Calcul d'un tiling optimal et ordonnancement topologique optimal

Rapport de D.E.A informatique de l'université de Versailles
mention: Parallélisme

présenté par: **Bouzdad Hicham**

directeur de recherche: **François Irigoin**

Centre de Recherche en Informatique
Ecole Nationale Supérieure des Mines de Paris

septembre 1999
Document EMP-CRI E/221/CRI

Résumé

Les programmes de calcul scientifique comme les équations différentielles, les interpolations spatiales et la programmation dynamique ... passent la majeure partie de leurs temps d'exécution dans des nids de boucles, ayant des dépendances régulières.

Une bonne localité des données et une utilisation efficace des ressources parallèles de la machine permettent d'obtenir de bonnes performances sur des boucles imbriquées.

Le partitionnement en supernoeuds est une technique de transformation de programmes, qui permet d'atteindre ces deux objectifs, sur un nid de boucles parfaitement imbriquées. L'objet de ce travail consiste à calculer un tiling optimal avec comme fonction objective le temps total d'exécution.

Nous présentons aussi, une technique d'ordonnancement des tuiles sur les processeurs en prenant en compte leur topologie physique.

abstract

Supernode partitionning is a transformation program technic , who allows data locality and efficient use of parallel ressources, over a perfectly nested loop.

The goal of this work is to compute an optimal tiling who gives the minimal execution time.

We present a scheduling method of tiles on processors with constraints on their physical topology.

Remerciement

Je tiens tout d'abord à remercier mon responsable de formation Monsieur le professeur Wiliam Jalby pour la confiance qu'il m'a accordé, ainsi que pour son soutien dans ma phase de recherche de stage de DEA.

Tout naturellement, je remercie du fond du coeur le directeur de mes recherches Monsieur le professeur François Irigoin pour m'avoir accepté au sein de son équipe sympathique, mais surtout pour m'avoir assisté tout au long de mon stage grâce à ses conseils importants et ses remarques pertinentes.

Je lui dois reconnaissance pour sa sympathie et la grande disponibilité dont il a fait preuve.

Je remercie également tout ceux qui ont collaboré à ce travail de près ou de loin, par leur présence, aide et encouragement; je pense particulièrement à madame Corine Ancourt, à monsieur Laurent Daverio et à monsieur Fabien Coleho. Une pensée chaleureuse à monsieur Jean Marie.

Je rend grâce surtout à mes parents, sans lesquels, ce travail, ni même ma présence à l'université n'aurait pu être possible.

Qu'Allah les protège, les garde et les récompense.

Chapitre 1

Introduction

1.1 Problèmatique

L'architecture des superordinateurs modernes présente plusieurs niveaux de parallélisme. Cette architecture doit absolument être prise en compte lors de la programmation, si l'on veut utiliser pleinement les possibilités offertes par ces machines, et ainsi obtenir de bonnes performances.

En effet, il faut d'une part saturer le parallélisme vectoriel ainsi que les différents processeurs élémentaires; d'autre part, assurer une bonne localité des références mémoires pour ne pas se trouver limiter par la bande passante de la mémoire principale; et finalement, garder un coût de synchronisation acceptable.

Les architectures des processeurs de plus en plus pérformante, il est vrai, deviennent très compliquées; du fait de l'introduction de plusieurs niveaux de parallélismes - i.e unités vectorielles, registres vectorielles, multiples processeurs, plusieurs niveaux de caches - ; ce qui nécessite aujourd'hui plus que jamais des outils automatiques puissants, pour faire profiter les utilisateurs de la puissance de ces processeurs. Ceci évite des optimisations manuelle très fastidieuses même pour des calculs matricieles simples.

Jusqu'à aujourd'hui encore, les solutions proposées pour masquer ce problème restent des bibliothèques de calculs scientifiques optimisées - monoprocesseurs et multiprocesseurs - adaptées à une machine donnée (exp: BLAS 1,2 et 3 , NAG, SCALAPACK ...) . Il existe aussi des préprocesseurs pour des compilateurs Fortran, qui explore le parallélisme d'un programme en découpant les boucles vectorielles pour en délivrer des trançons aux processeurs élémentaires; malheureusement, sans exploiter vraiment la puissance offerte par la machine. Les paralléliseurs automatiques donnent de bons résultats, mais dépendent fortement des techniques de transformation de programmes implémentées.

Le but est de concevoir une méthode qui assure aux processeurs élémentaires un travail en permanence. Ce travail devra être exécuté le plus efficacement possible par les unités vectorielles, qu'il s'effectue avec un minimum de données (bien utiliser la hiérarchie mémoire) , et gasspiller le moins de temps possible en synchronisations.

Pour satisfaire ces objectifs, il faut résoudre les trois problèmes suivants:

- 1) La constitution des taches -i.e partitionnement - .
- 2) L'affectation statique/dynamique des taches aux processeurs, en fonction des relations entre les taches d'une part, et des liens de communications entre processeurs d'autre part.
- 3) L'ordonnancement des taches, lorsque le nombre de taches exécutables en parallèle est supérieur à celui des processeurs.

Nous allons restreindre notre étude aux boucles parfaitement imbriquées, car ce type est fréquent dans les programmes de calcul scientifique. De plus, elles sont responsable de la majorité du temps d'exécution dans un programme.

Il existe néanmoins plusieurs techniques de transformation de programmes plus ou moins adaptées selon le type de machine utilisée et le code source exploré.

1.2 Optimisations monoprocesseur

Nous allons introduire quelques techniques de transformations de programmes sans pour autant vouloir être exhaustif.

Les techniques d'optimisation mono-processeur sont importantes, puisqu'il est inutile de vouloir gagner en performance avec un superordinateur si les taches monoprocesseurs ne sont pas optimisées.

Pour une optimisation manuelle efficace, il faut d'abord faire un *profiling*, ç.a.d repérer les parties du code qui consomment la plus part du temps processeur. Cette partie est souvent un nid de boucles, qu'il va falloir optimiser; sans oublier que les compilateurs ainsi que le matériel - i.e des superordinateurs - sont si compliqués que l'aide d'outils automatiques est indispensable.

Néanmoins, la part des optimisations manuelle est aussi à privilégier car elle concerne des domaines qui ne peuvent être assuré par le compilateur, comme par exemple, l'utilisation d'un bon algorithme -i.e un tri linéaire même optimisé ne peut être comparé au tri par tas , de même que l'intégrateur Rosenbruck dépasse celui de Range-kutta -, ou l'utilisation d'optimisation de haut niveau , voir même l'écriture en langage d'assemblage.

Les optimisations quelles soient manuelles ou automatiques sont de trois natures :

- 1) locales : - pour une procédure donnée -
  il faut privilégier l'utilisation des registres pour les résultats temporaires, voir pour les variables. Ainsi que le remplacement des multiplications et divisions -i.e coûteuses - par des *shift* et des *additions.*
- 2) globales : - aidé par le graphe des dépendances -
- 3) inter-procédurales.

Nous proposons maintenant de voir l'état de l'art en matière de transformation de programmes :

- **échange de boucles (** *loop interchange)*:

  Dans un nid de boucles de profondeur quelconque, il est possible de changer l'ordre des boucles de façon à avoir les boucles vectorisables à l'intérieur, et celles parallélisables à l'extérieur; ou tout simplement diminuer les références aux données - i.e bonne utilisation du cache -.

  Avant:

  ```
              do i = 1 , N
                  do j = 2 , M
  (S)                         A(i,j) = A(i,j-1) + B(i,j)
                  enddo
              enddo
  ```

  Après:

  ```
              do j = 2 , M
                  do i = 1 , N
  (S)                         A(i,j) = A(i,j-1) + B(i,j)
                  enddo
              enddo
  ```

  Cette technique doit absolument être précédée d'une analyse de dépendance -i.e construction d'un graphe de dépendance - car l'ordre d'exécution est évidement différent et risque d'altérer la sémantique du code.

- **Distribution de boucles (** *loop distribution)*:

  Une boucle qui contient plusieurs instructions peut être vectorisée en mettant chacune des instructions dans une boucle différente. Cette méthode est aussi utilisée pour réduire la taille des données, et ainsi augmenter la localité des données.

Avant:

```
        do i = 1 , N
(S1)        A(i) = A(i) + B(i)
(S2)        C(i) = A(i) * D(i)
        enddo
```

Après:

```
        do i = 1 , N
(S1)        A(i) = A(i) + B(i)
        enddo
        do i = 1 , N
(S1)        C(i) = A(i) * D(i)
        enddo
```

– (*strip mining*):

Lorsque la taille du code est supérieur à la ressource qu'on veut optimiser - i.e registre vectoriel, cache ... - , il est nécessaire de décomposer le traitement en plusieurs tranches qui puissent tenir dans le registre vectoriel (ou autres).

Avant:

```
        do i = 1 , N
(S)         A(i) = B(i) * C(i)
        enddo
```

Après:

```
    do si = 1 , N , 64
        do i = si , min( N , si+63 )
(S)         A(i) = B(i) * C(i)
        enddo
    enddo
```

– (*scalar expansion*):

L'utilisation de variables temporaires entraîne souvent des apparitions de dépendances, ces dernières sont éliminées par le remplacement des scalaires par un tableau temporaire.

Avant:

```
        do i = 1 , N
(S1)        tmp = B(i) * C(i)
(S2)        C(i) = C(i) / tmp
        enddo
```

Après:

```
        do i = 1 , N
```

```
(S1)                TMP(i) = B(i) * C(i)
(S2)                C(i) = C(i) / TMP(i)
        enddo
```

Ce n'est pas à proprement parler une optimisation, mais elle permet d'effectuer d'autres transformations .

– **déroulement de boucles (** *loop unrolling* **)**:

L'existence des pipelines peut réduire considérablement le temps d'exécution à condition de les saturer. Cette technique permet de transformer une boucle en une autre boucle comportant plusieurs instructions indépendantes exécutées à la chaîne.

Avant:

```
        do i = 1 , N
(S)                A(i) = B(i) + C(i)
        enddo
```

Après:

```
        do i = 1 , N ,3
(S1)                A(i) = B(i) * C(i)
(S2)                A(i+1) = B(i+1) * C(i+1)
(S2)                A(i+2) = B(i+2) * C(i+2)
        enddo
```

Chapitre 2

Tiling

2.1 Notions de base

Dans cette section, nous allons voir rapidement le contexte d'utilisation de cette méthode de transformation de programmes, et le but dans lesquels elle est appliquée. Enfin, nous donnerons les caractéristiques de cette technique.

2.1.1 Machines cibles et architectures

Les multiprocesseurs (machines à mémoire répartie - i.e à passage de messages -) offrent aujourd'hui de bonnes performances, mais souffrent du manque d'outils automatiques adéquats. En effet le paradigme de programmation par passage de messages employé pour ce type de machines est différent de celui habituel des machines à mémoire partagée. Il requiert que les processeurs communiquent entre eux en s'envoyant explicitement des paquets encapsulant données et autres informations de contrôle. Malheureusement, le coût d'établissement d'une communication - i.e start-up - est élevé, ce qui rend les communications fréquentes très pénalisantes (sur un Intel iPSC/860, le start-up est de $80\mu s$, et le transfert d'un réel double précision d'un noeud à un voisin prend $1\mu s$).

