Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Macro-Pipelines dans les programmes Data-Parallèles

Julien ZORY

Stage de DEA sous la responsabilité de
Frédéric Desprez
DEA d'Informatique de Lyon - Juin 1996

Résumé

La parallélisation automatique d'applications scientifiques tente aujourd'hui de masquer au programmeur les caractéristiques complexes de la machine cible en développant une analyse poussée par le compilateur. Il semble que la puissance de calcul des processeurs dans les Machines Parallèles à Mémoire Distribuée augmente beaucoup plus vite que la vitesse des communications dans le réseau de processeurs. Or le Data-Parallélisme impose une succession de phases de calculs et phases de communications bloquantes entre les processeurs. C'est pourquoi les méthodes de recouvrement des calculs/communications ont très rapidement interessé les développeurs, comme optimisations indispensables des compilateurs de langages Data-parallèle. Plusieurs techniques plus ou moins complexes ont été étudiées et agissent tant sur le grain de calcul que sur les communications elles mêmes.

Mots-clés: Parallélisation Automatique, HPF, Compilation, Recouvrements, Macro-Pipelines.

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80

INSA LYON
Institut National des Sciences Appliquées de Lyon
20 av. Albert Einstein - 69621 Villeurbanne Cedex - France
Téléphone : (33) 72 43 83 83 Télécopie : (33) 72 43 85 00

Remerciements

- *Je remercie tout d'abord Frédéric Desprez (ou plutot Fred), avec qui j'ai pris beaucoup de plaisir à travailler et qui a su aussi m'apporter tout son soutient dans ma quête de la thèse perdue. Bon courage pour l'Ironman et rendez-vous à Paris l'an prochain...*

- *Je tiens ensuite à remercier Alain Darte et Yves Robert, qui m'ont aidé en début d'année scolaire à mettre en place mon stage de DEA/PFE, et qui ne m'ont pas laissé faire des "cônes pointés" comme le pauvre Georges S. :-)*

- *J'attribue une mention spéciale à Thomas Brandes, qui a fait preuve d'un dévouement et d'une patience énormes pour m'expliquer les fondements d'Adaptor; et aussi pour son accent allemand si sympathique et sa joie de vivre.*

- *Une pensée émue pour Georges S. qui s'est engagé dans l'armée pendant trois ans (DRET?)... Moi qui pensait qu'il voulait faire une thèse?*

- *Une autre pour Fabien F. qui n'est pas sûr de revoir un jour les 150 cafés que je lui dois... peut-être qu'une facturation s'impose!*

- *Et enfin, un grand merci à*

  - *Catherine - féministe mangeuse d'hommes (au poivre)*
  - *Ioan - mais où est donc passé Vincent B.?*
  - *Manu - éternel vacancier en provenance de Lumini*
  - *Matthieu - un gros rouge qui fume et qui soupire*
  - *Mohamadou - stagiaire voyageur?*
  - *Nicolas - fr.rec.humour.gros-lourd.nicolas*

  pour leur éternelle bonne humeur....

Table des matières

1 Introduction

Les langages de programmation évolués permettent aujourd'hui au programmeur de s'abstraire complètement de l'architecture de la machine séquentielle. En effet, les progrès réalisés dans les techniques de compilation sont tels que la plupart des compilateurs génèrent du code hautement optimisé. La situation est très différente dans le domaine des architectures parallèles et, le programmeur ne peut, au stade actuel, se permettre d'ignorer les caractéristiques de la machine cible sans risquer d'obtenir des résultats déplorables. Il est donc très important de l'aider à prendre en compte ces contraintes de la manière la plus immédiate possible. Deux écoles s'opposent aujourd'hui à ce sujet; certains désirent conserver un parallélisme explicite et demandent au programmeur de générer lui même toutes les routines de "bas niveau", alors que d'autres tentent au contraire de soulager un maximum le programmeur, en générant "automatiquement" du code parallèle à partir de renseignements apportés sur la sémantique de l'algorithme. L'étude présentée ici se place dans le contexte de la *parallélisation automatique* de programmes, à laquelle vient s'ajouter une phase de compilation des programmes *Data-Parallèles* sur *Machines Parallèles à Mémoire Distribuée*.
Actuellement, de nombreux travaux de recherche sont effectués dans ce domaine. Les paralléliseurs automatiques disponibles aujourd'hui ont encore des performances en moyenne assez décevantes, et souvent catastrophiques dès que le code à paralléliser devient complexe. Cependant, les techniques de compilation et de parallélisation automatique constituent l'un des axes de recherche prioritaire dans le domaine de l'informatique du parallélisme et l'aboutissement de ces travaux permettra sans doute aux supercalculateurs d'être utilisés par un public plus large. La première étape marquante de cette avancée a été la mise en place d'un standard de programmation Data-Parallèle appelé *High Performance Fortran*, pour lequel commencent à apparaître actuellement des compilateurs.

L'approche la plus naturelle pour implémenter des algorithmes Data-Parallèles sur les Machines Parallèles à Mémoire Distribuée (MIMD), est la programmation SPMD (Single Program, Multiple Data). Ce type de programmes se compose d'une alternance entre phases de calcul pur et phases de communications bloquantes. Malheureusement, le gain en performance se trouve fortement freiné, voire anéanti, par le surcoût introduit par les communications. Pour obtenir les temps d'exécution "promis" par le parallélisme, il est donc primordial de masquer ce dernier. Plusieurs solutions existent, et de nombreuses études ont été réalisées à ce sujet.
Tout d'abord, le choix d'une bonne distribution des données sur les processeurs demeure la première étape indispensable pour minimiser le nombre et la taille des communications nécessaires. Le temps de latence introduit par les couches matérielles, par la bufferisation et surtout par les couches logicielles, doit être ensuite réduit au maximum, afin d'autoriser l'envoi de "petits" messages. Ceci étant, toutes les communications restantes doivent alors être masquées le mieux possible. Le mécanisme des *Communications Asynchrones* offre une solution intéressante pour cacher ce surcoût, en autorisant le traitement simultané d'un calcul et d'une communication par deux entités distinctes. Cependant, cette technique n'est plus utilisable dès lors qu'il existe une dépendance entre la communication et le calcul qui la recouvre. Le *Macro-Pipeline*, découpage du calcul à traiter en plusieurs petites entités, va permettre de contourner de manière efficace le problème. Le surcoût des communications n'est pas la seule faille du Data-Parallélisme. En effet, les points de synchronisation introduits par la programmation SPMD sont à l'origine de l'apparition dans le code de schémas dits *pipelines*, donnant lieu à une séquentialisation de l'algorithme complet. C'est en modifiant le grain de calcul, que le macro-pipeline va alors permettre d'en extraire un parallélisme partiel.

De nombreux travaux ont été entrepris dans ce domaine , et nous allons à travers le paragraphe 2, tenter d'en présenter les principaux résultats. Après un bref rappel des notions de base utiles à notre étude suivi d'une rapide présentation des techniques de recouvrements, nous exposerons l'état de l'art actuel du macro-pipeline. Nous aborderons dans le paragraphe 3, une étude de l'efficacité du macro-pipeline en réponse à un article récent de Quinn et Hatcher. Cette approche nous permettra de montrer que le gain obtenu par l'utilisation du macro-pipeline sur P processeurs est important et qu'il est de l'ordre de P dans certains cas. Nous présenterons ensuite dans le paragraphe 4, plusieurs résultats contribuant à étendre cette technique à des contextes moins triviaux, tel que le *macro-pipeline multidimensionnel*. Cette méthode de recouvrement trouve son intérêt à travers une implémentation dans un compilateur. Nous étudierons donc l'implémentation que nous avons mise en place dans le compilateur HPF *Adaptor*, puis nous présenterons les premiers résultats prometteurs obtenus sur diverses Machines Parallèles à Mémoire Distribuée. Enfin, nous donnerons les conclusions de cette étude, en rappelant les contributions importantes, mais aussi les échecs rencontrés. Nous terminerons alors par l'exposé des travaux qu'il est aujourd'hui envisageable d'entreprendre, pour donner au macro-pipeline toute sa valeur.

2 Problématique

2.1 Notions de base

Tout au long de cette étude, nous ferons appel à des notions de base du domaine de la parallélisation automatique. Nous ne faisons ci-dessous qu'une brève présentation des quelques points importants, mais le lecteur pourra trouver si besoin dans les références bibliographiques proposées, toutes les explications nécessaires.

2.1.1 Analyse de Dépendance

L'analyse de dépendance a pour objectif de contrôler si une transformation particulière peut être appliquée sans changer la sémantique des calculs ([4, 5]), mais aussi celui de déterminer les communications. **Une dépendance est une relation entre deux calculs qui impose une contrainte sur leur ordre d'exécution** [3].

- Il y a une **dépendance de contrôle** entre une condition C et une instruction S lorsque C détermine si S sera ou non exécutée.

- Il y a une **dépendance de données** entre une instruction S_1 et une instruction S_2 lorsqu'il y a utilisation conflictuelle d'une même variable. Il existe trois sortes de dépendances de données dues à l'ordre dans lequel le programme accède, aussi bien en lecture qu'en écriture, aux différents emplacements de la mémoire.
  - **Les dépendances de flot :** Il y a une *dépendance de flot* d'une instruction I_1 vers une instruction I_2 si I_2 est placée après I_1 dans le code et si I_2 accède en lecture à un emplacement mémoire qui a été modifié par I_1.
  - **Les anti-dépendances :** Il existe une *anti-dépendance* d'une instruction I_1 vers une instruction I_2 si I_2 est placée après I_1 dans le code et si I_2 accède en écriture à un emplacement mémoire qui est lu par I_1.
  - **Les dépendances de sortie :** Il existe une *dépendance de sortie* d'une instruction I_1 vers une instruction I_2 si I_2 est placée après I_1 dans le code et si I_2 modifie un emplacement mémoire précédemment modifié par I_1.

Dans de nombreux travaux effectués sur l'analyse de dépendances [19], on considère que la seule *véritable* dépendance est la dépendance de flot (appelée souvent **dépendance vraie**). Les autres dépendances ne sont que les conséquences de la réutilisation d'une zone de mémoire, et sont couramment appelées *pseudo-dépendances*.

2.1.2 Nids de boucles et dépendances

Nids de boucles: On appelle *nid de boucles* un ensemble d'instructions uniquement constituées de boucles `do` (boucles pour) et d'instructions d'affectation qui utilisent des variables scalaires ou des éléments de tableau [3, 9, 19].

Les dépendances dans les nids de boucles:

- Une dépendance est *indépendante de la boucle* si elle existe même lorsque l'on ignore les boucles englobantes.
- Une dépendance est *portée par les boucles* si elle dépend des indices de boucles englobantes.

2.1.3 High Performance Fortran

La percée du parallélisme dans le monde industriel est très nettement freinée par la difficulté à écrire des programmes portables. Pour essayer de résoudre ces difficultés, un forum réunissant des chercheurs et des constructeurs de machines s'est créé au début des années 90 [18, 23]. L'objectif est la définition d'extensions du langage FORTRAN 90 qui permettraient le développement de compilateurs paralléliseurs. *High Performance Fortran* (HPF) qui est le résultat de ce travail, est un langage de type Fortran possédant des extensions pour définir l'alignement des données, leur distribution sur les processeurs, ainsi qu'un certain nombre de constructions spécifiques à la parallélisation de boucles. HPF utilise le modèle du parallélisme de données (data-parallèle). La parallélisation est assistée par l'utilisateur qui guide le compilateur en insérant des directives dans son programme, utilisées par ce dernier pour générer le code SPMD correspondant (code unique exécuté sur tous les processeurs). Les directives HPF sont insérées dans une ligne de commentaire afin qu'elles puissent être ignorées par un compilateur Fortran standard. Ces lignes commencent par la suite de caractéres **!HPF$**. Elles indiquent au compilateur des stratégies afin d'améliorer les performances du programme, mais ne modifient en rien la sémantique de celui-ci. Dans un premier temps, les tableaux sont alignés (!HPF$ ALIGN) entre eux ou sur de templates et, dans un second temps, les différents groupes de tableaux sont distribués (!HPF$ DISTRIBUTE) sur une grille virtuelle de processeurs (!HPF$ PROCESSORS) (Figure 1).

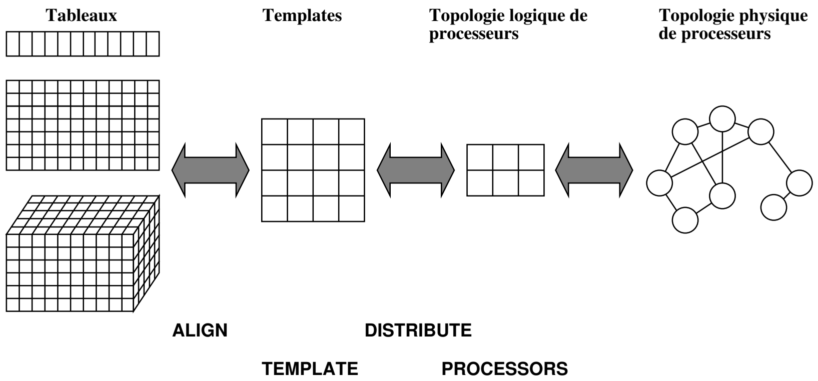

FIG. 1 - *Principe de l'alignement et de la distribution des données dans le langage HPF.*

Le langage HPF n'est pas adapté aux structures de données autres que matricielles; ce qui interdit pour l'instant la prise en compte des problémes irréguliers. De nombreuses recherches sont en cours sur ce sujet, *et déjà, un nouveau forum, appelé HPF-2, est mis en place.*

2.2 Techniques de Recouvrements calculs/communications

Dans un programme SPMD, tous les processeurs exécutent le même code, et le programme alterne entre des phases de calcul pur, et des phases de communications entre les processeurs. Au cours des phases de calcul, chaque processeur travaille à une fréquence qui lui est propre, et ne manipule que des données au niveau local (*owner computes rule*). Les phases de communications permettent au processeurs d'échanger des données, avec tous les processeurs (échange total, diffusion, ...) ou une partie seulement (voisin à voisin,...). Ces communications se font par envois de messages (message passing) entre les entités concernées, et demandent une certaine synchronisation.

