\mathbb{R}$. Pour les autres algorithmes d'algèbre linéaire tels que la décomposition LU et QR, la construction de l'allocation finale à partir de la solution générique est légèrement plus sophistiquée que pour l'algorithme de décomposition de matrice, mais les contraintes pour construire la solution générique initiale sont identiques.

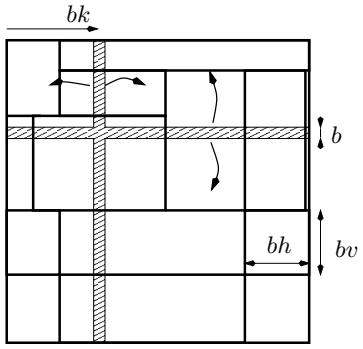


FIG. 8.1: *Partitionnement d'une matrice sur 13 processeurs hétérogènes. Phase de communication pour la k-ième étape de la multiplication de matrices : le k-ième bloc de b colonnes est diffusé horizontalement et le k-ième bloc de b lignes est diffusé verticalement.*

Ainsi, de manière purement géométrique, les deux problèmes d'optimisation s'expriment de la manière suivante : comment partitionner un carré unitaire en p rectangles d'aires fixées $s_1, s_2, \ldots, s_p$ (tels que $\sum_{i=1}^{p} s_i = 1$) afin de minimiser

- soit la somme des demi-périmètres des rectangles dans le cas de communications séquentielles,
- soit le plus grand des demi-périmètres des rectangles dans le cas de communications parallèles.

Remarquons qu'il est toujours possible de paver le carré en p bandes verticales de largeur $s_1, s_2, \ldots, s_p$. La difficulté du problème réside en la minimisation des fonctions objectives.

Considérons l'exemple avec $p = 5$ rectangles $R_1, \ldots, R_5$ d'aires $s_1 = 0.36$, $s_2 = 0.25$, $s_3 = s_4 = s_5 = 0.13$. Une partition possible est schématisée Figure 8.2. La taille de chaque rectangle est la suivante : $0.61 \times \frac{36}{61}$ pour R_1, $0.61 \times \frac{25}{61}$ pour R_2, et $0.39 \times \frac{1}{3}$ pour R_3, R_4, et R_5. Le demi-périmètre maximal est celui de R_1, approximativement 1.2002, très proche de la borne inférieure absolue $2\sqrt{0.36} = 1.2$ atteinte lorsque le plus grand rectangle est un carré (ce qui n'est pas possible dans cet exemple). En ce qui concerne la seconde fonction objective, la somme des demi-périmètres vaut 4.39, alors que la borne absolue inférieure vaut $\sum_{i=1}^{p} 2\sqrt{s_i} \approx 4.36$ (atteinte lorsque tous les rectangles sont des carrés, ce qui n'est encore pas possible dans cet exemple). Ainsi, la partition s'avère très satisfaisante pour chacune des deux fonctions objectives. L'interprétation géométrique de la somme des demi-périmètres est la suivante : cela correspond à la somme des longueurs des lignes tracées pour effectuer la partition, augmentée de 2 (correspondant à une bordure horizontale et à une bordure verticale du carré unitaire).

Les résultats principaux de ce chapitre sont les preuves de NP-Complétude au sens faible de ces deux problèmes ainsi que la description d'heuristiques garanties. Ainsi, nous explicitons formellement les différents problèmes d'optimisation PERI-SUM (minimisation de la somme des

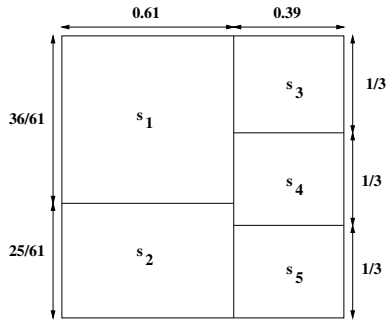


FIG. 8.2: *Un partitionnement du carré unitaire à l'aide de* 5 *rectangles.*

demi-périmètres) et PERI-MAX (minimisation du demi-périmètre maximum), et établissons leur NP-Complétude (Paragraphe 8.2 et Paragraphe 8.3). Puis nous proposons une heuristique garantie au problème d'optimisation PERI-SUM (Paragraphe 8.4) ainsi qu'au problème PERI-MAX (Paragraphe 8.5). Enfin, nous décrivons différents problèmes d'optimisation relatifs aux nôtres (Paragraphe 8.6) avant de conclure (Paragraphe 8.7).

8.2 NP-Complétude de PERI-SUM

Dans ce paragraphe, nous décrivons formellement le premier problème d'optimisation traité dans ce chapitre : PERI-SUM. Nous prouvons alors la NP-complétude du problème de décision associé.

Il nous faut paver le carré unitaire à l'aide de p rectangles R_i, d'aire s_i $(1 \leq i \leq p)$ où $\sum_{i=1}^{p} s_i = 1$. La forme de chaque R_i correspond aux degrés de liberté.

Définition 15 *PERI-SUM(s) : Soient p nombres réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition du carré unitaire en p rectangles R_i d'aire s_i et de forme $h_i \times v_i$, tels que $\hat{C} = \sum_{i=1}^{p}(h_i + v_i)$ soit minimisé.*

Il existe une borne inférieure triviale au problème PERI-SUM(s) :

Lemme 16 *Pour toute solution de PERI-SUM(s), $\hat{C} \geq 2\sum_{i=1}^{p} \sqrt{s_i}$.*

Preuve. Le demi-périmètre de chaque rectangle R_i sera toujours plus grand que lorsque c'est un carré, c'est à dire $2\sqrt{s_i}$. Bien sûr, paver un carré en p carrés d'aires s_i n'est pas nécessairement possible, et donc, la borne inférieure pour PERI-SUM(s) n'est pas nécessairement atteignable. ■

Le problème de décision associé au problème d'optimisation PERI-SUM est le suivant :

Définition 16 *PERI-SUM(s,K) : soient p nombres réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$ ainsi qu'une borne réelle positive K, existe-t-il une partition du carré unitaire en p rectangles R_i, d'aire s_i et de forme $h_i \times v_i$, telle que $\sum_{i=1}^{p}(h_i + v_i) \leq K$?*

Notre premier résultat établit la difficulté intrinsèque du problème d'optimisation PERI-SUM :

Théorème 15 *PERI-SUM(s,K) est NP-complet.*

Preuve. Les principales idées de la preuve sont les suivantes :
Réduction PERI-SUM à ASP

1. Dans un premier temps, nous effectuons la réduction de PERI-SUM(s,K) à un problème géométrique (ASP) dont le problème est de vérifier l'existence d'une partition du carré unitaire en carrés d'aires fixées.
2. Ensuite, nous prouvons la NP-Complétude de ASP à l'aide d'une réduction polynomiale au problème de 2-Partition-Equal qui lui même est NP-Complet [47]. Cette réduction est fondée sur les idées suivantes :

Réduction ASP à 2-Partition-Equal

1. Nous démarrons d'une instance arbitraire du problème de 2-Partition-Equal défini Page 130, c'est à dire d'un ensemble $\mathcal{A} = \{a_1, \ldots, a_n\}$ de n entiers, qu'il faut partitionner en deux sous-ensembles disjoints *de même cardinal* et de même somme.
2. L'idée est de construire une instance équivalente à ce problème à l'aide d'un ensemble $\mathcal{B} = \{b_1, \ldots, b_n\}$ de n entiers vérifiant $b > \frac{2}{3}\max b_i$. On définie simplement et polyniomalement pour cela $b_i = 2(a_i + 2nmax_k a_k)$. Sous certaines considérations techniques, nous montrons qu'il existe une solution au problème de 2-Partition-Equal initial si et seulement si $\mathcal{B}$ peut être partitionné en deux sous-ensembles de même somme, mais *pas nécessairement de même cardinal.*
3. Finalement, nous construisons à partir de $\mathcal{B}$ une instance de ASP à l'aide de trois types de carrés : de larges carrés dénotés par $A_{i,j}$ Figure 8.3, n carrés de taille $b_i \times b_i$ dénotés par A_{b_i} dans la Figure 8.4 et un nombre polynomial de carrés de remplissage dénotés par $A_{\overline{b_i},j}$ dans la Figure 8.4.
4. Nous montrons alors que la seule configuration possible est celle décrite Figure 8.3. Dans cette configuration, il y a deux zones rectangulaires d'aire $m \times S$ où $M = \frac{4}{3}\max b_i$ et $S = \sum_i \frac{b_i}{2}$, partitionnées comme schématisé Figure 8.3. Grâce à la condition $b_i > \frac{M}{2}$, nécessairement les carrés A_{b_i} d'aire $b_i \times b_i$ ne peuvent pas être superposés et doivent agencés les uns à coté des autres dans la direction S. Ainsi, pour chacune des deux zones, la somme des b_i qu'elle contient vaut S.
5. Intuitivement, les rectangles $A_{i,j}$ sont introduit afin de créer ces deux zones non adjacentes d'aire $M \times S$; les n carrés A_{b_i} doivent alors être alignés à l'intérieur de ces deux zones, et les carrés restant $A_{\overline{b_i},j}$ sont introduit afin de remplir les zones restantes.

Clairement, PERI-SUM(s,K) $\in$ NP. Nous utilisons la réduction suivante :

Lemme 17

$$\textit{2P-eq} \leq_P \textit{ASP} \leq_P \textit{PERI-SUM},$$

où 2P-eq est le problème de décision défini dans le Paragraphe 7.3.3 et ASP est défini comme suit :

Définition 17 *All-Squares-Partition (ASP)*
Soit un ensemble $\mathcal{L} = \{l_1, \ldots, l_p\}$ de p nombres réels positifs tels que $\sum_{i=1}^{p} l_i^2 = 1$, existe-t-il une partition du carré unitaire en p carrés S_i de largeur l_i ?

Comme 2P-eq est NP-complet, le Lemme 17 suffit à démontrer le Théorème 15.

8.2.1 Réduction : ASP $\leq_P$ PERI-SUM(s,K)

Nous commençons par la partie simple de la preuve du Lemme 17, c'est à dire ASP $\leq_P$ PERI-SUM(s,K). Considérons un ensemble $\mathcal{L} = \{l_1, \ldots, l_p\}$ constitué de p nombres réels positifs tels que $\sum_{i=1}^{p} l_i^2 = 1$. Résoudre ASP est équivalent à résoudre PERI-SUM(s,K) avec

$$\begin{cases} K = 2\sum_{i=1}^{p} l_i \\ \forall i,\ s_i = l_i^2 \end{cases}$$

et ainsi,

$$\text{ASP} \leq_P \text{PERI-SUM}.$$

8.2.2 Réduction : 2P-eq $\leq_P$ ASP

Dans ce paragraphe, nous considérons une instance arbitraire du problème de 2-Partition-Equal, c'est à dire la donnée d'un ensemble $\mathcal{A} = \{a_1, \ldots, a_n\}$ de n nombres entiers. Nous supposons, sans perte de généralité, que $n \geq 400$. Il nous faut transformer polynomialement cette instance en une instance du problème ASP ayant une solution si et seulement si l'instance originale du problème de 2-Partition-Equal a une solution.

Définissons pour cela $\{b_1, \ldots, b_n\}$ par $\forall i, b_i = 2(a_i + 2n \max_k a_k)$. Soit $N = \max_k b_k$ Ainsi, $b_i \geq \frac{2N}{3}$, et b_i est pair. De plus, si on note $M = \frac{4N}{3}$ et $S = \frac{\sum_i b_i}{2}$, alors $S \geq 100M$. On a aussi, $\frac{M}{2} < b_i < \frac{3M}{4}$ pour tout i. La raison pour laquelle nous introduisons M est de pouvoir paver les n rectangles de taille $b_i \times (M - b_i)$ en un nombre minimal de carrés $KS(i)$, à l'aide de la procédure de Kenyon [63]. Afin d'avoir un nombre polynomial de carrés $KS(i)$, le rectangle $\overline{R_i}$ ne doit pas être trop allongé, ce qui est assuré grâce à l'inégalité $M - b_i < b_i < 3(M - b_i)$. On obtient alors de [63] que $KS(i) \leq 3 + C \log b_i$, où C est une constante universelle. par la suite, nous dénotons par $w(\overline{b_i}, j)$ pour $1 \leq j \leq KS(i)$ la largeur des $KS(i)$ carrés obtenus par la procédure de [63] qui pave le rectangle $\overline{R_i}$ de taille $b_i \times (M - b_i)$.

Nous construisons ainsi l'instance du problème (mis à l'échelle d'un rapport $20S + 17M$) ASP suivante (noté abusivement ASP$(b_1, \ldots, b_n)$) : existe-t-il une partition du carré de taille $(20S + 17M) \times (20S + 17M)$ en $14 + n + \sum_k KS(i)$ carrés de largeur

$$\begin{cases} l_{13,11} &= (13S + 11M) & (\times 1), \\ l_{7,6} &= (7S + 6M) & (\times 3), \\ l_{4,3} &= (4S + 3M) & (\times 2), \\ l_{3,3} &= (3S + 3M) & (\times 2), \\ l_{3,2} &= (3S + 2M) & (\times 2), \\ l_{2,2} &= (2S + 2M) & (\times 4), \\ l_{bi} &= b_i & (\forall i, 1 \leq i \leq n), \\ l_{\overline{b_i},j} &= w(\overline{b_i}, j) & (\forall i,\ 1 \leq i \leq n,\ \forall j,\ 1 \leq j \leq KS(i))? \end{cases}$$

A partir de maintenant, $A_{x,y}$ représente un carré de largeur $l_{x,y} = (xS + yM)$, A_{b_i} un carré de largeur $l_{b_i} = b_i$ et $A_{\overline{b_i},j}$ représente le carré de Kenyon de largeur $l_{\overline{b_i},j} = w(\overline{b_i}, j)$.

Dans ce qui suit, nous allons montrer qu'une telle partition est nécessairement de la forme schématisée Figure 8.3, dans laquelle les deux petits rectangles $(M \times S)$ fléchés sont eux-même partitionnés en d'autres rectangles comme schématisé Figure 8.4. L'idée intuitive de la preuve est la suivante : les grands carrés sont utilisés afin de forcer les zones rectangles $M \times S$ à être séparées. Ainsi, ces deux surfaces doivent être remplies à l'aide des petits carrés restants d'aires b_i et des carrés de Kenyon. Ceci étant possible si et seulement si l'ensemble des b_i peut être partitionné

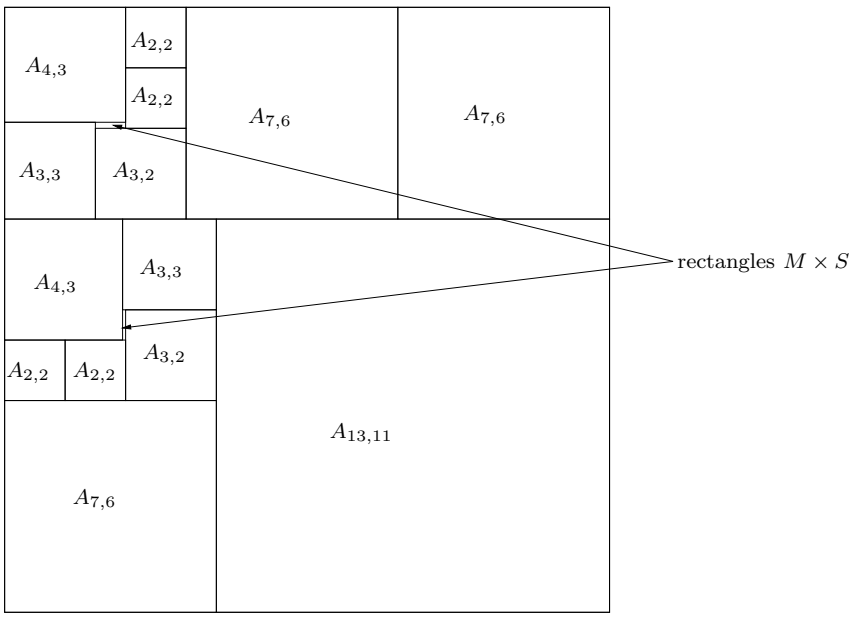


FIG. 8.3: *Positionnement des carrés.*

en deux sous-ensembles de même somme, ce qui est possible si et seulement si l'ensemble des a_i peut être partitionné en deux sous-ensembles de même cardinal et de même somme. Les carrés de Kenyon sont introduit afin de remplir les trous dans les deux zones rectangles et d'obtenir ainsi un pavage complet du carré initial.

Position des quatre plus grands carrés. La position générale des quatre plus grands carrés est donnée Figure 8.5a. Clairement, si on peut paver l'aire restante avec les carrés restants, il en est de même dans la configuration donnée Figure 8.5b. Ainsi, sans perte de généralité, nous considérons dès à présent que les quatre plus grands carrés sont positionnés comme montré Figure 8.5b.

Pavage de la surface restante. Analysons maintenant le pavage de la surface restante (partie non grisée de la Figure 8.5b). Toutes les dimensions sont fournies Figure 8.6. Une étude exhaustive décrite dans [12] montre que la seule configuration possible (d'autres solutions équivalentes sont possibles) est celle décrite dans la Figure 8.7.

