THESE

présentée par

Frédéric DESPREZ

pour obtenir le titre de

DOCTEUR de
l'INSTITUT NATIONAL POLYTECHNIQUE
de GRENOBLE
(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Informatique**

Procédures de Base Pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée

Date de soutenance:	6 Janvier 1994	
Composition du jury:		
Président	Denis	**TRYSTRAM**
Rapporteurs	Gaétan	**LIBERT**
	Yves	**ROBERT**
Examinateurs	Gérard	**AUTHIÉ**
	Michel	**COSNARD**
Directeur de thèse	Bernard	**TOURANCHEAU**
Invités	Jack	**DONGARRA**
	Marc	**GARBEY**

Thèse préparée au sein du

Laboratoire de l'Informatique du Parallélisme
URA 1398 du CNRS, ENS Lyon.

J'exprime ma plus profonde gratitude aux membres du jury :

à Denis Trystram pour l'honneur qu'il me fait en présidant ce jury, pour m'avoir fait découvrir et aimer le parallélisme et pour tous ses conseils qu'il m'a prodigué depuis mon DEA,

à Gaétan Libert pour ses remarques pertinentes et pour son accueil chaleureux à Mons,

à Yves Robert, pour sa lecture aussi rapide qu'efficace, sa constante bonne humeur et surtout pour cette bourse sans laquelle je n'écrirais pas ces lignes à l'heure qu'il est,

à Gérard Authié, pour avoir accepté de faire partie du jury et pour ses précieuses remarques,

à Michel Cosnard pour son accueil au laboratoire, pour ses conseils, pour n'avoir pas craqué à ma 2000-ième question administrative, pour nos footings/groupe de travail à l'heure du déjeuner, pour une relecture aussi précise que rapide de 235 pages d'une première version et pour m'avoir appris que même Store-and-Forward pouvait être traduit en français,

à Bernard Tourancheau pour m'avoir fait entrer au LIP, pour ses conseils sur-atlantiques et sub-gaussiques, pour m'avoir fait connaître le C.S. Department de UT et Jack Dongarra, et pour m'avoir appris l'autonomie,

à Jack Dongarra, pour son accueil à UT, pour ses conseils sur moults problèmes d'algèbre linéaire et pour les six prochains mois à PVM-land,

à Marc Garbey pour ses conseils, pour avoir expliqué à un ignorant la modélisation d'une flamme cellulaire, pour son acharnement à programmer un hypercube malgré 12387 bugs et pour sa constante bonne humeur,

qu'ils trouvent ici le témoignage de ma reconnaissance.

Table des matières

Chapitre 1

Introduction générale

Depuis quelques années, les machines séquentielles ont montré leurs limitations et la puissance des processeurs commence à atteindre son maximum. Les applications nécessitant de grandes puissances de calcul ont, par ailleurs, été adaptées à des résolutions sur ordinateurs grâce à des modèles adéquats, notamment en mécanique des fluides, en météorologie, ... Pour réussir à obtenir des résultats corrects en un temps raisonnable, une solution consiste à utiliser plusieurs processeurs en parallèle, chacun s'occupant d'une partie des données du problème. Les machines actuelles sont, pour la plupart, à mémoire distribuée et se programment à l'aide d'échanges de messages. De ce fait, l'utilisation de telles machines s'avère être compliquée et source d'erreurs. L'obtention de performances correctes est elle-même assez difficile, et ceci surtout pour des utilisateurs n'ayant pas une connaissance approfondie de la machine cible et du parallélisme. Il convient donc de rendre l'utilisation du parallélisme la plus transparente possible. Deux approches apparaissent actuellement pour le calcul numérique, celle des compilateurs paralléliseurs comme HPF (High Performance Fortran) et celle des bibliothèques. Les deux approches peuvent bien entendu être combinées, les compilateurs pouvant appeler des routines de bibliothèques pour une plus grande efficacité et un coût de développement moindre. On considère d'autre part que 90% des problèmes scientifiques peuvent se ramener à des noyaux d'algèbre linéaire. Ces noyaux primordiaux n'ont pas à être reprogrammés pour chaque application et surtout, dans un souci de portabilité, requièrent des appels de procédures similaires. Pour porter ceux-ci sur des machines à mémoire distribuée, il est nécessaire de créer des routines de communication pour les échanges de données non locales. De plus, afin que le gain en performances obtenu grâce aux bibliothèques de calculs ne soit pas perdu dans des mouvements de données aux coûts prohibitifs, il faut recouvrir ces communications au maximum, grâce à des routines le permettant.

Les machines cibles sont les machines parallèles à mémoire distribuée. Ces machines sont les précurseurs des machines massivement parallèles de l'avenir. En effet, des machines à mémoire partagée ne sont pas réalisables ou efficaces au-delà d'un petit nombre de processeurs. Nos machines cibles se programment à l'aide de primitives de communication de type échange de messages.

L'objet de cette thèse est d'étudier et d'implémenter des bibliothèques de communication, de calcul et de recouvrement des calculs/communications et ceci pour permettre à l'utilisateur d'une machine parallèle d'avoir un code simple, efficace et portable. Leur utilisation est validée par la parallélisation d'un code de simulation optimisant les communications à l'aide de recouvrements.

La thèse comporte cinq parties :

1. Présentation des modèles, techniques et machines utilisés, et des bibliothèques et environnements existants,

2. Etude de routines de communications globales sur réseau reconfigurable et étude de l'enchaînement de diffusions dans un hypercube,

3. Etude de routines de recouvrement calculs/communications,

4. Etude de routines de calcul,

5. Application à la simulation numérique.

Nous passons maintenant en revue les cinq parties précédentes.

Présentation générale

Cette partie, divisée en deux chapitres, sert d'introduction aux environnements matériels et logiciels, utilisés dans cette thèse. Dans un premier chapitre, nous présentons premièrement le type de parallélisme utilisé dans cette thèse, c'est-à-dire, avec des machines à mémoire distribuée, échangeant des données par passage de messages. Nous présentons les moyens de connecter les processeurs entre eux et les modèles de routage associés. Ceux-ci auront une grande influence dans le choix des algorithmes. Les différents moyens d'étudier les performances des algorithmes sur de telles machines sont alors détaillés. Ce chapitre se termine sur une présentation des machines utilisées. De nombreuses bibliothèques de calcul et de communication existent maintenant sur la plupart des machines et de nombreuses études, cherchant à définir des standards, sont en cours. Afin de comprendre certains choix effectués au cours de l'étude et l'implémentation de certaines bibliothèques, un second chapitre présente un tour d'horizon des bibliothèques et environnements de programmation disponibles sur les machines actuelles ou qui le seront bientôt. Nos études sont directement liées à ces outils : soit nous les utilisons pour produire nos bibliothèques, soit nous partons de bibliothèques existantes et nous les améliorons pour pouvoir les utiliser plus efficacement sur les machines parallèles à mémoire distribuée, et ainsi permettre à de nombreux utilisateurs d'accéder à des performances avec un coût de développement moindre.

Communications globales

Les routines de communications globales garantissent une utilisation des machines parallèles à passage de messages sans la connaissance de l'architecture sous-jacente. Grâce à ces routines, il n'est plus nécessaire d'avoir une connaissance approfondie de la topologie de la machine et de son modèle de communication pour obtenir des performances correctes. Nos études portent premièrement sur des routines pour réseaux reconfigurables. Les réseaux dynamiques utilisés sont construits à base de crossbars et permettent de relier entre eux différents processeurs classiques. Une particularité intéressante est que les connexions entre les processeurs peuvent être modifiées en cours d'exécution pour permettre d'optimiser les communications. Nous avons étudié un ensemble d'algorithmes de communication classiques tels que la diffusion personnalisée, la réduction, la diffusion, l'échange total et la multi-distribution sur de tels réseaux. Nous avons obtenu des résultats optimaux ou proche de l'optimal pour tous ces algorithmes. Tout d'abord conçus pour des machines parallèles à base de transputers et de crossbars reconfigurables, nos algorithmes peuvent être utilisés avec d'autres modèles de communication comme les réseaux optiques et les communications wormhole. Une deuxième étude concerne le problème des diffusions successives sur un hypercube, appliquées à des problèmes d'algèbre linéaire comme la factorisation de matrices denses. Si l'on ne tient pas compte de l'enchaînement des diffusions et que l'on cherche uniquement l'algorithme le plus rapide, les performances peuvent être réduites à cause des conflits. Nous étudions ce problème de manière théorique sur un hypercube. Nous avons obtenu un algorithme à un facteur deux de la borne inférieure. Des simulations à l'aide de Paragraph valident notre approche.