Il semble actuellement que la vitesse de calcul des processeurs dans les Machines Parallèles à Mémoire Distribuée (*MIMD*), augmente beaucoup plus rapidement que la vitesse de transmission des messages à travers le réseau d'interconnexion. Cette tendance a donc donné naissance à de nombreux travaux visant à réduire au maximum la perte de temps engendrée par les communications bloquantes (synchronisation) des programmes SPMD. En effet, le gain de temps issu de l'exécution parallèle des calculs, est parfois réduit à néant par la forte demande en communications. Le recouvrement des communications par les calculs est donc primordial pour atteindre les performances promises. Les communications asynchrones sont la première réponse apportée à ce problème dans les machines actuelles (Paragraphe 2.2.1). Par ailleurs, les processus légers (ou threads) font leur apparition aujourd'hui au niveau du système dans le but de permettre aux processeurs de toujours commencer les calculs le plus tôt possible (Paragraphe 2.2.2). Nous nous pencherons ici sur une autre technique de recouvrement appelée macro-pipeline; en effet, le placement des données sur les processeurs et l'existence de dépendances entre instructions ne permettent pas toujours une bonne *exécution parallèle* des calculs. L'approche *Data-parallèle* aboutit alors à une exécution plus ou moins séquentielle du code. C'est en agissant sur le grain des calculs que la technique du macro-pipeline apporte une solution (Paragraphe 2.3.1).

2.2.1 Les communications asynchrones

Recouvrir les communications au sein d'un processeur nécessite que celui-ci possède un moyen pour communiquer en même temps qu'il calcule : c'est le mécanisme des communications asynchrones. La communication est dans ce cas prise en charge par une autre entité (très souvent implantée physiquement au niveau du routeur de chaque nœud sur le réseau de processeurs), de telle sorte que l'on peut découper le temps de communication en trois étapes: le temps mis par le processeur pour fournir au buffer le message à envoyer (souvent appelé *message latency*), le temps de transmission du message à travers le réseau d'interconnexion, et enfin le temps nécessaire au processeur pour recevoir le message (notons que nous pourrions ajouter à cela le temps d'accusé de reception, mais celui-ci peut-être considéré comme négligeable dès lors que les messages ont une taille raisonnable). Seule la partie *Transmission* est réellement recouverte, même si l'utilisation du DMA [1] va permettre de rendre négligeables les temps de bufferisation. M. Quinn et P. Hatcher dans [22] ont étudié l'efficacité des communications asynchrones dans les programmes Data-Parallèles De nombreux exemples d'utilisation efficace de cette technique, comme la multiplication de matrices étudiée dans [6, 10, 11], peuvent être trouvés dans la littérature.

1. Direct Memory Access.

2.2.2 Les Processus Légers

Une autre manière de recouvrir les calculs et les communications consiste à utiliser les processus légers. A notre connaissance, ceux-ci ont été utilisés par deux fois dans la compilation de langages data-parallèles; une fois dans le compilateur Vienna-Fortran [14] et une fois dans le compilateur Pandore [2]. Ils ont été également utilisés au sein de programmes scientifiques dans [15].

Ils peuvent être utilisés soit pour créer des processeurs virtuels sans faire appel à une gestion de processus trop lourde, soit pour palier à des communications dont le comportemenet ne peut être facilement déterminé à l'avance. Quoiqu'il en soit, leur manipulation est encore assez lourde et le support d'éxécution d'un compilateur les utilisant doit être capable de les faire intervenir sans perte de temps. En effet, leur utilisation peut entraîner des surcoûts, non seulement pour leur activation, mais également du fait des vidages de caches entrainés par l'utilisation de données différentes.

2.3 Etat de l'art du Macro-Pipeline

2.3.1 La méthode

De nombreux travaux ont été effectués dans le domaine des recouvrements calculs communications depuis plusieurs années, et en particulier dans le domaine des *macro-pipelines*. Cette technique peut être grossièrement présentée comme une modification du grain de calcul en vue de diminuer le temps d'exécution total grâce à des recouvrements. Il existe cependant aujourd'hui dans la littérature deux justifications différentes de son utilisation :

- L'emploi du mécanisme des communications asynchrones présenté précédemment, est conditionnée par l'absence de dépendance entre la communication et le calcul qui la recouvre. La solution apportée par le macro-pipeline consiste alors à découper le calcul initial de taille n, en plusieurs paquets de taille réduite n_0, de sorte que le recouvrement soit possible entre le calcul $i+1$ et l'envoi du message i (Figure 2). Quinn et Hatcher ont étudié cette technique, et ont également analysé le gain qui est alors obtenu [22]. Nous reviendrons plus en détails sur ce point dans le Paragraphe 3.

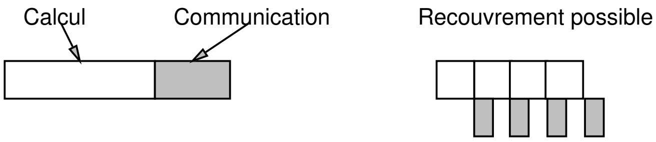

FIG. 2 - *Utilisation du macro-pipeline pour les communications asynchrones.*

- Les phases de synchronisation introduites par la programmation SPMD, sont à l'origine des *schémas pipelines* tels qu'ils sont présentés sur la Figure 3(A); chaque processeur attend les données de son voisin de gauche, effectue ses propres calculs, puis envoie une partie des nouvelles valeurs calculées à son voisin de droite. Une telle exécution sera évidemment moins performante que l'exécution séquentielle, puisque sont rajoutés les surcoûts de communication.

Lorsque cela est possible, c'est à dire lorsque les dépendances le permettent, le macro-pipeline va découper les phases de calculs de manière à envoyer des résultats partiels aux processeurs voisins, qui peuvent alors eux-mêmes commencer à calculer. Une première solution consiste à déclencher les communications au plus tôt, c'est à dire dès qu'une donnée attendue par un processeur distant a été calculée. Ceci permet un démarrage des travaux le plus tôt possible. On a donc un pipeline à grain fin ((B) sur la Figure 3). Cette solution introduit malgré tout un surcoût dû aux temps d'initialisation des communications. Ces temps sont toujours plus importants que le coût d'envoi d'un élément et il en résulte un temps total pouvant être plus important que le temps séquentiel. En envoyant les données plus tôt, le macro-pipeline permet aux processeurs de commencer leurs travaux sans attendre. Cependant, cette technique coûte cher en communications. Un compromis doit donc être trouvé, et consiste à prendre le grain de calcul (et donc de communication) qui minimisera le temps total (Paragraphe 2.3.2). On obtient alors un grain moyen ((C) sur la Figure 3).

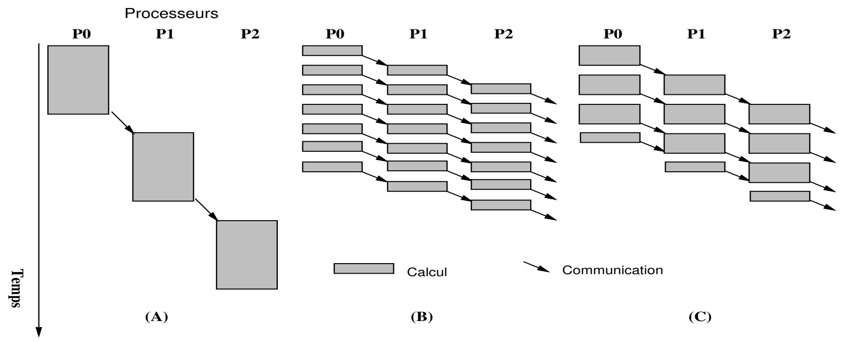

FIG. 3 - *La solution apportée par les Macro-Pipelines*

Le macro-pipeline est une opportunité offerte d'exploiter du *parallélisme partiel* en permettant aux processeurs de recouvrir leurs calculs mutuellement. Les schémas pipelines sont la plupart du temps issus d'une récurrence au sein d'un algorithme, qui se caractérisera très souvent par des calculs effectués sur une matrice donnée (nid de boucles). La Figure 2.3.1 donne un exemple de code source qui entrainera une exécution pipeline dont les performances seront catastrophiques. En effet, l'algorithme exécuté par chaque processeur sur ses données locales, sera celui de la Figure 2.3.1. Ainsi, lorsque la distribution des données sur les processeurs est telle qu'il existe une dépendance entre deux calculs effectués par deux processeurs différents (comme sur la Figure 5), l'exécution devient alors complètement séquentielle. Dans un cas comme celui-ci, la solution consistera bien évidemment à envoyer les données calculées par groupe de k lignes, après calcul du k optimal.

```
      REAL A(N,N)
!HPF$ DISTRIBUTE A(BLOCK,*)
      ...
      DO I=2,N
        DO J=1,N
  (S)       A(I,J) = f(A(I-1,J))
        END DO
      END DO
```

FIG. 4 - *Cette boucle sera exécutée de manière séquentielle.*

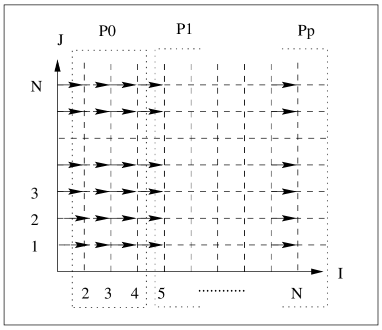

FIG. 5 - *Les dépendances traversent l'espace des processeurs.*

```
si (non premier processeur) recevoir A(ma_borne_inf-1,:)
DO I=ma_borne_inf, ma_borne_sup
   DO J=1,N
     A(I,J) = f(A(I-1,J))
   END DO
END DO
si (non dernier processeur) envoyer A(ma_borne_sup,:)
```

FIG. 6 - *Algorithme exécuté par chaque processeur sur ses donneés locales.*

Ces deux utilisations qui sont faites de la technique des macro-pipelines, sont en réalité très proches et pourront dans la plupart des cas être combinées. La Figure 7 présente une utilisation idéale du macro-pipeline, où les communications sont recouvertes, et pour laquelle on va masquer les retards entrainés par les synchronisations. Dans la suite de l'étude, nous nous limiterons la plupart du temps à du *macro-pipeline synchrone*, car l'ajout des communications asynchrones n'est intéressant que pour l'étude du gain obtenu et ne présente pas en soit de difficultés.

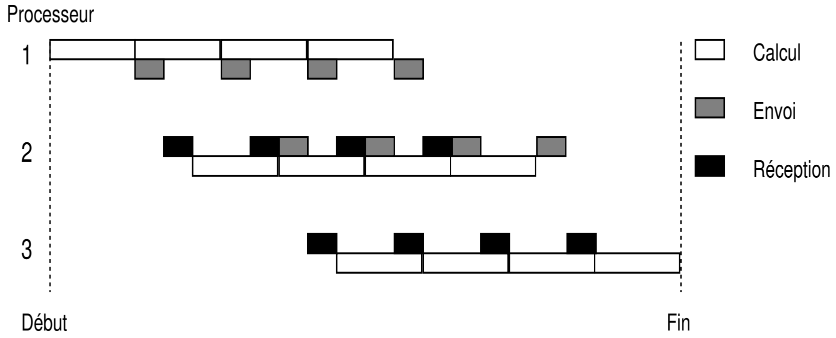

FIG. 7 - *Macro-Pipeline asynchrone sur P processeurs.*

2.3.2 Calcul du Grain Optimal

De nombreux travaux ont été entrepris pour modéliser la taille de paquet optimale. En effet, contrairement aux communications asynchrones, le macro-pipeline peut entraîner une augmentation du temps d'exécution si ce paramètre est mal choisi. La difficulté de détermination de ce paramètre réside dans le fait que sa valeur dépend à la fois des bornes de l'espace d'itération, des vecteurs de distance, ou encore de la distribution des données sur les processeurs et des paramètres de la machine cible.

Tseng dans [26] propose un résultat dans le cas simple à deux dimensions, pour un tableau de taille $m * n$ distribué selon une dimension sur un ligne de p processeurs. Il restreint son étude à des vecteurs de dépendances très simples. La modélisation est la suivante: soit B la taille de bloc recherchée et C le temps de communication (il choisit un modèle temps constant car il suppose que le temps de transfert est négligeable par rapport au temps d'initialisation). Le temps d'exécution global T, représenté sur la Figure 8, est égal au temps de démarrage du dernier processeur (1er terme de l'équation 1), auquel on ajoute le temps de calcul pour 1 processeur (second terme de l'équation 1).

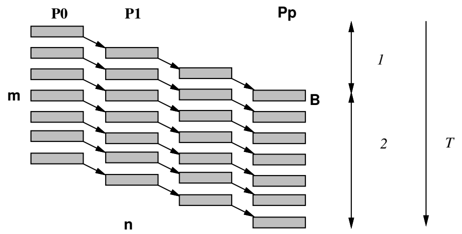

FIG. 8 - *Le temps d'exécution se compose de deux termes.*

$$T = (p-1)\left(\frac{nB}{p} + C\right) + \frac{m}{B}\left(\frac{nB}{p} + C\right) \tag{1}$$

En annulant la dérivée de T par rapport à B (équation 2), on obtient la taille de bloc recherchée (équation 3).

$$\frac{\partial T}{\partial B} = -\frac{mC}{B^2} + \frac{n(p-1)}{p} \tag{2}$$

$$\frac{\partial T}{\partial B} = 0 \Rightarrow B = \sqrt{\frac{mpC}{n(p-1)}} \sim \sqrt{\frac{m}{n}}\sqrt{C} \tag{3}$$

La recherche du grain optimal est une tâche qui très vite devient complexe, dès lors que l'on modifie le chemin critique de l'exécution. Cependant, la méthode de calcul reste la même, et la difficulté se trouve dans la modélisation du temps de calcul. Ohta, Saito, Kainaga et Ono dans [20] étendent l'étude de Tseng [26] à des vecteurs de dépendances plus complexes pour lesquels il devient nécessaire d'effectuer une transformation de boucle, et tentent une généralisation à N dimensions. Desprez, Ramet et Roman proposent quant à eux une bibliothèque de calcul de la taille de paquets optimale (cf. résultats obtenus sur la Figure 9). Il s'agit de la bibliothèque OPIUM (Optimal Packet sIze compUtation Methods [12]) qui offre un ensemble de routines permettant de calculer la taille de paquets en leur passant en paramètres la complexité des calculs et des communications. Cette bibliothèque est très utile pour plusieurs raisons: elle permet tout d'abord à l'utilisateur de bénéficier de la taille de paquets optimale sans avoir à connaître les méthodes de calcul. Par ailleurs, elle autorise des optimisations au moment de l'exécution et évite de recalculer la taille de bloc lorsque cela n'est pas nécessaire. Enfin, il sera possible de l'insérer directement au niveau du compilateur.