Conclusion partielle. Ainsi, tout pavage de la surface restante (cf Figure 8.6) est similaire à celui représenté Figure 8.7 : après avoir placé tous les gros rectangles $A_{x,y}$, il reste deux rectangles non adjacents de forme $M \times S$ à paver. Ainsi, le problème ASP admet une solution si et seulement si il est possible de paver ces deux rectangles restants à l'aide de n carrés de largeur b_i $(1 \leq i \leq n)$, et $\sum_k KS(i)$ carrés de Kenyon de largeur $w(\overline{b_i}, j)$. Comme $\min_k b_k > \frac{M}{2}$ et $\sum_k b_k = 2S$, il n'est pas possible de superposer deux rectangles A_{b_i} et A_{b_j} pour $i \neq j$ l'un au dessus de l'autre. Comme $\sum_i b_i = 2S$ chaque zone rectangle $M \times S$ doit être pavée comme décrit Figure 8.4 et pour chacune des deux zones la somme des b_i vaut S. Ainsi, notre instance du problème ASP admet une solution si et seulement si il existe une partition de $\{b_1, \ldots, b_n\}$ en deux sous-ensembles de même somme S.

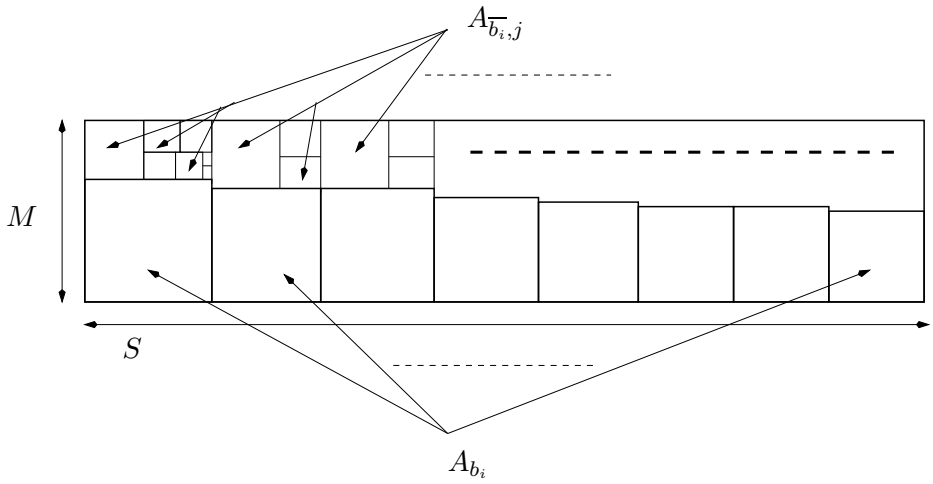


FIG. 8.4: *Description des rectangles $M \times S$.*

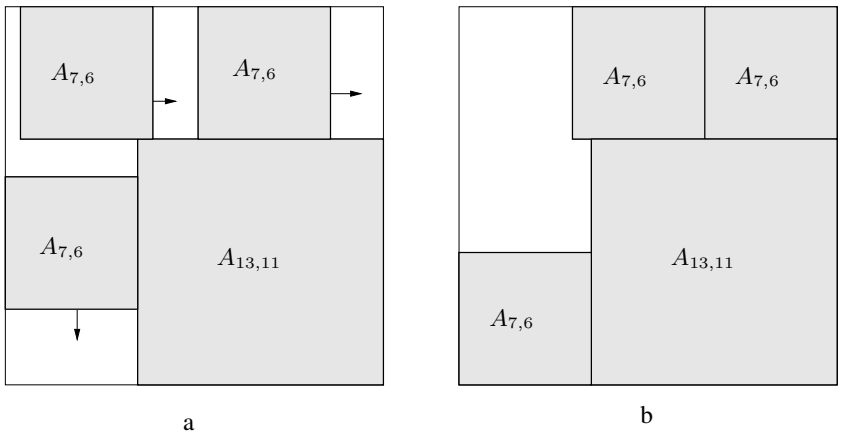


FIG. 8.5: *Position des quatre plus grands carrés.*

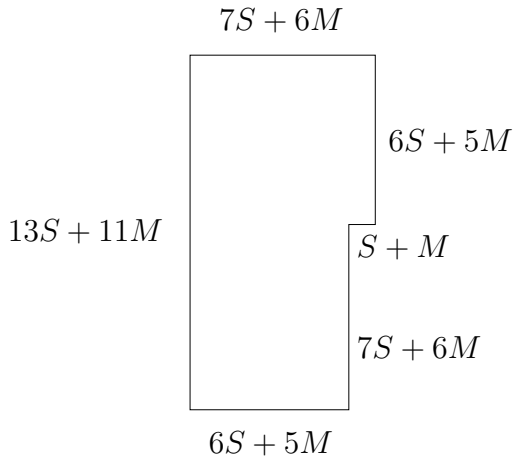


FIG. 8.6: *Surface restante.*

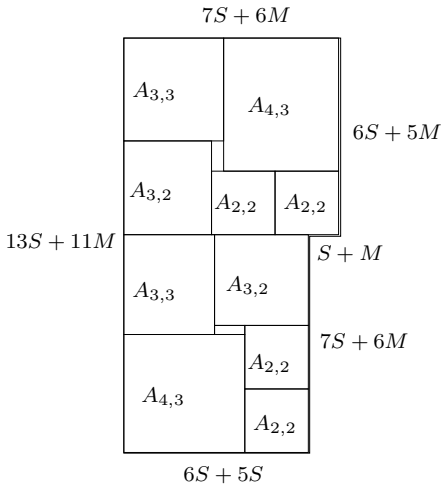


FIG. 8.7: *Pavage possible de la surface restante.*

Réduction finale. Pour compléter la réduction, il nous faut montrer qu'il existe une partition de l'ensemble $\{b_1, \ldots, b_n\}$ en deux sous-ensembles de même somme, si et seulement si l'instance originale du problème de 2P-eq admet une solution.

Supposons dans un premier temps que l'instance du problème de 2P-eq admet une solution, c'est à dire qu'il existe une partition de $\{1, \ldots, n\}$ en deux sous-ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ satisfaisant

$$\sum_{k \in \mathcal{A}_1} a_k = \sum_{k \in \mathcal{A}_2} a_k \text{ et } \text{card}(\mathcal{A}_1) = \text{card}(\mathcal{A}_2).$$

Rappelons que $b_k = 2(a_k + 2n \times \text{MAX})$, où $\text{MAX} = \max_k a_k$. Alors,

$$\begin{aligned} \sum_{k \in \mathcal{A}_1} b_k &= \sum_{k \in \mathcal{A}_1} a_k + 2n \times \text{MAX} \times \text{card}(\mathcal{A}_1) \\ &= \sum_{k \in \mathcal{A}_2} a_k + 2n \times \text{MAX} \times \text{card}(\mathcal{A}_2) \\ &= \sum_{k \in \mathcal{A}_2} b_k \end{aligned}$$

Il existe donc une partition acceptable de $\{b_1, \ldots, b_n\}$.

Réciproquement, supposons qu'il existe une partition de $\{1, \ldots, p\}$ en deux sous-ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ tels que

$$\sum_{k \in \mathcal{A}_1} b_k = \sum_{k \in \mathcal{A}_2} b_k.$$

Montrons que

$$\text{card}(\mathcal{A}_1) = \text{card}(\mathcal{A}_2)$$

On a,

$$\begin{aligned}\sum_{k\in\mathcal{A}_1} a_k &= \sum_{k\in\mathcal{A}_1} b_k - 2n \times \text{MAX} \times \text{card}(\mathcal{A}_1)\\ \sum_{k\in\mathcal{A}_2} a_k &= \sum_{k\in\mathcal{A}_2} b_k - 2n \times \text{MAX} \times \text{card}(\mathcal{A}_2)\\ \sum_{k\in\mathcal{A}_1} a_k - \sum_{k\in\mathcal{A}_2} a_k &= 2n \text{ MAX card}(\mathcal{A}_2 - \mathcal{A}_1)\end{aligned}$$

De plus, comme

$$\sum_{a_k\in\mathcal{A}_1} a_k - \sum_{a_k\in\mathcal{A}_2} a_k \leq n \text{ MAX},$$

on obtient

$$\text{card}(\mathcal{A}_1) = \text{card}(\mathcal{A}_2) \text{ et } \sum_{k\in\mathcal{A}_1} a_k = \sum_{k\in\mathcal{A}_2} a_k$$

En conséquence, l'instance d'origine du problème de 2P-eq admet une solution.

Consistance de la transformation. Le dernier point de la démonstration est la vérification de la consistance de la transformation : il nous faut prouver que l'expression de notre instance du problème ASP a une taille polynomiale en la taille de l'expression de l'instance du problème de 2P-eq d'origine.

Lemme 18 *Notons $MAX = \max_k a_k$, et représentons par $c(a)$ et $c(b)$ le codage des données a et b. Alors,*

$$\textit{Taille}(c(\mathcal{L})) = O(\textit{Taille}(c(\mathcal{A}))^3).$$

Preuve. Nous utilisons ici les notations classiques de comparaison asymptotique de fonctions O et Ω : on dit que $f(x) = O(g(x))$ s'il existe une constante c telle que $f(x) \leq cg(x)$ pour x suffisamment grand, et on dit que $f(x) = \Omega(g(x))$ s'il existe une constante c telle que $g(x) \leq cf(x)$ pour x suffisamment grand.

Pour l'encodage de l'instance initiale $\mathcal{A}$, on a $\text{Taille}(c(\mathcal{A})) = \Omega\left(\sum_i \log a_i\right)$. Comme

$$\sum_i \log a_i \geq \log \text{MAX} + (n-1)\log(\min_i a_i) \geq (n-1)\log 2 + \log \text{MAX},$$

on en déduit que

$$\text{Taille}(c(\mathcal{A})) = \Omega(n + \log \text{MAX}).$$

Pour l'encodage de l'instance ASP $\mathcal{L}$, on a

$$\begin{cases} \log b_i \leq \log((4n+2)\text{MAX}) = O(\log n + \log \text{MAX}),\\ \log M = O(\log n + \log \text{MAX}),\\ \log S = O(n(\log n + \log \text{MAX}),\\ \sum_i KS(i)\log b_i \leq \sum_i (3 + C\log b_i)\log b_i = O(n(\log n + \log \text{MAX})^2),\end{cases},$$

où C est la constante universelle donnée par Kenyon [63]. Ainsi,

$$\text{Taille}(c(\mathcal{L})) = O(\text{Taille}(c(\mathcal{A}))^3).$$

■

Ceci termine la preuve de NP-complétude du problème ASP, et achève donc la preuve de NP-complétude du problème PERI-SUM. ■

8.3 NP complétude de PERI-MAX(s)

Dans ce paragraphe, nous décrivons formellement le second problème d'optimisation traité dans ce chapitre : PERI-MAX. Nous prouvons alors la NP-complétude du problème de décision associé.

Il nous faut paver le carré unitaire à l'aide de p rectangles R_i, d'aire s_i $(1 \leq i \leq p)$ où $\sum_{i=1}^{p} s_i = 1$. La forme de chaque R_i correspond aux degrés de liberté.

Définition 18 *PERI-MAX(s) : Soient p nombres réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition du carré unitaire en p rectangles R_i d'aire s_i et de forme $h_i \times v_i$, tels que $\hat{M} = \max_{1 \leq i \leq p}(h_i + v_i)$ soit minimisé.*

Il existe une borne inférieure triviale au problème PERI-MAX(s) :

Lemme 19 *Pour toute solution de PERI-MAX(s), $\hat{M} \geq 2 \max_{1 \leq i \leq p} \sqrt{s_i}$.*

Preuve. Le demi-périmètre de chaque rectangle R_i sera toujours plus grand que lorsque c'est un carré, c'est à dire $2\sqrt{s_i}$. Bien sûr, paver un carré en p carrés d'aires s_i n'est pas nécessairement possible, et donc, la borne inférieure pour PERI-MAX(s) n'est pas nécessairement atteignable. ■

Le problème de décision associé au problème d'optimisation PERI-MAX est le suivant :

Définition 19 *PERI-MAX(s,K) : Soient p nombres réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$ ainsi qu'une borne réelle positive K, existe-t-il une partition du carré unitaire en p rectangles R_i, d'aire s_i et de forme $h_i \times v_i$, tels que $\max_{1 \leq i \leq p}(h_i + v_i) \leq K$?*

Théorème 16 *PERI-MAX(s,K) est NP-complet.*

Preuve. Nous donnons ici les principales lignes de la démonstration (cf [12]) pour une description plus précise), basée sur la réduction suivante :

Lemme 20

$$\textit{2P-0-4} \leq_P \textit{MSP} \leq_P \textit{PERI-MAX},$$

où MSP et 2P-0-4 sont définis comme suit :

Définition 20 *2-Partition-0-4 (2P-0-4)*
Soit un ensemble de $p+2$ *entiers* $\mathcal{A} = \{a_1, \ldots, a_p, a_{p+1} = 2, a_{p+2} = 2\}$ *tels que* $(\forall i \leq p, \quad a_i > 4$ *et* $a_i = 0 \bmod 4)$*, existe-t-il une partition de* $\{1, \ldots, p+2\}$ *en deux sous-ensembles* $\mathcal{A}_1$ *et* $\mathcal{A}_2$ *telle que*

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i \text{ ou } \sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i + 4 \ ?$$

Définition 21 *Max-Square-Partition (MSP)*
Soit un ensemble $\mathcal{A} = \{s_1, \ldots, s_p\}$ *de* p *nombres réels positifs tels que* $\sum_{i=1}^{p} s_i = 1$ *et* $s_1 \geq s_2 \geq \ldots \geq s_p$*, existe-t-il une partition du carré unitaire en* p *rectangles* R_i*, d'aire* s_i*, telle que* R_1 *soit un carré, et que les demi-périmètres des autres rectangles ne dépassent pas* $h_1 + w_1 = 2\sqrt{s_1}$ *?*

Comme 2P-0-4 est NP-complet (réduction triviale de 2-Partition [47]), le Lemme 20 prouve le Théorème 16.

8.3.1 Réduction : MSP $\leq_P$ PERI-MAX(s,K)

Débutons par la partie la plus facile de la preuve du Lemme 20, c'est à dire MSP $\leq_P$ PERI-MAX(s,K). Soit $\mathcal{A} = \{s_1, \ldots, s_p\}$ un ensemble de p nombres réels positifs tels que $\sum_{i=1}^{p} s_i = 1$ et $s_1 \geq s_2 \geq \ldots \geq s_p$. Résoudre MSP est équivalent à résoudre PERI-MAX(s,K) avec

$$K = 2\sqrt{s_1}$$

et alors,

$$\text{MSP} \leq_P \text{PERI-MAX}.$$

Il faut noter que la présence de la racine carrée dans le problème de décision n'est pas gênante car les deux membres de l'inégalité $\max_{1 \leq i \leq p}(h_i + v_i) \leq K$ peuvent être élevés au carré et la vérification peut ainsi être effectuée en temps polynomial.

8.3.2 Réduction : 2P-0-4 $\leq_P$ MSP

Dans ce paragraphe, nous considérons une instance arbitraire du problème de 2-Partition-0-4, c'est à dire la donnée d'un ensemble $\mathcal{A} = \{a_1, \ldots, a_n, a_{n+1} = 2, a_{n+2} = 2\}$ tel que $(\forall i \leq p, \quad a_i > 4$ et $a_i = 0 \bmod 4)$. Il nous faut transformer polynomialement cette instance en une instance du problème MSP ayant une solution si et seulement si l'instance d'origine du problème 2-Partition-0-4 admet une solution. Soit

$$S = \frac{\sum_{1 \leq i \leq n} a_i}{4}.$$

Nous construisons l'instance suivante (mise à l'échelle) du problème MSP (MSP$(a_1, \ldots, a_n)$) : existe-t-il une partition du carré de taille $(S+2) \times (S+2)$ en $(n+3)$ rectangles $R_S, R_1, \ldots, R_{n+2}$ d'aire

$$\begin{cases} R_S : & A_S & = & S^2, \\ R_i : & A_i & = & a_i \quad (\forall i, 1 \leq i \leq n), \\ R_{n+1} : & A_{n+1} & = & 2, \\ R_{n+2} : & A_{n+2} & = & 2, \end{cases}$$

où le rectangle R_S d'aire A_S est un carré et le demi-périmètre des autres rectangles R_i est inférieur à $2S$?

La position du plus grand carré est décrite Figure 8.8. Les différentes zones qui l'entourent, $\Sigma_{rem}, \Sigma_0, \Sigma_1$, sont aussi décrites Figure 8.8. Du fait que l'aire de la surface Σ_{rem} vaut 4, nous pouvons aisément prouver le lemme suivant :

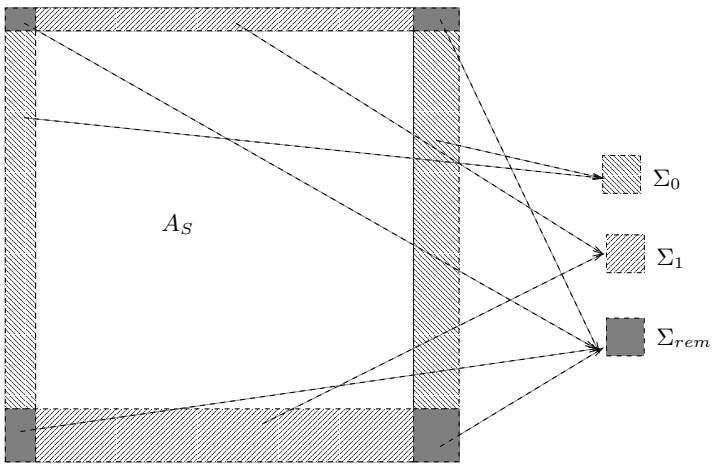


FIG. 8.8: *Position du plus gros carré.*

Lemme 21 *Soient*
- S_0 *: l'ensemble des rectangles ayant une intersection d'aire* non nulle *avec la surface* Σ_0.
- S_1 *: l'ensemble des rectangles ayant une intersection d'aire* non nulle *avec la surface* Σ_1.
- S_{rem} *: l'ensemble des rectangles restants.*

Alors, $\forall i \leq n$, R_i *appartient soit à* S_0 *soit à* S_1 *(*$a_i > 4$*). De plus, nous avons* $|\sum_{R_i \in S_0} A_i - \sum_{R_i \in S_1} A_i| \leq 4$, *c'est à dire* $|\sum_{R_i \in S_0} A_i - \sum_{R_i \in S_1} A_i| \in \{0, 2, 4\}$.