Routines de recouvrement calculs/communications

Le surcoût dû aux communications étant responsable de la perte de performances avec les machines parallèles à mémoire distribuée, il est naturel de chercher à recouvrir les communications par des calculs. Malheureusement, ceci n'est pas toujours possible du fait des dépendances entre les différentes tâches. De plus, en cas de déséquilibrage des charges, certains processeurs doivent attendre que d'autres aient terminé leurs calculs pour commencer à travailler. Des méthodes utilisant un macro-pipeline existent et ont été validées pour un grand nombre d'algorithmes. Notre but est d'offrir à l'utilisateur de machine parallèle à mémoire distribuée, une bibliothèque permettant de cacher l'utilisation de techniques de très bas niveau, proches de la machine et permettant d'avoir des recouvrements calculs/communications à l'intérieur d'un processeur et calculs/calculs entre des processeurs. Nous proposons donc une bibliothèque originale permettant de tels recouvrements, et ceci de manière transparente pour l'utilisateur. Des exemples d'application de cette bibliothèque sont également présentés dans ce chapitre, comme dans les suivants.

Routines de calcul

Pour effectuer de gros calculs, il est nécessaire de disposer de routines de calcul de haut niveau efficaces sans avoir à prendre en compte l'utilisation du parallélisme et les optimisations de bas niveau. Nous présentons donc nos études sur des routines BLAS de niveau 3, essentiellement le produit de matrices, qui intervient dans la majorité des programmes d'algèbre linéaire et la mise à jour de rang 2k qui est une application intéressante des bibliothèques présentées. Nous analysons différents produits de matrices, candidats à une implémentation dans une bibliothèque. Nous donnons également des résultats d'expériences très concluants sur la Volvox IS-860 d'Archipel. Ces routines BLAS de niveau 3 restent malgré tout d'assez bas niveau. Nous étudions donc une routine d'algèbre linéaire de plus haut niveau : la factorisation LU. Nous effectuons des analyses très précises de cette factorisation sur anneau et réseau complet avec une distribution de la matrice par colonnes. Nous présentons une méthode de calcul des temps d'attente issus de l'utilisation du pipeline. Nous présentons des résultats d'expériences sur iPSC/860 et Paragon d'Intel. Nos résultats théoriques sont corroborés par l'expérience. Ensuite, nous améliorons les performances de cette même factorisation sur une grille de processeurs avec, cette fois, un rangement circulaire par blocs. L'optimisation est obtenue grâce à une méthode de diffusion pipeline, recouverte par des calculs. Un second chapitre concerne l'optimisation du calcul de Transformées de Fourier Rapides bi-dimensionnelles sur hypercube. Nous donnons des méthodes d'optimisation des algorithmes classiques de FFT2D avec des recouvrements calculs/communications maximaux et une utilisation totale de la bande passante de l'hypercube à chaque pas.

Application à la simulation numérique

Ce dernier chapitre expose le résultat de l'utilisation des routines présentées dans les précédents chapitres pour un problème de simulation de front de flammes. Ce problème utilise des méthodes de décomposition de domaine. En partant d'un code séquentiel existant, nous avons parallélisé ce code et, grâce à des optimisations très précises des communications sur les machines cibles et à des modifications préalables de l'algorithme, nous avons obtenu des gains très importants par rapport à des machines vectorielles classiques comme les CRAY.

Conclusion

Ces différentes parties représentent un tour d'horizon des méthodes et algorithmes parallèles adaptés au calcul numérique. La production des bibliothèques présentées permet à des utilisateurs non spécialistes d'avoir accès à des machines très performantes, avec des codes simples et proches du code séquentiel. L'accès à ces routines leur évite d'avoir à connaître les caractéristiques internes de la machine. La production de telles bibliothèques, sur machines parallèles à mémoire distribuée, s'inscrit directement dans le cadre de la recherche de performances à coût de développement minimum. Les routines développées peuvent être ensuite appelées par un compilateur paralléliseur, après reconnaissance, dans le code séquentiel, du code pouvant être remplacé par un appel à une routine.

Chapitre 2

Parallélisme et machines parallèles

Depuis quelques années, un nouveau courant de programmation est apparu permettant de mettre plusieurs processeurs en concurrence pour réaliser une tâche plus rapidement. Le parallélisme, tout d'abord confiné à des études théoriques par manque de machines, s'est développé avec l'apparition de machines ayant plusieurs processeurs travaillant ensemble sur les données stockées dans une même mémoire partagée. Ensuite, à cause de limitations technologiques empêchant de construire des machines parallèles de tailles importantes, de nouvelles machines sont apparues avec leur mémoire distribuée sur les processeurs. Pour programmer ces machines de type MIMD (Multiple Instruction stream, Multiple Data stream), communiquant par échanges de messages, il faut prendre en compte un surcoût supplémentaire dû à ces échanges. Les contraintes amenées par la réalisation physique des machines parallèles de ce type réduisent le nombre de degrés de liberté disponibles lors de l'étude théorique des algorithmes. Une étude théorique d'un algorithme doit donc tenir compte de la machine sur laquelle il sera programmé si l'on souhaite avoir un gain en performances grâce au parallélisme. Nous rappelons que, dans toute la thèse, nous nous restreignons aux machines parallèles MIMD à mémoire distribuée (ou "à passage de messages").

Dans une première partie, nous présentons les caractéristiques des réseaux de processeurs, et les contraintes liées aux topologies statiques ou dynamiques utilisées sont explicitées. Le surcoût dû aux communications étant le plus important dans les machines de type "à passage de messages", nous passons en revue les différents modes de communication dans les réseaux équipant les machines actuelles. Un paragraphe suivant présente les différents moyens d'analyser les performances des algorithmes parallèles et enfin, un dernier paragraphe passe en revue les machines utilisées dans cette thèse.

2.1 Les topologies de processeurs

Les processeurs d'une machine parallèle à mémoire distribuée sont reliés entre eux par l'intermédiaire d'un réseau d'interconnexion, statique ou dynamique. La topologie de connexion des processeurs a une grande importance, voire une très grande importance suivant les modes de communications utilisés. La plupart des machines actuelles possèdent des processeurs dédiés aux communications. Leur but est de permettre l'acheminement des messages vers les processeurs destination et de ranger les données dans leur mémoire. Les nœuds des topologies présentées par la suite sont donc constitués d'un ou plusieurs processeurs de calcul, de mémoire, et d'un processeur de communication.

2.1.1 Classification des méthodes d'interconnexion

Les possibilités de connexion des processeurs entre eux sont nombreuses et un grand nombre de projets ont été initiés, tant dans la recherche que dans l'industrie, afin d'obtenir le meilleur moyen d'interconnecter les processeurs pour communiquer ensuite à moindre coût. Une classification des architectures en fonction de leur capacité à modifier leur topologie en cours de programme est donnée par :

- **Les topologies statiques :** le réseau d'interconnexion est fixé par le constructeur et ne peut être modifié. C'est le cas de la plupart des architectures actuelles du commerce (Intel iPSC/860, Paragon, Meiko CS, CRAY T3D, nCUBE NC/2, NC/3, ...). Par contre, cela ne limite en rien l'utilisation de processeurs de routage qui peuvent donner l'illusion que la machine est en fait un réseau complet.

– **Les topologies quasi-statiques :** le réseau d'interconnexion est fixé par logiciel avant exécution du programme. Il reste inchangé pendant son exécution. Des exemples de telles machines sont toutes celles à base de transputers sans environnement permettant de modifier la topologie des processeurs en cours d'exécution (Archipel Volvox, Telmat Tnode, ...).

– **Les topologies quasi-dynamiques :** le programme est divisé en phases et chacune des phases utilise sa propre topologie qui est "construite" après synchronisation des processeurs et avant exécution de la phase suivante. Un exemple, de machine et d'environnement, utilisé dans la thèse, est le Tnode de Telmat avec l'environnement C_NET [ABB+91].

– **Les topologies dynamiques :** la topologie peut être modifiée en cours d'exécution et un processeur peut demander un lien pour une communication à un gestionnaire de connexions. C'est le cas de l'architecture DAMP, décrite dans [BBM91].

Dans les deux paragraphes suivants, nous présentons les topologies statiques et dynamiques les plus utilisées.

2.1.2 Les topologies statiques

Les topologies statiques utilisées dans les prochains chapitres sont essentiellement : le réseau complet, l'anneau, la grille, le tore et l'hypercube (Figure 2.1). Mais de nombreux algorithmes peuvent être exécutés sur d'autres topologies grâce au plongement d'une topologie dans une autre [CT93b].

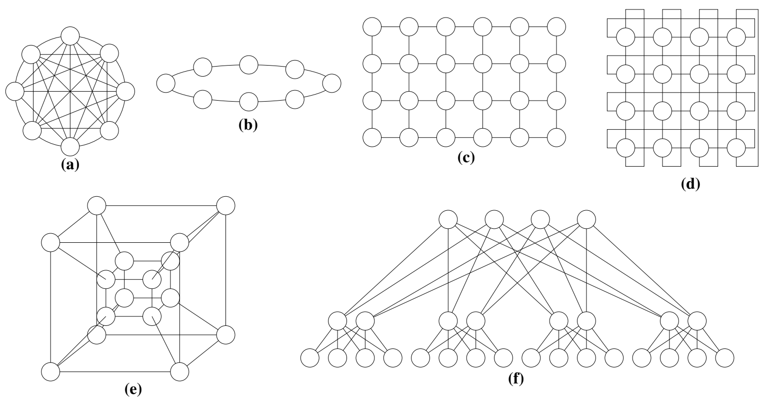

FIG. 2.1 - Différentes topologies de processeurs.