2.3.3 D'autres résultats

Le macro-pipeline est étudié depuis plusieurs années déjà, et a suscité l'intérêt de nombreuses équipes de recherche. Cependant, c'est une technique dont l'objectif est d'être inséré comme optimisation pour la compilation des programmes Data-Parallèles. Or, si la méthode est aujourd'hui bien connue, les contributions permettant d'étendre la technique à des cas moins triviaux (donc réels) ne sont encore que très limitées. Plusieurs paramètres sont à prendre en compte pour tenter de généraliser ces travaux : Ce sont principalement la distribution, les dépendances, et la dimension de l'espace d'itération. Parmi les résultats connus aujourd'hui, il est important de distinguer les résultats théoriques, des implémentations réelles qui en ont été faites. En effet, certaines méthodes telles que la torsion de boucle, rendent difficile l'implémentation du macro-pipeline dans un compilateur.

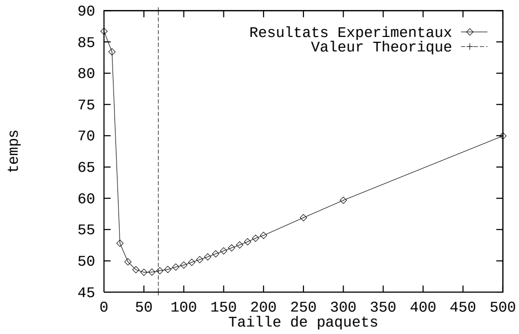

FIG. 9 - *Temps de factorisation obtenu pour une matrice de taille 1024 en fonction de la taille de paquet.*

Le Tableau 1 résume les différentes études qui ont été faites pour un espace d'itération à deux dimensions (c'est principalement là que se situent toutes les études jusqu'à ce jour). Pour chaque cas de figure, nous notons les références des personnes en gras lorsque l'implémentation existe, et en italique, lorsque les travaux n'en sont toujours qu'à la phase théorique. Le terme *tout le monde* est utilisé ici pour faire référence à tous les travaux sur le macro-pipeline (et en particulier les plus intéressants [1, 8, 10, 16, 20, 21, 25, 26]). L'extension à un espace d'itération à N dimensions, est étudiée par de nombreux auteurs, et ne présente pas de réelle difficulté dans le cas simple d'une dépendance du type $(d_1 > 0, d2 = 0)$. Cependant, personne ne s'est intéressé à la possibilité de faire du macro-pipeline selon plusieurs dimensions. Nous présenterons dans le Paragraphe 4, les diverses contributions que nous avons pu apporter au domaine.

	(Block , *)	**(Block , Block)**
$(d_1 > 0, d_2 = 0)$	**Tout le monde**	**[ce rapport]**
$(d_1 > 0, d_2 < 0)$	**[20]**	—
$(d_1 > 0, d_2 > 0)$	*[20] [ce rapport]*	*[ce rapport]*

TAB. 1 - *Etudes du macro-pipeline - Matrice 2D*

3 Efficacité du macro-pipeline

3.1 Contexte

Plusieurs articles ont présenté cette technique d'optimisation ainsi que les gains qui peuvent être obtenus en les utilisant. Le dernier en date est un rapport de M.J. Quinn et P.J. Hatcher [22] qui affirme que l'utilisation des Recouvrements calculs/communications conduit à un gain négligeable **sur un processeur** dans les programmes parallèles classiques. Dans ce paragraphe, nous présentons une analyse qui prouve que le gain n'est pas négligeable dès lors que l'on tient compte du temps d'exécution total de l'algorithme **sur l'ensemble des processeurs**.

Il existe plusieurs moyens de recouvrir les calculs et les communications suivant le système utilisé et bien sûr suivant l'algorithme. Comme cela a été présenté au cours du Paragraphe 2.2.1, le premier moyen de recouvrir les calculs et les communications et de communiquer un ensemble de données pendant que l'on travaille sur un autre. Cette méthode est appelée *Recouvrement Systolique*[2] ou *Recouvrement Général*[3] selon le cas dans [22].

La dernière solution, que nous avons présentée dans le paragraphe précédent, est le **macro-pipeline** (appelé *Pipelined Communication-Computation Overlap* dans [22]). Notre modélisation suppose que la machine cible est capable de recouvrir les calculs et les communications en utilisant des moyens matériels et (ou) logiciels. Premièrement, le matériel doit permettre les recouvrements en utilisant par exemple des DMA. Ensuite, le logiciel doit également permettre les recouvrements en offrant à l'utilisateur l'accès à une bibliothèque de communications permettant les envois et réceptions non-bloquants.

3.2 Modèle d'Architecture

Nous utilisons ici les même notations que Quinn and Hatcher [22] afin de pouvoir comparer nos résultats entre eux.

Nous supposons que notre machine cible possède une mémoire distribuée et utilise un réseau de communication pour échanger les données entre les processeurs. Par la suite, nous appellerons $T(n)$ le coût de communication de n éléments de données entre deux processeurs. Ceci est généralement une fonction linéaire de la taille du message envoyé. Nous écrirons donc $T(n) = \lambda_c + \beta_c n$. Nous exprimons la latence des communications par $L(n) = \lambda + \beta n$, également une fonction linéaire de la taille des messages. Elle inclue le coût de la bufferisation des messages et le temps pour prendre en compte l'appel à `send`. Nous appellerons également $L'(n)$ le coût de réception d'un message. Les latences d'émission et de réception sont considérées comme égales. Nous supposons que le coût de bufferisation est toujours inférieur au coût de communication, soit $T(n) > L(n)$. Cette supposition est logique puisque la bufferisation fait appel à des mouvements de données à l'intérieur de la mémoire et la communication en elle-même utilise le réseau qui est bien moins rapide.

3.3 Analyse de Quinn et Hatcher

Dans ce paragraphe, nous présentons les résultats principaux de Quinn et Hatcher sur les gains obtenus en utilisant les recouvrements calculs / communications [22]. Notons tout d'abord que l'analyse présentée dans cet article ne concerne que les optimisations basées sur les recouvrements et les gains obtenus sur **un processeur**. Leur analyse utilise le modèle présenté dans le Paragraphe 3.2.

Un programme contient des segments de calculs entrelacés avec des appels à des fonctions de communications. Les auteurs supposent que le programme est "toujours" bâti comme celui présenté sur la Figure 10. Une première optimisation peut être effectuée puisque le segment de code 1 peut recouvrir la communication 2. `Com2` sera translaté entre `CS1` et `CS2`. Ici, ils utilisent simplement un appel à une routine de communication asynchrone et placent une routine d'attente de complétion de communication juste après `CS2` pour s'assurer que la communication est bien terminée avant de commencer `CS3`. Le segment de code C_o peut être recouvert par la communication 2 et le segment de code C_r ne peut pas être recouvert simplement (mais peut être pipeliné (Figure 11)).

Il y a donc quatre cas, suivant que les segments de code C_o et C_r sont vides ou pas. Les quatre cas sont donnés avec leur nom générique dans le tableau 2.

Les recouvrements systolique et général sont proches car dans les deux cas, un appel à une fonction de communications asynchrones est utilisée. Nous allons maintenant, pour chaque cas,

2. *Systolic Overlap.*
3. *General Overlap.*

```
<Com1>   Communication 1
<CS1>    Segment de code 1 (incorporant des variables dans M)
<CS2>    Segment de code 2 (n'incorporant pas des variables dans M)
<Com2>   Communication 2 avec le message M
<CS3>    Segment de code 3 (n'incorporant pas des variables dans M)
<CS4>    Segment de code 4 (incorporant des variables dans M)
<Com3>   Communication 3
```

FIG. 10 - *Algorithme de référence pour les différentes méthodes de recouvrement.*

```
<Com1>
<CS1>
<Com2>
<CS2>          } Co
<CS3>          }
<CS4>          } Cr
<Com3>
```

FIG. 11 - *Algorithme de référence après la première optimisation.*

donner les résultats de Quinn et Hatcher sur le gain maximum en performances qui peut être obtenu en utilisant le recouvrement.

3.3.1 Recouvrement systolique

Le gain maximum en performances qui peut être atteint est donné par

$$S(n) = 1 + \frac{\min(C_0(n), T(n))}{L(n) + \max(C_0(n), T(n))} \tag{4}$$

Quinn and Hatcher donnent deux corrolaires sur les bornes qui peuvent être obtenus pour $S(n)$.

- Corollaire 1: $S(n) \leq 2$,
- Corrolaire 2: si $\lim_{n \to +\infty} T(n)/C_0(n) = 0$ et $\lim_{n \to +\infty} L(n)/C_0(n) = 0$, alors $\lim_{n \to +\infty} S(n) = 1$.

Le premier corrolaire est évident, puisque pour A et B positifs, on a $(A+B)/\max(A,B) \leq 2$. Pour le second, si le coût des calculs est bien plus important que les coûts de communication et de latence, le recouvrement des communications ne conduira évidemment pas à un gain important.

		Segment C_o	
		Contient du code	Vide
Segment C_r	Contient du code	Recouvrement général	Les dépendances de données empêchent les recouvrements
	Vide	Recouvrement systolique	Aucun calcul à recouvrir avec les communications

TAB. 2 - *Quatre cas pour le problème du recouvrement.*

3.3.2 Recouvrement général

Le gain maximum sur les performances qui peut être atteint est donné par

$$G(n) = 1 + \frac{\min(C_0(n), T(n))}{L(n) + C_r(n) + \max(C_0(n), T(n))} \tag{5}$$

Dans ce cas, le gain obtenu est moins important que celui du cas systolique puisqu'un autre terme $(C_r(n))$ est ajouté au dénominateur.

3.3.3 Recouvrement pipeliné

Le gain maximum sur les performances qui peut être atteint est donné par

$$P(n) = \frac{L(n) + T(n) + C_r(n)}{\frac{n}{n_0}L(n_0) + \left(\frac{n}{n_0}L(n_0) - 1\right)\max(T(n_0), C_r(n_0)) + T(n_0) + C_r(n_0)} \tag{6}$$

La latence peut rendre le temps d'exécution plus important que celui de la version non pipelinée. Dans leur article, Quinn et Hatcher affirment que si $\lambda < \min(T(n_0, C_r(n_0))$, alors le temps d'exécution sera réduit. Cela signifie que si la latence du message est trop importante, alors il est plus intéressant d'envoyer un gros message que plusieurs petits messages.

Le gain obtenu est tel que $\lim_{n\to+\infty} S(n) \leq 1$ si $\lim_{n\to+\infty} \frac{T(n)}{C_r(n)} = 0$ et $\lim_{n\to+\infty} \frac{L(n)}{C_r(n)} = 0$. Il est évident que si la complexité des communications est négligeable face à celle des calculs, alors il n'y aura pas de gain obtenu sur **un processeur**. Nous verrons par la suite que cela n'est plus le cas si l'on prend le temps d'exécution total sur **tous les processeurs**.

3.4 Cas mono-dimensionnel

Nous donnons à présent notre analyse du problème du recouvrement pipeline. Nous affirmons qu'un gain significatif peut être obtenu lorsque le temps d'exécution est pris pour l'application totale sur l'ensemble des processeurs. Dans la suite du paragraphe, nous utiliserons le terme de macro-pipeline au lieu de pipeline.

Soit P_{np}^{1D} le temps total d'exécution quand le macro-pipeline n'est pas utilisé et P_p^{1D}, le temps d'exécution de la version macro-pipeline. Nous supposons que la taille optimale de paquets a été calculée (par exemple en utilisant la bibliothèque OPIUM) et est utilisée. L'équation 9 donne le gain maximum pour le cas uni-dimensionnel.

Nous considérons ici le cas où une seule dimension est distribuée. Soit n la taille de cette dimension. Nous supposons que nous utilisons une taille de paquets optimale égale à n_0 (qui divise n), et donc il y a n/n_0 paquets. Le coût de calcul de la matrice totale est $C(n)$ et le coût de calcul d'un paquet est égale à $C(n_0)$. Le coût de calcul restant constant que l'on utilise un macro-pipeline ou pas, nous avons $\frac{n}{n_0}C_r(n_0) = C_r(n)$ (voir Figure 12).