Sans perte de généralité, supposons que $(\sum_{R_i \in S_0} A_i \geq \sum_{R_i \in S_1} A_i)$. Il faut alors considérer trois cas différents en fonction de la valeur de $(\sum_{R_i \in S_0} A_i - \sum_{R_i \in S_1} A_i)$:

- $(\sum_{R_i \in S_0} A_i - \sum_{R_i \in S_1} A_i) = 0$. Dans ce cas, soit $\exists i$, $R_{n+1} \in S_i$ et $R_{n+2} \in S_{1-i}$, ou bien à la fois R_{n+1} et R_{n+2} appartiennent à S_{rem}. Dans le premier cas, (S_0, S_1) représente une partition de $\mathcal{A} = \{a_1, \ldots, a_n, a_{n+1} = 2, a_{n+2} = 2\}$ en deux sous-ensembles de sommes égales. Dans le second cas, $(S_0 \bigcup R_{n+1}, S_1 \bigcup R_{n+2})$ représente une partition de $\mathcal{A}$ en deux sous-ensembles de même somme.
- $(\sum_{R_i \in S_0} A_i - \sum_{R_i \in S_1} A_i) = 2$. Dans ce cas, exactement un des deux rectangles R_{n+1} ou R_{n+2} appartient à S_0 ou S_1, et l'autre appartient à S_{rem}. Supposons, sans perte de généralité, que $R_{n+1} \in S_i$. Alors $(S_0, S_1 \bigcup R_{n+2})$ représente une partition de $\mathcal{A}$ en deux sous-ensembles de même somme.
- $(\sum_{R_i \in S_0} A_i - \sum_{R_i \in S_1} A_i) = 4$. A nouveau dans ce cas, soit $\exists i$, $R_{n+1} \in S_i$ et $R_{n+2} \in S_{1-i}$, soit à la fois R_{n+1} et R_{n+2} appartiennent à S_{rem}. Dans le premier cas, (S_0, S_1) représente une partition de $\mathcal{A}$ en deux sous-ensembles dont les sommes diffèrent de 4. Dans le second cas, S_0 et $S_1 \bigcup R_{n+1} \bigcup R_{n+2}$ représente une partition de $\mathcal{A}$ en deux sous-ensembles de même somme.

Ainsi, si le problème MSP a une solution, alors il existe une partition de $\mathcal{A} = \{a_1, \ldots, a_n, a_{n+1} = 2, a_{n+2} = 2\}$ en deux sous-ensembles dont la somme diffère de 0 ou de 4. Afin de compléter la réduction, il nous faut montrer la réciproque.

- Supposons qu'il existe une partition de $\{1, \ldots, n+2\}$ en deux sous-ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ tels que
$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i + 4.$$
Soit $S_0 = \bigcup_{i \in \mathcal{A}_1} \{R_i\}$ où R_i représente un rectangle de taille $2 \times \frac{a_i}{2}$, et $S_1 = \bigcup_{i \in \mathcal{A}_2} \{R_i\}$, où R_i représente un rectangle de taille $\frac{a_i}{2} \times 2$. Nous pavons alors l'aire $(S+2) \times (S+2)$ comme

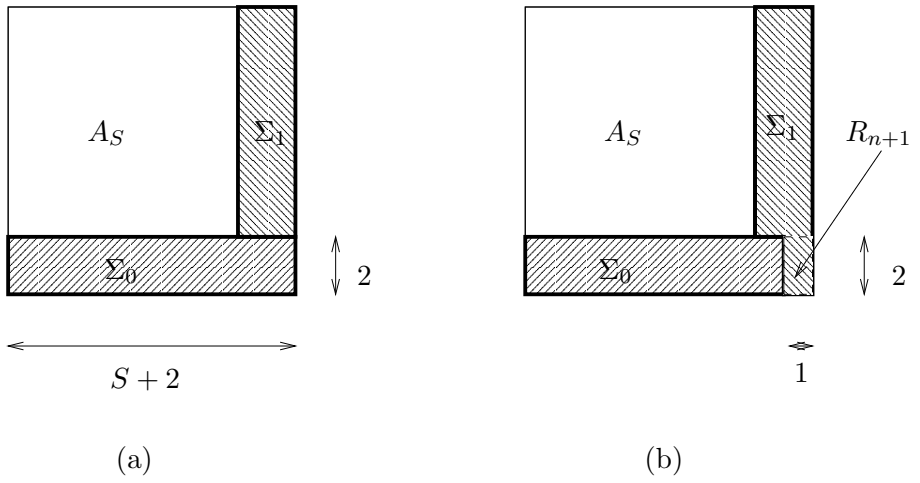


FIG. 8.9: *Pavage du carré à partir de l'instance du problème MSP.*

indiqué Figure 8.9a. Comme il est possible de paver à la fois Σ_0 et Σ_1 (avec les éléments de S_0 et S_1), il est donc possible de résoudre le problème MSP.

– Supposons qu'il existe une partition de $\{1, \ldots, n+2\}$ en deux sous-ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ tels que

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i.$$

Soit $S_0 = \bigcup_{i \in \mathcal{A}_1} \{R_i\}$ où R_i représente un rectangle de taille $2 \times \frac{a_i}{2}$, et $S_1 = \bigcup_{i \in \mathcal{A}_2} \{R_i\}$ où R_i représente un rectangle de taille $\frac{a_i}{2} \times 2$. Supposons, sans perte de généralité que R_{n+1} appartient à S_1. Nous pavons alors l'aire $(S+2) \times (S+2)$ comme indiqué Figure 8.9b. Comme il est possible de paver à la fois Σ_0 et Σ_1 (avec les éléments de S_0 et S_1), il est donc possible de résoudre le problème MSP.

Ainsi, notre instance du problème MSP admet une solution si et seulement si il existe une partition de $\mathcal{A} = \{a_1, \ldots, a_n, a_{n+1} = 2, a_{n+2} = 2\}$ en deux sous-ensembles dont les sommes diffèrent de 0 ou de 4. Cela achève la preuve de NP-complétude de MSP et par là même de PERI-MAX.

■

8.4 Heuristiques garanties pour PERI-SUM

Il existe plusieurs solutions heuristiques possibles au problème PERI-SUM. Mais en général, prouver une borne garantie sur ces heuristiques, s'avère délicat. Nous débutons dans le Paragraphe 8.4.1 par la description d'un partitionnement par colonnes. La complexité de cet algorithme est faible et l'approximation semble expérimentalement efficace. Malheureusement la garantie *théorique* que nous sommes parvenu à obtenir n'est pas très bonne : la formule présentée dans le Paragraphe 8.4.1.3 dépend de la taille relative des rectangles utilisés pour le pavage. Ainsi, nous proposons dans le Paragraphe 8.4.2 une solution définie récursivement, plus complexe à décrire, mais pour laquelle nous sommes capables de fournir une borne théorique convenable.

8.4.1 Heuristique par colonnes

Comme PERI-SUM(s) est NP-complet, nous considérons le problème plus contraint COL-PERI-SUM(s) dans lequel nous imposons le fait que le pavage soit constitué de colonnes de processeurs, comme décrit Figure 8.10. En d'autres termes, COL-PERI-SUM(s) est la restriction de PERI-SUM(s) aux partitionnements par colonnes. Dans ce paragraphe, nous proposons une solution optimale en un temps polynomial ($O(p\sqrt{p}\log p)$) à ce problème. Notons que ce problème a déjà été abordé par Kadourra et al. [60], mais qu'ils ne fournissent qu'une solution heuristique.

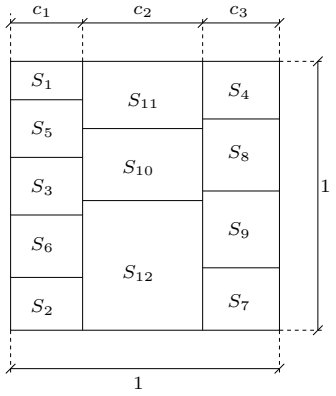


FIG. 8.10: *Partitionnement par colonnes du carré unitaire : le nombre de colonnes vaut $\mathcal{C} = 3$; la première colonne est composée de $k_1 = 5$ rectangles, la seconde de $k_2 = 3$ rectangles et la troisième de $k_3 = 4$ rectangles.*

8.4.1.1 Description

Donnons une description plus formelle du problème COL-PERI-SUM(s) : soit $s_1, \ldots, s_p$ p nombres réels positifs tels que $\sum_i s_i = 1$. Nous cherchons à paver le carré unitaire en $\mathcal{C}$ colonnes (où $\mathcal{C}$ est à déterminer) de largeur $c_1, \ldots, c_{\mathcal{C}}$. Chaque colonne C_i est elle-même partitionnée en k_i lignes (à déterminer aussi) d'aires $s_{\sigma(i,1)}, \ldots, s_{\sigma(i,k_i)}$. Bien entendu, le partitionnement final est composé de $\sum_{i=1}^{\mathcal{C}} k_i = p$ rectangles, et chaque aire $s_1, \ldots, s_p$ est représentée une fois et une seule. Le but est de construire un tel pavage qui minimise la somme des demi-périmètres des rectangles.

Algorithme. L'algorithme, fondé sur la technique de programmation dynamique, utilise les idées suivantes :

1. Renumérotation des variables $s_1, \ldots, s_p$ telle que $s_1 \leq s_2 \leq \ldots \leq s_p$.
2. Construction itérative des fonctions $f_{\mathcal{C}}^{perimeter}$, où $\mathcal{C}$ est incrémenté de 1 à la valeur désirée. Pour $q \in \{1, \ldots, p\}$, $f_{\mathcal{C}}^{perimeter}(q)$ représente le périmètre total du partitionnement optimal par $\mathcal{C}$ colonnes et q rectangles d'aires respectives $s_1, \ldots, s_q$, du rectangle de hauteur 1 et de largeur $(\sum_{i=1}^{q} s_i)$.

Afin de comprendre le principe, nous appliquons l'algorithme sur l'exemple suivant : soient $p = 8$ aires de valeurs (0.02, 0.04, 0.06, 0.08, 0.2, 0.2, 0.2, 0.2). Le résultat de l'algorithme est décrit dans la Table 8.1. Chaque colonne $\mathcal{C}_i$ contribue à la valeur de la somme des demi-périmètres comme suit : la ligne verticale compte 1, et l'ensemble des k_i lignes horizontales de largeur c_i compte $k_i \times c_i$.

Dans cet exemple, le partitionnement optimal est obtenu avec 3 colonnes ($f_3(8) = 5.4$). La dernière colonne de largeur $c_3 = s_7 + s_8 = 0.4$ est composée de deux éléments. La seconde, de

	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8
$\mathcal{C}=1$	1.02 | 0	1.12 | 0	1.36 | 0	**1.8** | **0**	3 | 0	4.6 | 0	6.6 | 0	9 | 0
$\mathcal{C}=2$		2.06 | 1	2.18 | 2	2.4 | 2	2.92 | 3	**3.6** | **4**	4.6 | 4	5.8 | 5
$\mathcal{C}=3$			3.12 | 2	3.26 | 3	3.6 | 4	4.12 | 5	4.72 | 5	**5.4** | **6**
$\mathcal{C}=4$				4.2 | 3	4.46 | 4	4.8 | 5	5.32 | 6	5.92 | 7
$\mathcal{C}=5$					5.4 | 4	5.66 | 5	6 | 6	6.52 | 7
$\mathcal{C}=6$						6.6 | 5	6.86 | 6	7.2 | 7
$\mathcal{C}=7$							7.8 | 6	8.06 | 7
$\mathcal{C}=8$								9 | 7

TAB. 8.1: *Tableau contenant les valeurs de* $f_{\mathcal{C}}^{perimeter}(q)$ *et a séparés par* $|$. $f_{\mathcal{C}}^{perimeter}(q) = \min_{a\in[\mathcal{C}-1,q-1]}\Big(1+\big(\sum_{a<i\leq q} s_i\big)\times(q-a)+f_{\mathcal{C}-1}^{perimeter}(a)\Big)$ *et a correspond à la valeur minimisant l'expression ci-dessus. Les valeurs écrites en caractères gras correspondent à la solution optimale : pour* $\mathcal{C}=3$, $f_{\mathcal{C}}^{cut}(8)=6$, *ainsi les deux premières colonnes contiennent 6 éléments ; pour* $\mathcal{C}=2$, $f_{\mathcal{C}}^{cut}(6)=4$, *et la première colonne contient 4 éléments.*

largeur $c_2 = s_5 + s_6 = 0.4$ est aussi composée de deux éléments. Finalement, la dernière, de largeur $c_1 = s_1 + s_2 + s_3 + s_4 = 0.2$ est composée des 4 plus petits éléments. Le partitionnement optimal est représenté Figure 8.11.

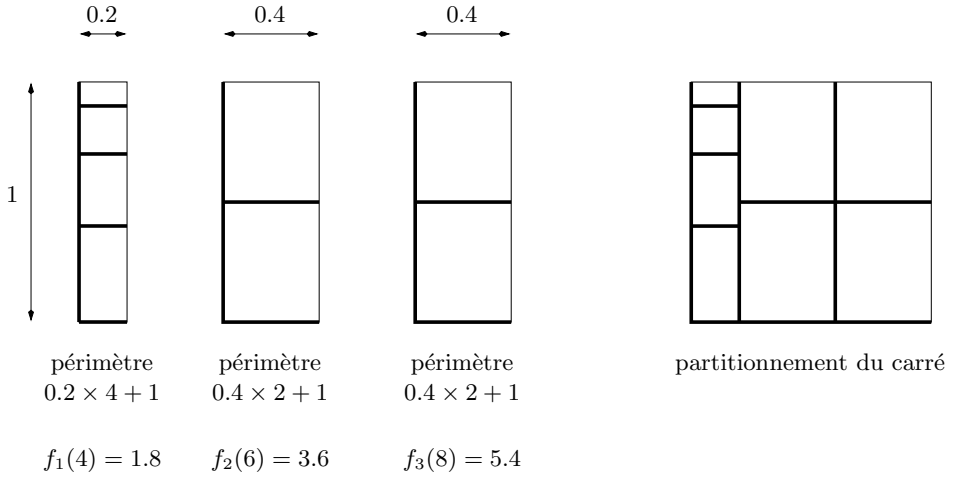


FIG. 8.11: *Partitionnement par colonnes optimal. Les lignes plus épaisses correspondent aux lignes comptabilisées dans la somme des demi-périmètres.*

L'algorithme est le suivant :

Algorithme 12. *Construction des fonctions $f_{\mathcal{C}}^{perimeter}$ ($f_{\mathcal{C}}^{cut}(q)$ correspond au nombre de blocs utilisés dans les $\mathcal{C}-1$ premières colonnes, ce qui laisse $q - f_{\mathcal{C}}^{cut}(q)$ dans la colonne $\mathcal{C}$).*

$S = 0$
for $q = 1, p$
  $S = S + s_q$
  $f_1^{perimeter}(q) = 1 + S \times q$
  $f_1^{cut}(q) = 0$
for $\mathcal{C} = 2, p$
  for $q = \mathcal{C}, p$
    $f_{\mathcal{C}}^{perimeter}(q) = \min_{a \in [\mathcal{C}-1, q-1]} \left(1 + \sum_{q-a<i\leq q} s_i(q-r) + f_{\mathcal{C}-1}^{perimeter}(a)\right)$
    $f_{\mathcal{C}}^{cut}(q) = q - a_{opt}$ {Où a_{opt} atteint le minimum dans l'expression ci-dessus}

Dans le pire des cas, la complexité de l'algorithme est de $O(p^3)$. Il faut noter qu'en pratique, la complexité est plus petite : en effet, la fonction $f_{\mathcal{C}}^{perimeter}(p)$ est une fonction de $\mathcal{C}$ décroissante puis croissante. Toutes les fonctions $f_{\mathcal{C}}^{perimeter}$ ne seront donc pas construites et l'on peut envisager un coût de $p^2\mathcal{C}_{opt} \approx p^{2.5}$.

Le partitionnement final correspondant à la fonction $f_{\mathcal{C}_{opt}}^{perimeter}(p) = \min_{1\leq\mathcal{C}\leq p} f_{\mathcal{C}}^{perimeter}(p)$ est trouvé à l'aide de l'algorithme suivant :

Algorithme 13. *Reconstruction de la solution optimale à partir des fonctions $f_{\mathcal{C}}^{cut}$*

$q = p$
for $\mathcal{C} = \mathcal{C}_{opt}, 2 : -1$
  $k_{\mathcal{C}} = q - f_{\mathcal{C}}^{cut}(q)$
  $q = f_{\mathcal{C}}^{cut}(q)$
$k_1 = q$

Cet algorithme correspond à parcourir en sens inverse le Tableau 8.1 en passant par les valeurs représentées en caractères gras. Le carré unitaire est partitionné en $\mathcal{C}_{opt}$ colonnes. La i-ième colonne contient les rectangles $s_{d_i}, s_{d_i+1}, \ldots, s_{d_i+k_i}$ avec $d_i = k_1 + k_2 + \ldots + k_{i-1}$.