D'ou l'idée de regrouper plusieurs itérations et de les exécuter de façon atomique, les communications et synchronisations étant faites avant et après l'exécution, ce qui amortit le coût élevé du start-up car les messages sont plus gros. Or ceci implique un partitionnement judicieux et un ordonnancement efficace des calculs et des communications.

Un autre problème des machines, celui-ci concernant à la fois les machines à mémoire partagée et celles à mémoire distribuée, vient du fait que la différence toujours grandissante entre la vitesse des mémoires et celle des processeurs a conduit les concepteurs à l'utilisation d'une hiérarchie mémoire de plus en plus profonde. La majorité des machines aujourd'hui disposent d'au moins deux niveaux de caches, les mémoires locales servent de troisième niveau dans les grappes de stations de travail. S'il on prend en compte les registres on arrive à cinq niveaux hiérarchiques , et ceci devra s'accentuer dans les années à venir.

Le parallélisme au niveau de la mémoire n'est pas le seul à poser des difficultés pour la conception de programmes scientifiques ayant de bonnes performances, ou même de compilateurs paralléliseurs; en effet, toujours dans le but de gagner en puissance, les concepteurs ont - depuis longtemps - introduit du parallélisme au niveau du traitement. Il se traduit par une multiplication des processeurs, ou de celles des unités fonctionnelles internes à un processeur, voir des unités pipelinées traitant à la suite des données indépendantes.

Les deux derniers problèmes n'en sont pas à proprement parler. Ce sont deux grandes facilitées offerte qu'il va falloir utiliser, de manière à profiter de toute la puissance de traitement de ces machines.

2.1.2 Définition du tiling

Le tiling se définit donc comme une méthode de transformation de programme qui permet à la fois une bonne utilisation de la hiérarchie mémoire, en assurant une bonne localité des données, et la saturation du parallélisme vectorielle et des différentes unités de traitement.

La façon dont le tiling satisfait ces objectifs est décrite dans ce qui suit.

Le tiling est une agrégation d'un certain nombre d'itération d'une boucle - ou d'un nid de boucles parfaitement imbriquées - en un objet appelé *une tuile* , et qui sera exécuté **atomiquement** sur un processeur; cette opération de constitution des taches - i.e tuiles - s'appelle *partitionnement* . Le processeur qui va traiter une tuile n'exécutera les itérations de cette tuile que lorsque toutes les données nécessaires à l'exécution de toute la tuile seront disponibles. Une fois l'exécution terminé et seulement à ce moment la, le processeur pourra transmettre à son tour les données résultats aux autres processeurs qui en ont besoin.

Le partitionnement de l'espace d'itération en tuiles permet d'assurer une bonne localité des données dans la mesure ou chaque tuile correspond à un nombre de données qui pourra tenir dans la mémoire la plus rapide du processeur. Il permet aussi, d'avoir des tuiles indépendantes et concurrentes qui soient exécutables sur des processeurs différent à un instant donnée; sans oublier qu'à l'intérieur d'une tuile, quelques opérations peuvent être indépendantes, elles permettront ainsi de saturer le parallélisme vectorielle et celui des divers unités de traitements.

2.1.3 Remarques sur le tiling

On a vu que le tiling repose sur un partitionnement de l'espace des itération en des tuiles qui sont exécutées de façon atomique, les communications et synchronisations sont exclusivement faite avant et après l'exécution d'une tuile - i.e jamais pendant - . Or ce partitionnement peut conduire à une situation d'interblocage, qui se traduit par l'existence d'au moins un cycle dans le graphe de dépendance des tuiles, - remarquez que le temps totale d'exécution est définit par le plus long chemin dans ce graphe - . De plus, pour que la génération de code soit simplifiée, il est de règle d'avoir des tuiles ayant toutes une forme identique et le même nombre d'éléments, sauf les tuiles des bords de l'espace d'itération.

2.1.4 Analyse des dépendances

On a vu que pour effectuer une transformation de programmes qui modifie la structure du code en général, et le tiling en particulier, il faut d'abord faire une analyse profonde des dépendances entre les données. Il existe plusieurs types de dépendances et seuls quelques-unes nous intéresse ici:
soient S_x, S_y deux instructions pas forcément différentes,

- dépendance de flot :

  une dépendance existe de S_x à S_y ssi ces instructions sont dans l'ordre séquentiel suivant , tel que :

$$(S_x) \qquad A = ...$$

$$(S_y) \qquad ... = A...$$

- anti-dépendance :

$$(S_x) \qquad ... = A...$$

$$(S_y) \qquad A = ...$$

- dépendance de sortie :

$$(S_x) \qquad A = ...$$

$$(S_y) \qquad A = ...$$

L'analyse des dépendances permet aussi de construire le graphe de l'espace d'itération.

2.1.5 Graphe de l'espace d'itération

Pour un nid de boucles d'ordre k, avec un ensemble d'index $i_1, i_2, ..., i_k$,on représente les noeuds du graphe - i.e instructions - par des points dans l'espace cartésien de dimension k; il existe un arc entre l'itération définie par i_1 et celle de i_2 s'il existe une dépendance entre les instructions dans la boucle constituant l'itération i_1 et i_2.

A chaque noeud on fait correspondre un vecteur d'itération $\vec{j}$. Il a les valeurs des indices de boucles comme coordonnées.

Les contraintes de dépendances doivent être respectés lors de l'ordonnancement. On définit alors le *vecteur de dépendance* par

$$\vec{d} = \vec{j_2} - \vec{j_1}$$

et la matrice des dépendances par $D = [\vec{d_1}, ..., \vec{d_n}]$ .voir l'exemple ci-dessous:

(S)

```
do i = 2 , N
    do j = 2 , N-1
        A(i,j) = A(i-1,j+1) + A(i-1,j) + A(i,j-1)
    enddo
enddo
```

les vecteurs de dépendances sont:

$$\vec{d_1} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \vec{d_2} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \vec{d_3} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

et

$$\vec{d_1} = \begin{pmatrix} i - (i-1) \\ j - (j+1) \end{pmatrix}$$

2.2 État de l'art

On a vu que le problème principale dans la méthode du tiling consiste à trouver un partitionnement sans interblocage, de plus il doit être optimal dans le sens ou il permet d'exploiter la puissance de la machine. [Irig] montre que trouver les partitions revient à déterminer le cône pour un ensemble de vecteurs de dépendance; il présente une approche pour trouver un partitionnment en tuiles non bloquante, minimisant le volume des communications.

2.2.1 Définition d'un partitionnement

Le partitionnement de l'espace d'itération est la constitution des taches - i.e tuiles - qui seront exécutées par les processeurs. Les tuiles dans un espace k-dimentionel sont définit par k familles d'hyperplans parallèles, chaque hyperplan est de dimension $k-1$.[1]

On obtient ainsi des tuiles qui ont la forme d'un parallélpipède, et qui sont des sous-ensembles de l'espace de dimension k. Leur formes dépend des familles d'hyperplans et leur tailles - i.e nombre d'éléments - est définit par la distance entre deux plans consécutifs pour chaque famille.

Il est bon de rappeler ici que la forme et la taille des tuiles générées doivent être identiques pour toutes les tuiles - sauf près des bords -, ceci facilitent à la fois la génération du code et l'estimation du temps d'exécution totale.

Un tiling est alors complètement défini par une matrice $H \in Q^{n*n}$ qui divise l'espace d'itération en des parallélpipèdes de forme et taille identiques. Chaque ligne de H est une normale à une face du parallelipiped. il peut aussi être défini par une matrice $P \in Z^{n*n}$ ou les colonnes de P sont les arètes du parallélpipède[irig].

Une autre définition du tiling [xue] se fait par une matrice H et une fonction de mapping

$$\tau : Z^n \longrightarrow Z^{2n}$$
$$I \longrightarrow \begin{pmatrix} \lfloor HI \rfloor \\ I - H^{-1}\lfloor HI \rfloor \end{pmatrix}$$

où la première coordonnées est celle de la tuile , et la seconde celle du noeud dans la tuile par rapport à l'origine de celle-ci.

2.2.2 Légalité d'un partitionnement

La légalité d'un partitionnement est en rapport avec les vecteurs de dépendances. Intuitivement un tiling ne conduit pas à une situation d'interblocage si le graphe de dépendance des tuiles est un arbre, ou plus précisément une arborescence - i.e sans aucun cycle -. Formellement, si n un noeud et S un supernoeud:

$$\forall n_1, n_1' \in S_1, n_2, n_2' \in S_2 \neq S_1, \qquad n_1 \prec n_2 \Rightarrow \neg(n_2' \prec n_1')$$

ou $\prec$est la relation de précédence.

[Irig] apporte une condition suffisante pour satisfaire cette contrainte, il introduit la notion de cône de dépendance R.

Si l'on considère un espace d'itération d'ordre k, R déduit de $D = [\vec{d_1}, ..., \vec{d_n}]$ la matrice des dépendances et H tel que ses colonnes sont $\vec{h_1}, ..., \vec{h_k}$, ou le vecteur $\vec{h_i} \in Q^N$ définit un hyperplan; alors le partitionnemnt est légal ssi

1. dans un espace à deux dimensions - i.e k=2 -, l'hyperplan est alors une ligne.

$$\forall i = 1..k, \qquad h_i^T.R \geq \vec{0}$$

Ou ce qui est équivalent:

$$H.R \geq 0$$

[irig] montre que pour avoir un partitionnement en supernoeuds identiques et bornés il faut avoir H libre et générateur de l'espace d'itération.

En conclusion, pour trouver un partitionnement d'un espace d'itération de dimension N respectant les contraintes précitées, il faut trouver une matrice H formant une base, avec les $\vec{h_i}$ valides - i.e $\forall i = 1..N, \qquad h_i^T.R \geq \vec{0}$.

Ce qui revient à calculer une matrice H dont les lignes sont les normales aux faces des parallélpipèdes, qui satisfait ces conditions:

- H non singulière - i.e det(H) non nul -.
- $H^{-1} \in Z^{nxn}$ intégral.
- $HD \geq 0$ - i.e condition suffisante pour respecter les dépendances -.

A noter que l'inverse du déterminant de H correspond au volume d'un supernoeud[irig].

Il est donc possible de définir un partitionnement par une base P. Deux noeuds ayant des coordonnées identiques dans cette base sont dans le même supernoeud:

$$\vec{j_1}, \vec{j_2} \in S \longleftrightarrow \lfloor \vec{j_1^P} \rfloor = \lfloor \vec{j_2^P} \rfloor$$

[irig] montre qu'un partitionnement par une base P est valide ssi $P^{-1}D \geq 0$, - il suffit d'avoir $P^{-1}R \geq 0$ -. Dans ce cas il équivaut à un partitionnement par hyperplan H ou $H^t = P^{-1}$.

[xue] part du fait que la condition 3 de [irig] est une condition suffisante, et trouve une qui soit nécessaire: $\lfloor HD \rfloor \geq 0$. mais le problème est que cette formule est non linéaire, on ne pourra donc pas l'utiliser par la suite.