Les topologies statiques peuvent être décrites en terme de graphes où les sommets sont des processeurs et les arêtes les liens de communication. Les caractéristiques des réseaux sont données par :

Nombre de nœuds (processeurs) P **:** C'est la principale caractéristique d'une machine car elle borne le degré de parallélisme utilisable.

Degré k **:** Nombre d'arêtes partant d'un nœud. Pour les topologies non-régulières, on peut distinguer le degré minimum δ et le degré maximum Δ. Par la suite $k = \Delta$ car il correspond au nombre de liens du processeur de communication.

Diamètre D : La distance étant la longueur du plus court chemin entre deux nœuds, le diamètre est le maximum des distances.

Nombre de liens N_l : C'est le nombre de liens total de la topologie.

Largeur de bissection L_B : Minimum du nombre de liens nécessaires pour relier deux moitiés égales d'une topologie entre elles.

Nous appellerons k_{min}, le nombre de liens minimal pour pouvoir réaliser la topologie. En effet, rien n'empêche de réaliser, par exemple, une topologie où chaque connexion entre deux processeurs est réalisée avec plusieurs liens physiques, afin d'augmenter la bande passante. Remarquons que, dans la suite de la thèse, nous travaillons avec des machines où le degré k des processeurs est constant pour une topologie donnée.

Dans les paragraphes suivants, nous présentons plus précisément quelques topologies statiques. Un résumé de leurs principales caractéristiques est donné dans le tableau 2.1.

- **Le réseau complet** C'est le réseau idéal. Tous les processeurs sont à une distance de 1 (Figure 2.1 (a)). Par contre, le nombre k de liens par processeur est de $P-1$, soit un total de $P(P-1)/2$. Ceci n'est réalisable que pour un nombre de processeurs très petit. Grâce aux routeurs présents sur la plupart des machines actuelles, le réseau d'interconnexion peut être vu comme un réseau complet mais le prix à payer est la présence de contentions lorsque deux messages veulent utiliser un même lien.

- **L'anneau** est l'une des topologies les plus simples (Figure 2.1 (b)). De nombreux algorithmes parallèles l'utilisent pour sa simplicité. Nous verrons par la suite que certains algorithmes utilisent la grille ou le tore comme un assemblage d'anneaux, verticaux ou horizontaux.

- **La grille 2D** Le degré maximum des processeurs est égal à 4 (Figure 2.1 (c)). Un des défauts de la grille est son manque de symétrie. En effet, les processeurs sur les bords de la grille ont des caractéristiques différentes des processeurs centraux et nécessitent, de ce fait, des programmes particuliers. Par contre, un avantage est sa parfaite adéquation pour des problèmes où l'on doit distribuer une matrice et où les calculs se font voisins-à-voisins sans bouclage comme en imagerie par exemple [Ube93]. Un autre avantage concerne sa faisabilité matérielle avec beaucoup de processeurs (scalabilité). Il suffit de rajouter des processeurs sur les bords. La grille 2D est d'ailleurs à la base de machines parallèles comme la Delta ou la Paragon d'Intel.

- **Le tore 2D** est facilement dérivé de la grille 2D en reliant les processeurs sur les bords entre eux (Figure 2.1 (d)). Cela a pour effet de réduire le diamètre. On peut également augmenter la connectivité des processeurs en réalisant un tore 3D. C'est le cas des nouvelles machines de CRAY [CRA92] et de Parsytec (GigaCube) [Sit92].

- **L'hypercube** a été, pendant de nombreuses années, la topologie la plus utilisée pour la construction de machines parallèles ((e) sur la Figure 2.1). Ses propriétés les plus intéressantes sont sa construction récursive qui permet d'avoir des algorithmes par dimensions et son faible diamètre (logarithmique). Par contre, son nombre de liens croît avec le nombre de processeurs ce qui ne lui permet pas d'être un bon candidat pour les machines massivement parallèles. On caractérise un hypercube par sa dimension. Remarquons que l'anneau, la grille et le tore (avec des nombres de processeurs puissances de 2) peuvent être plongés dans l'hypercube avec une dilatation égale à 1 [SS85]. Nous ne détaillerons pas ici l'hypercube dont les caractéristiques ont été de nombreuses fois reportées [HR91, JH86, JH87, SS88]. Plusieurs machines du commerce ont utilisé cette topologie pour leur système de communication comme les iPSC/2 et iPSC/860 d'Intel, les machines nCUBE, la CM-2 de Thinking Machine Corporation, ...

Topologie	**Nbre de proc.** P	**Degré min.** k	**Diamètre** D	**Nbre de liens** N_l	**Larg. Bisec.** L_B
Réseau complet	P	$P-1$	1	$P(P-1)/2$	$(P/2)^2$
Anneau	P	2	$\lfloor P/2 \rfloor$	P	2
Grille 2D	$\sqrt{P}\sqrt{P}$	$2 \rightarrow 4$	$2(\sqrt{P}-1)$	$2P-2\sqrt{P}$	$\sqrt{P}$
Tore 2D	$\sqrt{P}\sqrt{P}$	4	$2\lfloor \sqrt{P}/2 \rfloor$	$2P$	$2\sqrt{P}$
Hypercube	$P=2^d$	$d=\log(P)$	d	$P\log(P)/2$	$P/2$

TAB. 2.1 - Principales caractéristiques de quelques topologies.

2.1.3 Les topologies dynamiques

Dans ce paragraphe, nous décrivons quelques réseaux "réarrangeables" qui permettent d'avoir une topologie dynamique, c'est à dire évoluant au cours de l'exécution d'un programme [CT93b].

- **Le fat-tree** C'est la topologie choisie par Thinking Machine Corporation pour équiper la CM-5 [Lei85]. Il s'agit d'un arbre binaire dont la section (bande passante) augmente au fur et à mesure que l'on arrive vers la racine (Figure 2.1 (f)). Ce type d'arbre assure qu'il n'y aura pas de dégradation de la bande passante et que le nombre de processeurs de la machine pourra être augmenté sans que les performances des communications s'en trouvent diminuées. Au niveau des feuilles de l'arbre sont placés les processeurs eux-mêmes. Grâce au fat-tree, un réseau est redondant par nature et donc tolérant aux fautes.

- **Les modules de commutation, réseaux Oméga et Baseline** Ces réseaux sont construits à l'aide de modules permettant de connecter des entrées à des sorties. Les différences résident principalement dans le nombre de modules et de fils nécessaires pour réaliser n'importe quelle topologie. Ces réseaux sont décrits dans [Hwa93]. Les réseaux construits à l'aide de tels modules sont appelés réseaux multi-étages. Leur coût est moins important que celui des crossbars que nous verrons dans le paragraphe suivant mais le contrôle nécessaire au choix des chemins est plus difficile à mettre en œuvre. Certaines nouvelles machines utilisent de tels réseaux comme la SP1 de IBM, la CS-2 de PCI et la VPP500 de Fujitsu.

- **Les réseaux Crossbar** Un réseau Crossbar peut être vu comme un réseau de commutation à un étage. Il est constitué d'une grille de points de connexions entre les entrées et les sorties. Chaque point peut faire une connexion entre une entrée et une sortie de manière dynamique. Le coût important d'un tel réseau est dû au nombre de modules nécessaires pour le réaliser [Hwa93]. Le circuit C004 d'INMOS, équipant les machines à base de transputers comme le Tnode de TELMAT ou Volvox d'ARCHIPEL, est de ce type.

– **Les réseaux optiques** La tendance actuelle est d'utiliser des *space optics* avec reconfiguration, pour le parallélisme. Des architectures comme le Crossbar Electro-Optique ont été proposées; elles utilisent une technique de reconfiguration hybride pour connecter les processeurs. Le problème le plus important est la transformation des signaux optiques en signaux électriques, qui reste très coûteux.

2.2 Modèles de routage

L'étude de la complexité des algorithmes parallèles nécessite une bonne connaissance de la machine cible et de ses environnements de programmation. Une étude préalable de la machine doit être faite pour en extraire des modèles précis qui permettront par la suite d'étudier, de la manière la plus réaliste, le comportement futur des algorithmes lors de leur exécution.

Le coût le plus pénalisant sur une machine à mémoire distribuée est le coût des communications. Si celles-ci sont mal gérées, les performances d'un algorithme parallèle s'en trouvent amoindries. Il convient donc de bien équilibrer les communications dans les algorithmes (et si possible de les masquer) et d'utiliser à bon escient les environnements de programmation disponibles, afin d'obtenir le plus rapidement possible les meilleures performances.

Dans ce paragraphe, nous présentons les modèles de routage des messages les plus utilisés actuellement dans les machines parallèles. Les différents modèles présentés seront utilisés par la suite pour l'étude de la complexité de nos algorithmes. La description se veut sommaire, des renseignements plus amples pouvant être récupérés dans l'abondante littérature sur le sujet [CT93b, FL91, Rum93, Sys92]. Nous nous limitons volontairement à quelques modèles effectivement constatés sur les machines utilisées. D'autres modèles plus théoriques existent néanmoins comme les modèles à temps constants par exemple (avec ou sans regroupement des messages) [FL91].