Optimalité. Les algorithmes 12 et 13 fournissent la solution optimale du problème COL-PERI-SUM. La seule difficulté consiste à montrer que l'on peut réduire la recherche à des séquences ordonnées $s_1 \leq s_2 \ldots \leq s_p$:

Définition 22 (partitionnement bien-ordonné) *Un partitionnement est dit bien-ordonné si pour toute paire de colonnes $\mathcal{C}_i$ et $\mathcal{C}_j$, soit tous les éléments de $\mathcal{C}_i$ sont plus petits ou égaux à tous les éléments de $\mathcal{C}_j$, soit l'inverse.*

La Figure 8.12 illustre cette définition.

Considérons donc un partitionnement composé de $\mathcal{C}$ colonnes de tailles $k_1 \geq k_2 \geq \ldots k_{\mathcal{C}}$. Supposons les s_i indexés de telle manière que $s_1 \leq s_2 \leq \ldots \leq s_p$; soit τ une permutation de $\{1, 2, \ldots, p\}$ telle que la i-ième colonne du partitionnement contienne les rectangles $s_{\tau(d_i+1)}, \ldots, s_{\tau(d_i+k_i)}$ où $d_i = k_1+k_2+\ldots+k_{i-1}$. Maintenant, rappelons que le coût d'une colonne C_i est $1+k_i \sum_{j=d_i+1}^{d_i+k_{i-1}} s_{\tau(j)}$.

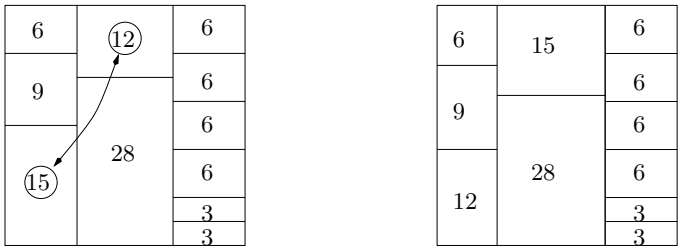


FIG. 8.12: *Deux partitionnements de là-même instance. Celle de droite est bien ordonnée, celle de gauche ne l'est pas.*

Ainsi, le demi-périmètre total vaut

$$\begin{aligned}\mathcal{C} \quad &+ \quad k_1 s_{\tau(1)} + k_1 s_{\tau(2)} + \ldots + k_1 s_{\tau(k_1)} \\ &+ \quad k_2 s_{\tau(k_1+1)} + k_2 s_{\tau(k_1+2)} + \ldots + k_2 s_{\tau(k_1+k_2)} \\ &+ \quad \ldots \\ &+ \quad k_{\mathcal{C}} s_{\tau(k_1+\ldots+k_{\mathcal{C}-1}+1)} + k_{\mathcal{C}} s_{\tau(k_1+\ldots+k_{\mathcal{C}-1}+2)} + \ldots + k_{\mathcal{C}} s_{\tau(k_1+\ldots+k_{\mathcal{C}})}\end{aligned}$$

Comme $k_1 \geq k_2 \geq \ldots \geq k_{\mathcal{C}}$, cette expression est minimisée pour $\tau =$ Identité. Ce qui correspond à un partitionnement bien-ordonné. La preuve est obtenue par récurrence sur le nombre d'inversions dans la permutation τ. Ainsi, pour tout partitionnement, il existe un partitionnement bien-ordonné correspondant meilleur ou équivalent.

8.4.1.2 Comparaison expérimentale avec la borne inférieure absolue

Comme montré dans le Paragraphe 8.2, la borne inférieure absolue de la somme $\hat{C}$ des demi-périmètres est deux fois la somme des racines carrées des aires, c'est à dire $LB = 2\sum_{i=1}^{p}\sqrt{s_i}$. Bien entendu, cette borne n'est pas nécessairement atteignable : considérons par exemple une instance du problème de PERI-SUM(s) composée seulement de deux rectangles, $s_1 = 1 - \epsilon$ et $s_2 = \epsilon$, où $\epsilon > 0$ est arbitrairement petit. Le partitionnement du carré en ces deux rectangles nécessite de tracer une ligne de longueur 1, d'où $\hat{C} = 3$. Or, $LB = 2(\sqrt{1-\epsilon} + \sqrt{\epsilon}) > 2$ tend vers 2 quand ϵ tend vers 0.

Dans cette partie, nous comparons expérimentalement, en générant aléatoirement des ensembles de surfaces, la valeur de $\hat{C}$ donnée par notre partitionnement, à la borne inférieure absolue LB. Du fait qu'une répartition uniforme des surfaces ne permet pas d'atteindre les pires cas, nous utilisons plutôt une répartition exponentielle des surfaces à laquelle on rajoute une forte probabilité d'avoir des surfaces égales. Ensuite, comme le rapport $r = \frac{\max s_i}{\min s_i}$ joue un rôle important dans l'évaluation du coût du pire des cas, nous avons effectué nos tests pour différentes valeurs de r variant de 2 à l'infini. Les résultats sont rapportés Figure 8.13.

Notons qu'en pratique, il est peu probable que l'on soit amené à utiliser *sur le même réseau* des processeurs de vitesses très différentes. Ainsi, $r = 20$ semble être un cas assez limite, et l'allocation par colonnes semble donc très satisfaisante dans la majeure partie des cas, d'autant que cette configuration assez simple permet d'envisager différentes stratégies de redistribution pour une allocation semi-statique.

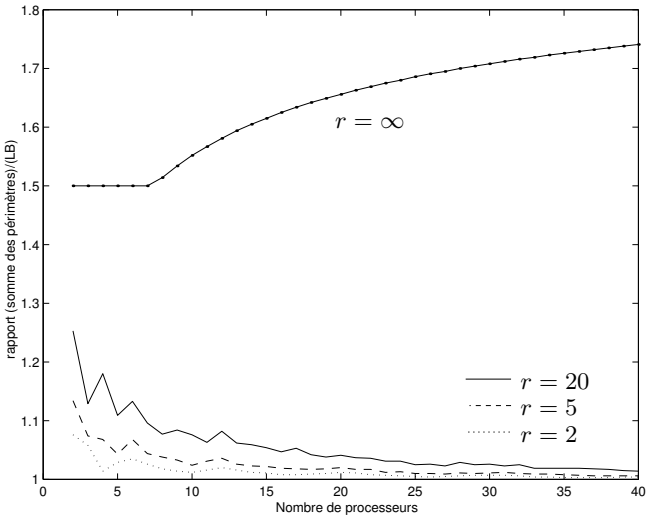


FIG. 8.13: *Pour chaque nombre de processeurs variant entre 2 et 40, et pour différentes valeurs de* $r = \frac{\max s_i}{\min s_i}$*, 10000 ensembles de* s_i *ont été générés aléatoirement à l'aide d'une distribution exponentielle avec une forte probabilité d'avoir des valeurs égales. Dans chaque cas, nous avons calculé le rapport de la somme* $\hat{C}$ *des demi-périmètres de notre partitionnement sur la borne inférieure absolue* LB*. Les pires des cas sont retenus, et ainsi les valeurs maximales de ces rapports sont reportés dans chacune des quatre courbes.*

8.4.1.3 Comparaison théorique avec la borne inférieure

Dans ce paragraphe, nous montrons que le partitionnement par colonnes fournit une bonne approximation, surtout lorsque le rapport entre la plus grande aire $\max s_i$ et la plus petite aire $\min s_i$ est faible.

Théorème 17 *Soit* $r = \frac{\max s_i}{\min s_i}$*. Notons* $\hat{C}$ *la somme des demi-périmètres des rectangles du partitionnement par colonnes optimal, et* $LB = 2\sum_{i=1}^{p} \sqrt{s_i}$*. Alors,*

$$\frac{\hat{C}}{LB} \leq \sqrt{r}\left(1 + \frac{1}{\sqrt{p}}\right) = \sqrt{\frac{\max s_i}{\min s_i}}\left(1 + \frac{1}{\sqrt{p}}\right)$$

Si $r = 1$, c'est à dire si tous les processeurs ont la même vitesse, le partitionnement par colonnes est asymptotiquement optimal. En contrepartie, lorsque r est grand, la borne est très pessimiste.

Preuve. Considérons le partitionnement par colonnes constitué de $\mathcal{C} = \lceil \sqrt{r} \sum \sqrt{s_i} \rceil$ colonnes. Les rectangles sont également distribués sur les colonnes de telle manière que le nombre de rectangles dans chaque colonne est soit $\lfloor \frac{p}{\mathcal{C}} \rfloor$ soit $\lceil \frac{p}{\mathcal{C}} \rceil$. Notons $\hat{C}^*$ la somme des demi-périmètres de ce partitionnement, on a :

$$\begin{aligned} \hat{C}^* &\leq \lceil \sqrt{r} \sum \sqrt{s_i} \rceil + \lceil \frac{p}{\lceil \sqrt{r} \sum \sqrt{s_i} \rceil} \rceil \\ &\leq 2 + \sqrt{r} \sum \sqrt{s_i} + \frac{p}{\sqrt{r} \sum \sqrt{s_i}} \end{aligned}$$

Ainsi,

$$\frac{\hat{C}^*}{2\sum \sqrt{s_i}} \leq \frac{1}{\sum \sqrt{s_i}} + \frac{\sqrt{r}}{2} + \frac{p}{2\sqrt{r} \sum \sqrt{s_i}}.$$

De plus,

$$\begin{aligned} \sum s_i = 1 &\implies p \max s_i \geq 1 \\ &\implies \min s_i \geq \frac{1}{pr} \end{aligned}$$

et ainsi,

$$\sum \sqrt{s_i} \geq p\sqrt{\min s_i} \geq \sqrt{\frac{p}{r}}.$$

Soit finalement,

$$\begin{aligned} \frac{\hat{C}^*}{2\sum \sqrt{s_i}} &\leq \sqrt{\frac{r}{p}} + \frac{\sqrt{r}}{2} + \frac{\sqrt{r}}{2} \\ &\leq \sqrt{r}(1 + \frac{1}{\sqrt{p}}). \end{aligned}$$

Comme $\widehat{C}$ correspond à la meilleure solution parmi toutes les solutions possibles par colonnes, $\hat{C}$ vérifie $\hat{C} \leq \hat{C}^*$, ce qui achève la preuve. ■

8.4.2 Heuristique récursive

La motivation de ce paragraphe est purement théorique et provient de notre défaite à *prouver* théoriquement une borne convenable pour notre partitionnement par colonnes : nous proposons ici une heuristique définie récursivement, fournissant un bon facteur de garantie. Au premier niveau de récursion, le partitionnement considéré est un partitionnement en guillotine, c'est à dire que le carré est scindé en deux parties, la partie de droite est elle même scindée en deux rectangles, etc. Le second niveau de récursion de l'algorithme ainsi que les suivants sont fondés sur un partitionnement par colonnes. L'idée ici est d'imposer à tout niveau de la récursion, aux différents rectangles de ne pas être trop allongés : grâce à cela, les rectangles finaux ayant une forme proche d'un carré, la borne est assuré. C'est pour cela que nous développons ici un algorithme de partitionnement par colonnes différent de celui présenté dans le paragraphe précédent, permettant de transmettre l'hypothèse de récurrence au niveau de récursion suivant.

Ainsi, cette partie se décompose de la manière suivante :

1. dans un premier temps, nous décrivons l'algorithme de partitionnement par colonnes pour un rectangle aplati de tailles horizontale h et verticale v vérifiant $h \leq v \leq 4h$. *Cet algorithme est bien entendu valable pour le problème symétrique d'un partitionnement par lignes d'un rectangle étiré vérifiant* $v \leq h \leq 4v$.
2. Ensuite, nous décrivons le principe de construction de l'arbre de récursion.
3. Finalement, nous décrivons l'algorithme de partitionnement fondé sur l'arbre de récursion préalablement construit.

8.4.2.1 Partitionnement par colonnes

Le but, ici, est de paver un rectangle R (pas un carré !) de taille $h \times v$ (tel que $h \leq v \leq 4h$) en p rectangles d'aires $s_1 \geq s_2 \geq \ldots \geq s_p$, où $\sum_{i=1}^{p} s_i = hv$. Nous supposons que le rapport r vérifie

$r = \frac{s_1}{s_p} \leq 2$. Dans ce cas, nous montrons qu'il existe un pavage de R par colonnes, composé de rectangles d'aire $s_i = h_i \times v_i$ tel que

$$(CR) : \ \frac{1}{4} v_i \leq h_i \leq 4 v_i \qquad \forall i, 1 \leq i \leq p.$$

Remarquons que cette condition est équivalente à $\frac{\sqrt{s_i}}{2} \leq (h_i, v_i) \leq 2\sqrt{s_i}$. Pour les besoins de la preuve, nous scindons cette condition en deux, et définissons pour la suite :

$$\begin{cases} (CR)_h & \text{la condition} \quad h_i \geq \frac{1}{4} v_i \\ (CR)_v & \text{la condition} \quad h_i \leq 4 v_i \end{cases}$$

L'algorithme est composé de deux phases distinctes :

Algorithme 14. *Pavage d'un rectangle de taille $h \times v$ par colonnes en p rectangles de tailles $s_1, \ldots, s_p$. Dans l'algorithme, c est un indice de colonnes, c_{max} est le nombre de colonnes final et $\mathcal{C}_c$ est un ensemble contenant les éléments de la c-ième colonne. Par ailleurs, l est le nombre d'éléments de la colonne en cours et $s'_1 \geq s'_2 \ldots \geq s'_l$ ses éléments.*

Procédure Partitionne_parcolonnes$(h, v, s_1, \ldots, s_p)$
{Précondition : $s_1 \geq s_2 \ldots \geq s_p$}
{Première_phase}
$c = 1$, $\mathcal{C}_c = \{s_1\}$
$l = 1$, $s'_1 = s_1$
for $j = 2, p$
  if $\frac{\sum_{k=1}^{l} s'_k}{v} \leq 2\sqrt{s'_l}$ {Si $(CR)_v$ est vérifiée par tous les éléments de $\mathcal{C}_c$}
    $c = c + 1$, $\mathcal{C}_c = \{s_j\}$
    $l = 1$, $s'_1 = s_j$
  else
    $\mathcal{C}_c = \mathcal{C}_c \cup \{s_j\}$
    $l = l + 1$, $s'_l = s_j$
$c_{max} = c$

{Seconde_phase}
if $v \leq 4h$ {N'est effectuée que si $v \leq 4h$.}
  if $\frac{\sum_{k=1}^{l} s'_k}{v} > 2\sqrt{s'_l}$ {Si $(CR)_v$ n'est pas vérifiée par tous les éléments de $\mathcal{C}_{c_{max}}$}
    for $j = 1, l$
      $\mathcal{C}_{j+c-l-1} = \mathcal{C}_{j+c-l-1} \cup \{s'_j\}$
    $\mathcal{C}_c = \emptyset$
    $c_{max} = c - 1$
Retourne$(c_{max}, \mathcal{C}_1, \ldots, \mathcal{C}_{c_{max}})$

La Figure 8.14 représente le partitionnement par colonnes du rectangle de taille 3×3.6 à l'aide de 7 rectangles d'aires : $(2, 2, 1.9, 1.5, 1.2, 1.2, 1)$.

Théorème 18 *Si $h \leq v \leq 4h$ alors la condition (CR) est vérifiée par tous les rectangles du partitionnement obtenu par l'Algorithme 14.*

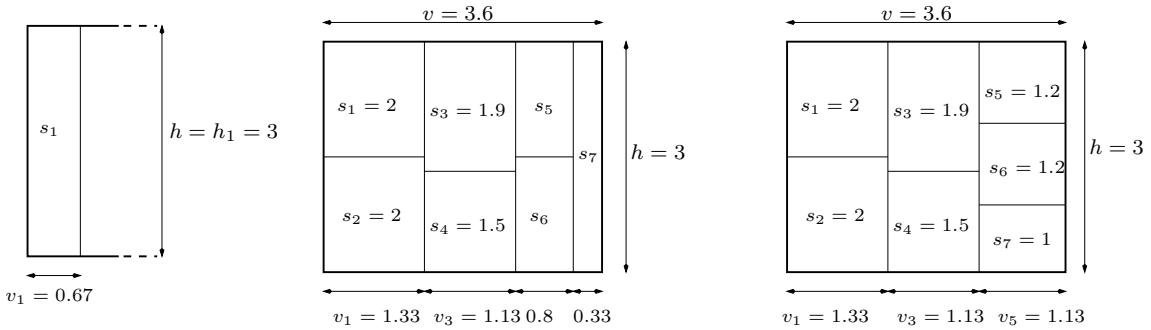


FIG. 8.14: *Les deux premières figures correspondent à la première phase de l'algorithme : les colonnes sont remplies en utilisant $(CR)_v$ comme condition d'arrêt. Ainsi, la première colonne ne peut pas être composée seulement de s_1 car la condition $(CR)_v$ n'est alors pas vérifiée. En effet, $\frac{h_1}{4} = \frac{3}{4} = 0.75 > 0.67 = v_1$. Un nouvel élément doit être ajouté à la colonne. On obtient alors $h_1 = h_2 = 1.5$ et $v_1 = 1.33$. La condition étant vérifiée, l'algorithme passe au remplissage de la colonne suivante. La dernière figure correspond à la seconde phase de l'algorithme : du fait que les éléments de la dernière colonne ne vérifient pas tous la condition $(CR)_v$, ils sont distribués sur les autres colonnes (Ici, un seul élément $(s_7 = 1)$ doit être distribué).*

Preuve. C'est une conséquence des trois affirmations suivantes :

- $[(CR)_h$ *Début*] : considérons toute colonne (même la dernière) après la première phase : alors tout élément de cette colonne vérifie la condition $(CR)_h$.
- $[(CR)_h$ *Fin*] : si on ajoute un élément à *une* colonne, la condition $(CR)_h$ reste vérifiée par tous les éléments de cette colonne.
- [*Nombre de colonnes suffisant*] : supposons qu'il y a $(c+1)$ colonnes à la fin de la première phase. Si les l éléments de la dernière colonne ne vérifient pas la condition $(CR)_v$, alors $l \leq c$.