Le temps total d'exécution de l'algorithme sans macro-pipeline est donné par

$$P_{np}^{1D} = PC_r(n) + (P-1)[2L(n) + T(n)] \tag{7}$$

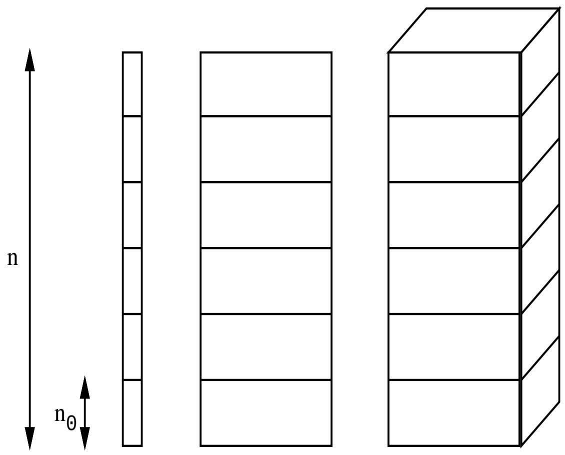

FIG. 12 - *Volume de calcul avec ou sans macro-pipeline pour un vecteur, une matrice 2D et 3D.*

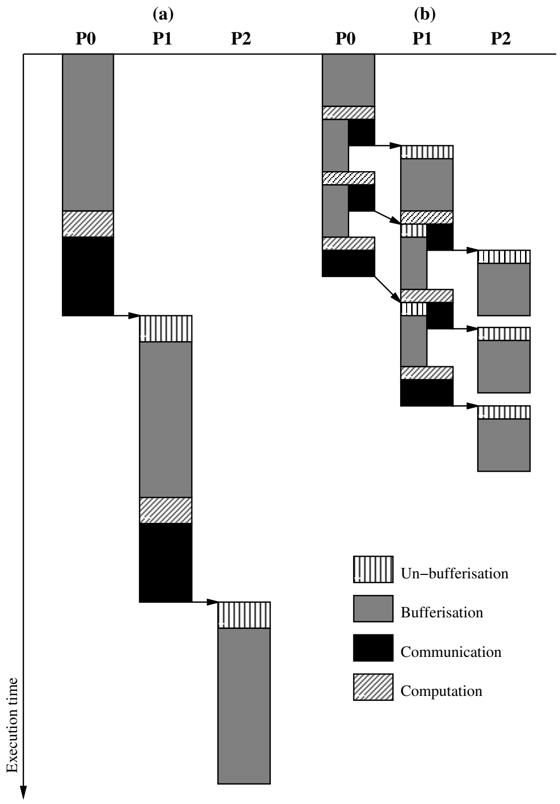

FIG. 13 - *Gain en performances pour P processeurs quand $C_r(n) > T(n)$.*

FIG. 14 - *Gain en performances pour P processeurs quand $C_r(n) < T(n)$.*

Comme le montrent les diagrammes de Gantt des Figures 13 et 14, le temps total d'exécution de la version macro-pipeline est donné par

$$\begin{aligned} P_p^{1D} &= C_r(n_0) + L(n_0) + T(n_0) + (P-2)[C_r(n_0) + 2L(n_0) + T(n_0)] \\ &\quad + \left(\frac{n}{n_0} - 1\right)[\max(L(n_0) + C_r(n_0), T(n_0)) + L(n_0)] + L(n_0) + C_r(n_0) \\ &= PC_r(n_0) + (P-1)[2L(n_0) + T(n_0)] \\ &\quad + \left(\frac{n}{n_0} - 1\right)[\max(L(n_0) + C_r(n_0), T(n_0)) + L(n_0)] \end{aligned} \tag{8}$$

Nous devons maintenant calculer le gain en performances obtenu par l'utilisation du macro-pipeline. Il est égal au rapport entre les temps d'exécution sans macro-pipeline et avec macro-pipeline. Soit $G(n)$ ce rapport, nous avons

$$\begin{aligned} G(n) &= \frac{P_{np}^{1D}}{P_p^{1D}} \\ &= \frac{PC_r(n) + (P-1)[2L(n) + T(n)]}{PC_r(n_0) + (P-1)[2L(n_0) + T(n_0)] + \left(\frac{n}{n_0} - 1\right)[\max(L(n_0) + C_r(n_0), T(n_0)) + L(n_0)]} \end{aligned} \tag{9}$$

Afin de calculer le gain optimal quand n est grand, nous ferons différentes suppositions sur les rapports entre les coûts de calculs, de communications et de latence.

Proposition 1

Si $\lim_{n\to+\infty} \frac{T(n)}{C_r(n)} = 0$, alors le gain optimal est toujours inférieur à P, le nombre de processeurs.

Preuve

Le coût de calcul est alors plus important que le coût de communication. Nous avons donc

$$\begin{aligned} \lim_{n\to+\infty} G(n) &\leq \frac{PC_r(n)}{PC_r(n_0) + \left(\frac{n}{n_0} - 1\right) C_r(n_0)} \\ &\leq \frac{PC_r(n)}{(P-1)C_r(n_0) + C_r(n)} \\ &\leq P \end{aligned} \tag{10}$$

L'hypothèse que nous prenons ici est celle qui a été choisie par Quinn et Hatcher dans leur étude. Elle suppose que le coût des communications devient négligeable lorsque n est très grand. C'est ce cas de figure que l'on retrouve dans des algorithmes tels que la multiplication de matrices, ou l'élimination de Gauss par pivot partiels.

CQFD

Proposition 2

Si $\lim_{n\to+\infty} \frac{T(n)}{C_r(n)} = 1$, alors le gain optimal est toujours inférieur à $2P-1$ (P le nombre de processeurs).

Preuve

Le coût de communication est équivalent au coût de calcul, nous avons alors

$$\begin{aligned} \lim_{n\to+\infty} G(n) &\leq \frac{PC_r(n) + (P-1)T(n)}{PC_r(n_0) + (P-1)T(n_0) + \left(\frac{n}{n_0} - 1\right) C_r(n_0)} \\ &\leq \frac{(2P-1)C_r(n)}{(2P-1)C_r(n_0) + C_r(n)} \\ &\leq 2P - 1 \end{aligned} \tag{11}$$

Cette fois-ci, nous supposons que le coût des communications reste équivalent à celui des calculs lorsque n devient grand. Cette hypothèse est tout à fait justifiée, étant donné qu'elle reflète particulièrement bien la situation observée dans la plupart des codes scientifiques réels. Rappelons que le gain calculé ici est le gain obtenu par le macro-pipeline, par rapport à l'exécution parallèle sans macro-pipeline, et non par rapport à la version séquentielle. Ce qui explique que le gain soit ici supérieur à P.

CQFD

3.5 Cas 2-dimensionnel

Les macro-pipelines peuvent également être utilisés avec une grille bi-dimensionnelle. Ici, nous supposons que nous utilisons une grille carré avec P processeurs ($\sqrt{P} \times \sqrt{P}$). Les résultats suivants peuvent être facilement adaptés à des grilles non-carrés. Le choix de la taille et de la forme de la grille dépendent de la taille et de la forme des matrices impliquées et, bien sûr, des paramètres de la machine cible. Puisque la taille de la matrice est n, chaque processeur reçoit une sous-matrice de taille $n' = n/\sqrt{P}$.

Si nous avons des dépendances de type $(d_1, 0)$ et $(0, d_2)$ avec $d_1, d_2 > 0$, nous pouvons utiliser un macro-pipeline mono-dimensionnel selon chacune des deux dimensions. La Figure 15 montre un macro-pipeline uni-dimensionnel dans la première dimension de la grille 2D. Les zones grisées montrent le chemin critique de l'exécution. Celui-ci est dû à la différence de temps d'exécution des processeurs "rapides" qui sont sur le bord et n'ont donc pas à attendre les données du processeur précédent pour commencer à calculer le bloc suivant.

Soit P_{np}^{2D} le temps total d'exécution lorsqu'aucun macro-pipeline n'est utilisé et P_p^{2D} quand le macro-pipeline est utilisé.

Le temps total d'exécution de l'algorithme sans macro-pipeline est donné par

$$P_{np}^{2D} = \left(2\sqrt{P} - 1\right) C_r(n) + \left(2\sqrt{P} - 2\right) [2L(n) + T(n)] \tag{12}$$

D'après la Figure 15, le temps total d'exécution de la version macro-pipeline de l'algorithme est donné par la somme des temps d'exécution des trois parties du chemin critique.

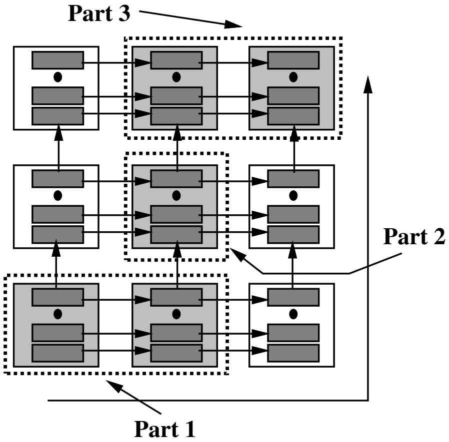

FIG. 15 - *Macro-pipeline 1D sur une grille 2D.*

$$
\begin{aligned}
P_{p1}^{2D} &= C_r(n_0) + L(n_0) + T(n_0) \\
&+ \left(\frac{n'}{n_0} - 1\right)(\max(L'(n_0) + C_r(n_0), T(n_0)) + L(n_0)) \\
&+ C_r(n_0) + \max(L(n'), T(n_0)) + L(n_0) + T(n') \qquad (13) \\
P_{p2}^{2D} &= (\sqrt{P} - 2)[L'(n') + \left(\frac{n'}{n_0} - 1\right)(\max(L'(n_0) + C_r(n_0), T(n_0)) + L(n_0)) \\
&+ C_r(n_0) + L(n_0) + \max(L(n'), T(n_0)) + T(n')] \qquad (14) \\
P_{p3}^{2D} &= L'(n') + C_r(n_0) + L(n_0) + T(n_0) \\
&+ (\sqrt{P} - 3)[L'(n_0) + C_r(n_0) + L(n_0) + T(n_0)] \\
&+ \left(\frac{n'}{n_0} - 1\right)(\max(L'(n_0) + C_r(n_0), T(n_0)) + L(n_0)) + C_r(n_0) \qquad (15)
\end{aligned}
$$

Comme pour le cas mono-dimensionel, notre étude a consisté rechercher quel est le gain obtenu pour des problèmes de grande taille. Les résultats obtenus seront présentés dans un prochain rapport de recherche du LIP.

4 Généralisations

4.1 Extensions à d'autres distributions

La plupart des travaux réalisés autour de la technique du macro-pipeline, se limitent à l'étude de la méthode dans le cas "simple". Les calculs sont effectués sur une matrice à deux dimensions, distribuée selon une dimension sur une ligne de P processeurs (avec des schémas de dépendances simples). Afin de rendre le macro-pipeline utilisable dans d'autres contextes, il était important d'étendre cette étude à d'autres distributions, et ceci pour des matrices à N dimensions. Les langages de programmation Data-Parallèles tels que HPF, offrent une certaine liberté à l'utilisateur dans le choix des distributions. Ainsi, il existe principalement deux façons de distribuer une matrice; la distribution par blocs (**BLOCK**) et la distribution cyclique (**CYCLIC**) (représentées sur la Figure 16).

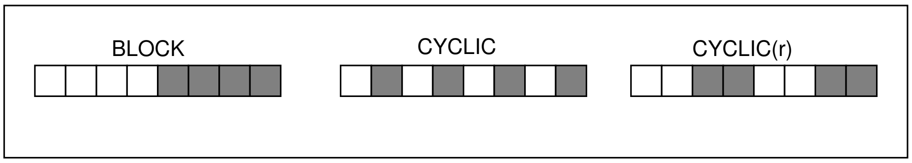

FIG. 16 - *Répartitions par blocs, cyclique et par blocs cycliques de huit données sur deux processeurs.*

4.1.1 La distribution Cyclique

L'étude de la distribution par blocs est la plus courante, car elle présente une certaine simplicité d'implémentation. En effet, l'utilisation de la distribution cyclique est plus complexe pour le compilateur qui doit générer les communications nécessaires. Cependant, c'est une distribution qui s'avère intéressante car elle se caractérise par une grande régularité. La modification du grain n'est pas plus compliquée que pour la distribution par blocs; par contre, la modélisation du temps de communication nécessaire pour une étude poussée du grain optimal, et surtout pour la compilation, est importante à étudier. Considérons une matrice $N \times N$ distribuée comme indiqué sur la Figure 17.

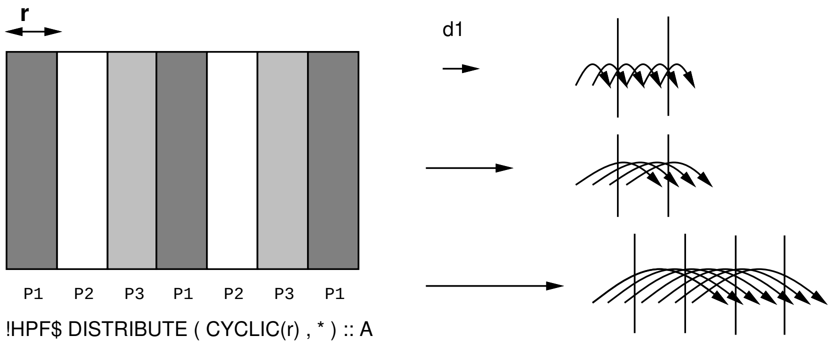

FIG. 17 - *Répartition cyclique de taille r d'une matrice 2D.*

Les communications entre les processeurs vont cette fois-ci dépendre du paramètre r. Le temps de communication sera modélisé par $T(n) = \lambda_c + \beta_c n$, où λ_c décrit le temps de latence, β_c est inversement proportionnel à la vitesse de transmission, et n représente la taille du message à envoyer. Ce dernier paramètre est évidemment fonction du grain de macro-pipeline (noté B), de la taille d'un élément unique du tableau (noté l), et du nombre d'éléments à envoyer. Trois cas sont à envisager selon la valeur du vecteur de dépendance $\vec{d} = (d_1, d_2)$:

- $d_1 \leq r$. Les communications se feront uniquement entre deux processeurs voisins ($P_i \rightarrow P_{i+1}$). On aura alors $T(n) = \lambda_c + \beta_c l d_1 B$.

- $r < d_1 \leq 2r$. Deux types de communications sont cette fois-ci nécessaires, car la dépendance est plus longue. Le processeur P_i envoie un message vers le processeur P_{i+1} avec $T(n) =$

$\lambda_c + \beta_c l(2r - d_1)B$. Mais il envoie aussi un message vers le processeur P_{i+2} avec $T(n) = \lambda_c + \beta_c l(d_1 - r)B$.

- $d_1 > 2r$. Lorsque la dépendance atteint une certaine taille, les communications ne se font plus qu'entre processeurs distants. Le processeur P_i envoie des données au processeur $P_{i+(d_1 \bmod r)}$ avec $T(n) = \lambda_c + \beta_c l(r - d_1 \bmod r)B$. Et il envoie de même un message vers le processeur $P_{i+(d_1 \bmod r)+1}$ avec $T(n) = \lambda_c + \beta_c l(d_1 \bmod r)B$. Notons que dans ce dernier cas de figure, il faudrait tenir compte pour calculer le temps d'exécution total, du fait que $(d_i \bmod r)$ processeurs peuvent travailler de manière complètement parallèle.