Dans les paragraphes suivants, nous supposons que deux processeurs x et y, non voisins, souhaitent s'échanger un message M de taille L.

2.2.1 Modèle à commutation de messages (CM)

Dans ce protocole, chaque processeur intermédiaire sur le chemin de la communication reçoit et stocke le message M complètement avant de le réémettre en direction du processeur destination. Ce modèle est également appelé *Store-and-Forward* (SF).

Modélisation du modèle CM

Si un message de taille L est transmis d'un processeur x vers un processeur y distants de $d(x,y)$, alors la modélisation de ce protocole est donné par $d(x,y)(\beta + L\tau)$, où β est le temps d'initialisation (ou *startup time*) et τ le temps de propagation d'un octet. Notons que β et τ sont sujets à des variations plus ou moins importantes suivant que plusieurs liens sont utilisés en parallèle (modèle k-port si k liens sont utilisés en parallèle) ou bien que le lien est utilisé en unidirectionnel (modèle half-duplex) ou en bidirectionnel (modèle full-duplex). L'influence de l'environnement de programmation utilisé est non-négligeable puisque suivant le confort d'utilisation offert, les coûts augmentent ou diminuent [DGJP93]. Une modélisation précise des communications doit prendre en compte toutes ces variations. Ce protocole de communication est utilisé par un grand nombre de machines parallèles, notamment celles à base de transputers T800 (TELMAT Tnode, Volvox IS-860, MEIKO Computing Surface, ...) ainsi que l'iPSC/1 d'Intel.

Le message peut être découpé en paquets de taille variable ou non. Cela donne une variation dans le coût total de la communication. Ce découpage est inhérent au processeur gérant les communications [Fra92]. Dans le paragraphe suivant, nous étudions cette amélioration classique, gérée par l'utilisateur.

Amélioration du coût du modèle CM : le pipeline

Le pipeline est un moyen de réduire le coût de communication dû à la distance entre les processeurs concernés par la communication [SS89, SW87]. Pour réduire le coût total, on divise le message en $\frac{L}{\mu}$ paquets de taille μ. Ensuite, les paquets sont envoyés les uns à la suite des autres à partir du processeur x vers le processeur y. Le premier paquet atteint le processeur y après $d(x,y)$ étapes de coût $\beta + \mu\tau$. Si $\frac{L}{\mu}$ est entier, les $\frac{L}{\mu} - 1$ suivants arrivent au processeur terminal en $(\frac{L}{\mu} - 1)(\beta + \mu\tau)$. Le coût total est donc $\left(d(x,y) + \frac{L}{\mu} - 1\right)(\beta + \mu\tau)$. On optimise μ en dérivant le temps total pour obtenir : $\mu_{opt} = \sqrt{\frac{L\beta}{(d(x,y)-1)\tau}}$, ce qui donne un coup total : $\left(\sqrt{L\tau} + \sqrt{(d(x,y)-1)\beta}\right)^2$. Le coût total est donc en $o(\sqrt{L}) + L\tau$. Ce qui est asymptotiquement optimal [Fra92].

Cette méthode sera utilisée dans de nombreux algorithmes par la suite. Elle sera aussi associée à des calculs qui recouvreront les communications. Dans ce cas, nous ferons référence à la méthode du ***macro-pipeline***.

2.2.2 Modèle "cut-through" (CT)

Dans ce modèle, pour lequel on peut distinguer trois protocoles, le message acheminé n'a pas besoin d'arriver entièrement sur un noeud pour être renvoyé vers une autre destination. Si le nombre de processeurs sur le chemin entre la source et la destination est P, alors le modèle communément utilisé est : $\beta + (P-1)\delta + L\tau$, où δ est le surcoût dû au calcul du chemin utilisé. Sur la plupart des machines actuelles $\delta \ll \beta$ (Paragon, iPSC/860, Delta, CS-2, ...). Les trois protocoles sont les suivants :

- **Commutation de circuit (CC)** Ce modèle est également appelé *Circuit-switching*. Un chemin est créé avant émission des premiers octets constituant le message. Après création de ce chemin, le message est acheminé directement entre la source et la destination. Notons que les processeurs sur le chemin sont immobilisés au niveau des communications pendant toute la durée de l'échange. Ce modèle est utilisé entre autres par Intel pour l'iPSC/860.

- **Wormhole (WH)** Pour ce mode d'acheminement, l'adresse du destinataire est placée dans l'en-tête du message. Le routage se fait sur chaque processeur, la sortie d'un processeur étant calculée par celui-ci à la lecture de l'adresse destination. Le message est découpé en petits paquets appelés *flits*. En cas de blocage, les flits sont stockés dans les registres internes des routeurs des processeurs intermédiaires. De nombreux algorithmes de routage existent [BS92, Sys92]. C'est le modèle utilisé pour les machines de type Paragon d'Intel ainsi que les futures machines à base de T9000 et de C104.

- **Virtual cut-through** Dans ce dernier cas, en cas de blocage, le message entier est stocké sur le dernier processeur atteint. Il nécessite donc des capacités infinies de stockage [Fra92]. A notre connaissance, aucune machine n'a implémenté ce modèle.

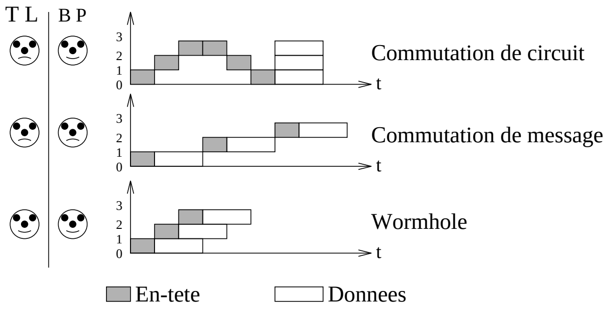

FIG. 2.2 - Différents modèles de communication.

2.2.3 Possibilités d'utilisation des liens d'un processeur

Grâce à un circuit DMA (Direct Memory Access), on peut utiliser plusieurs liens en parallèle. Suivant les machines, un seul lien pourra être utilisé à la fois (modèle 1-port), ou tous les liens pourront être utilisés en parallèle. Un modèle intermédiaire étant l'utilisation d'une partie des liens du processeur en parallèle ou *link bound* [FL91]. Il faut noter qu'une partie de la communication (initialisation) est réalisée en séquentiel. Elle correspond à la mise en place et au démarrage de la communication (par exemple : passage des informations aux circuits de DMA (Direct Memory Access)).

Une autre caractéristique importante est la possibilité d'utiliser les liens dans un seul sens (*half-duplex*) ou dans les deux sens (*full-duplex*). Dans un seul sens, nous signifions qu'un lien est utilisable dans un seul sens à un instant donné, non du fait qu'il s'agit d'un graphe orienté. La plupart du temps, certaines informations du protocole de communication (comme les acquittements de messages) transitent de manière transparente pour l'utilisateur et ralentissent le flot de données.

2.2.4 Comparaison des différents modèles

A cause de la technologie et des connaissances en matière de routage de l'époque, le modèle à commutation de messages fut le premier utilisé sur les machines parallèles de type à "passage de messages" (*message passing*). Les modèles cut-through (CC et WH) sont plus efficaces dans ce sens qu'ils masquent la distance entre les processeurs communiquant de manière matérielle sans utilisation de buffers trop importants. Entre le wormhole et le modèle à commutation de circuit, l'avantage revient au WH qui construit sa route tandis que le message avance dans le réseau au contraire du CC qui construit son chemin (et l'acquitte) avant l'envoi des premiers paquets de données.

La Figure 2.2 montre une comparaison des différents modèles (CM, CC et WH) grâce à un exemple illustrée par un diagramme de Gantt pour la communication d'un message entre deux nœuds distants de trois unités. La partie grisée représente le coût d'initialisation (en-tête du message) et la partie blanche les données à envoyer. En abscisse, nous avons le temps et en ordonnée les processeurs. Sur la gauche de la figure, des visages souriants ou tristes donnent les qualités et les défauts du modèle en termes de temps de latence (TL) ou de bande passante (BP).

2.3 Etude des performances

L'étude théorique des performances des algorithmes parallèles permet de comprendre certains dysfonctionnements et pertes de performances. Tous les paramètres d'un algorithme destiné à être parallélisé doivent être connus ainsi que ceux de la machine cible. Il existe de nombreuses mesures de performances qui permettent de connaître par exemple l'efficacité d'un programme, le gain issu de l'utilisation du parallélisme, son comportement futur dans le cas de l'augmentation du nombre de processeurs [Wor91]. Dans ce paragraphe, nous passons en revue les moyens d'analyse des performances qui seront utilisés par la suite. Pour une étude très complète des différents moyens d'analyse des programmes parallèles, se référer au livre de Hwang [Hwa93].

2.3.1 Terminologie

Dans cette section, nous donnons les termes qui seront utilisés pour l'étude des performances. Nous suivons les notations données par Gupta et Kumar dans [GK92].