En effet, considérons une colonne composée de k éléments $s'_1 \geq \ldots \geq s'_k$. Alors pour chaque élément $s'_i = h'_i \times v'_i$ de cette colonne, nous avons $h'_i = \frac{s'_i}{\sum_{j=1}^{k} s'_j} \times h$ et $v'_i = \frac{\sum_{j=1}^{k} s'_j}{h}$. En conséquence, $(CR)_h$: $\sqrt{s'_k} \geq \frac{1}{2}\frac{\sum_{j=1}^{k} s'_j}{h}$ et $(CR)_v$: $\sqrt{s'_1} \leq 2\frac{\sum_{j=1}^{k} s'_j}{h}$. En particulier, à partir du moment où la condition $(CR)_v$ est vérifiée pour un élément d'une colonne, alors elle restera vérifiée si l'on ajoute un élément à cette colonne.

$(CR)_h$ Début. Notons $s'_1 \geq s'_2 \geq \ldots \geq s'_k$ les éléments de la colonne considérée. Soit v'_1 sa largeur et $\forall 1 \leq i \leq k$, h'_i la hauteur du rectangle d'aire s'_i. Deux cas sont possibles :

1. $[k = 1]$: on a $v'_1 \leq v \leq 4h = 4h'_1$
2. $[k > 1]$: du fait que $(CR)_h$ n'est plus vérifiée si l'on enlève le k-ième élément, on a $\forall i \in \{1, \ldots, k\}$, $\frac{\sum_{j=1}^{k-1} s'_j}{h} < \frac{1}{2}\sqrt{s'_1} < \frac{1}{\sqrt{2}}\sqrt{s'_i}$. En conséquence, $v'_1 = \frac{\sum_{j=1}^{k} s'_j}{h} < \frac{k}{k-1} \times \frac{1}{\sqrt{2}}\sqrt{s'_i} < 2\sqrt{s'_i}$.

$(CR)_h$ Fin. Soient $s'_1 \geq s'_2 \geq \ldots \geq s'_k$ les éléments de la colonne considérée, et $s = s'_{k+1}$ l'élément à ajouter à cette colonne. Une fois cet élément ajouté, nous définissons les variables v'_1 et h'_i comme précédemment. Trois cas sont possibles :

1. $[k = 1]$: dans le pire des cas, il y a un seul élément dans la dernière colonne : en effet, considérons une colonne $s''_1, \ldots, s''_l$ de $l > 1$ éléments. Alors, $v''_1 = \frac{\sum_{i=1}^{l} s''_i}{h} \geq \frac{l}{2}\frac{s'_1}{h} \geq \frac{1}{4}h'_1 =$

$\frac{1}{4}h > \frac{1}{4}h''_i$. En conséquence, $\frac{s}{h} < \frac{h}{4}$. Mais $s'_1 \leq 2s$. D'où, $v'_1 = \frac{s'_1+s}{h} < \frac{3}{4}h$. Avec deux éléments dans une colonne, $h'_i \geq \frac{h}{3}$, d'où finalement, $v'_1 < \frac{9}{4}h'_i$.

2. $[k = 2]$: dans ce cas, $\frac{s'_1}{h} < \frac{h}{4}$ et $s'_1 \geq s'_2 \geq s$. Ainsi, $v'_1 = \frac{s'_1+s'_2+s}{h} < \frac{3}{4}h$. S'il y a trois éléments dans cette colonne, alors $h'_i \geq \frac{h}{5}$. En conséquence, $v'_1 < \frac{15}{4}h'_i$.
3. $[k \geq 3]$: dans ce cas $v'_1 = \frac{\sum_{i=1}^{k+1} s'_i}{h} < \frac{k+1}{k-1} \times \frac{\sum_{i=1}^{k-1} s'_i}{h} < \frac{k+1}{k-1} \times \frac{\sqrt{s'_1}}{2} < \sqrt{2}\sqrt{s'_i}$.

Nombre de colonnes suffisant. Soit v_i (pour $1 \leq i \leq c+1$) la largeur de la i-ième colonne à la fin de la première phase. Nous avons montré ci-dessus (cf Paragraphe $(CR)_h$ *Début*) que $\forall 1 \leq i \leq c,\ 1 \leq j \leq p, v_i < \sqrt{2}\sqrt{s_j}$. Comme $(CR)_v$ n'est pas vérifiée par les éléments de la $(c+1)$-ième colonne, $v_{c+1} < \frac{1}{\sqrt{2}}\sqrt{s_j} < \sqrt{2}\sqrt{s_j}$ et $\forall 1 \leq i \leq l,\ 1 \leq j \leq p,\ h'_i > \sqrt{2}\sqrt{s_j}$. En conséquence, $\forall 1 \leq i \leq c+1,\ 1 \leq j \leq l, v_i < h'_j$. De plus, comme $v \geq h$, $\sum_{i=1}^{c+1} v_i = v$ et $\sum_{j=1}^{l} h'_j = h$, il est clair que $l \leq c$.

■

Corollaire 1 *Dans le cas où $r \leq 2$, il existe une borne plus précise que celle fournie par le Théorème 17 :* $\frac{\hat{C}}{LB} \leq \frac{5}{4}$.

8.4.2.2 Partitionnement défini récursivement

L'idée principale de ce partitionnement est de partitionner la liste $s_1 \geq s_2 \geq \ldots \geq s_p$ des aires en sous-listes dans lesquelles les éléments diffèrent d'au plus un rapport 2 ($r \leq 2$). Par exemple, si $S = (0.49, 0.2, 0.2, 0.1, 0.01)$,nous obtenons les trois sous-listes (0.49), $(0.2, 0.2, 0.1)$ et (0.01). Ensuite, on somme les éléments de chaque sous-liste, et on considère ces sommes comme notre nouveau problème. Dans notre exemple, nous obtenons le problème réduit suivant $S = (0.5, 0.49, 0.01))$. Puis, on réitère le processus jusqu'à ce que plus aucun regroupement ne soit possible. Dans notre exemple, le processus s'arrête après la troisième étape avec $S = (0.99, 0.01))$. A la fin du processus, comme $\forall i,\ \ s_i > 2s_{i+1}$, les inégalités suivantes sont alors vérifiées :

$$\sum_{j>i} s_j < \frac{s_i}{2} + \frac{s_i}{4} + \ldots < s_i.$$

Arbre de récurrence.
De l'exemple précédent, on obtient l'ensemble récurrent $S = (((0.2, 0.2, 0.1), 0.49), 0.01)$ dont l'arbre correspondant est représenté Figure 8.15. *Dans cet arbre, et dans la description de l'algorithme, nous utilisons les notations abusives suivantes : à chaque étape, les éléments de l'ensemble récurrent considéré sont indexés $S_1, \ldots, S_k$, et pour chaque S_i, on note s_i l'aire associée et k_i son cardinal.* Le résultat est noté de manière récurrente : $RS = \mathcal{H}(RS_1, \ldots, RS_k)$ (respectivement $RS = \mathcal{V}(RS_1, \ldots, RS_k)$) signifie que la zone considérée est scindée horizontalement (respectivement verticalement) en k zones elles-mêmes partitionnées comme spécifié par RS_i. Ainsi, le résultat de la Figure 8.16 peut être noté :

$$\begin{array}{ll}\mathcal{H}(& \mathcal{V}(\mathcal{H}(0.05, 0.05), \mathcal{H}(0.05, 0.05), \mathcal{H}(0.06, 0.05), \mathcal{H}(0.06, 0.06), \mathcal{H}(0.06, 0.06)), \\ & \mathcal{V}(0.26, \mathcal{H}(0.12, \mathcal{V}(0.024, 0.026, \mathcal{H}(0.01, 0.01)))))\end{array}$$

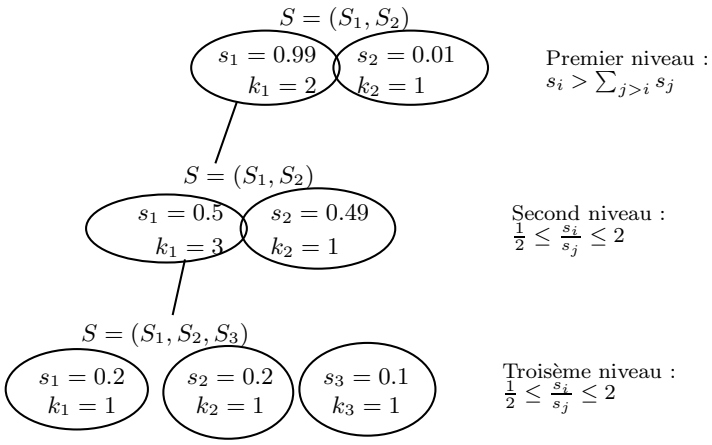


FIG. 8.15: *Arbre de récurrence de profondeur 3 associé à l'ensemble récurrent* $S = (((0.2, 0.2, 0.1), 0.49), 0.01)$.

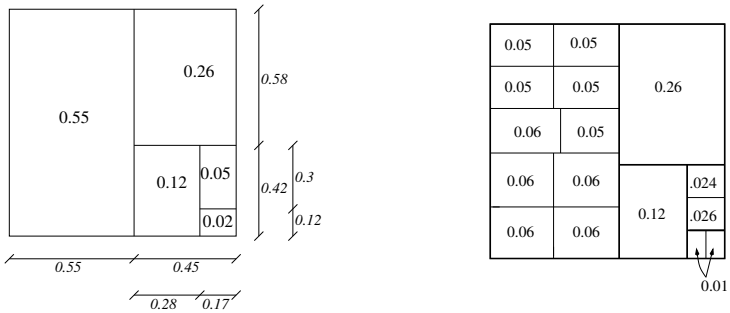
L'algorithme de partitionnement est défini à l'aide de deux fonctions principales, comme décrit ci-dessous :

Algorithme 15. *Partitionnement initial du carré (profondeur 0 de l'algorithme)*

Procédure Carré_Initial$(h, v, S = (S_1, \ldots, S_k))$
{ Nécessairement, les s_i doivent vérifier la condition $s_i > \sum_{j>i} s_j$}
if $k > 1$
  if $v \geq h$
    { partitionne $h \times v$ en $h \times v_1 = s_1$ et $h \times v_2 = \sum_{j>i} s_j$}
    $v_1 = \frac{s_1}{h}$, $v_2 = v - v_1$
    $RS_1 =$**Par_colonnes**(h, v_1, S_1)
    $RS_{autres} =$**Carré_initial**$(h, v_2, (S_2, \ldots, S_k))$
    $RS = \mathcal{V}(RS_1, RS_{autres})$
  else
    { partitionne $h \times v$ en $h_1 \times v$ et $h_2 \times v$}
    $h_1 = \frac{s_1}{v}$, $h_2 = h - h_1$
    $RS_1 =$**Par_colonnes**(h_1, v, S_1)
    $RS_{autres} =$**Carré_initial**$(h_2, v, (S_2, \ldots, S_k))$
    $RS = \mathcal{H}(RS_1, RS_{autres})$
Retourne(RS)

Algorithme 16. *partitionnement par colonnes des rectangles (profondeur supérieure)*

Procédure Par_colonnes$(h, v, S = (S_1, \ldots, S_k))$
{ Nécessairement, les s_i doivent vérifier la condition $\frac{1}{2} \leq \frac{s_i}{s_j} \leq 2$}
if $h < v$
  $(c_{max}, \mathcal{C}_1, \ldots, \mathcal{C}_{c_{max}})$ =Partitionne_parcolonnes$(h, v, s_1, \ldots, s_k)$
  for $c = 1, c_{max}$
    $\forall i \in [1, k]$, if $s_i \in \mathcal{C}_c$
      $h_i = \frac{\sum_{s_k \in C_c} s_k}{v}$
      $v_i = \frac{s_i}{h_i}$
      if $k_i > 1$
        RS_i =Par_colonnes(h_i, v_i, S_i)
      else $RS_i = s_i$
    $RS_{\mathcal{C}_c} = \mathcal{V}_{s_i \in \mathcal{C}_c}(RS_i)$
  $RS = \mathcal{H}_{c=1}^{c_{max}}(RS_{\mathcal{C}_c})$
else
  $(c_{max}, \mathcal{C}_1, \ldots, \mathcal{C}_{c_{max}})$ =Partitionne_parcolonnes$(v, h, s_1, \ldots, s_k)$
  for $c = 1, c_{max}$
    $\forall i \in [1, k]$, if $s_i \in \mathcal{C}_c$
      $v_i = \frac{\sum_{s_k \in C_c} s_k}{h}$
      $h_i = \frac{s_i}{v_i}$
      if $k_i > 1$
        RS_i =Par_colonnes(h_i, v_i, S_i)
      else $RS_i = s_i$
    $RS_{\mathcal{C}_c} = \mathcal{H}_{s_i \in \mathcal{C}_c}(RS_i)$
  $RS = \mathcal{V}_{c=1}^{c_{max}}(RS_{\mathcal{C}_c})$
Retourne(RS)



(a) Partitionnement initial (b) Partitionnement final

FIG. 8.16: *Partitionnement du carré à l'aide des surfaces suivantes :* $S = (0.26, 0.12, 0.06, 0.06, 0.06, 0.06, 0.06, 0.05, 0.05, 0.05, 0.05, 0.05, 0.026, 0.024, 0.01, 0.01)$. *Dans ce cas, la convergence est atteinte après une étape de l'algorithme de regroupement, et alors* $S = (0.55, 0.26, 0.12, 0.05, 0.02)$. *Les rectangles du partitionnement initial ont été eux-mêmes (récursivement) partitionnés à l'aide de l'algorithme "Par_colonnes".*

Théorème 19 *Soit $\hat{C}$ la somme des demi-périmètres des rectangles du partitionnement récursif défini dans ce paragraphe. Soit $LB = 2\sum_{i=1}^{p}\sqrt{s_i}$. Alors,*

$$\hat{C} \leq 1 + \frac{5}{4}LB$$

Preuve. Il nous faut montrer que le coût supplémentaire à payer pour l'ensemble des rectangles qui ne vérifient pas la condition (CR) est inférieur à 1. Pour le partitionnement *d'un rectangle donné*, appelons cette grandeur *surcoût*. En fonction de la profondeur de récursion, deux types de partitionnement sont effectués :

- Si *tous* les éléments diffèrent d'un ratio inférieur à 2, alors le rectangle est partitionné en colonnes.
- Si *tous* les éléments diffèrent d'un ratio supérieur à 2, alors le rectangle est partitionné en deux parties. La plus petite de ces deux parties est elle-même partitionnée etc.

En conséquence, la preuve est composée de deux parties :

1. [*Surcoût du partitionnement par colonnes*] Dans ce cas, nous montrons que le surcoût du partitionnement d'un rectangle de taille $h \times v$ ($v \geq h$) est inférieur à $(v - h)$.
2. [*Surcoût du partitionnement du carré initial*] Dans ce cas, nous montrons que le surcoût du partitionnement d'un rectangle de taille $h \times v$ ($v \geq h$) est inférieur à h.

Surcoût du partitionnement par colonnes.

1. S'il y a un seul rectangle, alors le surcoût est de $v + h - 2\sqrt{vh} \leq v - h$.
2. Supposons qu'il y ait plus d'une surface à positionner, et que le rectangle initial soit tellement allongé, que le partitionnement ne soit alors composé que d'un élément par colonne. Dans ce cas, pour chaque rectangle d'aire s_i sa forme est $h \times v_i$ et son surcoût est inférieur à $(v_i - h)$. Ainsi, le surcoût global est inférieur à $\sum_{i=1}^{c}(v_i - h) = v - ch < v - h$.
3. Supposons qu'il y ait plus d'un seul rectangle par colonne, de telle manière que seule la dernière colonne soit déséquilibrée (ces éléments ne vérifient pas la condition (CR)). Pour chaque élément de la dernière colonne, le surcoût est inférieur à $(h_i - v_c)$. Ainsi, le surcoût global est inférieur à $\sum_{i=1}^{l} h_i - v_c = h - lv_c < h$ et, comme $4h < v$, majoré par $(v - h)$.