L'utilisation du macro-pipeline sur une matrice distribuée de manière cyclique est donc rendue difficile par les communications distantes. Cette augmentation du nombre de messages à gérer est ressentie tant au niveau de la modélisation des temps d'exécution, qu'au niveau de l'implémentation dans un compilateur. En effet, la détermination du chemin critique lorsque plusieurs communications doivent partir ou arriver d'un processeur au même instant, dépend complètement du modèle de communication choisi. Prenons comme exemple le cas de figure intermédiaire $r < d_1 \leq 2r$, et étudions le temps d'exécution du macro-pipeline sur P processeurs en choisissant un modèle de communication 1-port (l'entité chargée de la communication peut recevoir ou envoyer un message sur un seul port à la fois [24]). Étant donné cette hypothèse, le retard au démarrage entre le processeur P_i et le processeur P_{i+1}, est donné par l'équation 16. Finalement, le temps d'exécution total sera alors donné par l'équation 17

$$R = \max(T_{bloc}, T_{comm\_distants}) + T_{comm\_voisins} \tag{16}$$

$$\begin{aligned} T \;=\; & (P-2)\left[\max(T_{bloc}, T_{comm\_distants}) + T_{comm\_voisins}\right] + T_{bloc} + T_{comm\_voisins} \\ & + \frac{N_1}{B} \times \frac{N_2}{Pr}\left[\max(T_{bloc}, T_{comm\_distants}) + T_{comm\_voisins}\right] \end{aligned} \tag{17}$$

4.1.2 Macro-pipelines multidimensionnels

Le macro-pipeline est, comme nous l'avons vu, une modification du grain de calcul. Les travaux précédents se limitaient très souvent à des matrices à deux dimensions, et effectuaient toujours un découpage selon *la dimension qui n'était pas distribuée*. Or, il est tout à fait possible de distribuer la matrice selon n'importe laquelle de ses dimensions, voire même selon toutes ses dimensions.

Nous apportons ici deux **contributions** importantes à la technique du macro-pipeline, en autorisant aujourd'hui le découpage (notion de grain), selon *plusieurs dimensions*, qui peuvent *être ou non distribuées*. La Figure 18 donne un exemple d'utilisation de la technique du macro-pipeline, selon plusieurs dimensions. Les dimensions i et k sont distribuées, et un macro-pipeline est effectué selon les dimensions i et j. Bien que la dimension i soit ici distribuée, il est intéressant de modifier le grain de calcul selon cette dimension, afin d'envoyer plus tôt les données aux processeurs P_3 et P_4. Notons que la présence d'un macro-pipeline selon la dimension i, va ralentir le macro-pipeline selon la dimension j (et vice versa).

Ces premiers résultats laissent d'ores et déjà apparaître plusieurs perspectives de travaux. En effet, il serait aujourd'hui très intéressant de déterminer pour un ensemble {*matrice, dépendances, distribution*}, quelles seront les dimensions selon lesquelles le macro-pipeline sera appliqué de la manière la plus efficace. En effet, ce choix dépend bien évidemment de beaucoup de paramètres, et il y aurait sans aucun doute quelques heuristiques à trouver. Par ailleurs, une fois ce choix effectué,

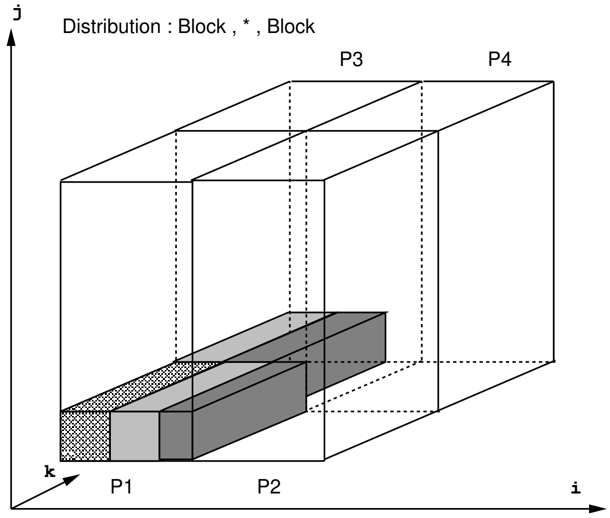

FIG. 18 - *Un macro-pipeline selon deux dimensions.*

il est absolument nécessaire pour des raisons d'efficacité, de déterminer pour chaque dimension à découper, le grain de calcul optimal. Cette tâche devient rapidement complexe, étant donné qu'elle est basée sur la recherche du chemin critique. Là encore, une valeur exacte n'est pas obligatoire, étant donné que l'intervalle sur lequel les performances sont données est relativement important. Nous n'avons pas abordé ces deux questions pour le moment, car elle nécessiteraient une étude prolongée impossible dans le cadre de ce stage de DEA.

4.2 Généralisations à d'autres dépendances

C'est à ce niveau que réside la principale difficulté à rendre le macro-pipeline plus *général*. En effet, le macro-pipeline est pour le moment limité pour deux raisons principales: premièrement, cette technique est étroitement liée à la présence de communications entre les processeurs, et l'avancement actuel des méthodes de compilation de programmes SPMD ne permet pas d'envisager des schémas trop compliqués. On peut espérer que cette limite sera rapidement effacée. Par ailleurs, la modification du grain de calcul est une technique complexe, et qui n'est malheureusement pas toujours envisageable. Ainsi, considérons l'exemple de la Figure 19.

La présence de ces deux dépendances interdit la modification du grain de calcul tel qu'il est. L'instruction $S(i,j)$ supporte deux dépendances portées par la boucle notées $\vec{d} = (1,0)$ (en pointillé) et $\vec{d'} = (1,-1)$ (en trait continu). Le premier bloc, représenté par un rectangle gras sur l'espace d'itération, ne peut débuter à calculer avant de connaître la valeur de $A(2,4)$, qui est mise à jour par le second bloc (en pointillé). De même, ce second bloc ne peut commencer avant de connaître la valeur de A(2,3) mise à jour par le premier bloc. L'exécution pipeline est donc impossible sans redistribution.

Nous nous approchons ici de la technique bien connue du Tiling ([17, 7]), dont l'objectif sera de modifier les indices de boucles et d'effectuer des permutations afin de permettre l'application du grain moyen. Cependant, ces techniques se placent à un niveau plus "abstrait" dans l'échelle de la parallélisation automatique, et nécessiteront souvent une redistribution des données sur les processeurs. Lorsque la redistribution ne dégrade pas les performances, nous appliquerons le macro-pipeline sur les nids de boucles résultats des modifications du tiling. Nous nous intéresserons ici,

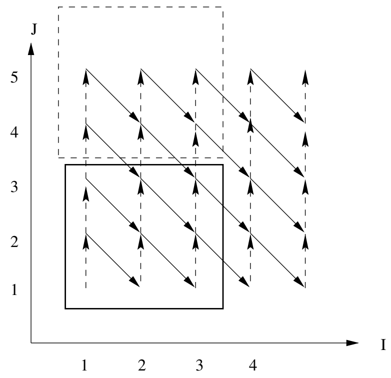

FIG. 19 - *Macro-pipeline impossible sans redistribuer.*

uniquement aux cas où la distribution est imposée. Ce sera souvent le cas, lorsque plusieurs calculs seront appliqués sur une même structure de données, avec des schémas de dépendances différents.

La généralisation des techniques de modification du grain de calcul, est étudiée dans [20], pour des dépendances plus "compliquées". Cet article se place en fait à l'intersection entre les techniques de Tiling, DO_ACCROSS et Macro-pipeline. On y trouve des résultats intéressants, mais qui se limitent à une distribution selon une seule dimension. Le principal résultat qu'ils proposent est la généralisation du macro-pipeline à une dépendance du type $\vec{d} = (d_1 > 0, d_2 < 0)$ à l'aide d'une torsion de l'espace d'itération.

L'existence d'une dépendance $\vec{d} = (d_1 > 0, d_2 > 0)$ n'interdit pas l'utilisation de la technique classique du macro-pipeline; celle-ci n'est cependant pas du tout optimale. Les auteurs de [20] introduisent l'idée qu'une torsion de l'espace d'itération peut permettre d'extraire une plus grande quantité de parallélisme. Nous sommes donc allés plus loin dans l'étude de ce mécanisme.
Considérons un nid de boucles donnant lieu à des dépendances telles qu'elles sont représentées sur la Figure 20(A) (on accède à une matrice A de taille $N_1 \times N_2$). Grâce à une torsion de l'espace d'itération, on se ramène à la situation proposée sur la Figure 20(B).

La matrice A est distribuée par blocs sur une ligne de P processeurs. On remarque en analysant la figure que chaque processeur possède des données qu'il peut immédiatement commencer à calculer (marqué par des triangles en gras). Nous allons découper l'espace d'itération en deux parties : la partie inférieure notée *startup* est la zone de calcul qui va permettre d'améliorer les performances obtenues par rapport à un pipeline standard. La partie supérieure, notée *macro-pipeline*, va être traitée de manière habituelle.

Phase de Startup Afin de tirer profit au maximum du parallélisme, l'exécution de cette phase peut avoir lieu comme indiqué sur la Figure 21(A). Chaque processeur travaille sur une taille de bloc B_{start} qui est déterminée par l'inclinaison de l'espace et la distribution des processeurs ($B_{start} = \frac{N_1 d_2}{P d_1}$). L'exécution est parfaitement parallèle avec des phases de synchronisation régulières. Le temps d'exécution de cette phase de startup est alors donné par l'équation 18.

$$T_{start} = \frac{N_1^2 d_2}{2P^2 d_1} t + (P-1) \left[\lambda_c + \frac{N_1 d_2}{P d_1} \beta_c l d_1 + \frac{N_1^2 d_2}{P^2 d_1} t \right] \tag{18}$$

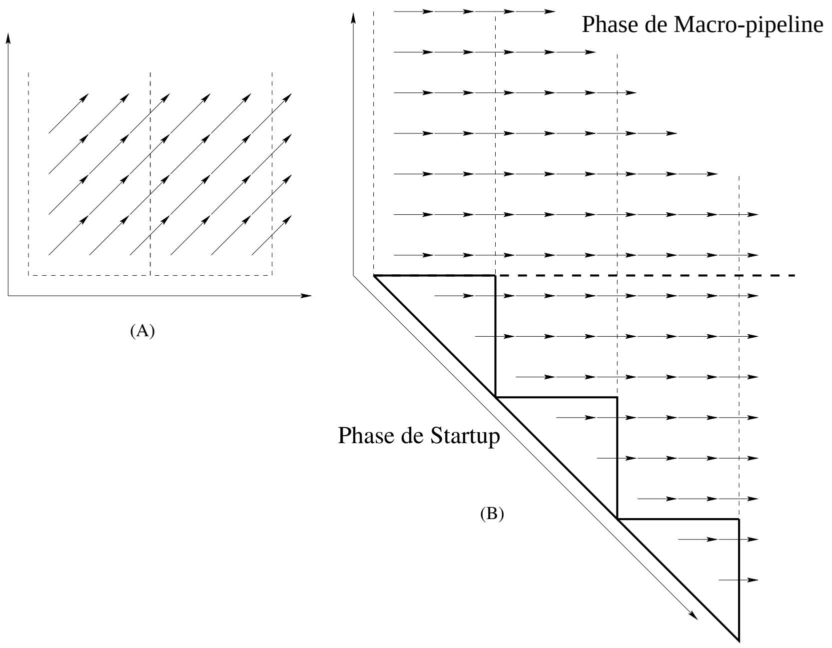

FIG. 20 - *Torsion de l'espace d'itération.*

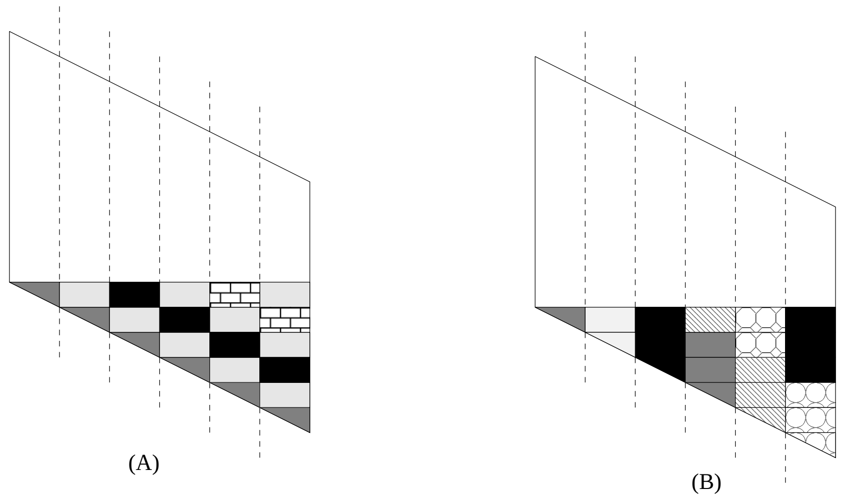

FIG. 21 - *Phase de startup.*

Bien que le maximum de parallélisme soit ici utilisé, les nombreuses synchronisations nécessaires peuvent devenir très coûteuses. En effet, si l'on considère une situation dans laquelle l'inclinaison est faible et la matrice très grande, l'exécution contiendra alors beaucoup trop de petites communications (et donc de λ_c). Ainsi, il peut s'avérer intéressant de retarder un peu le démarrage des processeurs, pour ensuite gagner beaucoup en temps de communication (moins de temps de latence). C'est cette situation que représente la Figure 21(B). Il est dans ce cas nécessaire de déterminer la taille de bloc optimale (notée B'_{start}). La modélisation du temps d'exécution est celle d'un pipeline classique sur un espace d'itération *tordu*. Le temps d'exécution est alors donné par l'équation 19.

$$\begin{aligned} T'_{start} &= (P-1)\left[\frac{N_1^2 d_2}{2P^2 d_1}t + \frac{N_1}{P}t\left(B'_{start} - \frac{N_1 d_2}{P d_1}\right)\right] \\ &+ \frac{N_1 d_2}{B'_{start} d_1}\left[B'_{start}\frac{N_1}{d_1}t + \lambda_c + B'_{start}\beta_c l d_1\right] \end{aligned} \tag{19}$$

En annulant la dérivée de T'_{start} par rapport à B'_{start}, on obtient la taille de bloc optimale: $B'_{start} = \sqrt{\frac{d_2 \lambda_c P}{t d_1 (P-1)}}$. Finalement, au moment de l'exécution, lorsque les paramètres du problème sont connus, on peut choisir entre les deux méthodes et définir la taille de blocs en fonction du résultat de $T_{start} - T'_{start}$.