Il est à noter ici, que la matrice H n'est pas unique, ce qui s'explique par l'existence de nombreux partitionnements légaux possibles. D'ou l'existence du problème de trouver l'optimum.

2.2.3 Choix d'un partitionnement minimisant les communications

Trouver un partitionnement optimal revient à définir une forme et une taille aux supernoeuds ainsi générés. La taille des tuiles doit être suffisamment petite pour tenir dans la mémoire la plus rapide.

Le nombre de tuiles doit être suffisamment grand pour pouvoir assurer un bon équilibrage de charge des différents processeurs - supposé infini -, mais aussi petit que possible pour garder un coût de synchronisation et de transfert acceptable.

[Ram] montre qu'un partitionnement peut être définit soit par des vecteurs définissant les hyperplans, soit par des extrêmes vecteurs puisque ces derniers sont les colonnes de la matrice H^{-1}. A noter qu'ici le vecteur $\vec{h_i}$ est la i^{eme} ligne de H.

[Ram] propose un algorithme - basé sur la programmation linéaire en nombres entiers - en $O(n)$ qui calcule une matrice H triangulaire inférieure à déterminant 1 ou -1, ayant la forme - pour une dimension 3 -

$$H = \begin{pmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & c & 1 \end{pmatrix}$$

avec $HD \geq 0$
et qui minimise le volume des communications:
$\sum \sum \sum H_{i,k} D_{k,j}$
[xue] montre que trouver un tiling minimisant les communications est équivalent à la résolution du problème de programmation linéaire suivant:

minimiser le volume de communication d'une tuile

$$V_{com} = v. \sum_{i=1}^{n} \sum_{j=1}^{m} (HD)_{ij}$$

sujet à: $v = \frac{1}{det(H)}$ et $HD \geq 0$

Il montre aussi que dans le cas particulier de $D = I_d$

on a $H_{opt} = diag(1/v^{1/n}, ..., 1/v^{1/n})$ et
$P_{opt} = diag(v^{1/n}, ..., v^{1/n})$
ainsi que d'autres tilings pour des cas particuliers de D.

2.3 Calcul d'un partitionnement optimal

Le problème de déterminer un tiling optimal est complexe, car il faut prendre en compte une multitude de variables. En effet, la taille des tuiles détermine son temps d'exécution mais aussi le volume des communications qu'il faut minimiser, si l'on prend aussi en compte la forme des tuiles on détermine alors le chemin critique des tuiles, qui lui est en relation étroite avec le temps totale d'exécution du code.

On prend par la suite l'hypothèse d'une machine ayant un nombre très grand de processeurs, ce qui nous permet de calculer le temps total d'exécution. Si on considère que c est la longueur du chemin critique, et t_e le temps d'exécution d'une tuile, v_{com} le temps de transfert du volume des communications, alors:

$$T_{total} = (1 + c) * t_e + c * v_{com}$$

Le temps d'exécution d'une tuile est une fonction de son volume:

$$t_e = cte * \frac{1}{det(H)}$$

et la constante est connue a l'avance par expérimentation sur la machine cible.

Le volume des communications d'une tuile est égale à

$$\frac{1}{det(H)} * \sum_{ij} (HD)_{ij}$$

On voit alors que pour trouver un tiling optimal, il faut à la fois minimiser le volume de transfert et le chemin critique.

Formellement, résoudre le problème linéaire:
chercher $H \in Q^{n*n}$ qui

minimise $\frac{cte*(1+c)}{det(H)} + c * \frac{\sum_{ij}(HD)_{ij}}{det(H)}$

sujet à:

$$HD \geq 0$$

$$H^{-1} \in Z^{n*n}$$

Ce problème linéaire n'est pas complet, car c est une fonction de H qu'il faudra trouver.

2.3.1 Calcul du plus long chemin

Pour calculer le chemin critique, nous allons utiliser un certain nombre de résultats:

résultat 1 *H définit un tiling $\Rightarrow \forall \vec{j}$ noeud $\in S$ tuile, alors $\lfloor H\vec{j} \rfloor$ sont les coordonnées de S dans le nouveau repère.*

résultat 2 $\forall \vec{d} \in D, \exists \vec{j} \in S, \vec{j'} \in S'$ *tel que* $\vec{j} + \vec{d} = \vec{j'} \Leftrightarrow$

$\exists \vec{d'} \in D'$ *entre S,S' tel que* $S + \vec{d'} = S'$

et $D' = \lfloor HD \rfloor$.

Maintenant que nous avons calculé les dépendances entre les tuiles, nous pouvons calculer le plus long chemin c dans le graphe des tuiles, pour un partitionnement H donnée.

Soit les indices des boucles $(i_1, i_2, .., i_k)$, et les bornes de début et fin d'un indice i_j : (i_{jD}, i_{jF}) - i.e $i_{jD} \leq i_j \leq i_{jF}$- .

Alors on a:

$$A = H * ((i_{1F}, i_{2F}, .., i_{kF}) - (i_{1D}, i_{2D}, .., i_{kD}))$$

la distance entre le point de départ et celui de la fin du nid de boucles exprimé en nombre de tuiles dans chaque dimension.

Alors, la longueur d'un chemin quelconque s'exprime par $c' = \sum_{i=1}^{k} \alpha_i$ avec $\alpha_i \in N$ tel que

$$A = \sum_{i=1}^{k} \alpha_i * \vec{d'_i}$$

Donc, le plus long chemin $c = max(c')$ peut être calculé par:

maximiser $c = \sum \alpha_i$

sujet à:

$$A = \textstyle\sum_{i=1}^{k} \alpha_i * \vec{d'_i}$$

Il faut noter que la longueur du chemin critique est exprimé en nombre de dépendances entre tuiles, et non en nombre de tuiles, ce qui fait que lors du calcul du temps total d'exécution, on ajoute un au facteur du temps d'exécution d'une tuile.

2.3.2 Exemple

Soient un nid de boucles d'ordre deux avec les dépendances

$$(\vec{d_1}, \vec{d_2}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

et les indices des boucles i, j qui vérifient:

$$1 \leq i \leq N, \qquad 1 \leq j \leq M.$$

Supposons que nous ayons calculé un tiling:

$$H = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Alors on peut exprimer la distance A en nombre de tuiles, entre le point de calcul initiale $(1,1)$ et le dernier point (N,M) dans chaque dimension:

$$A = H * (N-1, M-1)^T = \begin{pmatrix} N-1 \\ M-1 \end{pmatrix}$$

Le calcul des dépendances entre tuiles donne $D' = \lfloor H * D \rfloor = D$. Donc le calcul du plus long chemin se traduit par :

maximiser $c = \alpha_1 + \alpha_2$ tel que

$$A = \begin{pmatrix} N-1 \\ M-1 \end{pmatrix} = \alpha_1 * \vec{d'_1} + \alpha_2 * \vec{d'_2}$$

ce qui donne directement: $\alpha_1 = N-1, \qquad \alpha_2 = M-1$, c'est à dire $c = N+M-2$

2.4 Génération de code

Les paralléliseurs automatiques génèrent du code pour multiprocesseurs.[Anc] leur ajoute une phase supplémentaire qui tient compte de la hiérarchie mémoire. Ceci en générant un code de transfert des données de la mémoire globale vers la mémoire locale du processeur qui utilisera ses données, puis après exécution - i.e code de calcul -, un autre code de transfert allant cette fois de la mémoire locale à la globale.

Ceci revient à calculer la projection d'un polyèdre (domaine d'itération) dans une nouvelle base. Le polyèdre résultant peut ne pas être connexe, il est alors approchés par celui connexe, minimal qui l'englobe. Cette approximation ne peut se faire lors du transfert des résultats vers la mémoire globale pour des rasions de cohérence.

Ceci traduit toute la difficulté de la génération de code dans le tiling, c-a-d que chaque processeur ne doit transférer que les données utilisées et uniquement celles-ci.

Nous pouvons alors utilisés les résultats de [Anc] dans le transfert des résultats entre processeurs voisins. Ceci est important puisqu'on cherche à minimiser le coût des transferts; le choix de la base de parcours des références du tableau est donc primordial.

On cherche donc à générer un code de transfert de la forme:

do i = ...
    do j = ...

transmetre $T(\varphi(i,j,..))$ vers $T'(\varphi'(i,j,..))$

    enddo
enddo

Soient F la matrice représentant la fonction d'accès φ au tableau, et $\vec{f_0}$ sa partie constante, - i.e , pour T(2i+j-5,3j+1), on a :

$$F = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}, \vec{f_0} = \begin{pmatrix} -5 \\ 1 \end{pmatrix}$$

Pour toute matrice F, il existe deux matrices unimodulaire P et Q tel que: $PFQ = H$, avec H forme réduite de Hermite associé à P. On a alors d'après [Anc]

$$\varphi = P^{-1}HQ^{-1}\vec{j} + \vec{f_0}$$

avec $Q^{-1}\vec{j} = \vec{t}$, ou $\vec{j}, \vec{t}$ sont les vecteurs d'itération dans l'ancienne et la nouvelle base.

2.5 L'ordonnancement des tuiles

Une fois le tiling optimal - i.e minimisant le temps total d'exécution - déterminé, et le chemin critique calculé, on peut choisir un ordonnancement qui minimise les transferts entre processeurs ainsi que le temps total d'exécution, sous la condition d'un nombre illimité de processeurs.

En effet, les tuiles qui forment le chemin critique doivent être exécutées par le même processeur, ce qui élimine les coûts de transfert des résultats intermédiaires, et ainsi permet d'obtenir un temps d'exécution total minimal. Ceci est vrai dans le cas ou le processeur critique - i.e exécutant les tuiles du chemin critique - n'ait pas à attendre les résultats des autres processeurs. On voit donc qu'il faut prendre en compte aussi la topologie physique des processeurs.

Nous avons donc choisi un ordonnancement en dimension 2-D et 3-D qui tient compte de la topologie physique des processeurs, et qui conduit à un temps d'exécution minimal, voir la figure 2.1.

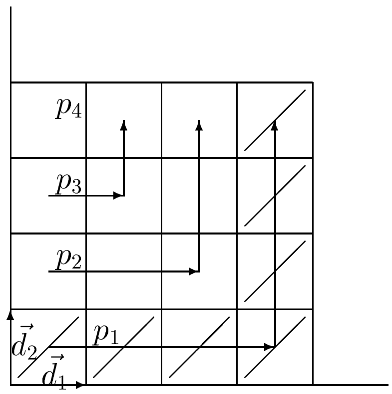

FIG. 2.1 – *ordonnancement des tuiles sur 4 processeurs*

Sur la figure 2.1, le chemin critique C est celui qui est hachuré, ses tuiles sont exécutées par le processeur critique p_1.

La figure 2.2 montre les dates de lancement des tuiles à l'exécution sur les différents processeurs.

Le processeur p_2 qui doit être un voisin physique de p_1 - i.e pour minimiser les coûts de transfert - exécute le second chemin critique[2] C_2 - i.e $\forall C'$ second chemin, $c > c_2 > c'$ -.