Surcoût du partitionnement du carré initial. Supposons sans perte de généralité que le rectangle de taille $h \times v$ (où $v \geq h$) soit partitionné en deux rectangles $h \times v_1$ et $h \times v_2$ (où $v_1 > v_2$). Alors, le rectangle $h \times v_1$ est partitionné à l'aide de l'algorithme par colonnes, et le surcoût de ce rectangle est inférieur à $(v_1 - h)$ si $v_1 > h$ et vaut 0 sinon du fait que $h \leq v < 2v_1$.
Deux situations sont alors possibles pour le rectangle restant :

- Si $v_2 \geq h$, alors le surcoût de ce rectangle est inférieur à v_2, soit $v_1 > v_2 \geq h$. Ainsi, le surcoût global est inférieur à $v_1 - h + v_2 < v$.
- Si par contre $v_2 \leq h$, alors le surcoût de ce rectangle est inférieur à h. D'où, soit $v_1 \geq h$, et alors le surcoût global est majoré par $v_1 - h + h < v$; soit $v_1 < h$, et le surcoût global est majoré par $h \leq v$.

En conséquence, le surcoût total du partitionnement du carré initial est inférieur à 1. ■

8.5 Heuristique garantie pour PERI-MAX

De même que pour PERI-SUM, notre recherche d'heuristique s'est initialement orientée vers une solution par colonnes. En d'autres termes, nous considérons dans ce paragraphe le problème plus contraint COL-PERI-MAX, correspondant à la restriction de PERI-MAX aux partitionnements par colonnes. Mais contrairement à COL-PERI-SUM, nous ne connaissons pas de solution optimale polynomiale à ce problème. Le but de ce paragraphe est donc de montrer, dans un premier temps, la NP-Complétude de COL-PERI-MAX(s), puis de proposer une solution heuristique garantie à COL-PERI-MAX (et donc à PERI-MAX).

8.5.1 NP-Complétude de COL-PERI-MAX(s)

Définition 23 *COL-PERI-MAX(s,K) : soient p nombres réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$ ainsi qu'une borne réelle positive K, existe-t-il une partition du carré unitaire, par colonnes, en p rectangles R_i, d'aire s_i et de forme $h_i \times v_i$, telle que $\max_{i=1}^{p}(h_i + v_i) \leq K$?*

Notre premier résultat établit la difficulté intrinsèque du problème d'optimisation COL-PERI-MAX :

Théorème 20 *COL-PERI-MAX(s,K) est NP-complet.*

Preuve. De même que pour les théorèmes de NP-complétude précédemment établis, la preuve est basée sur une réduction au problème de 2-Partition :

Lemme 22

$$2P \leq_P \text{COL-MSP} \leq_P \text{COL-PERI-MAX},$$

où COL-MSP est défini comme suit :

Définition 24 *Col-Max-Square-Partition (COL-MSP)*
Soit un ensemble $\mathcal{A} = \{s_1, \ldots, s_p\}$ de p nombres réels positifs tels que $\sum_{i=1}^{p} s_i = 1$ et $s_1 \geq s_2 \geq \ldots \geq s_p$, existe-t-il une partition par colonnes du carré unitaire, en p rectangles R_i, d'aire s_i, telle que R_1 soit un carré ($h_1 = w_1$) et que les demi-périmètres des rectangles ne dépassent pas $h_1 + w_1 = 2\sqrt{s_1}$?

La preuve de la réduction COL-MSP $\leq_P$ COL-PERI-MAX(s,K), similaire à la preuve de la réduction MSP $\leq_P$ PERI-MAX(s,K) donnée dans le Paragraphe 8.3.1. Il reste à montrer que 2P $\leq_P$ COL-MSP, et 2P étant NP-complet, ceci achèvera la preuve du Théorème 20.

Considérons pour cela une instance arbitraire du problème de 2P, c'est à dire la donnée d'un ensemble de *n entiers* positifs $\mathcal{A} = \{a_1, \ldots, a_n\}$. Il nous faut transformer polynomialement cette instance en une instance du problème COL-MSP ayant une solution si et seulement si l'instance d'origine du problème 2P admet une solution. Soient

$$\begin{cases} S = \frac{\sum_{1 \leq i \leq n} a_i}{2}, \\ L = 2S + 1 \end{cases}$$

Nous construisons l'instance suivante (mise à l'échelle) du problème COL-MSP (représenté, par abus de notations, par COL-MSP($a_1, \ldots, a_n$)) : existe-t-il une partition par colonnes du carré de

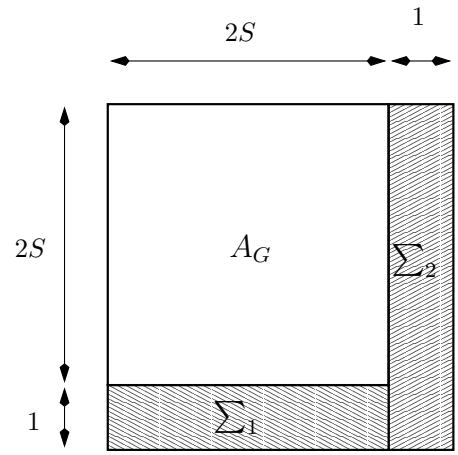


FIG. 8.17: *Schéma de partitionnement du carré après permutation des colonnes.*

taille $L \times L$ en $n+2$ rectangles $R_G, R_\epsilon, R_1, \ldots, R_n$ d'aires

$$\begin{cases} R_G : & A_G & = & 4S^2, \\ R_\epsilon : & A_\epsilon & = & 1 \\ R_i : & A_i & = & 2a_i \qquad (\forall i, 1 \leq i \leq n), \end{cases}$$

où les demi-périmètres des rectangles R_G, R_i et R_ϵ sont inférieurs à $4S$?

Supposons dans un premier temps qu'il existe une solution à cette instance. On peut considérer, sans perte de généralité, que le carré R_G est situé en haut de la première colonne. Ainsi, le partitionnement correspond au schéma de la Figure 8.17. Ainsi, si il existe une solution au problème COL-MSP$(a_1, \ldots, a_n)$, il existe un partitionnement du rectangle $\sum_1$ d'aire $2S \times 1 = 2S$ à l'aide d'un sous-ensemble de l'ensemble des rectangles $\{R_1, \ldots, R_n, R_\epsilon\}$. Ce qui signifie, en d'autres termes, qu'il existe un sous-ensemble I_1 de $I = \{1, 2, \ldots, n, \epsilon\}$ tel que $\sum_{i \in I_1} A_i = 2S$. Comme pour tout $i \in [1, n]$, A_i est pair, on a $\epsilon \notin I_1$, et alors $\sum_{i \in I_1} a_i = S$. De plus, $\sum_{i \in I - \{\epsilon\}} a_i = 2S$, et ainsi $(I_1, I - (I_1 \cup \{\epsilon\}))$ constitue une solution à l'instance initiale du problème 2P.

Réciproquement, supposons qu'il existe une solution au problème de 2-Partition initial, et montrons qu'il existe alors une solution au problème COL-MSP$(a_1, \ldots, a_n)$. A une permutation des indices prêt, considérons que $\sum_{i=1}^{m} a_i = \sum_{i=m+1}^{n} a_i = S$. Il suffit de partitionner le carré comme indiqué Figure 8.18. Clairement, comme $1+(1+2S) \leq 2S+2S$ (correspondant au périmètre du rectangle $\sum_2$), tous les rectangles de ce partitionnement ont un périmètre inférieur à $4S$ correspondant au carré R_G. Ceci achève la preuve de NP-Complétude de COL-MSP, et donc de COL-PERI-MAX. ■

8.5.2 Heuristique par colonne

Dans ce paragraphe, nous proposons une heuristique polynomiale, avec garantie, aux problèmes PERI-MAX et COL-PERI-MAX. C'est donc une solution par colonne que nous décrivons ici. Nous considérons deux cas distincts en fonction de l'aire du plus grand rectangle. Soient $s_1 \geq s_2 \ldots \geq s_p$ les aires des différents rectangles.

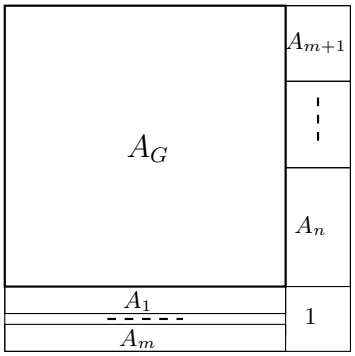


FIG. 8.18: *Si le problème de 2-Partition admet comme solution* $\sum_{i=1}^{m} a_i = \sum_{i=m+1}^{n} a_i = S$*, alors le carré peut être partitionné comme schématisé ci-dessus.*

Si s_1 est supérieur à $\frac{1}{3}$, nous utilisons une première heuristique. Dans ce cas, une colonne est créée par élément. Ainsi, le demi-périmètre d'un rectangle d'aire s_i est $1 + s_i$. On a donc,

$$\forall 1 \leq i \leq p,\ r_i = \frac{1+s_i}{2\sqrt{s_1}} \leq r_1 \leq \frac{2}{\sqrt{3}}.$$

Par contre, lorsque s_1 est inférieur à $\frac{1}{3}$, nous utilisons une seconde heuristique légèrement plus sophistiquée, assurant le fait que

$$\forall 1 \leq i \leq p,\ r_i = \frac{h_i+v_i}{2\sqrt{s_1}} \leq \frac{2}{\sqrt{3}}.$$

L'algorithme peut être décrit comme suit :

Algorithme 17. *Solution heuristique par colonnes au problème PERI-MAX, dans le cas* $s_1 < \frac{1}{3}$*. c est un indice de colonne,* c_{max} *le nombre de colonnes final, et* $\mathcal{C}_c$ *est un ensemble contenant les éléments de la c-ième colonne.*

Procédure Peri-max_Par_colonnes$(p, S = (s_1, \ldots, s_p))$
{Première_phase}
$c = 1$
$\mathcal{C}_c = \{s_1\}$
for $j = 2, p$
  if $\sum_{s \in \mathcal{C}_c} s \geq 2\sqrt{\frac{s_1}{3}} - \sqrt{\frac{4s_1}{3} - \max_{s \in \mathcal{C}_c} s}$.
    $c = c + 1$
    $\mathcal{C}_c = \{s_j\}$
  else $\mathcal{C}_c = \mathcal{C}_c \bigcup \{s_j\}$
  $c_{max} = c$

{Seconde_phase}
if $\sum_{i \in \mathcal{C}_{c_{\max}}} s_i \leq 2\sqrt{\frac{s_1}{3}} - \sqrt{\frac{4s_1}{3} - \max_{i \in \mathcal{C}_{c_{\max}}} s_i}$
  $\mathcal{C}_1 = \mathcal{C}_1 \bigcup \mathcal{C}_c$
  $\mathcal{C}_c = \emptyset$
  $c_{max} = c_{max} - 1$
Retourne$(c_{max}, \mathcal{C}_1, \ldots, \mathcal{C}_{c_{max}})$

La configuration de l'une des colonnes $\mathcal{C}_c$ est décrite dans la Figure 8.19. Le plus grand périmètre des rectangles de la colonne $\mathcal{C}_c$ est

$$\sum_{s\in\mathcal{C}_c} s + \frac{\max_{s\in\mathcal{C}_c} s}{\sum_{s\in\mathcal{C}_c} s}.$$

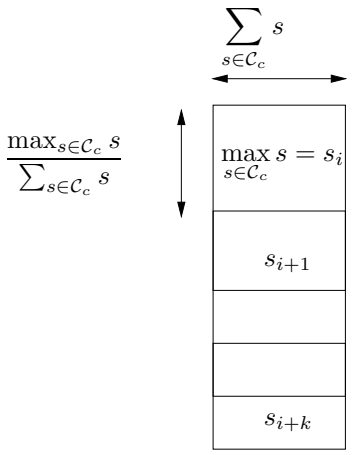


FIG. 8.19: *Configuration de la colonne $\mathcal{C}_c$.*

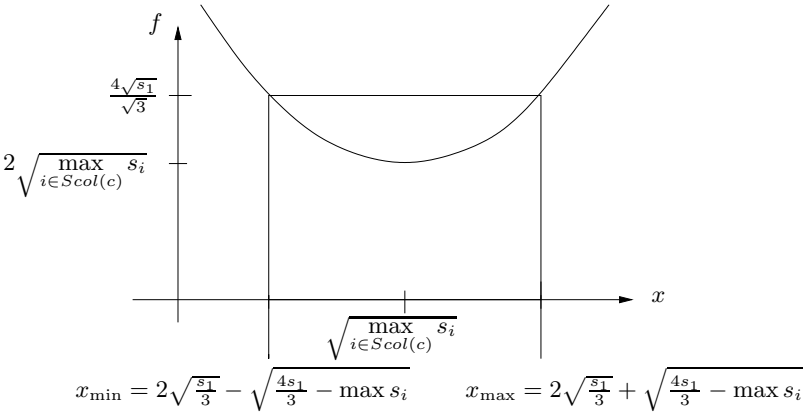


FIG. 8.20: *Tracé de la fonction $f(x) = x + \frac{\max_{s\in\mathcal{C}_c} s}{x}$.*

Comme montré Figure 8.20, la condition

$$\forall i \in \mathcal{C}_c, \quad \max(h_i + v_i) \leq \frac{4\sqrt{s_1}}{\sqrt{3}}$$

est vérifiée si et seulement si

$$2\sqrt{\frac{s_1}{3}} - \sqrt{\frac{4s_1}{3} - \max_{i\in\mathcal{C}_c} s_i} \leq \sum_{i\in\mathcal{C}_c} s_i \leq 2\sqrt{\frac{s_1}{3}} + \sqrt{\frac{4s_1}{3} - \max_{i\in\mathcal{C}_c} s_i}.$$

Ainsi, la condition

$$\forall i \in \mathcal{C}_c, \quad \max(h_i + v_i) \leq \frac{4\sqrt{s_1}}{\sqrt{3}}$$

est vérifiée pour toutes les colonnes, hormis la première ($\mathcal{C}_{c_{\max}}$ a pu être ajouté). En effet, comme

$$x_{\max}(\mathcal{C}_c) - x_{\min}(\mathcal{C}_c) = 2\sqrt{\frac{4s_1}{3} - \max_{i \in \mathcal{C}_c} s_i} \geq \frac{2\sqrt{s_1}}{\sqrt{3}} \geq s_1,$$

il n'est pas possible de sauter de $x_{\min}(\mathcal{C}_c)$ à $x_{\max}(\mathcal{C}_c)$ par l'ajout d'un seul rectangle à la colonne $\mathcal{C}_c$.

Afin de prouver la correction de notre algorithme, il nous faut donc prouver les deux points suivants :

- Il y a au moins deux colonnes à la fin de la première étape de l'algorithme. En effet,

$$\sum_{i \in \{1..n\}} s_i = 1 > \sqrt{s_1} \geq x_{\min}(\mathcal{C}_1).$$

- Supposons que la dernière colonne $\mathcal{C}_{c_{\max}}$ ne vérifie pas la condition

$$\forall s_i \in \mathcal{C}_{c_{\max}}, \quad \max(h_i + v_i) \leq \frac{4\sqrt{s_1}}{\sqrt{3}}.$$

  Alors la condition

$$\forall s_i \in \mathcal{C}_1, \quad \max(h_i + v_i) \leq \frac{4\sqrt{s_1}}{\sqrt{3}}$$

  reste vérifiée après

$$\mathcal{C}_1 = \mathcal{C}_1 \bigcup \mathcal{C}_{c_{\max}}.$$

  En effet, dans ce cas, nous savons que

$$\sum_{s \in \mathcal{C}_{c_{\max}}} s \leq x_{\min}(\mathcal{C}_{c_{\max}}) = 2\sqrt{\frac{s_1}{3}} - \sqrt{\frac{4s_1}{3} - \max_{s \in \mathcal{C}_{c_{\max}}} s} \leq \sqrt{\frac{s_1}{3}}$$

  et

$$\sum_{s \in \mathcal{C}_1} s \leq x_{\min}(\mathcal{C}_1) + s_1 \leq \sqrt{\frac{s_1}{3}} + s_1.$$

  D'où, comme $s_1 \leq \frac{1}{3}$,

$$\sum_{s \in \mathcal{C}_{c_{\max}} \bigcup \mathcal{C}_1} s \leq 2\sqrt{\frac{s_1}{3}} + s_1 \leq \sqrt{3s_1} = x_{\max}(\mathcal{C}_1).$$

En résumé, par l'utilisation de l'une ou l'autre de nos deux heuristiques en fonction de la valeur de s_1, nous imposons la garantie suivante :

Proposition 4 *Soit $\hat{M}$ le maximum des demi-périmètres des rectangles du partitionnement obtenu à l'aide de l'heuristique décrite ci-dessus. Soit $LB = 2\sqrt{s_1}$. Alors,*

$$\frac{\hat{M}}{LB} \leq \frac{2}{\sqrt{3}}$$

Remarquons qu'il n'est pas possible d'obtenir une meilleure garantie sans tenir compte des valeurs relatives des s_i. En effet, si on considère le cas $s_1 = s_2 = s_3 = \frac{1}{3}$, alors la solution optimale satisfait l'égalité

$$\hat{M} = \max_i(h_i + v_i) = \frac{2}{\sqrt{3}} 2\sqrt{s_1} = \frac{2}{\sqrt{3}} LB.$$

8.6 Travaux connexes

Dans cette partie, nous effectuons une synthèse des problèmes d'optimisation géométrique existants, similaires aux problèmes PERI-SUM et PERI-MAX :

Recouvrement d'un carré à l'aide de rectangles Alon et Kleitman [3] considèrent le recouvrement d'un carré unitaire en n rectangles. Il n'y a aucune contrainte sur l'aire des rectangles. Ils montrent que nécessairement l'un des rectangles a un périmètre supérieur à $4(2m+1)/(n+m(m+1))$, où $m = \lfloor\sqrt{n}\rfloor$. Ce résultat est atteignable lorsque $n = m(m+1)$ ou $n = m^2$.