Phase de macro-pipeline La phase de macro-pipeline n'est qu'une simple application de la méthode classique, à un espace d'itération encore une fois *tordu* (Figure 22(A)). La difficulté réside maintenant dans la recherche du chemin critique.

Proposition 3

Si $B_{pipe} \geq \frac{N_1 d_2}{P d_1}$, alors les calculs de la phase de startup pour le processeur P_i seront terminés avant que le processeur P_{i-1} ne soit en mesure de lui envoyer les données dont il a besoin pour commencer la phase de macro-pipeline.

Preuve

Démonstration par Récurrence

Hypothèse: $B_{start} = B'_{start}$ (on se limite ici à l'étude du pire cas)

- *Processeur P_1:* Aucune contrainte
- *Processeur P_2:*

$$\begin{aligned} B_{pipe}\frac{N_1}{P}t + \lambda_c + B_{pipe}\beta_c l d_1 &\geq \frac{N_1 d_2}{P d_1}\frac{N_1}{P}t + \lambda_c + \frac{N_1 d_2}{P d_1}\beta_c l d_1 \\ B_{pipe} &\geq \frac{N_1 d_2}{P d_1} \end{aligned}$$

- *Processeur P_i*:
  Hypothèse de Récurrence: la proposition est vraie au niveau P_{i-1}

$$\begin{aligned} (P-1)\left[\frac{N_1}{P}tB_{pipe} + \lambda_c + B_{pipe}\beta_c l d_1\right] &\geq (P-1)\left[\frac{N_1 d_2}{P d_1}\frac{N_1}{P}t + \lambda_c + \frac{N_1 d_2}{P d_1}\beta_c l d_1\right] \\ (P-1)\left(\frac{N_1 t}{P} + \beta_c l d_1\right)\left[B - \frac{N_1 d_2}{P d_1}\right] &\geq 0 \\ B_{pipe} &\geq \frac{N_1 d_2}{P d_1} \end{aligned}$$

CQFD

Le chemin critique de l'exécution sera donc celui représenté en gras sur la Figure 22(B) lorsque l'on a $B_{pipe} \geq \frac{N_1 d_2}{P d_1}$, et celui représenté en pointillé sinon. En tirant profit du parallélisme inhérent à la boucle, nous pouvons donc masquer le temps d'exécution de la phase de startup, et ainsi augmenter encore le gain obtenu par rapport à une exécution sans pipeline.

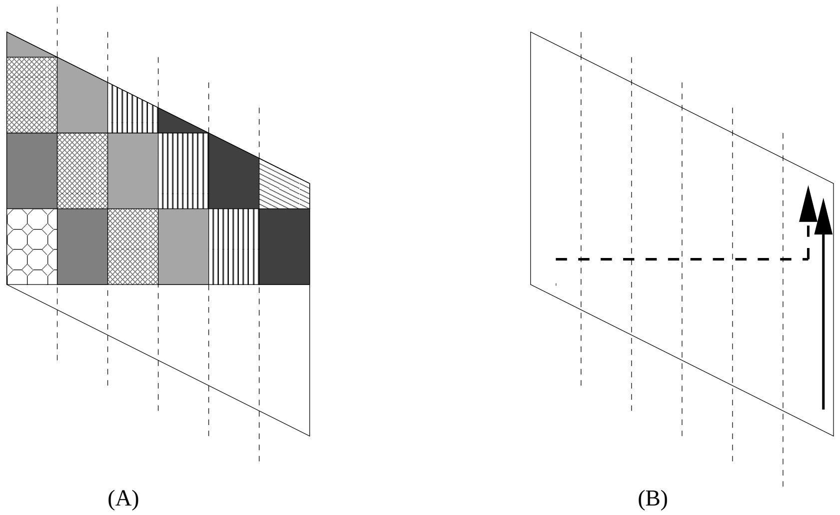

FIG. 22 - *Phase de macro-pipeline.*

5 Implémentation et Résultats

Tous les travaux effectués au cours de ce stage se situent dans le cadre du projet **TransTOOL** dirigé par Frédéric Desprez. L'objectif de ce projet est de fournir un outil interactif d'aide à la parallélisation d'applications scientifiques sur Machines Parallèles à Mémoire Distribuée. Cet outil sera utilisé comme interface avec un grand nombre d'autres outils, tels que des compilateurs, des bibliothèques, des outils de simulation ou visualisation. TransTOOL permettra ainsi aux scientifiques de produire une version parallèle de leur code source séquentiel. La plupart des applications scientifiques étant écrites en Fortran, le langage cible choisi est HPF.
Notre objectif est donc de permettre aux utilisateurs de TransTOOL, de profiter de la technique des macro-pipelines, pour optimiser leurs programmes. La première action proposée à l'utilisateur sera donc celle de sélectionner un nid de boucles, pour savoir si celui-ci correspond effectivement à un schéma d'exécution type *pipeline*. Dans ce cas, ce dernier sera annoté de façon particulière, puis selon le compilateur choisi, le nid de boucles sera remplacé par un appel à une routine effectuant le découpage par bloc.

5.1 Implémentation des macro-pipelines

La technique du macro-pipeline nécessite l'ajout de communications de bas niveau qui rendent la programmation difficile et vont à l'encontre de l'idée de parallélisation automatique. Par ailleurs, le programmeur doit être capable d'identifier les schémas pipelines au sein de son programme. L'intégration dans un compilateur est donc indispensable pour rendre cette technique plus abordable.

5.1.1 Identification

La première étape consiste à identifier les nids de boucles qui sont adaptés à l'utilisation de la technique *macro-pipeline*. Tseng a introduit dans [26] la notion de **Cross Processor Loops**, et propose l'algorithme de la figure 23.

– Une boucle i est appelée *Cross Processor* si ses itérations sont distribuées le long des processeurs, et si elle possède une dépendance vraie portée par la boucle $\vec{d} = (0, .., d_i, .., 0)$ avec $d_i \neq 0$. Dans ce cas, un processeur a besoin des résultats calculés par son voisin pour commencer sa propre itération.

/* Entrée de l'algorithme */
  Nids de boucles $\{L_1, L_2, ..., L_n\}$ avec les indices $\{I_1, I_2, ..., I_n\}$
  Liste de Toutes les dépendances vraies portées par la boucle
  Distribution de toutes les matrices accédées

/* Sortie de l'algorithme */
  CP_Loops = Liste de boucles *Cross Processor*

/* Algorithme */
CP_Loops $\leftarrow \emptyset$
Pour chaque dépendance vraie portée par la boucle
    entre deux références $A(f_1, ..., f_m)$ et $A(f_1, ..., f_m)$ **faire**
  Pour chaque dimension k distribuée de A **faire**
    $\{f_k$ et g_k sont les indices de la dimension k de A $\}$
    Si $f_k \neq g_k$
      Pour chaque variable I_j dans f_k ou g_k **faire**
        Si L_j contient les deux références à A Alors
          CP_Loops $\leftarrow$ CP_Loops $\cup \{L_j\}$
        finSi
      finPour
    finSi
  finPour
finPour

FIG. 23 - *Algorithme de Tseng d'Identification des boucles Cross Processor.*

Cet algorithme permet effectivement de déterminer quels sont les boucles *Cross Processor*. Cependant, il est très important de ne faire l'amalgame entre cette notion, et la notion de macro-pipeline. En effet, le nid de boucles présenté sur la Figure 5.1.1 est composé de deux boucles qui seront toutes les deux identifiées comme Cross-Processor par cet algorithme; pourtant il ne sera pas forcement possible d'utiliser le macro-pipeline sans effectuer au préalable une redistribution. Ainsi, comme on peut le voir sur la Figure 25, l'espace d'itération du tableau A (ici un vecteur) n'est pas aligné avec l'espace d'itération du nid de boucle (correspondant *réellement* aux calculs). En effet, tous les calculs tels que $i + j = cste$ accèdent à la même case mémoire. On ne pourra donc pas prendre de décision sur la possibilité de faire ou non du macro-pipeline, sans faire une étude plus poussée des communications. Les *dépendances* et *communications*, sont deux notions liées l'une à l'autre, mais ne le sont pas toujours directement.

Tous les nids de boucles que nous avons considéré jusqu'à présent, étaient tels que l'instruction $S(i, j)$ accédait à des tableaux de la forme $a(i, j)$. Celà n'est évidemment pas toujours le cas. L'algorithme de Tseng ne doit donc être considéré que comme une première étape dans l'identification automatique des boucles adaptées au macro-pipeline. Nous pourrions notamment l'utiliser en vérifiant au préalable que le nid de boucle considéré est parfait et *fully-permutable* et que les tableaux accédés sont de dimension pleine [3, 19, 9]. Nous ne chercherons pas pour l'instant compléter cette étude, mais elle sera très importante en vue d'étendre le macro-pipeline à des schémas de dépendances compliqués. Nous pourrions simplement dans l'immédiat utiliser l'algorithme en vérifiant au préalable que le nid de boucle considéré est parfait et fully-permutable et que les tableaux accédés sont de dimension pleine [3, 9, 19]. Sous ces hypothèses, toutes les boucles qui seront annotées *cross processor* par l'algorithme de Tseng seront adaptés au macro-pipeline (le contraire est faux). Elles seront alors marquées par une directive spéciale (par exemple !ADP$ MACRO_PIPELINE).

```
       REAL A(N)
!HPF$  DISTRIBUTE A(BLOCK)
       ...
       DO I=1,N
         DO J=1,N
  (S)        A(I+J) = f(A(I+J-1))
         END DO
       END DO
```

FIG. 24 - *Un nid de boucle complexe.*

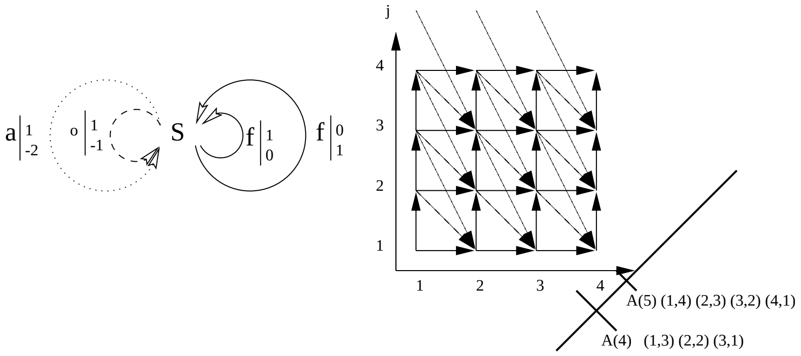

FIG. 25 - *Les dépendances portées par l'instruction S et l'espace d'itération du nid de boucle .*

Selon le compilateur choisi, le nid de boucle annoté *Cross Processor* sera remplacé par un appel à une routine (que nous appelerons ici le driver) s'occupant du découpage par blocs et de la génération des communications de bas niveau. Une routine locale contenant le code à traiter par chaque processeur sera par ailleurs inserée. Il suffira pour cela de recopier le nid de boucle initial en ne modifiant que les indices de boucles, qui devront être gérés directement par le driver.

5.1.2 Les Macro-pipelines dans Adaptor

Plusieurs travaux récents, envisagent la mise en place des macro-pipelines dans leur compilateur (FortranD [26], PARADIGM, [21]), mais à notre connaissance, personne n'a pour l'instant considérablement avancé. Nous avons implémenté une première version des macro-pipelines, en travaillant en collaboration avec Thomas Brandes sur le compilateur Adaptor.

Le compilateur HPF Adaptor **ADAPTOR** (Automatic DAta Parallelism translaTOR) est un outil de compilation du domaine public (développé en Allemagne à GMD/SCAI) qui effectue une transformation source-à-source (Figure 26). Il prend en entrée un programme Data-Parallèle HPF, et le traduit en un programme SPMD qui sera exécuté par chaque noeud. Un support d'éxécution appelé **DALIB** (Distributed Array LIBrary) a été développé dans le but d'y placer toutes les fonctions de transposition, redistribution, réduction, etc. La bibliothèque ayant été développée pour pratiquement toutes les machines parallèles classiques (MIMD), le programme généré peut être exécuté sur toutes ces machines.

La routine `dalib_loccs_driver` Plusieurs approches sont possibles pour implémenter les macro-pipelines dans un compilateur. Nous avons préféré l'approche runtime à la génération de code directement par le compilateur (comme le propose Tseng par exemple). En effet, celle-ci est relativement complexe, et il est quasiment obligatoire de passer par l'intermédiaire d'un langage intermédiaire.

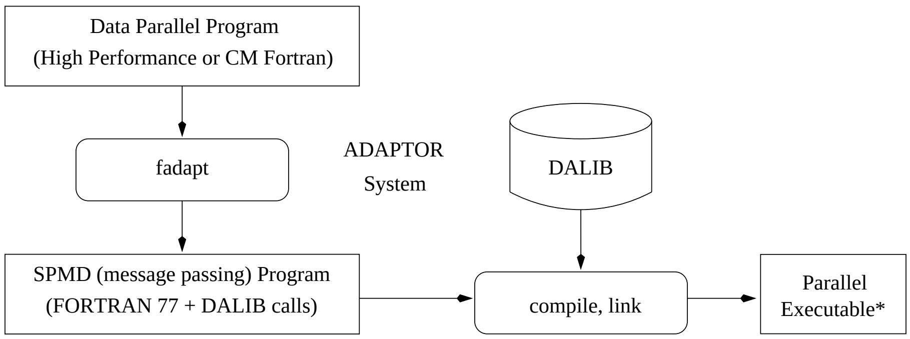

FIG. 26 - *Une vue d'ensemble d'ADAPTOR.*

D'autre part, l'utilisation d'une routine du support runtime facilite la conservation de l'aspect *portabilité* d'Adaptor. Enfin, l'approche *runtime* permettra de considérer avec succès les cas où la distribution n'est connue qu'à l'exécution.