2. Un second chemin part du nouveau point de départ au nouveau point final. le nouveau point de départ est la première tuile perte à l'exécution et qui n'appartient pas au chemin critique. - i.e la tuile ayant les coordonnées (1,2) -

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
p_1	*	*	*	*	*	*	*
p_2		*	*	*	*	*	
p_3			*	*	*		
p_4				*			

FIG. 2.2 – *dates de lancement des tuiles sur les 4 processeurs*

Remarque:

Pour que la condition de non-attente du processeur p_1 soit vérifiée, il faut que les taches soient exécutées au *plus tôt* au sens donné en terminologie d'ordonnancement .

On peut généraliser ce processus pour l'algorithme suivant:

algorithme d'ordonnancement

données: H optimal.

résultat: Ordonnancement optimal.

- 1. déterminer le chemin critique C, et sa longueur c.
- 2. lancer l'exécution des tuiles de C sur le processeur p_1.
- 3. déterminer les nouveau points de départ d, et ceux finaux f.
- 4. déterminer pour chaque (d, f) le second chemin critique C_2, et sa longueur c_2.
- 5. lancer l'exécution au plus tôt des tuiles de C_2 sur un processeur p_2 voisin de p_1.
- 6. recommencer en 3 en remplaçant p_1 par le dernier processeur utilisé.

exemple:

soit le graphe des tuiles de la figure 2.1, pour lequel, nous ne considérons que les trois premières lignes et colonnes pour simplifier. Les dépendances entre tuiles sont

$$D' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Cet exemple illustre l'algorithme informel d'ordonnancement présenté ci-dessus.

calcul des points d et f:
on a $d = (1,1)$ et $f = (3,3)$ d'après le code.

calcul de C :
$(3,3) - (1,1) = (2,2) = \alpha\vec{d_1} + \beta\vec{d_2}$
d'ou $(\alpha, \beta) = (2,2)$ donc $c = \alpha + \beta = 4$
alors le chemin critique C est $2\vec{d_1} + 2\vec{d_2}$

ordonnancement sur P_1 :
les tuiles qui seront ordonnancées sur p_1 sont :
$d, d + \vec{d_1}, d + 2\vec{d_1}, d + 2\vec{d_1} + \vec{d_2}, d + 2\vec{d_1} + 2\vec{d_2}$
c-a-d: $(1,1), (2,1), (3,3), (3,2), (3,3)$

calcul des points d et f:
on a $d = d + \vec{d_2}$, on choisit $\vec{d_2}$ car $d + \vec{d_1} \in C$.
et $f = f - \vec{d_1}$
donc $d = (1,2), f = (2,3)$.

calcul de C_2 :
$(2,3) - (1,2) = \alpha\vec{d_1} + \beta\vec{d_2}$
d'ou $(\alpha, \beta) = (1,1)$ donc $c_2 = \alpha + \beta = 2$
alors le second chemin critique C_2 est $\vec{d_1} + \vec{d_2}$

ordonnancement sur P_2 voisin de p_1 :
les tuiles qui seront ordonnancées sur p_2 sont :
$d, d + \vec{d_1}, d + \vec{d_1} + \vec{d_2}$
c-a-d: $(1,2), (2,2), (2,3)$

calcul des points d et f:
on a $d = d + \vec{d_2}$, on choisit $\vec{d_2}$ car $d + \vec{d_1} \in C_2$.
et $f = f - \vec{d_1}$
donc $d = (1,3), f = (1,3)$.

calcul de C_3 :
$(1,3) - (1,3) = (0,0)$ donc $c_3 = \alpha + \beta = 0$

alors le second chemin critique C_3 est nul.

ordonnancement sur P_3 voisin de p_2 :
les tuiles qui seront ordonnancées sur p_3 sont: $d = (1, 3)$

fin
car $d = f$ et il ne reste plus de points qui ne soit pas ordonnancés.

remarque :

La topologie idéale pour cet exemple est bien sûr un bus, ou toute topologie pouvant le simuler comme l'anneau, le tore, ou l'hypercube.

Il est a noter que cet algorithme fonctionne aussi pour des dimensions supérieures, seulement en dimension 3 par exemple, chaque processeur devra avoir deux processeurs voisins libres, ce qui fait penser à un arbre binaire.

2.5.1 Nombre maximal de processeurs

Nous avons supposé tout le long de ce travail que le nombre de processeur était illimité, évidement cette condition à elle seule est irréaliste. On peut facilement calculer le nombre de processeur maximal dans le cas de l'ordonnancement présenté à la section précédente.

La notion de maximalité ici est floue, car ce nombre ne peut être dépassé sans rajouter des coûts de transferts inutiles et donc diminuer le temps d'exécution; elle ne peut non plus pas être sous-estimé sans limiterer le parallélisme induit et donc augmenter le temps d'exécution total.

Dans l'hypothèse de notre ordonnancement, le nombre de processeurs maximal utile est

$$min(N', M')$$

avec N' nombre de tuiles sur la première dimension, et M' celui sur la dimension deux.
Ce qui donne pour l'exemple au-dessus, le nombre de processeurs utiles est

$$min(4, 4) = 4$$

Il est bon de rappeler ici, que

$$\begin{pmatrix} N' \\ M' \end{pmatrix} = \lfloor H * \begin{pmatrix} N \\ M \end{pmatrix} \rfloor$$

Chapitre 3

Cas d'école

3.1 Versions de base

Soit l'exemple de nid de boucles parfaitement imbriquées suivant :

```
          do i = 2 , N
                do j = 2 , M
(S)                        A(i,j) = ( A(i-1,j) + A(i,j-1) )* 0.5
                enddo
          enddo
```

Ce code comporte des dépendances de données, car l'instruction (S) dépend d'elle même ;

en effet l'analyse de dépendances révèle l'existence de deux vecteurs de dépendances :

$$\vec{d_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \text{et} \qquad \vec{d_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Les performances en temps d'exécution sont alors influencées. De plus ces dépendances empêchent une parallélisation efficace.

Fort heureusement elles sont régulières comme le montre la figure 3.1 [1] .

Le but de ce qui suit est de modifier ce programme de façon à réduire l'influence des dépendances, et ainsi obtenir des performances acceptables.

Ce code peut dans un premier lieu, être écrit de trois façons différentes, tout en respectant les dépendances :

1. les petits cercles représentent les instances de (S) , et les flèches, les flots de données.

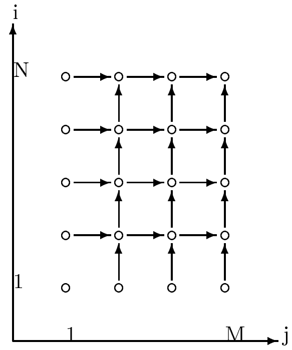

FIG. 3.1 – *Graphe des dépendances*

version ij :

C'est exactement le code donné plus haut ; ici le calcul s'effectue par lignes, voir la figure 3.2 .

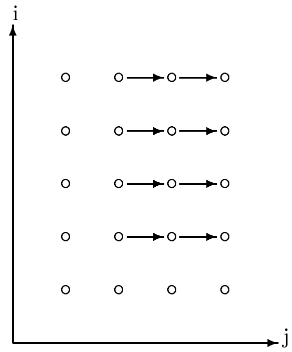

FIG. 3.2 – *ordre de calcul de ij*

version ji :

Le calcul s'effectue par colonnes, ce qui correspond au mode de stockage des données par le langage Fortran; ainsi, cette version devrait être la plus performante, voir la figure 3.3.

```
do j = 2 , M
        do i = 2 , N
                A(i,j) = ( A(i-1,j) + A(i,j-1) )* 0.5
        enddo
```

enddo

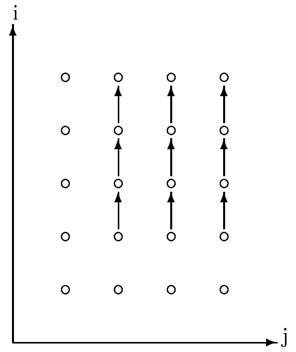

FIG. 3.3 – *ordre de calcul de ji*

version tp :

Cette version est une modification de la version **ij** selon le changement de base suivant, voir la figure 3.4:

$$\begin{cases} \vec{p} = \vec{i} - \vec{j} \\ \vec{t} = \vec{j} \end{cases}$$

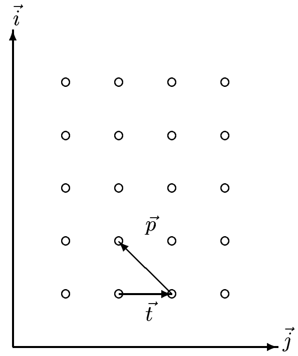

FIG. 3.4 – *changement de base*

ce qui donne le changement de variables[2] :

2. (x, y) exprimé dans la base $(\vec{i}, \vec{j}) \Rightarrow (x, y) = x\vec{i} + y\vec{j} = x\vec{p} + (x + y)\vec{t} \Rightarrow (x, x + y)$ sont ses

$$\left\{ \begin{array}{c} p = i \\ t = i + j \end{array} \right.$$

Ici le calcul s'effectue par diagonales, voir la figure 3.5, ce qui est à priori pénalisant .

```
do t = 4 , N+M
        do p = max(2,t-M) , min(N,t-2)
                A(p,t-p)= ( A(p-1,t-p) + A(p,t-p-1) )* 0.5
        enddo
enddo
```

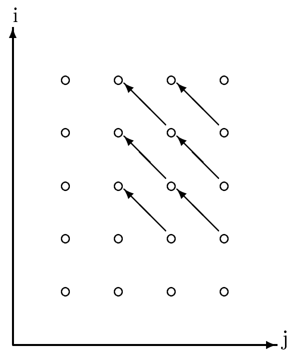

FIG. 3.5 – *ordre de calcul de tp*

Comme le calcul se fait par diagonales, et qu'il n'y a pas de dépendances entre les éléments d'une même diagonale, alors ces calculs peuvent être fait en parallèle, voir la figure 3.5 .

Pour générer le code de la version **tp** - voir la figure 3.6 - ,il faut calculer les bornes des indices de boucles t et p , $t = i + j$ implique que le calcul de cet indice est directe , par contre celui de p se déduit des droites qui définissent le polyèdre convexe résultant.

coordonnées dans la nouvelle base $(\vec{p}, \vec{t})$.

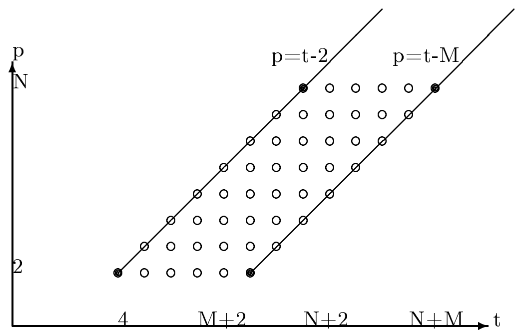

FIG. 3.6 – *polyhedre convexe*

3.2 Expérimentations

Il s'agit ici de faire une expérimentation sur trois versions de notre programme, avec $N * M$ constant et petit de façon à ce que le tableau tienne dans le cache L1. Ceci évite de prendre en compte d'autres critères comme la taille des pages..