Nombre de processeurs, P : le nombre d'unités de calcul utilisées pour résoudre un problème.

Taille du problème, N

Temps séquentiel, W : le temps pris par un algorithme pour résoudre un problème sur un seul processeur. Si l'algorithme utilisé est le même, il est aussi égal à la somme des temps de travail utiles exécutés par les P processeurs pour résoudre un même problème. Comme ce temps varie avec la taille du problème N, on le notera également $W(N)$. Par exemple, pour le produit de deux matrices de taille $N \times N$, on considère que $W(N) = O(N^3)$.

Temps d'exécution parallèle, T_P : le temps total pris par P processeurs pour résoudre un problème donné. C'est une fonction de la taille du problème, du nombre de processeurs et des paramètres de la machine.

Temps de surcoût parallèle, T_0 : la somme des surcoûts dus à l'utilisation du parallélisme (communications, mise en place de processus, recopies mémoire, temps d'attente, ...). Si les communications sont recouvertes (c'est-à-dire exécutées en même temps que des calculs et de coûts inférieurs aux temps de calculs), elle n'entrent bien entendu pas dans ce surcoût. T_0 est également fonction du nombre de processeurs, de la taille du problème et des paramètres de la machine. Il est donc écrit : $T_0(N, P)$. Il est égal à : $T_0(N, P) = PT_P(N) - W(N)$.

2.3.2 Mesures de performances

Les mesures données dans cette section sont les plus utilisées. De nombreuses études ont été faites essayant de donner la meilleure mesure [Gus88a, SG91].

Vitesse

C'est une des mesures les plus données. Elle est calculée à partir du nombre d'opérations flottantes du programme et du temps d'exécution et s'exprime donc en *flops*. Si le nombre d'opérations flottantes est N_{of} et le temps d'exécution t, alors le nombre de flops est donné par : N_{of}/t. Avec les machines actuelles, les résultats sont donnés en Mflops (Mégaflops ou 10^6 flops), Gflops (Gigaflops ou 10^9 flops), l'objectif de la plupart des constructeurs étant à présent le Tflops (Téraflops ou 10^{12} flops). Ceux-ci donnent généralement les performances en crête de leur machine, calculées à partir du nombre d'instructions exécutées à chaque cycle (processeurs superscalaires) et du temps de cycle de base de la machine. Ces performances ne tiennent pas compte des problèmes d'accès à la mémoire, aux caches et des problèmes de conflits. Il ne s'agit donc là que d'une borne supérieure sur les performances.

Facteur d'accélération S

Le facteur d'accélération (ou *speedup*) est le ratio $S = \frac{W}{T_P}$. Plus il est proche de P, meilleur est l'algorithme parallèle. On peut parfois constater un facteur d'accélération supérieur à P. Dans ce cas, nous dirons qu'il est superlinéaire. Cela arrive quand l'algorithme parallèle permet de gagner sur certaines parties du code original grâce au traitement parallèle (comme par exemple une recherche aléatoire), ou permet une meilleure utilisation des hiérarchies mémoire et ainsi réduit le temps total [HD89]. Cela est bien sûr également possible si l'algorithme séquentiel de référence n'est pas le même que l'algorithme utilisé pour la parallélisation, et qu'il est moins efficace.

Efficacité E

C'est le ratio $E = \frac{S}{P}$ et donc $E = \frac{W}{PT_P} = \frac{1}{1+\frac{T_0}{W}}$.

Accélération de Gustafson

L'accélération de Gustafson correspond à une réévaluation de la loi d'Amdahl [Amd67] qui dit que le facteur d'accélération d'un programme parallèle est borné par l'inverse du pourcentage de calcul séquentiel d'un programme. Si on note $W = T_s + T_p$ où T_s est la fraction séquentielle du programme et T_p la fraction parallélisable, alors l'accélération $1/(T_s + T_p/P)$ tend vers $1/T_s$ lorsque le nombre de processeurs P tend vers l'infini. Comme le pourcentage n'est jamais nul, cela démontre l'inutilité du parallélisme massif. Gustafson [Gus88b], a proposé une renormalisation du problème pour calculer l'accélération. On considère la plus grande instance du problème qui peut être résolue sur une machine et on détermine le temps d'exécution nécessaire pour le même problème sur un seul processeur. Le rapport des temps donne l'accélération. Ce type d'étude d'accélération a été utilisé dans le cadre de l'élimination de Gauss dans [CRT89].

Isoefficacité et Scalabilité

La scalabilité d'un algorithme fait référence à sa capacité à conserver une efficacité donnée au fur et à mesure que le nombre de processeurs augmente. Pour un tour d'horizon des mesures de scalabilité des algorithmes parallèles, se référer aux rapports de Gupta et Kumar [GK91a, GK92].

La fonction d'Isoefficacité, définie par Gupta et Kumar, est une mesure de scalabilité d'un algorithme parallèle. Elle est une combinaison de la complexité de l'algorithme parallèle et de la taille de l'architecture et relie la taille du problème au nombre de processeurs nécessaires pour maintenir une efficacité fixée ou délivrer un facteur d'accélération croissant linéairement avec le nombre de processeurs.

L'efficacité d'un système parallèle est donnée par $E = \frac{W(N)}{W(N)+T_0(N,P)}$. Généralement, le temps de surcoût parallèle croît avec N et P. Pour une taille de problème fixée, l'efficacité décroît quand P croît, car $T_0(N,P)$ croît avec P. Pour une taille de machine fixée, l'efficacité augmente avec N car généralement $T_0(N,P)$ augmente moins rapidement que $W(N)$. Si c'est le cas, alors on peut dire que l'on a affaire à un système *scalable* [GK91b]. La scalabilité est donc fonction des croissances relatives entre $W(N)$ et $T_0(N, P)$.

- Si $W(N)$ doit augmenter exponentiellement pour maintenir une efficacité donnée, alors le système est dit peu scalable,
- si $W(N)$ doit augmenter linéairement pour maintenir une efficacité donnée, alors le système est dit très scalable.

Les fonctions d'isoefficacité de nombreux systèmes parallèles sont des fonctions polynômiales de P (c.-à.-d. $O(P^N)$ avec $N \geq 1$). Plus la puissance de P est petite, plus le système est scalable.

Si $T_0(N, P)$ est le surcoût total du système parallèle alors l'efficacité est égale à $E = \frac{1}{1+\frac{T_0(N,P)}{W(N)}}$. Pour maintenir une efficacité donnée constante, $W(N)$ doit être proportionnel à $T_0(N, P)$, où la relation suivante doit être satisfaite :

$$W = \gamma T_0(N, P) \text{ où } \gamma = \frac{E}{1 - E}. \tag{2.1}$$

γ est constant pour une efficacité fixée E. L'équation 2.1 est utilisée pour déterminer l'isoefficacité. On peut donc définir la fonction d'isoefficacité $f_E(P)$, telle que :

$$f_E(P) = \gamma T_0(N, P). \tag{2.2}$$

Si le travail $W(N)$ doit croître avec $f_E(P)$ pour maintenir une efficacité E, alors $f_E(P)$ est définie comme étant la fonction d'isoefficacité de la combinaison algorithme/architecture pour l'efficacité E [GK91a]. Si $W(N)$ croît aussi vite que $f_E(P)$, alors une efficacité constante peut être maintenue pour la combinaison algorithme/architecture. Une caractéristique importante de l'analyse d'isoefficacité est qu'une simple expression peut caractériser l'algorithme parallèle et l'architecture sur laquelle il est implémenté. Grâce à l'isoefficacité, on peut tester les performances d'un algorithme sur un petit nombre de processeurs puis prédire ses performances sur un plus grand nombre de processeurs.

Des études similaires ont été effectuées dans le cadre de la bibliothèque ScaLAPACK avec la définition d'une fonction d'isogranularité [CDW92].

Les benchmarks

Parallèlement aux études du comportement des algorithmes sur les machines parallèles (ou sur des classes de machines parallèles), des tests sont effectués sur les machines afin de déterminer leur aptitude à résoudre tel ou tel problème. Ces tests sont appelés *benchmarks*. De nombreuses études et groupes de travail ont été initiés pour déterminer un ensemble de benchmarks permettant de donner les caractéristiques d'une machine en terme de performances et ce, pour différentes classes de problèmes. On peut noter que la plupart des benchmarks existants se réfèrent aux performances numériques des machines. Le tableau 2.2 rappelle quelques benchmarks d'aujourd'hui avec leurs caractéristiques et leurs unités de mesures. Pour une comparaison des différents benchmarks actuels, se référer à l'article de Weiker [Wei91].

A part LINPACK qui est maintenant le standard (parfois critiqué) des benchmarks pour les problèmes d'algèbre linéaire, aucun des autres benchmarks n'a réellement émergé. LINPACK est maintenant porté sur des machines parallèles à mémoire distribuée et permet ainsi de connaître les qualités en communication des machines testées. Le tableau 2.3 donne les performances de LINPACK pour quelques machines actuelles [DMS93]. Nous donnons la puissance en crête de la machine et entre parenthèses le nombre de processeurs utilisés pour le test.