Le driver s'occupant des macro-pipelines se trouve dans la bibliothèque Dalib et se nomme `dalib_loccs_driver`. Cette routine sera appelée de la manière suivante dans le programme HPF (voir Figure 36) :

```
call DALIB_LOCCS_DRIVER (subroutine, n, m,
                         a1, d1, a2, d2, ..., an, dn,
                         b1, ..., bm)
```

- Le premier paramètre `subroutine` est le nom de la routine locale qui contient le code exécuté par chaque noeud (nid de boucle). (voir algorithme ADI Figure 36)
- Le paramètre `n` donne le nombre de tableaux qui devront être découpés par le driver.
- Le paramètre `m` donne le nombre d'arguments (les b_i) qui seront passés à la routine locale, comme paramètres fortran usuels.
- Chaque paramètre a_i est un tableau; tous les tableaux accédés doivent être alignés les uns avec les autres. Dans le cas contraire, il faudrait éclater la boucle et appeler plusieurs fois le driver.
- Chaque paramètre d_i décrit les dépendances spécifiques à un tableau a_i. Cette information est nécessaire pour mettre en place les communications.

Le principe général de la routine `dalib_loccs_driver` est présenté sur la Figure 27. A chaque étape, une structure de données est utilisée pour déterminer quelles sont les données à recevoir (ou envoyer) et quel est le processeur avec qui aura lieu cette communication.

5.2 Premiers résultats

Il est actuellement possible d'utiliser de manière très efficace les macro-pipelines avec le compilateur Adaptor. Les matrices peuvent être de dimension quelconque, et distribuées selon une ou plusieurs dimensions. De même, le macro-pipeline peut être effectué selon autant de dimensions que souhaité (que celles-ci soient ou non distribuées).

Déterminer les dimensions qui seront pipelinées. (`dalib_choose_pipelining`)
Calculer la taille de bloc optimale pour chaque dimension pipelinée.
Tant qu'il reste des *blocs de calcul* **Faire**
  déterminer l'intervalle de travail du bloc (`dalib_loccs_set_job`)
  recevoir les données (`dalib_loccs_receive`)
  calculer les données (`dalib_loccs_do_job`)
  envoyer les données (`dalib_loccs_send`)
finTantque

FIG. 27 - *Principe de fonctionnement du driver.*

Nous avons installé Adaptor sur trois Machines Parallèles à Mémoire Distribuée afin de pouvoir effectuer nos tests de performances.

- Le **CRAY T3D** (128 processeurs) au *CENG*,
- Le **PARAGON** (30 processeurs) à *l'Université de Lyon1.*
- Le **SP2** (16 processeurs) au *Labri- Université Bordeaux 1.*

Nous avons rencontré plusieurs difficultés au cours de ces expérimentations; en effet, la SP2 est une machine toute récemment installée, et pose pour le moment des problèmes de stabilité des résultats (les coûts communications sont très irréguliers). La Paragon se caractérise quant à elle par des coûts souvent perturbateurs de manipulation des données dans le cache. Ceci a rendu impossible la comparaison entre la version séquentielle et la version Parallèle; les matrices distribuées sur P processeurs entraient dans le cache, alors que les matrices de même taille prises en compte par un unique processeur, nécessitaient sans arrêt des entrées-sorties hors cache. Enfin la CRAY T3D est une machine très puissante, mais qui n'est malheureusement pas souvent libre... Les tests de performances présentés en annexe ont été effectués pour la plupart sur la T3D, car cette dernière donne des résultats plus constants (nous avons calculé des temps moyens sur 25 Itérations).

Sur toutes les figures, `seq` représente le temps séquentiel sur un processeur, `normal` (ou `norm`) représente le temps parallèle sans utilisation du macro-pipeline (c'est-à-dire en gros le temps séquentiel plus le surcoût des communications entre les processeurs), et `pipeline` (ou `pip`) le temps du programme utilisant le macro-pipeline.

Deux types de résultats sont intéressants à étudier : il est tout d'abord important de bien observer l'évolution du temps d'exécution en fonction du grain de calcul choisi (Figure 29 en annexe). Celui-ci est très important pour une taille de bloc réduite, car les communications coûtent cher. Il diminue ensuite considérablement jusqu'à atteindre le meilleur compromis possible entre coûts de communications et retard au démarrage du dernier processeur. Puis, il recommence à augmenter, de manière plus lente.
Par ailleurs, le gain obtenu par le macro-pipeline par rapport d'une part, à l'exécution séquentielle et d'autre part, à l'exécution parallèle sur P processeurs sans macro-pipeline est intéressant à observer. Notons qu'il est impossible d'atteindre ici des gains de l'ordre de ceux calculés au Paragraphe 3, car la version asynchrone de la routine *dalib_loccs_driver* n'est pas encore disponible. Nous obtenons cependant des gains tout à fait prometteurs comme ceux présentés sur les Figures 28 et 30 (la Figure 31 présente les mêmes résultats que la Figure 30 mais avec une échelle semi-logarithmique qui permet de mieux voir la taille optimale de blocs).

Les résultats des Figures 32 et 33 ne sont pas vraiment bons, et sont en celà assez intéressants. La Figure 32, présente le speedup atteint sur 16 processeurs pour une matrice de taille très limitée. Nous voyons ici que le gain obtenu par le macro-pipeline n'est pas suffisant pour compenser le surcoût introduit par les communications et que l'exécution séquentielle serait toujours meilleure. Rappelons tout de même, que dans le cadre d'un compilateur Data-Parallèle, le gain réel est celui obtenu par rapport à l'exécution sur P processeurs sans macro-pipeline. De plus, ce genre de résultats ne serait plus vrai sur une machine comme la Paragon, qui comme nous l'avons vu introduit le facteur *cache*. La Figure 33 présente le speedup atteint pour le macro-pipeline selon deux dimensions d'une matrice 3D. Les gains obtenus sont beaucoup plus faibles que ceux que l'on pourrait attendre. Ceci s'explique par le fait qu'il faudrait dans ce contexte faire varier de manière indépendante les deux tailles de bloc (selon les deux dimensions à découper), pour obtenir des résultats corrects.

6 Conclusion

Nous avons tout au long de cette étude choisi de mener de front une approche théorique basée sur la modélisation du macro-pipeline, et une approche plus concrète visant à implémenter les diverses techniques dans un compilateur Data-Parallèle. Les aboutissements de tous ces travaux sont très prometteurs, et nous laissent entrevoir de nombreuses perspectives. Nous avons en effet pu prouver l'efficacité des macro-pipelines en apportant à la fois une nouvelle modélisation de cette méthode, et des résultats réels très encourageants. Nous avons par ailleurs permis d'envisager le macro-pipeline de façon plus générale, en introduisant la notion de macro-pipeline multidimensionnel, qui offrira des gains beaucoup plus importants pour des matrices à N-dimensions.

Le succès des techniques de recouvrement vient entre autres de l'échec rencontré aujourd'hui par les compilateurs Data-Parallèles, dont les programmes générés ont des performances assez médiocres sur de “vraies” applications. La prise en compte de la distribution des données sur les processeurs est un facteur capital, qui fait sans aucun doute défaut aux méthodes classiques de parallélisation de boucles. Ainsi, les techniques telles que le Tiling ou la méthode de l'hyperplan sont très efficaces lorsqu'elles ont toute liberté sur le choix de la distribution. Cependant, quand celle-ci est imposée par le programme (ce qui est souvent le cas dans les programmes réels), le coût de la redistribution dégrade énormément les gains obtenus par l'extraction du parallélisme.

Nous avons apporté une contribution au domaine des recouvrements calculs communications mais cependant les travaux sur le sujet sont loin d'être terminés. L'implémentation que nous envisageons de l'algorithme d'identification automatique des schémas pipelines a été retardé d'une part suite à la confusion exposée Paragraphe 5.1.1, et d'autre part, à cause de la difficulté technique que représente la récupération de toutes les informations nécessaires dans l'arbre syntaxique d'Adaptor. Par ailleurs, nous nous sommes heurté à des problèmes de détermination des communications à générer par le compilateur. Ces différents problèmes, sont finalement tout à fait franchissables, mais demanderaient un travail plus poussé, impossible en si peu de temps. Nous avons enfin rencontré quelques difficultés de modélisation des temps d'exécution, dès lors que nous nous sommes penchés sur des structures multidimensionnelles.

Plusieurs perspectives de travaux peuvent donc être envisagées suite à cette étude. La première étape est aujourd'hui de travailler à la mise en oeuvre d'une heuristique de détermination des dimensions à pipeliner. Celle-ci s'accompagnerait bien sûr de la recherche des tailles de blocs optimales pour chaque dimension. Par ailleurs, nous envisageons, dans le cas d'un macro-pipeline sur plusieurs instructions, d'étudier le regroupement lorsque cela est possible des différentes communications à engendrer(des communications communes à toutes les matrices). Ceci permettrait en effet de diminuer encore considérablement les temps de latence. L'extension des résultats à des configurations plus complexes, telles que par exemple la présence de dépendances affines, est nécessaire au bon développement des macro-pipelines. Cependant, ces études théoriques doivent être accompagnées de travaux sur les compilateurs, afin de permettre leur implémentation future. Toutes ces généralisations posent évidemment le problème de la définition de ce qu'est une boucle *cross processor*, et de son identification automatique. Enfin, il serait intéressant aujourd'hui d'étudier de manière plus formelle, les performances relatives obtenues par les différentes approches existantes (par exemple le macro-pipeline, le Tiling avec redistribution, la méthode de l'hyperplan, etc...). En effet, selon les caractéristiques de la machine et le cas traité, l'une ou l'autre de ces méthodes peut s'avérer plus performante. Il existe aujourd'hui beaucoup de techniques de parallélisation de boucles, et il serait judicieux d'essayer d'élaborer des critères de choix selon le contexte.

Références

[1] Adve (V.S.), Kennedy (K.) et Mellor-Crummey (J.). – *Performance Analysis of Data-Parallel Programs.* – Rapport technique n° CRPC-TR94405, Rice University, Center for Research on Parallel Computation, May 1994.

[2] André (F.). – *A Multi-Threads Runtime for the Pandore Data-Parallel Compiler.* – Rapport technique n° 986, IRISA, 1996.

[3] Bacon (D.), Graham (S.) et Sharp (O.). – Compiler transformations for high performance computing. *In: ACM Computing Surveys.* – 1994.

[4] Banerjee (U.). – *Dependence Analysis for Supercomputing.* – New York, USA, Kluwer Academic Publisher, 1988.

[5] Banerjee (U.). – An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, vol. 2, 1988, pp. 133–149.

[6] Berntsen (J.). – Communication Efficient Matrix Multiplication on Hypercubes. *Parallel Computing*, vol. 12, 1989, pp. 335–342.

[7] Boulet (Pierre), Darte (Alain), Risset (Tanguy) et Robert (Yves). – (Pen)-ultimate tiling? *INTEGRATION, the VLSI Journal*, vol. 17, 1994, pp. 33–51.

[8] Brandes (T.) et Desprez (F.). – Implementing Pipelined Computation and Communication in an HPF Compiler. – Submitted to Europar'96, 1996.

[9] Darte (A.). – *Techniques de Parallélisation Automatique de Nids de Boucles.* – Thèse de PhD, ENS-Lyon, Avril 1993.

[10] Desprez (F.). – *Procédures de Base pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée.* – Thèse de PhD, Institut National Polytechnique de Grenoble, janvier 1994. LIP ENS-Lyon.

[11] Desprez (F.) et Pourzandi (M.). – A Comparison of Three Matrix Product Algorithms on the Intel Paragon and Archipel Volvox Machines. *In: Advances in Numerical Methods and Applications NM&A - $O(h^3)$'94*, éd. par I.T.Dimov (Bl. Sendov) et P.S.Vassilevski. pp. 234–244. – World Scientific.

[12] Desprez (F.), Ramet (P.) et Roman (J.). – Optimal Grain Size Computation for Pipelined Algorithms. *In: Europar'96.* – to appear.

[13] Eicken (T. Von), Culler (D.E.), Goldstein (S.C.) et Schauser (K.E.). – Active Message: a Mecanism for Integrated Communication and Computation. *In: The 19th Annual Symposium on Computer Architecture*, éd. par Society (ACM IEEE Computer). pp. 256–266. – ACM Press.

[14] Fahringer (T.), Haines (M.) et Mehrotra (P.). – On the Utility of Threads for Data Parallel Programming. *In: International Conference on Supercomputing.*

[15] Felten (E.W.) et Namee (D. Mc). – Improving and the Performance of Message Passing Applications by Multithreading. *In: Scalable High-Performance Computing Conference*, éd. par Society (IEEE Computer). pp. 84–89. – IEEE Computer Society Press.

[16] Hiranandani (S.), Kennedy (K.) et Tseng (C.W.). – Compiling Fortran D for MIMD Distributed Memory Machines. *Communication of the ACM*, vol. 35, n° 8, août 1992, pp. 66–80.

[17] Irigoin (François) et Triolet (Rémi). – Supernode partitionning. *In : Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 319–329. – San Diego, CA, janvier 1988.

[18] Koelbel (C.), Loveman (D.), Schreiber (R.), Jr. (G. Steele) et Zosel (M.). – *The High Performance Fortran Handbook.* – Scientific and Engineering Computation Series, 1994.

[19] Kulkarni (D.) et Stumm (M.). – *Loop and Data Transformations: A tutorial.* – Rapport technique, University of Toronto, 1993.

[20] Ohta (H.), Saito (Y.), Kainaga (M.) et Ono (H.). – Optimal Tile Size Adjustement in Compiling General DOACROSS Loop Nests. *In : International Conference on Supercomputing*, éd. par Press (ACM). ACM SIGARCH, pp. 270–279. – Barcelona, Spain, juillet 1995.

[21] Palermo (D.J.), Su (E.), Chandy (J.A.) et Banerjee (P.). – Communication Optimizations Used in the Paradigm Compiler for Distributed Memory Multicomputers. *In : International Conference on Parallel Processing.* – St Charles, IL, August 1994.

[22] Quinn (M.J.) et Hatcher (P.J.). – On the Utility of Communication-Computation Overlap in Data-Parallel Programs. – To appear in Journal of Parallel and Distributed Computing, 1996. `http://www.cs.unh.edu/Personal/pjh.html`.

[23] Richardson (Harvey). – *High Performance Fortran: history, overview and current developments.* – Thinking Machines Corporation, 245 Firts Street, Cambridge, USA, March 1995.