La simulation a été effectué sur une station de travail monoprocesseur Sun Ultra Sparc II à 267 Mhz, ayant 16 Koctets de cache L1. De plus cette station possède deux unités d'exécutions flottantes pipelinées - i.e addition et multiplication - ayant chacune un débit d'un résultat par cycle d'horloge, et d'une latence de 4 cycles;

Sa performance théorique est donc de l'ordre de 267*2=534 Mflops.

Les codes sont compilés avec un compilateur Fortarn SUNWspro/SC5.0 avec l'option **-O4** pour les optimisations.

Les résultats obtenus - voir figure 3.7 - montrent que la version **ij** est la moins bonne, et la version **ji** est la plus performante.

L'analyse du code assembleur généré a montré que ceci s'explique par le fait que le compilateur Fortran - uniquement dans cette version **ji** - assure une bonne localité spatiale des références aux données - i.e registres -, ceci permet de n'effectuer qu'un load au lieu de deux dans les cas **ij , tp**.

Aussi, la version **tp** est plus rapide que **ij** car elle offre plus de parallélisme, donc les calculs peuvent être pipelinés si le *renaming* des registres est assuré par le processeur Ultra sparc II, comme on le voit sur les codes assembleur de la table 3.1.

Le compilateur n'assurant pas la localités des données, nous avons essayé de faire les optimisations nécessaires à la main - malheureusement - en rajoutant un scalaire qui sauvegarde la donnée calculé et nécessaire à l'itération suivante, ceci

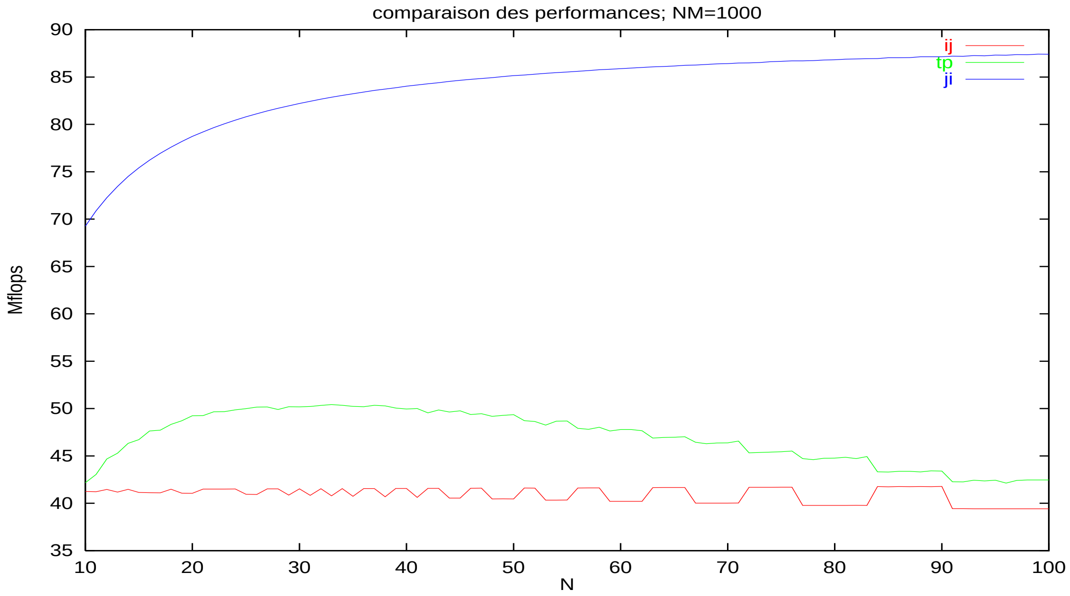

FIG. 3.7 – *faiblesse de la version* **ij**

permet de ne faire qu'un load au lieu de deux; cette optimisation est faite pour les deux versions **ij** et **ji** pour voir les gains en performances apportés.

Les codes de ces versions **ij2** et **ji2** ainsi que tous les autres sont à l'Annexe , et les résultats des performances sur la figure 3.8.

Il faut noter que le calcul des dépendances n'est pas effectuer par défaut dans l'option **-O4**, ce qui explique que le *renaming* des registres n'est pas assuré par le compilateur. Autrement dit, aucun load ne peut être fait, tant que le store n'est pas terminé (voir la table 3.1). Ceci explique donc, la faiblesse de la version **tp** qui ne peut profiter d'un haut degré de parallélisme.

En effet, pour que le compilateur effectue une analyse profonde des dépendances, et ainsi pouvoir faire des optimisations intéressantes, il faut compiler avec l'option **-depend**, que nous associons à l'option habituelle **-O4**. Les résultats obtenues sont spectaculaires, voir la figure 3.9.

On remarque la similitude entre les versions **ij, ji** ; ceci est normal, puisque le compilateur effectue un changement de boucles pour **ij** ; Quand à la version **tp**, elle profite de l'exécution pipelinés de ses instructions.

ji	tp	ij
ld [],f1	ld [],f3	ld [],f3
fadds f0,f1,f0	ld [],f0	ld [],f0
fmuls f0,f2,f0	fadds f0,f3,f0	fadds f0,f3,f0
st f0,[]	fmuls f0,f2,f0	fmuls f0,f2,f0
	st f0,[]	st f0,[]
	ld [],f3	ld [],f3
	ld [],f0	ld [],f0
	fadds f0,f3,f0	fadds f0,f3,f0
	fmuls f0,f2,f0	fmuls f0,f2,f0
	st f0,[]	st f0,[]

TAB. 3.1 – *codes assembleur générés*

Nous avons aussi remarqué au cours de ce travail, que l'option de compilation **-fast** conduit à des résultats erronés, puise qu'elle ne calcule aucune itération du nid de boucles; et ceci même si on demande un affichage des résultats après le calcul.

Néanmoins, une question se pose, l'Ultra Sparc II peut effectuer deux instructions par cycle, et dans notre cas seule la version **tp** se prête bien à ce type de parallélisme.

Peut-on conclure qu'une bonne utilisation de la mémoire la plus rapide - i.e registres - est plus importante que l'utilisation du parallélisme des unités de calculs? On est tenté de répondre oui, puisque nous somme dans le cache pour les trois versions.

3.3 Versions bloqués

Nous avons jusqu'à présent vu trois versions différentes de notre programme, ces versions sont différentes quand à leurs performances malgré les similitudes entre les versions **ij** et **ji** .

Chacune de ces versions peut être partitionée de façon à ce que les partitions constituées soient suffisamment petites pour tenir dans la mémoire la plus rapide .

Le partitionnement étant rectangulaire, il s'agit de déterminer la taille idéale de ce dernier qui offre les meilleurs performances.

Chaque version donne alors trois sous-versions , dans la mesure ou nous avons le choix dans le calcul à l'intérieur d'un supernoeud. Nous avons donc :

version ij bloqué en ij:

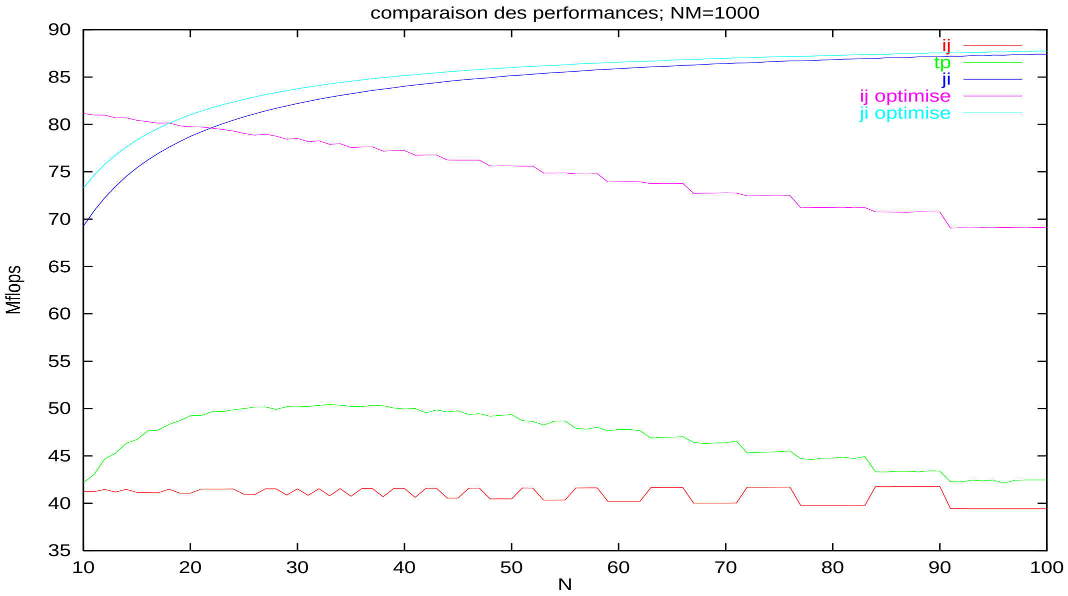

FIG. 3.8 – *faiblesse de la version* **tp**

Ici, l'ordre d'exécution des supernoeuds est en ligne, et dans chaque supernoeud , le calcul se fait aussi par ligne, voir la figure 3.10 pour remarquer l'ordre d'exécution des supernoeuds - i.e de 1 à 8 -, ainsi que celui des éléments d'un même supernoeud - i.e sens des flèches -.

Il faut aussi noter que les supernoeuds des bords n'ont pas toujours la même forme que les autres.

version ij bloqué en ji:

c'est une version similaire à la précédente sauf que l'ordre d'exécution des éléments internes est en colonne cette fois - i.e d'ou le **ji** - .

Les autres versions suivent le même raisonnement , comme leurs noms l'indiquent :

version ji bloqué en ij .

version ji bloqué en ji .

version tp bloqué en ji .

version tp bloqué en tp .

Remarquez que d'autres possibilités existent, mais ces versions ne nous intéressent pas pour des raisons de similitudes des performances - voir plus loin - .