Benchmark	Caractéristiques	Unité de mesure	Refs
Livermore	boucles pour des problèmes numériques	Mflops	[Mah86]
LINPACK	résolution système linéaire 1000x1000	Mflops	[Don90]
Drystone	boucles, appels de procs, calculs, ...	Drystones/s	[WD84]
Whestone	arith. entière et flottante, appels de procs	MWIPS/s	[CW76]
SLALOM	calculs de radiosité (temps fixé = 1')	Nbre de faces	[GREC91]

TAB. 2.2 - principaux benchmarks et leurs caractéristiques

Machine	Puiss. en crête (# procs) Mflops	LINPACK 1000×1000 (# procs) Mflops
TMC CM-5	131000 (1024)	59700 (1024)
KSR KSR-1	10240 (256)	3380 (256)
Intel Paragon	7000 (140)	3300 (140)
Intel iPSC/860	5000 (128)	2600 (128)

TAB. 2.3 - Performances LINPACK pour différentes machines parallèles actuelles.

2.4 Les architectures de machines

Durant les 10 dernières années le parallélisme a évolué d'un concept étudié en laboratoire à une réalité industrielle. Actuellement on assiste à une grande convergence aux plans matériel et logiciel. La machine type des années 90 est un réseau de processeurs du commerce (RISC 32 bits) avec une unité puissante de calcul flottant, disposant d'une grande mémoire privée et communiquant à l'aide d'un réseau d'interconnexion et de processeurs de communication spécifiques. Une gamme complète de machines est proposée allant de quelques processeurs à plusieurs milliers.

Dans ce paragraphe, nous présentons une partie des machines parallèles qui ont été utilisées au cours de la thèse. La connaissance précise de leurs caractéristiques permet de comprendre les choix réalisés dans la parallélisation des algorithmes. Dans un premier paragraphe, nous présentons les processeurs de calcul des machines utilisées et ensuite, nous passons en revue les machines en détaillant leurs caractéristiques au niveau matériel et logiciel.

2.4.1 Les processeurs

Dans ce paragraphe, nous décrivons rapidement quelques processeurs utilisés actuellement dans les machines parallèles ou qui le seront à leur sortie sur le marché.

T800

Le transputer INMOS T800 est un processeur RISC 32 bits spécialement dédié aux applications embarquées et à la construction de machines parallèles à mémoire distribuée [Bré88]. En interne, le transputer T800 possède une CPU et une unité flottante séparées, une unité matérielle de gestion de processus, deux timers, une interface de gestion de la mémoire externe, une mémoire interne de 4 Ko ainsi que quatre interfaces de communication possédant chacune deux DMA (soit un

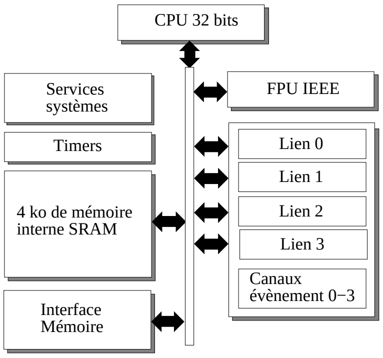

FIG. 2.3 - Transputer T800.

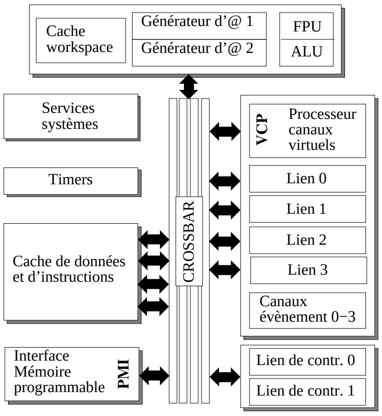

FIG. 2.4 - Transputer T9000.

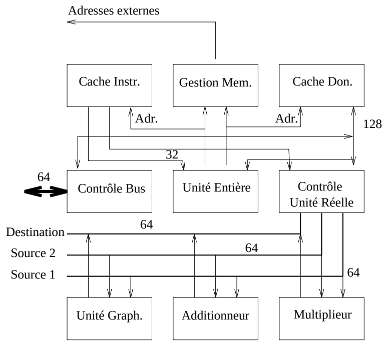

FIG. 2.5 - i860.

processeur 4-ports full-duplex). Ces DMA permettent au transputer de faire des communications bidirectionnelles en parallèle sur ses quatre liens, en même temps que des calculs [TV89]. Cette fonctionnalité extrêmement intéressante permet de masquer les communications par les calculs. La bande passante des liens est de 2.96 Mo/s en bidirectionnel. L'unité matérielle de gestion de processus fait du transputer un processeur multitâches très performant.

T9000

Le T9000 est le dernier né des processeurs INMOS/THOMSON. Dérivé du T800, il en garde les principales caractéristiques (quatre liens bidirectionnels, unités arithmétique et flottante, gestion multitâches matérielle, ...) mais de nouvelles évolutions lui permettent d'avoir des performances en calculs et communications supérieures au T800. L'architecture est maintenant de type superscalaire et ce nouveau processeur possède un groupeur d'instructions chargé d'alimenter au mieux le pipeline à 5 étages. Le protocole de communication a également été changé, passant de la commutation de message au wormhole. Les performances en crête annoncées sont de 200 MIPS et de 25 Mflops. La vitesse des liens annoncée est de 20 Mo/s. Ce processeur intègre également une unité de gestion de canaux virtuels (VCR). Il est associé à un crossbar C104 pour construire des architectures à base de T9000. Celui-ci a été décrit précisément dans [DFL93].

Intel i860

Le processeur i860, sorti en 1989, est devenu le fer de lance de la société Intel pour ce qui est de la course aux performances. Celui-ci constitue la brique de base de machines parallèles très performantes comme l'iPSC/860, la DELTA ou encore la Paragon. Ce processeur est également présent dans d'autres machines parallèles à mémoire distribuée, comme l'Archipel Volvox IS-860 où il est utilisé comme coprocesseur de calcul et sur des cartes additionnelles pour stations de travail. Il s'agit d'une architecture 32 bits délivrant en crête 75 Mflops (double précision) pour la version la plus performante (i860 XP à 50MHz). Doté d'un million de transistors, ce processeur possède une unité RISC, une unité flottante (FPU), une unité graphique 3D, une unité de gestion mémoire avec pagination et enfin deux caches instructions (4 Ko) et données (8 Ko) séparés (Figure 2.5). Le pipeline de l'unité RISC est constitué de quatre étages [Cha90]. L'i860 peut exécuter en crête trois instructions par cycle. Le mode superscalaire est sélectionnable (mode double-instruction). Ne possédant pas de liens avec l'extérieur, l'i860 doit être associé à un processeur de communication (Paragon Mesh Routing Chip (PMRC) sur la Paragon, T800 sur la Volvox d'ARCHIPEL), accessible par le biais d'une mémoire partagée. Sur la Paragon, on associe à l'i860 dédié aux calculs un autre i860 gérant les communications en liaison avec le PMRC.

Il faut également noter que la difficulté de gestion des caches internes, des registres et des chemins de données ne permet pas aux compilateurs actuels de dépasser 15 Mflops, et les routines codées en assembleur atteignent avec peine la moitié de la puissance en crête (environ 39 Mflops) [Moy91].

D'autres processeurs

Dès leur conception, les processeurs décrits précédemment ont été destinés à être inclus dans des machines parallèles. Avant 1992, la plupart des machines ont été construites avec de tels processeurs. On constate actuellement un changement de situation avec les nouvelles machines. On trouve de plus en plus de processeurs initialement dédiés au marché des stations de travail puissantes. Ces processeurs RISC, comme le SPARC [Gla91] et l'Alpha [Com92], sont maintenant capables de communiquer entre eux grâce à des mémoires partagées et des dispositifs matériels de gestion. Le coût de développement s'en trouve réduit et les constructeurs peuvent se focaliser sur le problème majeur des machines parallèles à mémoire distribuée : la vitesse des communications.

2.4.2 Exemples d'architectures parallèles

Dans ce paragraphe, nous donnons quelques exemples de machines parallèles qui ont été utilisées dans cette thèse. D'autres architectures ont été décrites dans [CD94].

Le projet SuperNode

Ce projet européen est à la base de calculateurs composés de transputers d'Inmos et utilisant des réseaux réarrangeables. La machine Tnode de TELMAT, issue de ce projet, est constituée de 32 transputers de calcul (et de communication) T800, de transputers de contrôle T414 et de réseaux d'interconnexion NEC et C004. Le transputer de contrôle assure la gestion des commutateurs et du bus de contrôle par des registres implantés en mémoire. Le commutateur NEC connecte les transputers de travail suivant n'importe quel graphe de degré inférieur ou égal à quatre. Un commutateur C004 permet la connexion vers l'extérieur du transputer de contrôle et du commutateur reliant les transputers de travail. Le commutateur utilisé entre les transputers de travail est conçu pour réaliser des configurations dynamiques (en cours de programmes).