Décomposition d'un carré en rectangles de périmètre minimum Kong et al. [67] déterminent comment paver le carré unitaire en p rectangles de même aire afin de minimiser le maximum des périmètres de ces rectangles. Ceci correspond exactement au problème PERI-MAX dans le cas où tous les rectangles ont même aire ($s_i = 1/p$ pour $1 \leq i \leq p$). Ce problème est polynomial [67]. La solution est l'un des deux arrangements par colonnes suivant : soit $m = \lfloor\sqrt{p}\rfloor$ ou bien $m = \lceil\sqrt{p}\rceil$. Le partitionnement est alors fait de m colonnes composées de $\lfloor\frac{n}{m}\rfloor$ ou $\lceil\frac{n}{m}\rceil$ rectangles. Cette solution est étendue au cas du partitionnement d'un rectangle (au lieu d'un carré) [66].

Partitionnement d'un rectangle avec des points intérieurs Un autre problème connexe consiste à chercher la partition de périmètre minimal passant par des points intérieurs : étant donné un rectangle R et un nombre fini de points (P) situés à l'intérieur du rectangle, trouver une partition en guillotine, telle que chaque point de P soit situé sur l'un des bords des rectangles ainsi construits. Le but est de minimiser la somme des longueurs des segments ainsi tracés. Ce problème est NP-complet comme montré dans [72] et des solutions heuristiques sont proposées dans [49, 50]. Le lien avec notre problème d'optimisation PERI-MAX est la coïncidence de la fonction objective à minimiser, mais la motivation originale [49, 50] était la minimisation des coûts de routage dans les machines VLSI et les contraintes sont différentes.

Partitionnement d'un tableau Le problème [64], consistant à partitionner un tableau en rectangles de poids minimum, est partiellement lié au problème d'optimisation PERI-MAX : on se donne un tableau de nombres positifs A, de taille $n \times n$, et l'on cherche à partitionner ce tableau en p rectangles (p fixé) afin que le maximum des poids des rectangles soit minimisé. Le poids d'un rectangle étant calculé comme la somme des nombres qu'il contient. Ce problème est NP-complet, et des heuristiques sont proposées dans [65, 64]

Rappelons finalement, qu'un certain nombre de problèmes d'optimisation géométrique sont répertoriés dans le "NP Compendium [34]", et que d'autre part le livre "Complexity and approximation" [10] fournit une synthèse sur le sujet.

8.7 Conclusion

Dans ce chapitre nous avons traité deux problèmes d'optimisation géométrique : comment partitionner un carré en p rectangles, d'aires fixées, afin de minimiser le maximum (PERI-MAX) ou la somme (PERI-SUM) de leur périmètre. Après avoir montré la NP-Complétude de ces problèmes, nous avons proposé des solutions heuristiques garanties. Ceci nous a amené à considérer les problèmes plus contraints suivants :

- COL-PERI-SUM est la restriction du problème PERI-SUM à un partitionnement par colonnes. Ce problème, polynomial, est résolu à l'aide d'un algorithme dynamique en $O(p^2\mathcal{C}_{opt}) \approx O(p^{2.5})$ étapes où $\mathcal{C}_{opt}$ est le nombre de colonnes du partitionnement final.
- Cette dernière solution ne fournissant pas de garantie théorique par rapport à la borne absolue inférieure, nous avons proposé une solution heuristique récursive en $O(p)$ étapes assurant le rapport $\frac{\hat{C}}{LB} \leq \frac{5}{4}$ où $\hat{C}$ correspond à la somme des périmètres de la solution.

- COL-PERI-MAX est la restriction du problème PERI-MAX à un partitionnement par colonnes. Ce problème est NP-complet, et nous proposons une solution heuristique en $O(p)$ étapes (commune au problème PERI-MAX), avec pour garantie $\frac{\hat{M}}{LB} \leq \frac{2}{\sqrt{3}}$, où $\hat{M}$ correspond au périmètre maximum de la solution et LB à la borne inférieure absolue.

Notons que la motivation initiale de ces problèmes est très importante : il s'agit d'allouer des tableaux de données sur un ensemble de processeurs hétérogène afin d'équilibrer parfaitement les charges de calcul, et de minimiser le coût de communication lors d'une exécution parallèle.

Chapitre 9

Conclusion

Dans cette thèse, nous avons abordé deux problèmes, le *pavage* que nous avons traité sous différentes approches (chapitres 2 à 5), et l'*équilibrage de charge* et l'*algorithmique hétérogène* pour des codes déjà partitionnés (chapitres 6 à 8). Nous rappelons chapitre par chapitre les résultats obtenus et les travaux qu'il faudrait effectuer pour les compléter :

Dans le Chapitre 2, nous avons calculé *la longueur du chemin critique* pour un nid de boucles aux *dépendances internes* $(0, 1)$ et $(1, 0)$. Nous avons déduit une forme de tuile "optimale", et avons appliqué ces résultats au pavage hiérarchique. Notre étude est réduite aux espaces d'itérations de forme trapézoïdale et aux bordures gauche et droite verticales. De plus, nous n'avons considéré que des allocations unidimensionnelles `cyclic(r)` ou `bloc-cyclic(r)`. Il serait intéressant de généraliser ces résultats à un espace d'itération de *forme triangulaire* avec une *allocation quelconque*, et chercher l'allocation qui, sous certains modèles de communication, minimise le chemin critique.

Le Chapitre 3 est consacré au problème de pavage de nids de boucles aux *dépendances externes*. Nous avons exprimé analytiquement une approximation de l'*empreinte cumulée d'une tuile*, c'est à dire de la quantité de données utilisées par le calcul de cette tuile. En fonction de la forme des tuiles, le recouvrement des données étant différent, l'empreinte varie. La minimisation de cette expression s'avérant difficile, nous avons proposé une solution heuristique. Exprimer analytiquement l'empreinte cumulée d'une tuile, est une question souvent rencontrée en conception de circuits. Malheureusement, notre solution *ne recouvre pas tous les cas de figure* : lorsque l'espace des données est plus petit que l'espace des tâches, les polytopes considérés n'étant plus des parallélépipèdes, cela complique énormément les choses, et les résultats présentés dans ce chapitre ne peuvent pas être élargis au cas général. Ceci devrait faire l'objet d'une étude approfondie, et l'approche suivie devra être probablement toute différente.

Dans le Chapitre 4 nous avons étudié l'*ordonnancement de tâches à l'intérieur des tuiles*. Pour notre modèle, nous avons donné des algorithmes optimaux, et fourni quelques éléments relatifs à la généralisation multidimensionnelle de notre approche. Ce travail ayant été initialement motivé par des méthodes de pavage sur système VLSI, une analyse de performances sur de telles machines manque cruellement à cette étude. Il faudrait comparer les différents algorithmes présentés dans ce chapitre et mesurer l'impact des différents paramètres (régularité de la permutation, période de l'ordonnancement, etc.) sur les performances.

Le Chapitre 5 est consacré à l'*implémentation d'un code réel* sur une machine à mémoire distribuée. Nous avons en particulier proposé une méthode, peu classique, fondée sur l'utilisation de *calculs redondants*. Le choix de cette méthode a été notamment motivé, par la volonté de générer

un code séquentiel le plus efficace possible, et par la complexité de la solution "parallélogramme". Il faudrait pour valider entièrement cette approche, mesurer, sur différentes plateformes, l'impact de la longueur du code sur les performances ainsi que le surcoût lié à la gestion d'un tableau supplémentaire contenant les données à communiquer. C'est ce que nous cherchons actuellement à faire, dans le cas simple où il n'y a pas de phases d'orthonormalisation, à l'aide de l'outil Omega [81]. Mais, il s'avère qu'une allocation `bloc-cyclic`, avec peu de processeurs, génére de nombreuses situations bloquantes, et la gestion des communications est complexe.

La majorité des résultats établis jusqu'à présent, incluant ceux présentés dans cette thèse, fournissent la taille et/ou la forme de tuiles optimales pour un cas particulier ; les modèles utilisés sont parfois critiquables, et les solutions proposées sont difficilement applicables directement, d'autant plus que les résultats manquent souvent de généralité. A ce niveau de maturité du pavage, il semble donc nécessaire de faire une synthèse, de redéfinir de nouvelles directions de recherche, et d'implémenter à l'intérieur d'un compilateur l'état de l'art concernant cette phase d'optimisation. A cette fin, il faudrait, d'une part répertorier l'ensemble des outils existants et comparer leurs approches. Il faudrait, d'autre part, effectuer des mesures sur plusieurs architectures différentes, afin de redéfinir les modèles et de préciser leurs domaines d'application.

Notons pour terminer qu'en général, le pavage d'un nid de boucles totalement permutables est effectué indépendamment des phases d'ordonnancement et d'alignement des données et des calculs. Ces deux dernières phases peuvent, dans une certaine mesure, être effectuées simultanément : en brisant la contrainte du "owner computes rule[1]", le graphe d'affinités ainsi construit [70], devient généralement acyclique [c4]. Mais il manque à cette approche un modèle de coût précis. Ceci est principalement lié au fait que le pavage est effectué *après* l'alignement et l'ordonnancement des tâches. En fait, il semble possible, à l'aide d'une analyse de flots de données, et en se restreignant à un pavage orthogonal et à une allocation `bloc-cyclic` de regrouper l'ensemble de ces phases et de fournir une solution comprenant à la fois l'ordonnancement des calculs et des communications.

Dans le Chapitre 6, nous avons abordé différents problèmes liés à l'implémentation de programmes sur un *ensemble hétérogène de ressources de calcul*, et nous nous sommes restreint pour cela à la recherche d'une *allocation statique unidimensionnelle* des données. Nous avons traité le problème de l'équilibrage de gros grains de calculs indépendants, puis proposé une allocation nommée incrémentale, adaptée à la décomposition LU. Finalement, le problème de pavage abordé dans le Chapitre 2 a été rediscuté dans le cadre de ressources hétérogènes. Ces résultats peuvent être généralisés au cas d'un réseau hétérogène de profondeur supérieure à 1 [c5] : l'hétérogénéité des communications dans le cas d'un cluster de clusters (etc.) peut être prise en compte en considérant le système d'un point de vue hiérarchique. La construction du motif est alors légèrement plus complexe [r7], et plusieurs choix se posent. Il faudrait dans ce cas comparer expérimentalement les différentes approches, et proposer une solution générale à notre problème statique.

Nous avons alors, dans le Chapitre 7, adapté ces résultats à des noyaux d'algèbre linéaire dans le cas d'une grille hétérogène bidimensionnelle. Nous avons décomposé ce problème en deux étapes : arrangement d'un ensemble de processeurs, en une grille la plus proche possible d'une grille parfaite, puis allocation des données sur cette grille. Le problème d'allocation, dans son ensemble, est malheureusement NP-complet, mais nous ne savons pas si la dernière étape n'est pas polynomiale. Nous proposons une heuristique, mais sans aucune garantie. La recherche d'une telle solution reste donc un problème ouvert.

La contrainte d'une allocation en grille n'étant pas nécessairement justifiée, nous avons, dans le

[1] le processeur qui détient la données située à gauche d'une affectation, effectue le calcul associé.

Chapitre 8, abordé le problème d'équilibrage de charge à l'aide d'une allocation libre. Cela, nous à conduit à quatre problèmes d'optimisation géométrique :

- PERI-SUM consiste à partitionner un carré en rectangles d'aire fixée, et de minimiser la somme des périmètres des rectangles ainsi construits. Ce problème est NP-complet. Nous proposons une solution heuristique définie récursivement.
- COL-PERI-SUM correspond au problème PERI-SUM avec la contrainte d'un partitionnement par colonnes. Nous proposons une solution optimale à ce problème de complexité polynomiale. Malheureusement, nous ne pouvons garantir aucune borne théorique acceptable en rapport avec le problème d'optimisation PERI-SUM.
- PERI-MAX consiste à partitionner un carré en rectangles d'aire fixée, et de minimiser le maximum des périmètres des rectangles ainsi construits. Ce problème est NP-complet, et nous proposons une heuristique garantie commune au problème d'optimisation COL-PERI-MAX suivant.
- COL-PERI-MAX correspond au problème PERI-MAX avec la contrainte d'un partitionnement par colonnes. Contrairement à COL-PERI-SUM, ce problème est NP-complet, et nous proposons une solution heuristique.

En résumé, nous avons montré que la stratégie d'une allocation statique des données et des calculs était nécessaire à l'exécution de codes très réguliers sur plateformes hétérogènes. En effet, les stratégies dynamiques réagissent mal à la présence de dépendances de données, ralentissant au rythme du plus lent l'ensemble des processeurs, et mettant souvent en jeu des redistributions coûteuses. Nous avons aussi montré que la recherche d'une allocation statique s'avérait être un problème difficile, même sur un simple problème tel que la multiplication de matrices. Puisque les ressources peuvent, dans certains cas être non dédiées, il faut aussi imaginer une stratégie semi-statique : les problèmes liés au variations de vitesse des processeurs peuvent ainsi être résolus par une réallocation des calculs et des données, de temps en temps, entre des phases statiques correctement identifiées. L'un des avantages des solutions heuristiques que nous avons décrites ici, est qu'elles se présentent toutes sous forme d'un partitionnement par colonnes, ce qui fournit une structure unifiée pour l'étude des stratégies de réallocation.

En conclusion, nous pensons que l'implémentation efficace de routines de calcul hétérogènes, semblable à la librairie ScaLAPACK sur plateformes homogènes, passera nécessairement par la construction de schémas efficaces *d'allocation statique* et de *réallocation*. D'autre part, la difficulté des problèmes algorithmiques rencontrés ne doit pas être sous-estimée : le partitionnement de données, l'ordonnancement et l'équilibrage de charges sont connus comme étant difficiles dans le contexte du parallélisme classique. Ils deviennent encore plus complexe dans le cadre de clusters hétérogènes, sans parler des plateformes de "metacomputing". Ceci constitue un challenge enthousiasmant à investir pour des algorithmiciens aventureux...

Bibliographie

[1] A. Agarwal, D.A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 6(9) :943–962, 1995.

[2] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Research and Development*, 38(6) :673–681, 1994.

[3] N. Alon and D.J. Kleitman. Covering a square by small perimeter rectangles. *Discrete Computational Geometry*, 1 :1–7, 1986.

[4] Stergios Anastasiadis and Kenneth C. Sevcik. Parallel application scheduling on networsk of workstations. *Journal of Parallel and Distributed Computing*, 43 :109–124, 1997.

[5] R. Andonov, P-Y Calland, S. Niar, S. Rajopadhye, and N. Yanev. First steps towards optimal oblique tiling of two-dimensional iterations. In *CPC2000, Compilers for Parallel Computers, Aussois, France, Janvier 4-7*, 2000. (Also available as RR-2000-01, LAMIH/ROI,Université de Valenciennes, `http://www.univ-valenciennes.fr/limav/andonov/pub/rech/`).

[6] Rumen Andonov, Hafid Bourzoufi, and Sanjay Rajopadhye. Two-dimensional orthogonal tiling : from theory to practice. In *International Conference on High Performance Computing (HiPC)*, pages 225–231, Trivandrum, India, 1996. IEEE Computer Society Press.

[7] Rumen Andonov and Sanjay Rajopadhye. Optimal orthogonal tiling of two-dimensional iterations. *Journal of Parallel and Distributed Computing*, 45(2) :159–165, 1997.

[8] D. Arapov, A. Kalinov, A. Lastovetsky, and I. Ledovskih. Experiments with mpc : efficient solving regular problems on heterogeneous networks of computers via irregularization. In *Irregular'98*, LNCS 1457. Springer Verlag, 1998.

[9] Bengt Aspvall, Magnus Halldorsson, and Fredrik Manne. Approximations for the generalized block distribution of a matrix. In *proceedings of 6th Scandinavian Workshop on Algorithm Theory, SWAT'98*, volume 1432, pages 47–58. Lecture Notes in Computer Science, Springer, 1998.

[10] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer, Berlin, Germany, 1999.

[11] Utpal Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2 :133–149, 1988.

[12] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix-matrix multiplication on heterogeneous platforms. Technical Report RR-00-02, LIP, ENS Lyon, France, January 2000. Available at `www.ens-lyon.fr/LIP/`.

[13] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid : Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.

[14] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack : A portable linear algebra library for distributed-memory computers - design issues and performance. In *Supercomputing '96*. IEEE Computer Society, 1996.

[15] 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, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[16] V. Boudet, F. Rastello, and Y. Robert. Algorithmic issues for (distributed) heterogeneous computing platforms. Technical Report RR-99-19, LIP, ENS Lyon, France, 1999. Available at `www.ens-lyon.fr/LIP/`.

[17] Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves Robert. Data allocation strategies for dense linear algebra kernels on heterogeneous two-dimensional grids. Technical Report RR-99-31, LIP, ENS Lyon, France, 1999. Available at `www.ens-lyon.fr/LIP/`.

[18] Vincent Boudet, Fabrice Rastello, and Yves Robert. Algorithmic issues for (distributed) heterogeneous computing platforms. In Rajkumar Buyya and Toni Cortes, editors, *Cluster Computing Technologies, Environments, and Applications (CC-TEA'99)*, pages 709–712. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-19.