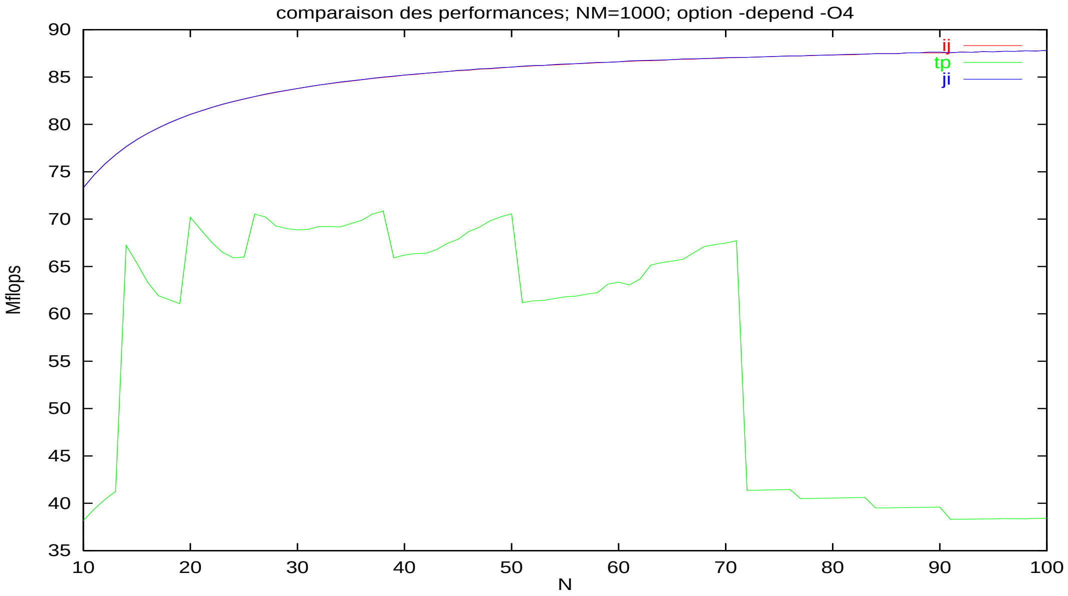

FIG. 3.9 – *résultat après analyse des dépendances par -depend*

3.4 Expérimentations

L'expérimentation a été effectuée dans les mêmes conditions qu'auparavant, en plus pour les versions bloquées, la taille du pavage varie.

Ceci nous permet de voir qu'elle est la taille de pavage qui assure la performance maximale, et ainsi de comparer la meilleur version bloquée avec la meilleur non bloquée.

Ce résultat devrait nous donner une idée sur le gain ou sur la perte induite par le partitionnement .

Donc , avec les codes produits (voir les sources dans l'Annexe) et testés sur un Ultra Sparc II a 267Mhz et 16Koctets de cache L1, on s'est intéressé aux pavages - i.e B1, B2 - qui donnent la performance maximale.

Dans la suite, seule cette performance nous intéresse.

La figure 3.11 donne les résultats des expérimentations pour les versions **ij, ji bloquées** en ij et ji. On remarque alors que les meilleurs performances sont pour celles qui sont bloquées en ji.

Ceci s'explique par le fait que la taille du pavage optimal est presque égale à celle de N - i.e longeur de la première dimension du tableau -, voir les figures 3.12 et 3.13.

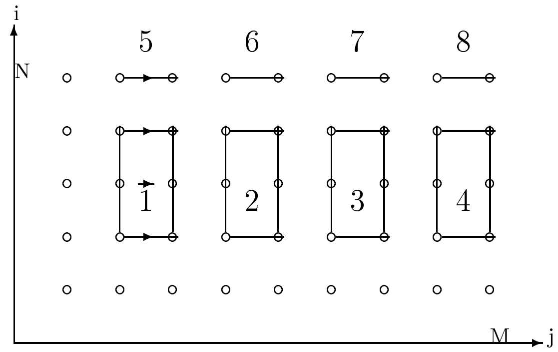

FIG. 3.10 – *Partitionnement possible,B1=3,B2=2*

La figure 3.15 compare les résultats de la meilleur versions bloquée et la moins bonne vus à la figure 3.11, avec les versions **tp bloquées**.

On constate que la version **ij bloquée en ji** est la meilleur de toutes. Il faut aussi ne pas oublier que la version **ji bloquée en ji** lui est très proche.

La figure 3.14 montre les tailles des pavages optimaux pour les versions **tp bloquées**.

3.5 conclusion

Après avoir testé les différentes versions d'un même nid de boucles parfaitement imbriquées, on constate que la meilleur version bloquée **ij bloquée en ji** est légérement plus performante que la meilleure version non bloquée qui est **ji**.

Cette conclusion à elle seule nous incite à croire en la technique du tiling comme moyen de gagner en performance sur un multiprocesseur.

Il reste maintenant à choisir la taille idéale du pavage - i.e B_1, B_2 - qui dans notre exemple a été trouvé par expérimentation.