INTEL iPSC/860

Cette machine, sortie en 1990, a été l'une des machines à mémoire distribuée les plus puissantes du moment grâce à ses processeurs i860. L'iPSC/860 est la deuxième machine du projet Intel/DARPA[1] appelé Touchstone et destiné à approcher le Téraflop [Int91]. Le réseau d'interconnexion est de type hypercube avec une dimension maximum de 7, soit 128 processeurs. Chaque processeur est connecté par l'intermédiaire de deux files FIFO à un DCM (Direct Connect Module) à 8 liens. Un des liens est réservé aux entrées/sorties (E/S). Chacun des canaux de communication est série et bidirectionnel full-duplex avec une bande passante de 2.8 Mo/s dans chaque direction. La liaison avec le i860 est réalisé à l'aide de deux DMA (1-port full-duplex). Le mode de routage

[1]Defense Advanced Research Projects Agency

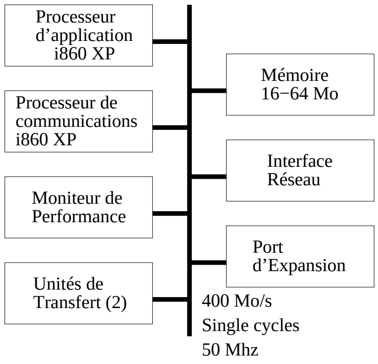

FIG. 2.6 - Architecture d'un nœud de la Paragon.

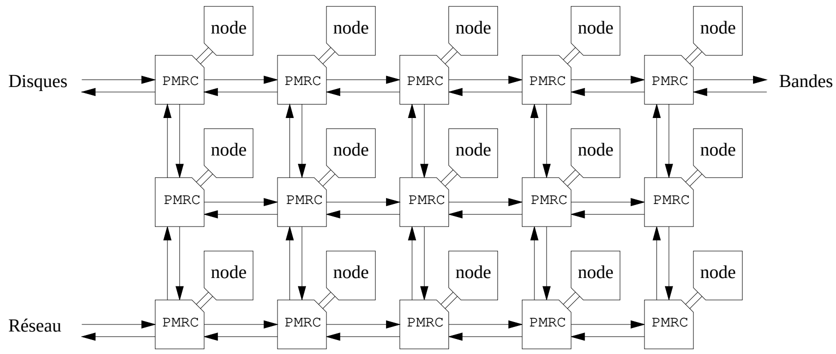

FIG. 2.7 - Architecture de la Paragon.

est la commutation de circuit. Avant l'envoi du message, un chemin est construit dynamiquement à l'aide de l'algorithme *e-cube*. Le système d'exploitation est NX/2.

INTEL Paragon

La Paragon est la dernière machine issue du projet d'Intel Touchstone. Elle est la version commerciale de la machine Delta avec quelques modifications d'importance : deux processeurs par nœud et un nouveau système d'exploitation (OSF/1). La première machine a été livrée en septembre 1992. Dans la gamme commerciale, la Paragon succède à l'iPSC/860 avec comme principale différence le changement de topologie (de l'hypercube à la grille) dans un souci de scalabilité. Le nombre de nœuds peut aller jusqu'à 2048, soit une puissance de crête de 150 Gigaflops.

Les nœuds de la machine sont tous basés sur le même schéma : deux bus (un de données de 64 bits et un d'adresses de 32 bits) relient entre eux les différents éléments (processeur application, moniteur de performance, unités de transfert, processeur de communication, mémoire, interface E/S et interface réseau) (Figure 2.6). Les nœuds d'application et de communication sont deux processeurs i860 XP (à 50Mhz). Le fait d'avoir un processeur dédié aux communications permet d'avoir un recouvrement des calculs par les communications encore plus important que pour l'iPSC/860. Ne possédant pas de liens avec l'extérieur, l'i860 doit être associé à un processeur de communication (Paragon Mesh Routing Chip (PMRC)). L'interface entre les nœuds de communication et les PMRC est effectuée, comme pour l'iPSC/860, par deux DMA.

Le réseau d'interconnexion est une grille 2D avec comme composant de routage le PMRC. Le choix de la grille repose sur des arguments de scalabilité et de simplicité de fabrication. De plus, grâce à son mode de routage wormhole, la distance entre deux nœuds du réseau a une moindre importance. Le système contient des nœuds pour trois tâches différentes : des nœuds de calcul, des nœuds de services et des nœuds d'entrées/sorties. Les nœuds de calcul sont utilisés pour l'exécution des programmes parallèles, les nœuds de services offrent la capacité d'un système UNIX (grâce au système OSF/1), incluant la compilation et les outils de développement de programmes et les nœuds d'E/S font l'interface entre tous ces nœuds et les mémoires de masse (comme les disques et les unités de bandes) (Figure 2.7). La Paragon possède deux réseaux de communication : un réseau dédié aux échanges de données et un réseau de diagnostic. Ce dernier est dédié à l'initialisation de la machine et au diagnostics. Le réseau de communication est constitué des circuits de connexion PMRC reliés entre eux par des canaux de largeur 16 bits. La bande passante théorique en crête est de 200 Mo/s en full-duplex. Les PMRC sont indépendants des nœuds avec lesquels ils sont attachés. Les côtés de la grille peuvent être connectés à des disques, des unités graphiques, des nœuds d'entrée/sortie, ...

Le processeur de communication opère en parallèle avec le processeur de calcul et partage la même mémoire. Sa tâche est de s'occuper de toutes les communications avec les processeurs voisins et de faire les communications globales. De ce fait, à la différence de l'iPSC/860, le processeur application n'est pas interrompu par les échanges de messages et les changements de contexte. Le processeur de communication s'occupe également du découpage des messages en paquets, de l'ajout des informations nécessaires au routage et de l'initialisation des transferts de messages. Le code de ce processeur est entièrement contenu dans les caches, ce qui a pour effet de réduire le temps de latence des communications. L'unité transférée entre les nœuds est le paquet. Sa taille peut varier de 8 à 1984 octets contrôlée par l'utilisateur. La taille par défaut est de 1024 octets. Des informations pour le routage sont ajoutées en tête de paquet. C'est donc l'en-tête du message qui indique au circuit PMRC la route à suivre à l'aide de son algorithme de routage [DS86]. L'algorithme de routage est très simple, les messages sont d'abord envoyés horizontalement puis verticalement. Cela évite les blocages. L'en-tête d'un message contient deux informations : l'orientation du chemin et le nombre de sauts à effectuer. Il y a deux entêtes, un pour chaque direction [EK93]. Le tableau 2.4 donne les performances LINPACK 1000×1000 pour différents modèles de Paragon.

Modèle	**XP/A4**	**XP/S5**	**XP/S10**
Nombre de nœuds	56	66	140
Puissance en crête	4200	3300	7000
LINPACK 1000×1000	1500	1900	3300

TAB. 2.4 - Performances du benchmark LINPACK pour la Paragon (Mflops).

ARCHIPEL Volvox IS-860

La machine ARCHIPEL Volvox IS-860 est une machine à mémoire distribuée constituée de 8 à 48 nœuds de calcul et de 4 nœuds d'entrées/sorties. Chaque nœud de la Volvox est constitué, comme pour la Paragon, de deux processeurs, un dédié aux calculs et un dédié aux communications. Le processeur de calcul est le i860, dans sa version XR avec une vitesse d'horloge de 33 MHz. Il y a peu de changements architecturaux entre le i860 de la Volvox et celui de la Paragon décrit précédemment [Cha93]. Le processeur de communication est le transputer T800. Les deux processeurs sont reliés entre eux par une mémoire partagée double port (Figure 2.8). Les communications entre les deux processeurs se font par l'intermédiaire de cette mémoire.

Il faut noter que les T800 peuvent être utilisés pour les calculs mais leurs performances en opérations flottantes sont plus modestes. Cela offre par contre un niveau de parallélisme supplémentaire. Grâce aux quatre liens bidirectionnels du T800 et au commutateur programmable associé, on peut créer n'importe quelle topologie de degré inférieur ou égal à quatre. On ne peut pas parler de topologie de la machine car elles sont multiples et programmables par l'utilisateur. Tous les nœuds sont reliés entre eux par des crossbars C004 d'INMOS permettant une reconfiguration.

Les communications se font en utilisant les liens des transputers de manière synchrone avec le modèle par commutation de message. Par contre, des outils logiciels permettent de virtualiser ces communications et de masquer les contraintes de topologie à l'utilisateur. Les i860 possèdent

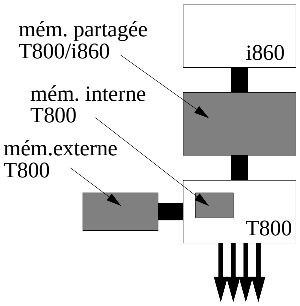

FIG. 2.8 - Architecture d'un nœud de la Volvox IS-860.

deux caches et les transputers une mémoire interne de 4 Ko. Par ailleurs, sur chaque nœud de la Volvox, les transputers possèdent une mémoire externe de 1 Mo. Enfin, la mémoire principale est la mémoire partagée entre les deux processeurs qui contient les données et le code des processeurs. Il faut noter que la mémoire interne des transputers est d'accès plus rapide que les autres mémoires. De ce fait, le code des transputers gagne à être rangé dans cette mémoire (40% de gain).