[24] Rumeur (J.D.). – *Communication Dans Les Réseaux de Processeurs.* – Masson, 1993.

[25] Siegel (B.S.) et Steenkiste (P.A.). – Controlling Application Grain Size on a Network of Workstations. *In : Supercomputing'95.* – http://www.cs.cmu.edu/afs/cs/project/nectar/WWW/gnectar_papers.html.

[26] Tseng (C.W.). – *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* – Thèse de PhD, Rice University, janvier 1993.

A Annexe

Macro-Pipeline Selon 1 Dimension
Matrice 2-D de taille 1000×1000

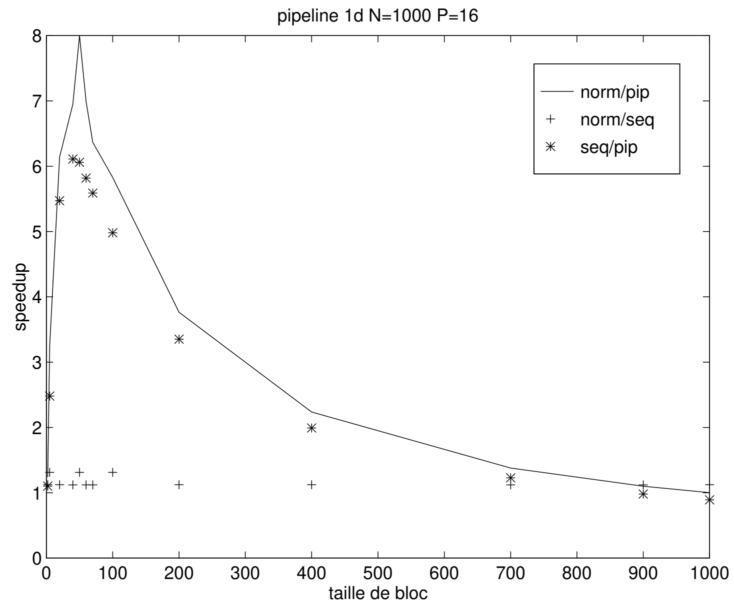

FIG. 28 - *Gains obtenus sur 16 processeurs.*

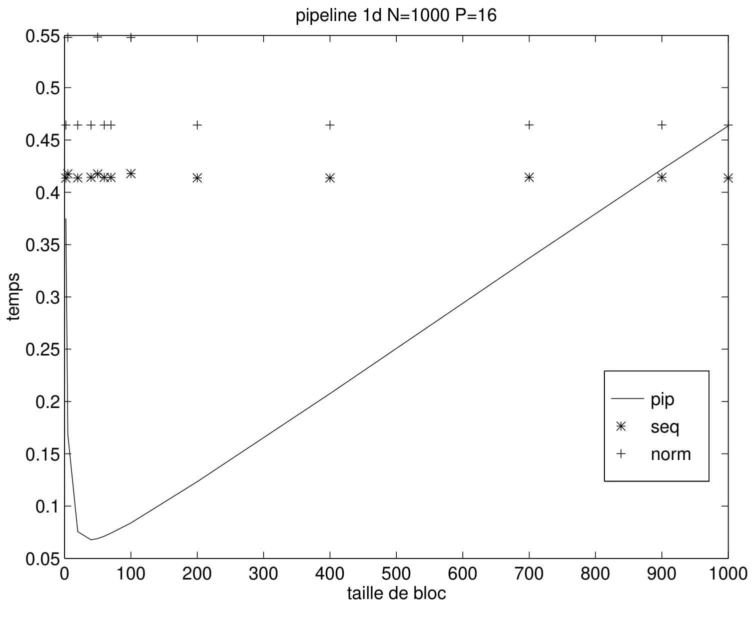

FIG. 29 - *Temps obtenus sur 16 processeurs.*

Matrice 2-D de taille 5000×5000

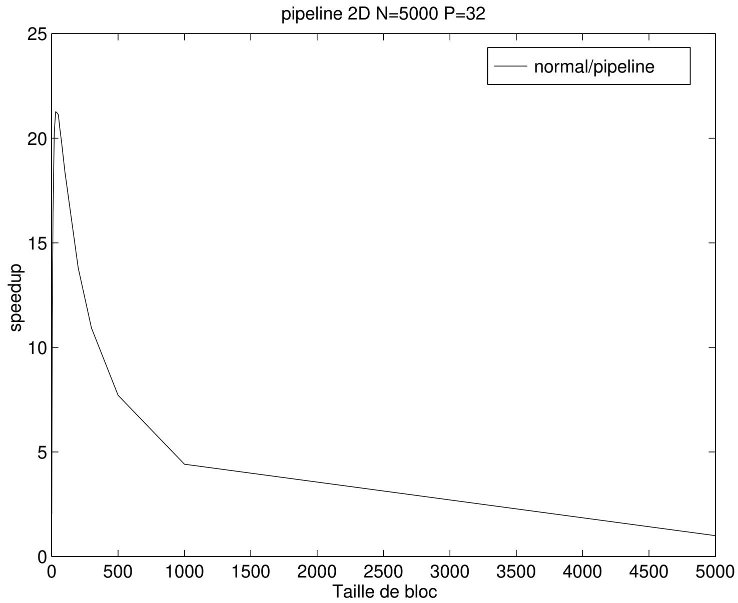

FIG. 30 - *Gains obtenus sur 32 processeurs.*

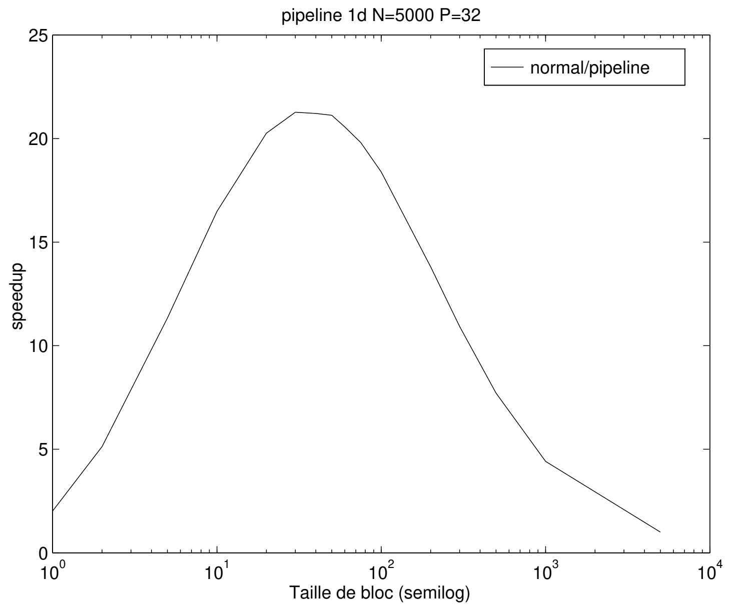

FIG. 31 - *Gains obtenus sur 32 processeurs (échelle semi-logarithmique).*

Matrice 2-D de taille 200×200

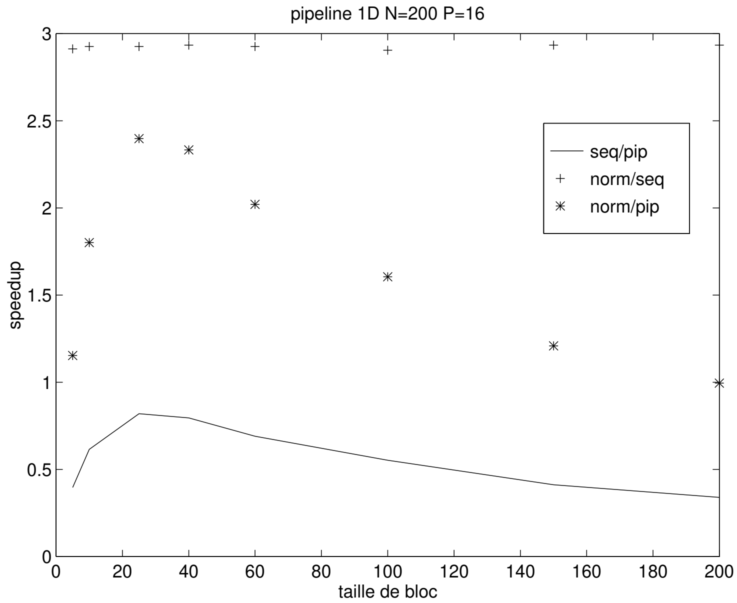

FIG. 32 - *Gains obtenus sur 16 processeurs.*

Macro-pipeline Multidimensionnel
Matrice 3-D de taille $200 \times 200 \times 200$

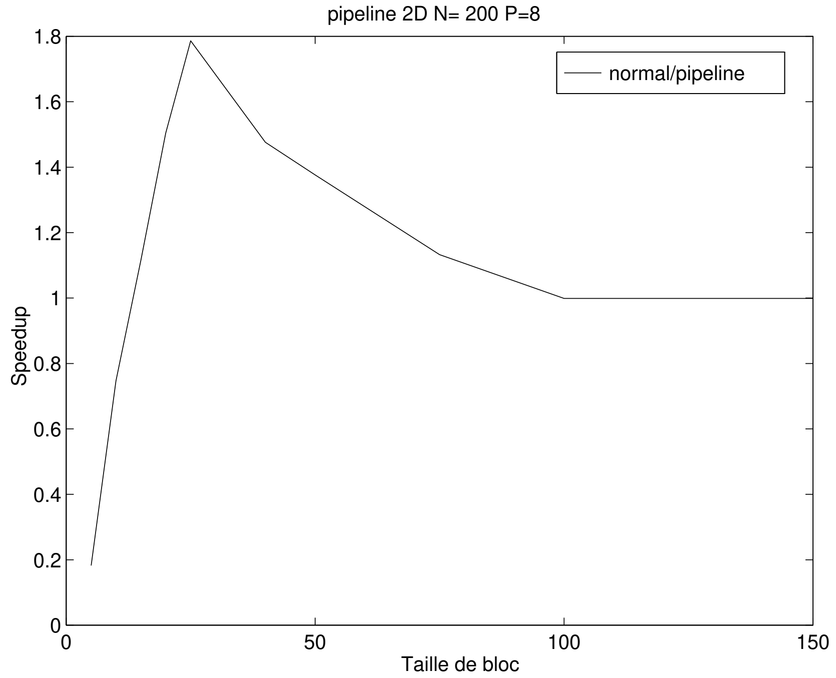

FIG. 33 - *Gains obtenus sur 8 processeurs.*

Parallélisation de l'algorithme ADI

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A, B
!HPF$ DISTRIBUTE (*,BLOCK) :: A, B
      ...
!     sweep along the columns
      DO I = 2, N
         DO J = 1, N
            A(I,J) = A(I,J) - A(I-1,J)*B(I,J)
         END DO
      END DO
!     sweep along the rows
      DO J = 2, N
         DO I = 1, N
            A(I,J) = A(I,J) - A(I,J-1)*B(I,J)
         END DO
      END DO
```

FIG. 34 - *Algorithme de départ.*

```
      REAL, DIMENSION (N,N) :: A, B, A1, B1
!HPF$ DISTRIBUTE (*,BLOCK) :: A, B
!HPF$ DISTRIBUTE (BLOCK,*) :: A1, B1
      ...
      B1 = B
      ...
      DO J = 1, N     !  exécution parallèlle
         DO I = 2, N
            A(I,J) = A(I,J) - A(I-1,J)*B(I,J)
         END DO
      END DO

      A1 = A        ! redistribution
      DO I = 1, N    ! exécution parallèlle
          DO J = 2, N
            A1(I,J) = A1(I,J) - A1(I,J-1)*B1(I,J)
         END DO
      END DO

      A = A1        ! nouvelle redistribution
```

FIG. 35 - *Première parallélisation de l'algorithme ADI (sans* DALIB_LOCCS_DRIVER*).*

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A, B
!HPF$ DISTRIBUTE (*,BLOCK) :: A, B
      ...
      DO I = 2, N     ! exécution parallèlle
         DO J = 1, N
            A(I,J) = A(I,J) - A(I-1,J)*B(I,J)
         END DO
      END DO

      CALL DALIB_LOCCS_DRIVER (BLOCK, 2, 0,
             A(:,2:N), [0,1], B(:,2:N), [0,0])
      ...
      EXTRINSIC (HPF_LOCAL) SUBROUTINE BLOCK (A, B)
      REAL A(:,:), B(:,:)
!HPF$ DISTRIBUTE *(*,BLOCK) :: A, B
      DO J=lbound(A,2),ubound(A,2)
        DO I=lbound(A,1),ubound(A,1)
           A(I,J) = A(I,J) - A(I,J-1)*B(I,J)
        END DO
      END DO
      END
```

FIG. 36 - *Seconde parallélisation de l'algorithme ADI (avec* DALIB_LOCCS_DRIVER*).*

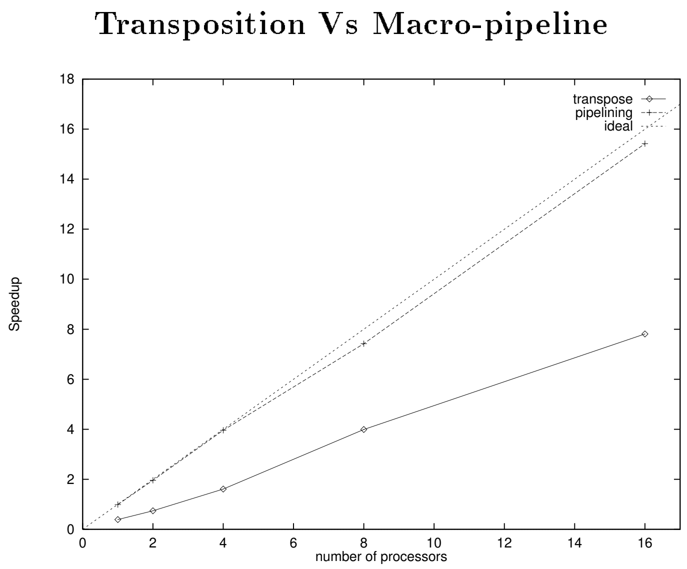

FIG. 37 - *Speedup ($N = 512$): transposition vs. macro-pipeline.*