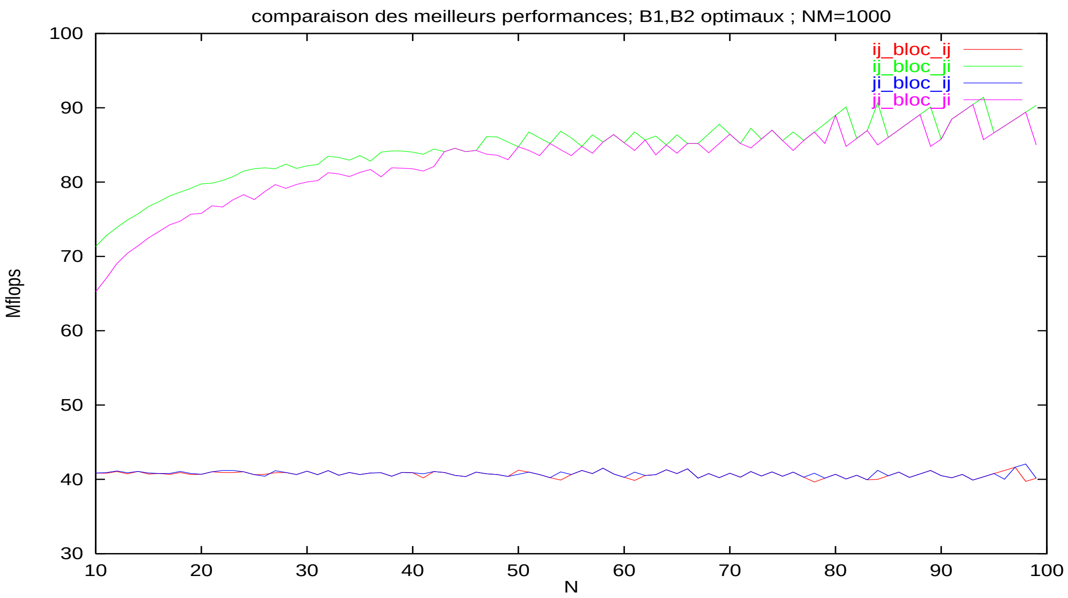

FIG. 3.11 – *similitudes entre versions bloquées*

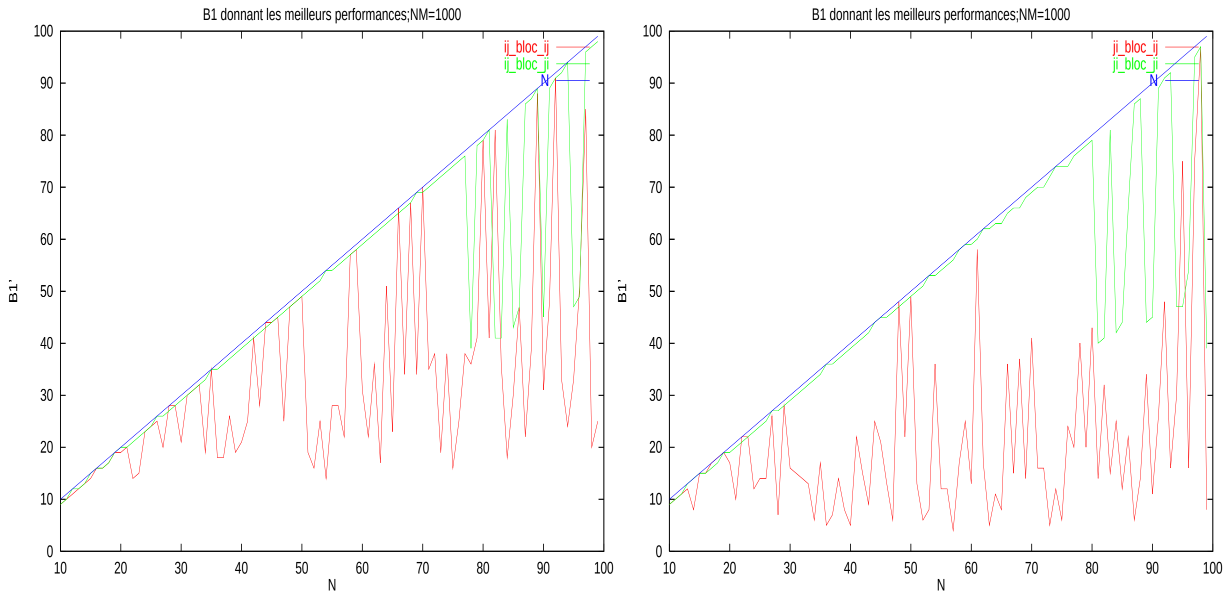

FIG. 3.12 – *similitude de la taille du pavage optimal avec* N

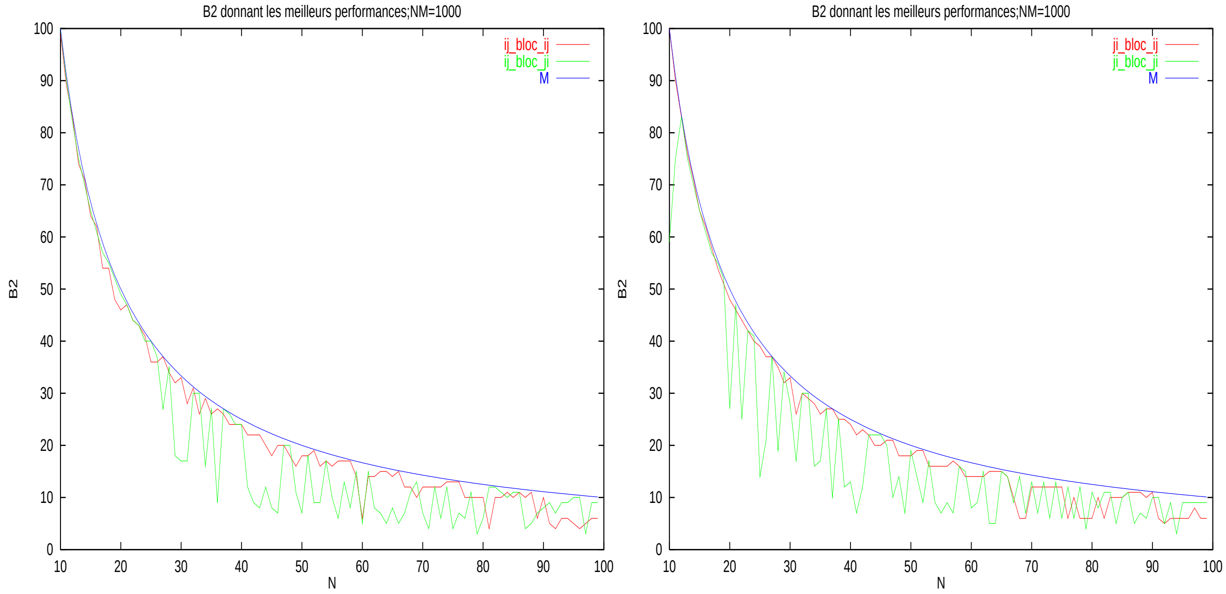

FIG. 3.13 – *similitude du pavage maximisant les performances*

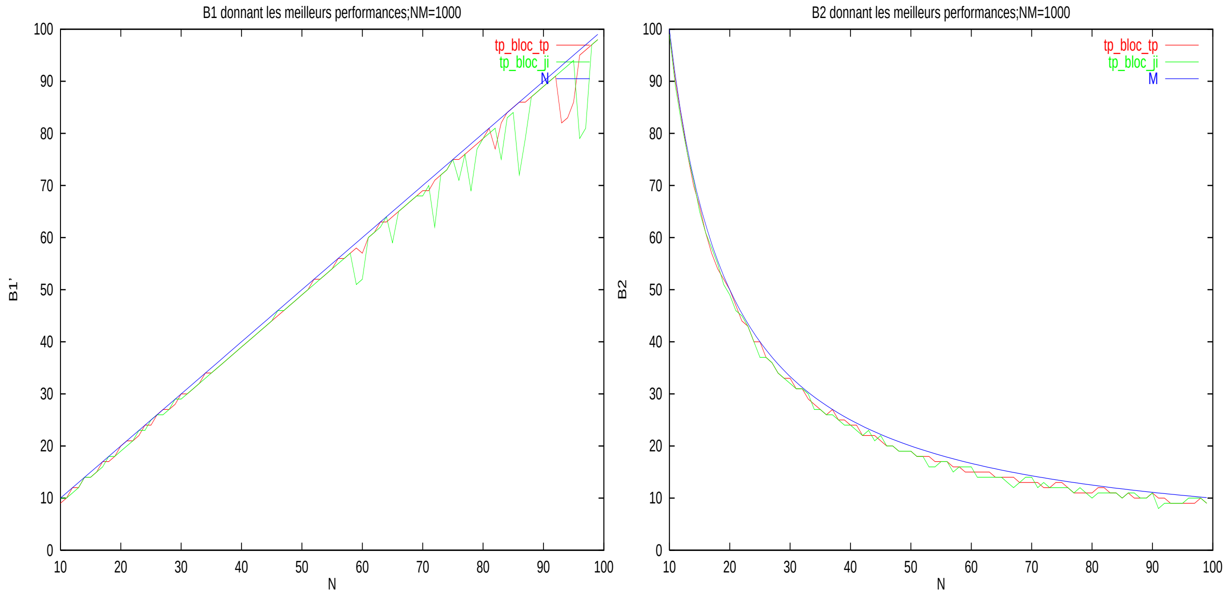

FIG. 3.14 – *similitude du pavage maximisant les performances*

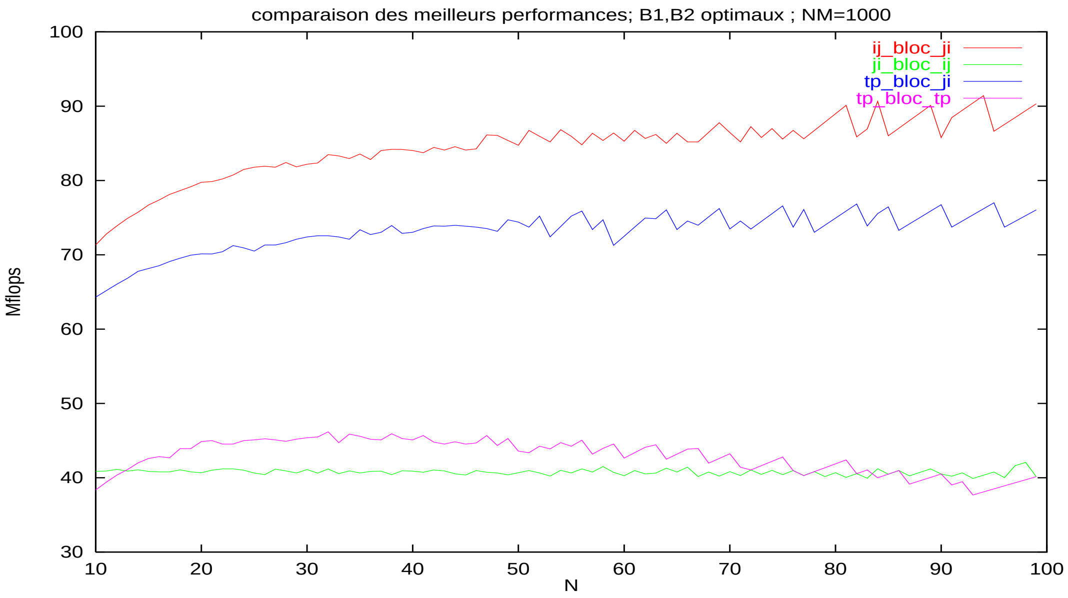

FIG. 3.15 – *similitudes entre versions bloquées*

Chapitre 4

Synthèse

Ce chapitre constitue une synthèse de tous les résultats cités dans les chapitres ultérieurs de façon superficielle. A savoir, le calcul du tiling optimal avec comme fonction économique, le temps d'exécution total, et la proposition d'un ordonnancement topologique optimal. Cet ordonnancement tient compte de la topologie physique des processeurs, et conduit a un temps d'exécution optimal sur une machine ayant un nombre de processeurs illimités.

La première partie de ce chapitre présente deux méthodes de calcul d'un tiling optimal. La seconde partie propose un algorithme de calcul du tiling optimal ainsi que son temps total d'exécution. La dernière partie ouvre la voie aux perspectives de recherches dans le domaine du tiling sous forme de questions et idées que nous n'avons pu bon explorer au cours de ce travail.

4.1 Calcul d'un tiling optimal

4.1.1 Première méthode:

La plus part des partitionnement optimaux calculés dans la littérature prenaient comme fonction objective, le volume des communications. Il est vrai que le traitement sur des superordinateurs souffre toujours du coût élevé du transfert des données; or cela ne concerne plus que les temps d'amorçage, puisque les débits sont aujourd'hui très élevés entre caches et les mémoires globales.

Il est donc très important de considérer que le volume des communications à lui seul, ne peut permettre de déterminer un tiling conduisant à un temps d'exécution minimal.

Cette conclusion nous a permis de prendre comme fonction objective, non plus le volume des transferts, mais la temps d'exécution total.

Nous supposons par la suite que la machine cible possède un nombre illimité de processeurs, ce qui nous permet de calculer le temps total d'exécution.

Soit t_e est le temps d'exécution d'une tuile par un processeur,
soit t_{vcom} le temps de transfert des résultats d'une tuile aux processeurs voisins,
et soit c la longeur du plus long chemin exprimée en nombre de tuiles.
Alors le temps total d'exécution s'exprime par

$$T = (1 + c) * t_e + c * t_{vcom}$$

Nous pouvons exprimer ces termes par:

$$t_e = \frac{taille\ d'une\ tuile * nombre\ d'operation\ dans\ l'instruction(S)}{10^6 * performance}$$

et

$$t_{vcom} = temps\ start\ up + \frac{taille\ d'une\ tuile * \sum_{ij} HD}{debits\ en\ reels\ par\ secondes}$$

D'ou la formule final du temps d'exécution

$$T = (1 + c) * \frac{det(P).nb_{op}}{10^6.perf} + c * (st_{up} + \frac{det(P).\sum_{ij} HD}{debit}$$

avec nb_{op} le nombre d'operation dans l'instruction (S), $perf, debit$ la performance estimé du code et le débit en réels par secondes.

Cette fonction sera donc, la fonction objective à minimiser sous un certain nombre de contraintes que nous allons voir.

Nous cherchons dans la suite un tiling valide qui minimise T. Or le nombre de tilings valides est infini. Nous devons donc restreindre notre domaine de recherche de H aux tiling valides unitaires -i.e $det(H) = 1$ -[1], et puis trouver les facteurs des lignes de H qui minimise T , et ceci pour chaque tiling unitaire valide.

Formellement, ceci revient a chercher :

- 1) un tiling H unitaire,
- 2) un ensemble de facteurs $k_i \in N$

s.c

- $HD \geq 0$
- $H^{-1} = P \in Z$
- T minimal

1. $det(H) = -1$ est élimimé, car il n'est autre chose que des colonnes interchangées

Le résultat est le tiling $P' = [k_1\vec{p_1}, ..., k_n\vec{p_n}]$
Le tiling P' est valide, voir le lemme.

lemme 1 *Si H est un tiling valide, $\vec{p_i}$ i^{eme} colonne de P,*
et $k_i \in N$, alors
$P' = [k_1\vec{p_1}, ..., k_n\vec{p_n}]$ est valide .

preuve

Soit $\vec{h_i}$ i^{eme} ligne de H.
on a H valide: $HD \geq 0$
alors $\forall \vec{h_i}$ on a $\vec{h_i}D \geq 0$
donc si $k_i > 0$ alors $\frac{1}{k_i} * \vec{h_i}D \geq 0$

en conclusion,
$H' = [\vec{h_1}/k_1, ..., \vec{h_n}/k_n]$ est valide.

4.1.2 Exemple illustratif

Nous allons voir au cours de cet exemple, comment calculer le tiling optimal.

Le nid de boucle choisi est celui vu au début. Nous avons alors les dépendances suivantes:

$$D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

L'ensemble des tiling valides unitaires est calculé par :

trouver: $H \in Q$

sujet à:
- $HD \geq 0$
- $H^{-1} \in Z$
- $det(H) = 1$

Ce qui revient à calculer

$$H = \begin{pmatrix} a & b \\ c & \frac{1+bc}{a} \end{pmatrix}$$

avec
$a, b, c \in N$ tel que $a \neq 0$ et $\frac{1+bc}{a} \in N^*$.

Nous avons donc un nombre infini de tiling H à traiter , ce qui est impossible, ceci nous oblige a faire des restrictions supplémentaires; pour notre part, nous retiendrons les tiling H pour les quels $a, b, c \leq cte$, où cte est suffisament petite pour générer un ensemble de tilings valides unitaires qui ne soit pas très grand.

Soit alors le tiling valide unitaire

$$H = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

nous allons chercher un couple $(k_1, k_2) \in N^*$ tel que le tiling déduit $P' = [k_1\vec{p_1}, k_2\vec{p_2}]$ soit optimal: conduit à un temps d'exécution total minimal.

Remarquez que le tiling optimal s'exprime alors par:

$$H' = \begin{pmatrix} 1/k_1 & 0 \\ 0 & 1/k_2 \end{pmatrix}$$

Le calcul des dépendances entre tuiles $D' = \lfloor H'D \rfloor$ donne

$$D' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Le nombre de tuiles entre le point de départ et la fin dans chaque dimension

$$A = H' * \begin{pmatrix} N-1 \\ M-1 \end{pmatrix} \qquad \textit{donne} \qquad A = \begin{pmatrix} \frac{N-1}{k_1} \\ \frac{M-1}{k_2} \end{pmatrix}$$

Nous pouvons maintenant exprimer c, la longueur du chemin critique par :

maximiser $c = \alpha + \beta$

sujet à: $\quad A = \alpha * d'_1 + \beta * d'_2$

Ce qui donne directement $c = \frac{N-1}{k_1} + \frac{M-1}{k_2}$

La derniere phase du calcul, consiste a résoudre le problème combinatoire suivant:

minimiser

$$T_{total} = (1 + \frac{N-1}{k_1} + \frac{M-1}{k_2}) * (\frac{k_1 k_2 * nb_{op}}{10^6 * perf}) + (\frac{N-1}{k_1} + \frac{M-1}{k_2}) * (st_{up} + \frac{k_1 + k_2}{debit})$$

sujet à:

– $\quad k_1 \leq N - 1$

– $\quad k_2 \leq M - 1$

– $k_1k_2 \leq$ taille du cache.