[19] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling ? *Integration, the VLSI Journal*, 17 :33–51, 1994.

[20] Pierre Boulet, Jack Dongarra, Fabrice Rastello, Yves Robert, and Frederic Vivien. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters*, 9(2) :197–213, 1999. Extended version available as LIP Technical Report RR-98-49.

[21] Pierre Boulet, Jack Dongarra, Yves Robert, and Frédéric Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25 :547–568, 1999.

[22] J. Brenner and L. Cummings. The Hadamard maximum determinant problem. *Amer. Math. Monthly*, 79 :626–630, 1972.

[23] Pierre-Yves Calland, Jack Dongarra, and Yves Robert. Tiling on systems with communication/computation overlap. *Concurrency : Practice and Experience*, 11(3) :139–153, 1999.

[24] Pierre-Yves Calland and Tanguy Risset. Precise tiling for uniform loop nests. In P. Cappello et al., editors, *Application Specific Array Processors ASAP 95*, pages 330–337. IEEE Computer Society Press, 1995.

[25] L. Carter, J. Ferrante, S. F. Hummel, B. Alpern, and K.S. Gatlin. Hierarchical tiling : a methodology for high performance. Technical Report CS-96-508, University of California at San Diego, San Diego, CA, 1996. Available at http ://www.cse.ucsd.edu/∼carter.

[26] Y-S. Chen, S-D. Wang, and C-M. Wang. Tiling nested loops into maximal rectangular blocks. *Journal of Parallel and Distributed Computing*, 35(2) :123–32, 1996.

[27] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK : A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97 :1–15, 1996. (also LAPACK Working Note #95).

[28] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5 :173–184, 1996.

[29] W.H. Chou and S.Y. Kung. Scheduling partitioned algorithms on processor arrays with limited communication supports. In Luigi Dadda and Benjamin Wah, editors, *Application Specific Array Processors ASAP 93*, pages 53–64. IEEE Computer Society Press, 1993.

[30] E. Chu and A. George. QR factorization of a dense matrix on a hypercube multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 11 :990–1028, 1990.

[31] Michal Cierniak, Mohammed J. Zaki, and Wei Li. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43 :156–162, 1997.

[32] Michal Cierniak, Mohammed J. Zaki, and Wei Li. Scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6) :356–372, 1997.

[33] S. Coleman and K. Mckinley. Tile Size Selection using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, volume 30(6), pages 279–290, June 1995.

[34] P. Crescenzi and V. Kann. A compendium of NP optimization problems. World Wide Web document, URL : `http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html`.

[35] D. E. Culler and J. P. Singh. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1999.

[36] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4) :379–392, 1997.

[37] Frédéric Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. Scheduling block-cyclic array redistribution. *IEEE Trans. Parallel Distributed Systems*, 9(2) :192–205, 1998.

[38] Michèle Dion. *Alignement et distribution en parallélisation automatique*. PhD thesis, École normale supérieure de Lyon, January 1996.

[39] Michèle Dion, Tanguy Risset, and Yves Robert. Resource-constrained scheduling of partitioned algorithms on processor arrays. *Integration, the VLSI Journal*, 20 :139–159, 1996.

[40] Val Donaldson, Francine Berman, and Ramamohan Pandri. Program speedup in a heterogeneous computing network. *Journal of Parallel and Distributed Computing*, 21 :316–322, 1994.

[41] J. Dongarra, R. van de Geijn, and D. Walker. Scalability issues affecting the design of a dense linear algebra library. *Journal of Parallel and Distributed Computing*, 22(3) :523–537, 1994.

[42] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2) :151–180, 1995.

[43] Xing Du and Xiadong Zhang. Coordinating parallel processes on networks of workstations. *Journal of Parallel and Distributed Computing*, 46 :125–135, 1997.

[44] G. Benettin et al. Lyapunov characteristic exponents for smooth dynamical systems and for hamiltonian systems ; a method for computing all of them. *Meccanica*, (9) :21, March 1980.

[45] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i : matrix multiplication. *Parallel Computing*, 3 :17–31, 1987.

[46] J.S. Frame, G. de B. Robinson, and R.M. Thrall. The hook graphs of the symmetric group. *Canad. J. Math*, (6) :316–325, 1954.

[47] Michael R. Garey and Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.

[48] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins, 2 edition, 1989.

[49] T.F. Gonzalez and S. Zheng. Improved bounds for rectangular and guilhotine partitions. *J. Symbolic Computation*, 7 :591–610, 1989.

[50] T.F. Gonzalez and S. Zheng. Approximation algorithm for partitioning a rectangle with interior points. *Algorithmica*, 5 :11–42, 1990.

[51] M. Grigni and F. Manne. On the complexity of the generalized block distribution. In *Parallel Algorithms for Irregularly Structured Problems, Third International Workshop IRREGULAR '96*, LNCS 1117, pages 319–326. Springer-Verlag, 1996.

[52] Andrew S. Grimshaw, Jon B. Weissman, Emily A. West, and Ed. C. Loyot Jr. Metasystems : an approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21 :257–270, 1994.

[53] Stanford Compiler Group. Suif compiler system. World Wide Web document, URL : `http://suif.stanford.edu/suif/suif.html`.

[54] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1) :27–45, 1992.

[55] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Technical Report UCSD-CS96-489, and on the WEB at http ://www.cse.ucsd.edu/∼carter.

[56] François Irigoin. *Partitionnement des boucles imbriquées, une technique d'optimisation pour les programmes scientifiques*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, June 1987.

[57] François Irigoin and Rëmy Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[58] Nathan Jacobson. *Basic Algebra I*. W.H. Freeman and Company, Yale University, 1974.

[59] Maher Kaddoura and Sanjay Ranka. Run-time support for parallelization of data-parallel applications on adaptive and nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 43 :163–168, 1997.

[60] Maher Kaddoura, Sanjay Ranka, and Albert Wang. Array decomposition for nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 36 :91–105, 1996.

[61] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *HPCN Europe 1999*, LNCS 1593, pages 191–200. Springer Verlag, 1999.

[62] W. K. Kaplow and B. K. Szymanski. Tiling for Parallel Execution - Optimizing Node Cache Performance. In *Workshop on Challenges in Compiling for Scaleable Parallel Systems, Eigth IEEE Symposium on Parallel and Distributed Processing*, 1996.

[63] R. W. Kenyon. Tiling a rectangle with the fewest squares. *J. Combin. Theory A*, 76 :272–291, 1996.

[64] S. Khanna, S. Muthukrishnan, and M. Paterson. On approximating rectangle tiling and packing. In *Proc. 9th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 384–393. ACM Press, 1998.

[65] S. Khanna, S. Muthukrishnan, and S. Skiena. Efficient array partitioning. In *Proc. 24th Int. Colloquium on Automata, Languages and Programming*, LNCS 1256, pages 616–626. Springer-Verlag, 1997.

[66] T.Y. Kong, D.M. Mount, and W. Roscoe. The decomposition of a rectangle into rectangles of minimal perimeter. *SIAM J. Computing*, 17(6) :1215–1231, 1988.

[67] T.Y. Kong, D.M. Mount, and M. Wermann. The decomposition of a square into rectangles of minimal perimeter. *Discrete Applied Mathematics*, 16 :239–243, 1987.

[68] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[69] S.Y. Kung. *VLSI Array Processors.* Prentice-Hall, 1988.

[70] Jingke Li and Marina Chen. Index domain alignment : Minimizing cost of cross-referencing between distributed arrays. In *Frontiers 90 : The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, College Park, MD, October 1990. IEEE Computer Society Press.

[71] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201–214. ACM Press, January 1997.

[72] A. Lingas, R.Y. Pinter, R.L. Rivest, and A. Shamir. Minimum edge length partitioning of rectilinear polygons. In *Proc. 20th Ann. Allerton Conference on Communication, Control and Computing*, 1982.

[73] R. Livi, A. Politi, and S. Ruffo. Distribution of characteristic exponents in the thermodynamic limit. *Journal of Physics A*, (19) :2033–2040, 1986.

[74] R. I. MacLachlan and P. Atela. The accuracy of symplectic integrators. *Nonlinearity*, (5) :541–562, 1992.

[75] Naraig Manjikian and Tarek S. Abdelrahman. Loop fusion for parallelism and locality. *IEEE Trans. Parallel Distributed Systems*, 8(2) :193–209, February 1997.

[76] Fredrik Manne and Tor Sørevik. Partitioning an array onto a mesh of processors. In *Proceedings of Para'96, Workshop on Applied Parallel Computing in Industrial Problems and Optimization*, volume 1184, pages 467–477. Lecture Notes in Computer Science, Springer, 1996.

[77] Nicholas Mitchell, Karin Högsted, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6) :641–670, June 1998. Special issue from 1997 LCPC workshop.

[78] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *1995 International Conference on Supercomputing*, pages 270–279. ACM Press, 1995.

[79] S. Pande. A Compile Time Partitioning Method for DOALL Loops on Distributed Memory Systems. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume III (Software), pages 35–44. IEEE Computer Society Press, 1996.

[80] Loïc Prylli and Bernard Tourancheau. BIP : a new protocol designed for high performance networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, volume 1388 of *Lect. Notes in Comp. Science*, pages 472–485. Held in conjunction with IPPS/SPDP 1998. IEEE, Springer-Verlag, April 1998.

[81] William Pugh and the Omega Team. World Wide Web document, URL : `http://www.cs.umd.edu/projects/omega/`.

[82] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2) :108–120, 1992.

[83] L.E. Reichl. *The transition to chaos*. Springer Verlag, 1989.

[84] M. Remoissenet. *Waves called solitons*. Springer Verlag, 1994.

[85] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, The University of Tennessee, Knoxville, TN, August 1990.

[86] D. J. Searles, D. J. Evans, and D. J. Isbister. The number dependence of the maximum lyapunov exponent. *Physica A*, (240) :96–107, 1997.

[87] S. Sharma, C.-H. Huang, and P. Sadayappan. On data dependence analysis for compiling programs on distributed-memory machines. *ACM Sigplan Notices*, 28(1), jan 1993. Extended Abstract.

[88] Andrew S. Tanenbaum. *Structured computer organization*. Prentice-Hall International Editions, 1990.

[89] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Center for Research in Parallel Computing, Rice University, January 1993. CRPC-TR93291-S.

[90] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing '98*. Sponsored by ACM SIGARCH and IEEE Computer Society, 1998. Winner, best paper in the systems category, SC98 : High Performance Networking and Computing.

[91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44. ACM Press, 1991.

[92] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4) :452–471, October 1991.

[93] M. J. Wolfe. More Iteration Space Tiling. In *Proceedings of Supercomputing'89*, pages 655–664, November 1989.

[94] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.

[95] Yong Yan, Xiadong Zhang, and Yongsheng Song. An effective and practical performance model for parallel computing on nondedicated heterogeneous now. *Journal of Parallel and Distributed Computing*, 38 :63–80, 1996.

Bibliographie personnelle

Chaque travail a, dans un premier temps, fait l'objet d'un rapport de recherche, puis a généralement été présenté en version courte à une conférence. Ensuite, le rapport, modifié et complété, est soumis pour publication dans une revue internationale.

Articles parus dans des revues internationales.

[j1] Frédéric Desprez, Jack Dongarra, Fabrice Rastello, and Yves Robert. Determining the idle time of a tiling : new results. *Journal of Information Science and Engineering*, 14 :167–190, 1998. Short version of [c1].

[j2] Pierre Boulet, Jack Dongarra, Fabrice Rastello, Yves Robert, and Frédéric Vivien. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters*, 9(2) :197–213, 1999. Extended version available as LIP Technical Report RR-98-49.

Conférences avec publication des actes.

[c1] Frédéric Desprez, Jack Dongarra, Fabrice Rastello, and Yves Robert. Determining the idle time of a tiling : new results. In *Parallel Architectures and Compilation Techniques PACT'97*, pages 307–317. IEEE Computer Society Press, 1997.

[c2] Fabrice Rastello, Amit Rao, and Santosh Pande. Optimal task scheduling to minimize inter-tile latencies. In *International Conference on Parallel Processing (ICPP'98)*, pages 172–179. IEEE Computer Society Press, 1998.

[c3] Fabrice Rastello and Yves Robert. Loop partitioning versus tiling for cache-based multi-processors. In *International Conference on Parallel and distributed Computing and Systems, PDCS'98, Las Vegas*, pages 477–483. IASTED Press, 1998.

[c4] Vincent Boudet, Fabrice Rastello, and Yves Robert. Alignment and distribution is NOT (always) NP-hard. In Chyi-Nan Chen and Lionel M. Ni, editors, *ICPADS'98, Taiwan*, pages 648–657. IEEE Computer Society Press, 1998.

[c5] Vincent Boudet, Fabrice Rastello, and Yves Robert. Algorithmic issues for (distributed) heterogeneous computing platforms. In Rajkumar Buyya and Toni Cortes, editors, *Cluster Computing Technologies, Environments, and Applications (CC-TEA'99)*, pages 709–712. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-19.

[c6] Vincent Boudet, Fabrice Rastello, and Yves Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). In Hamid R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1285–1291. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-17.

[c7] Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves Robert. Data allocation strategies for dense linear algebra kernels on heterogeneous two-dimensional grid. In *Parallel and*

Distributed Computing and Systems conference (PDCS'99), pages 561–569. IASTED Press, 1999.

[c8] Vincent Boudet, Fabrice Rastello, and Yves Robert. PVM implementation of heterogeneous ScaLAPACK dense linear solvers. In J. Dongarra, E. Luque, and T. Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, LNCS 1697, pages 333–340. Springer Verlag, 1999. Short extention of [r6] and [r7].

[c9] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Load balancing strategies for dense linear algebra kernels on heterogeneous two-dimensional grids. In *14th International Parallel and Distributed Processing Symposium, IPDPS'2000, Mexico*, pages 783–792. IEEE Computer Society Press, 2000. Extention of [c7].

[c10] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix-matrix multiplication on heterogeneous platforms. In *2000 International Conference on Parallel Processing (ICPP'2000)*, pages 289–298. IEEE Computer Society Press, 2000.

Rapports de recherche.

[r1] Frederic DESPREZ, Jack DONGARRA, Fabrice RASTELLO, and Yves ROBERT. Determining the idle time of a tiling. Technical Report RR-97-35, LIP, ENS Lyon, France, 1997. Available at `www.ens-lyon.fr/LIP/`.

[r2] Fabrice RASTELLO, Amit RAO, and Santosh PANDE. Task ordering in linear tiles. Technical Report RR-98-11, LIP, ENS Lyon, France, 1998. Avalaible `www.ens-lyon.fr/LIP/`. Older version of [c2].

[r3] Fabrice RASTELLO and Yves ROBERT. Loop partitioning versus tiling for cache-based multiprocessors. Technical Report RR-98-13, LIP, ENS Lyon, France, February 1998. Avalaible `www.ens-lyon.fr/LIP/`. Extented version of [c3].

[r4] P. Boulet, J. Dongarra, F. Rastello, Y. Robert, and F. Vivien. Algorithmic issues for heterogeneous computing platforms. Technical Report RR-98-49, LIP, ENS Lyon, France, 1998. Available at `www.ens-lyon.fr/LIP`. Extented version of [j2].

[r5] V. Boudet, F. Rastello, and Y. Robert. Alignment and distribution is not(always) np-hard. Technical Report RR-98-30, LIP, ENS Lyon, France, 1998. Available at `www.ens-lyon.fr/LIP`. Same version as [c4].

[r6] V. Boudet, F. Rastello, and Y. Robert. A proposal for an heterogeneous cluster ScaLAPACK (dense linear solvers). Technical Report RR-99-17, LIP, ENS Lyon, France, 1999. Available at `www.ens-lyon.fr/LIP`. Extented version of [c6].

[r7] V. Boudet, F. Rastello, and Y. Robert. Algorithmic issues for (distributed) heterogeneous computing platforms. Technical Report RR-99-19, LIP, ENS Lyon, France, 1999. Available at `www.ens-lyon.fr/LIP/`. Extented version of [c5].

[r8] Vincent BOUDET, Antoine PETITET, Fabrice RASTELLO, and Yves ROBERT. Data allocation strategies for dense linear algebra kernels on heterogeneous two-dimensional grids. Technical Report RR-99-31, LIP, ENS Lyon, France, 1999. Available at `www.ens-lyon.fr/LIP/`. Same version as [c7].

[r9] Olivier BEAUMONT, Vincent BOUDET, Fabrice RASTELLO, and Yves ROBERT. Matrix-matrix multiplication on heterogeneous platforms. Technical Report RR-00-02, LIP, ENS Lyon, France, January 2000. Available at `www.ens-lyon.fr/LIP/`. To be published.

[r10] Olivier BEAUMONT, Vincent BOUDET, Fabrice RASTELLO, and Yves ROBERT. Partitioning a square into rectangles : NP-completeness and approximation algorithms. Technical Report RR-00-10, LIP, ENS Lyon, France, February 2000. Available at `www.ens-lyon.fr/LIP/`. To be published.

[r11] Olivier BEAUMONT, Vincent BOUDET, Arnaud LEGRAND, Fabrice RASTELLO, and Yves ROBERT. Heterogeneity considered harmful to algorithm designers. Technical Report RR-00-24, LIP, ENS Lyon, France, june 2000. Available at `www.ens-lyon.fr/LIP/`.