Ce qui donne pour des valeurs de N, M de 1.000.000 , sur un **IBM SP2** ayant des performances sur ce nid de boucles de l'ordre de $25Mflops$, un temps de start-up de $45.7\mu s$, un débit de $80.5Mbits/s$ et un cache de $16Kbits$ les valeurs suivantes: $k_1 = 23$, $k_2 = 23$ et le temps totale d'exécution est de 7.417886 secondes.

Donc on a le tiling

$$P = \begin{pmatrix} 23 & 0 \\ 0 & 23 \end{pmatrix}$$

Il faut maintenant reprendre ce même calcul pour plusieurs tiling valides unitaires, et retenir celui qui conduit au temps total d'exécution minimal, ainsi que les valeurs de k_1, k_2 qui ont permis de l'atteindre.

Le seul problème ici, est le nombre de tiling valides unitaires, ce qui nous oblige d'ajouter une restriction supplémentaire. Pour notre part, la restrictions retenue sera sur les valeurs de H , c-a-d , prendre uniquement des valeurs $a, b, c \leq cte$.

Aussi le choix de la performance doit être judicieux, car les expérimentations ont montrés qu'il est très variable selon le nombre d'éléments dans chacune des dimensions, et ceci même si les éléments tiennent dans le cache.

4.1.3 Deuxième méthode

On a vu que la première méthode présente une petite faiblesse, du fait de la restriction apportée sur les valeurs de a, b, c, ce qui risque d'éliminer la solution optimale.

Nous allons donc suivre une seconde approche de calcul d'un tiling optimal. Le nombre de tiling valide unitaire, on l'a vu, est infini. La première approche ne considérait qu'un sous ensemble de celui-ci, pour les éléments duquels on calcule le temps total d'exécution.

Notre seconde approche permet de considérer la formule générale des tilings valides unitaires, et ne risque donc pas d'éliminer la solution optimale.

Nous procéderont de la même manière que la première approche , c-a-d, par un exemple, qui de plus est le même.

Nous avons vu que la formule des tiling valides unitaires est:

$$H = \begin{pmatrix} a & b \\ c & \frac{1+bc}{a} \end{pmatrix}$$

avec
$a, b, c \in N$ tel que $a \neq 0$ et $\frac{1+bc}{a} \in N^*$.

De la même manière le calcul du tiling recherché est:

$$H' = \begin{pmatrix} \frac{a}{k_1} & \frac{b}{k_1} \\ \frac{c}{k_2} & \frac{1+bc}{ak_2} \end{pmatrix}$$

Et

$$P' = \begin{pmatrix} \frac{k_1(1+bc)}{a} & -bk_2 \\ -ck_1 & ak_2 \end{pmatrix}$$

Aussi, les dépendances entre tuiles s'exprime par:

$$D' = \lfloor \begin{pmatrix} \frac{a}{k_1} & \frac{b}{k_1} \\ \frac{c}{k_2} & \frac{1+bc}{ak_2} \end{pmatrix} \rfloor$$

Finalement, la valeur en nombre de tuiles entre le point de départ et celui final:

$$A = \begin{pmatrix} \frac{aN+bM}{k_1} \\ \frac{cN+\frac{(1+bc)}{a}M}{k_2} \end{pmatrix}$$

On cherche donc a maximiser la longueur du chemin critique $c = \alpha + \beta$ avec $\alpha, \beta \in N$

S.c

$$\begin{pmatrix} \frac{aN+bM}{k_1} \\ \frac{cN+\frac{(1+bc)}{a}M}{k_2} \end{pmatrix} = \alpha \begin{pmatrix} \lfloor\frac{a}{k_1}\rfloor \\ \lfloor\frac{c}{k_2}\rfloor \end{pmatrix} + \beta \begin{pmatrix} \lfloor\frac{b}{k_1}\rfloor \\ \lfloor\frac{1+bc}{ak_2}\rfloor \end{pmatrix}$$

On suppose pour cela que $a, b \leq k_1$ et $c, \frac{1+bc}{a} \leq k_2$, ce qui donne:

$$0 \leq \frac{a}{k_1}, \frac{c}{k_2}, \frac{b}{k_1}, \frac{1+bc}{ak_2} \leq 1$$

Il faut noter ici, que les valeurs de a et d ne sont jamais nules.
Dans notre cas, nous avons les résultats suivants:

si $b = 0$ alors $\frac{b}{k_1} = 0 \Rightarrow \alpha = \frac{aN}{k_1}$

et si $c = 0$ alors $\beta = \frac{M}{ak_2}$

et si $c \neq 0$ alors $\alpha + \beta = \frac{cN+\frac{M}{a}}{k_2}$

si $b \neq 0$ alors $\lfloor\frac{b}{k_1}\rfloor = 1 \Rightarrow$

si $c = 0$ alors $\beta = \frac{M}{ak_2}$ et $\alpha + \beta = \frac{aN+bM}{k_1} -$

si $c \neq 0$ alors $\alpha + \beta = \frac{aN+bM}{k_1} = \frac{cN+(1+bc)M}{k_2}$

Nous avons donc, en résumé quatre cas possibles:

$b = 0, c = 0$: $c = \frac{aN}{k_1} + \frac{M}{ak_2}$

$b = 0, c \neq 0$: $c = max(\frac{cN+\frac{M}{a}}{k_2}, \frac{aN}{k_1})$

$b \neq 0, c = 0$: $c = max(\frac{aN+bM}{k_1}, \frac{M}{ak_2})$

$b \neq 0, c \neq 0$: $c = max(\frac{aN+bM}{k_1}, \frac{cN+\frac{(1+bc)}{a}M}{k_2})$

Pour chacun de ces quatre cas, nous allons essayer de calculer le tiling qui minimise le temps total d'exécution, - i.e trouver $a, b, c, k_1, k_2 \in N$ avec $a \neq 0$, et $d = \frac{1+b*c}{a} \neq 0$

4.2 Algorithme et Résultats

L'algorithme est dorénavent logique, il a comme résultat le tiling optimal P' ainsi que le temps d'execution total:

algorithme:

- **choisir une constante aussi petite** cte
- **pour chaque** $a \in [1, cte]$ **faire**
-     **pour chaque** $b \in [0, cte]$ **faire**
-         **pour chaque** $c \in [0, cte]$ **faire**
-             **si** $\frac{1+b*c}{a} \in N^*$ **alors**
-                 **pour chaque** $k_1 \in [max(a, b), taille_{cache}]$ **faire**

– **pour chaque** $k_2 \in [max(c,d), taille_{cache}/k_1]$ **faire**
– **calculer la longeur du** *chemin critique*
– **calculer le temps total d'execution** T_{total}
– **si** T_{total} **est minimal alors**
– **récupérer les valeurs de** a, b, c, d, k_1, k_2

Le résultat est donc le tiling optimal P':

$$P' = \begin{pmatrix} \frac{k_1(1+bc)}{a} & -bk_2 \\ -ck_1 & ak_2 \end{pmatrix}$$

Un programme a été ecrit en C sous Unix, implémentant cet algorithme, (voir à l'Annexe), pour les tests sur un **SP2** des mêmes conditions qu'auparavant -i.e débit, taille du cache ... -, nous avons les résultats suivants:

Le Tiling Optimal P' est donc:

$$P' = \begin{pmatrix} 16 & 0 \\ -16 & 32 \end{pmatrix}$$

d'ou

$$H' = \begin{pmatrix} \frac{1}{16} & 0 \\ \frac{1}{32} & \frac{1}{32} \end{pmatrix}$$

avec le temps total d'exécuition: $T_{total} = 5.352238$ secondes.
contre[2] $T_{total} = 527.771851$ secondes pour le tiling P_1:

$$P_1 = \begin{pmatrix} 738 & -738 \\ -81 & 82 \end{pmatrix}$$

2. Pour plus de détails quand aux résultats des différents tiling et leurs temps déxecutions, reportez-vous à l'Annexe.

4.3 Conclusion

Au cours de ce travail, nous avons mis en évidence les problèmes posés aux programmeurs de calculs scientifiques sur des superordinateurs. A savoir la dificulté de saturer le parallélisme, et bien utiliser la hiérarchie mémoire.

Nous avons alors introduit la technique de tiling qui est bien adapté à ce type de problème. Après un bref rappel de l'état de l'art de cette methode de transformation de programmes, nous avons proposé deux methodes pour calculer le tiling optimal avec comme fonction économique, le temps total d'exécution.

Aussi, Nous avons proposé une methode d'ordonnancement topologique, qui tient compte de la topologie physique des processeurs. Cet ordonnancement est de plus optimal.

Finalement, un algorithme donnant le tiling optimal P' avec son temps total d'exécution est donné. Une implémentation de cet algorithme est à l'Annexe, avec tous ses résultats.

Bibliographie

[Irig,Anc] François Irigoin, Corinne Ancourt "Scanning polyhedra with do loops" 1991 EMP-CRI A/195

[Irig] François Irigoin "Partitionnement des boucles imbriquées, une technique d'optimisation pour les programmes scientifiques" 1987 Thèse Université Pierre et Marie Curie

[Irig] François Irigoin, Remi Triolet "Supernode partitionning" 1987 EMP-CRI A 179 E 90 CAI

[Anc] Martine Ancourt "Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales" 1999 Thèse Université Paris VI

[Dart] Pierre Boulet, Alain Darte, Tanguy Risset, Yves Robert " Pen -ultimate tiling? " 1993 laboratoire LIP-IMAG, Integration, the VLSI journal 17 1994 33-51

[Xue] Jingling Xue " Communication-Minimal Tiling of union dependenc loops" 1996 University of new England Australia, LNCS 1239 SAN JOSE

[Xue] Jingling Xue "On tiling as a loop transformation" 1996 PPL vol 7 n 4 1997 409-424

[Ram] J.Ramanujam, P.Sadayappan "Tiling multidimensional iteration space for multicomputers" 1992, JPDC 16 108-120 1992

[Sar] Vivek Sarkar "loop transformation for hierarchical parallelism and locality" IBM Resherch, Thomas J.Watson Resherch Center, PO Box 704, Yorktown Heights, NY 10598, USA

[Hog] K.Hogstedt, L.Carter, J.Ferrante "Selecting Tile shape for minimal execution time" Dept of computer science and engineering, UCSD 9500 Gilman Drive, Lajolla, CA 92093-0114

[Mit] N.Mitchell, L.Carter, J.Ferrante, K.Hogstedt "Quantifying Multi-level nature of tiling interactions" Computer science and engineering, UCSD 9500 Gilman Drive, Lajolla, CA 92093-0114

[Anc] C.Ancourt, G.Oget "Tpips: command line interpreter" Ecole des Mines de Paris 1999

[coh] W.Cohen "Loop interchange" University of Alabama, Montsville, Alabama 35899=1998

Table des figures

Table des matières