2.5 Conclusion

Il ne fait plus de doute à présent que le parallélisme représente la solution d'avenir pour l'obtention de performances telles que celles nécessaires pour la résolution de problèmes présentés comme des "grands challenges" pour la météorologie, la médecine, la mécanique des fluides, les bases de données, le traitement d'images, ...

Le parallélisme est un champ très vaste et nous n'avons présenté ici que ses aspects qui nous serviront au cours de la présentation des travaux réalisés. Dans toute la thèse, nous nous restreignons aux machines de type MIMD à gros grain et à échange de messages. Ce modèle est actuellement adopté par tous les constructeurs de machines parallèles, même par ceux qui, comme Thinking Machine Corporation, prônaient l'utilisation d'un grand nombre de processeurs de très faible puissance.

Chapitre 3

Environnements de programmation et bibliothèques

Une des raisons de la difficulté d'utilisation des machines MIMD du passé résidait souvent dans le faible nombre de logiciels adéquats. En effet, ces machines ne possédaient en tout et pour tout qu'un compilateur pour le processeur de calcul et des routines de communication de bas niveau qui ne permettaient pas de recouvrir calculs et communications. Autant dire que la programmation de telles machines s'avérait très fastidieuse et que les programmes étaient liés à la machine sans aucun espoir d'être portés. De plus, si cette programmation restait envisageable pour un petit nombre de processeurs, l'augmentation de la taille de la machine rendait impossible le débogage des applications. Ces machines étaient donc réservées à quelques "gourous" du parallélisme, les demandeurs de grandes puissances de calcul ne pouvaient pas, quant à eux, raisonnablement les utiliser. De plus, le manque de portabilité et les temps de développement énormes laissaient le parallélisme à la porte du monde industriel. Dès l'apparition des machines parallèles, de nombreuses recherches ont vu le jour concernant différents niveaux d'environnement de programmation, des bibliothèques aux compilateurs paralléliseurs, en passant par les outils d'analyse de performance et de débogage et les systèmes d'exploitations. Les bibliothèques existaient déjà avec les calculateurs vectoriels et certains compilateurs effectuaient des optimisations sur les machines à mémoire partagée. Par contre, avec l'avènement des machines parallèles à mémoire distribuée, la tâche est encore plus ardue, car tous ces outils doivent tenir compte du coût de communication et de la répartition des données.

Dans ce chapitre, nous présentons rapidement des environnements de programmation et des bibliothèques de calcul et de communication utilisables sur machines parallèles à mémoire distribuée. Un outil d'analyse du comportement des programmes est également présenté. Ce tour d'horizon, loin d'être exhaustif, liste plutôt les outils de programmation les plus utilisés actuellement et dont la plupart l'ont été de manière intensive dans cette thèse.

3.1 Les bibliothèques d'algèbre linéaire

Comme nous l'avons vu précédemment, portabilité et efficacité sont les deux maîtres-mots de l'algorithmique parallèle d'aujourd'hui. De nombreuses études ont révélé que certains noyaux de calculs revenaient régulièrement dans les programmes d'algèbre linéaire (et plus généralement de calcul scientifique). Il convient donc d'avoir un certain nombre de routines de base, si possible optimisées. Ces noyaux permettent d'avoir des programmes plus simples et plus portables. Ils seront également plus efficaces si les routines sont optimisées. Avec les machines parallèles, les bibliothèques séquentielles doivent être entièrement repensées et leurs algorithmes adaptés aux contraintes de ces modèles de machines.

3.1.1 Les BLAS

Le but des BLAS (Basic Linear Algebra Subroutines) n'est pas de définir une bibliothèque performante pour une machine donnée mais plutôt de définir des routines d'algèbre linéaire assez générales pour être utilisées par un grand nombre de programmes. Les vendeurs de machines pourront alors développer les BLAS optimisés pour leur machine et, de ce fait, offrir aux utilisateurs potentiels une portabilité de leurs codes ainsi que des performances intéressantes. Les routines ont été développées à l'origine en FORTRAN 77 mais existent également en C. Par exemple, les bibliothèques codées en assembleur utilisées sur les machines d'Intel et d'ARCHIPEL peuvent aussi bien être appelées en C qu'en FORTRAN.

Les BLAS de niveau 1

Ces routines sont les premières d'algèbre linéaire définies à l'origine par C. Lawson, R. Hanson, D. Kincaid et F. Krogh [LHKK79]. Elles implémentent des opérations de base sur des vecteurs telles que la mise à jour d'un vecteur par un autre vecteur (_AXPY) : $y \leftarrow y + \alpha x$, et le produit scalaire (_DOT) : $s \leftarrow x^T y$, (où x et y sont des vecteurs de même dimension n et α et s sont des scalaires). Le nombre d'opérations est $O(n)$ sur $O(n)$ éléments.

Les BLAS de niveau 2

Afin de diminuer le nombre d'accès mémoire, d'augmenter le grain de calcul et surtout tenir compte des processeurs proposant des opérations accélérées sur des vecteurs grâce à des registres et des unités pipelines, les BLAS de niveau 2 ont été définies par J.J. Dongarra, J. Du Croz, S. Hammarling et R. Hanson [DCHH88]. Celles-ci sont basées sur des opérations entre matrices et vecteurs telles que le produit matrice-vecteur (_GEMV) : $y \leftarrow \beta y + \alpha Ax$, (où A est une matrice, x et y sont des vecteurs et α et β des scalaires). Le nombre d'opérations est de $O(n^2)$ sur $O(n^2)$ éléments.

Les BLAS de niveau 3

Les BLAS de niveau 3 ont été définies par J.J. Dongarra, J. Du Croz, I. Duff et S. Hammarling [DCDH90] pour utiliser au maximum les possibilités offertes par les hiérarchies mémoires et pour minimiser les mouvements de données entre les différentes mémoires. Les opérations sont de type matrice-matrice comme par exemple le produit de matrices (_GEMM) : $C \leftarrow \beta C + \alpha AB$, (où A, B et C sont des matrices de tailles compatibles). Par la suite, nous ne nous intéresserons qu'aux opérations BLAS sur des matrices réelles en simple ou double précision. Les résultats présentés peuvent être aisément adaptés à d'autres types de matrices, complexes par exemple.

Les routines BLAS de niveau 3 peuvent être classées en trois ensembles principaux : produit matrice-matrice, mise à jour de rang k et 2k et résolution de systèmes triangulaires. Nous allons maintenant décrire toutes les opérations qui constituent cette bibliothèque.

Produit matriciel et produit de matrices triangulaires Ce sont les routines _GEMM et _TRMM.

$$\begin{array}{ll} C \leftarrow \alpha AB + \beta C & C \leftarrow \alpha A^T B + \beta C \\ C \leftarrow \alpha AB^T + \beta C & C \leftarrow \alpha A^T B^T + \beta C \\ C \leftarrow \alpha TB & C \leftarrow \alpha T^T B \\ C \leftarrow \alpha BT & C \leftarrow \alpha BT^T \end{array}$$

Mise à jour de rang k et 2k Il s'agit des routines _SYRK et _SYR2K.

$$\begin{array}{ll} C \leftarrow \alpha AA^T + \beta C & C \leftarrow \alpha A^T A + \beta C \\ C \leftarrow \alpha AB^T + \alpha BA^T + \beta C & C \leftarrow \alpha A^T B + \alpha B^T A + \beta C \end{array}$$

Résolution de système triangulaire C'est la routine _TRSM.

$$\begin{array}{ll} C \leftarrow \alpha T^{-1} B & C \leftarrow \alpha T^{-T} B \\ C \leftarrow \alpha B T^{-1} & C \leftarrow \alpha B T^{-T} \end{array}$$

Il faut noter que cette bibliothèque a été très utilisée pour le développement de la bibliothèque LAPACK [BDD+89] et pour sa version parallèle ScaLAPACK [DDGW93].

De plus amples renseignements concernant les BLAS de niveau 3 (comme par exemple les paramètres d'appel) pourront être trouvés dans les documents de définition [DCDH90].

Granularité et niveaux de BLAS

Les BLAS de niveau 1 utilisent très mal les hiérarchies mémoire et les opérations vectorielles. De nombreux accès à des données sont effectués. Il faut noter que certaines routines des BLAS de niveau 2 réussissent à atteindre des performances très proches des performances de crête sur plusieurs machines vectorielles telles que les CRAY X-MP ou Y-MP ou des machines Convex C-2, mais qu'elles sont limitées dans le cas de machines CRAY-2 ou bien IBM 3090 à cause des vitesses d'accès plus ou moins lentes entre les différents éléments de la hiérarchie mémoire [DDSV91]. Les BLAS de niveau 3 permettent une meilleure réutilisation des données déjà présentes dans les caches. Ces routines effectuent $O(n^3)$ opérations arithmétiques (contre $O(n^2)$ pour les BLAS de niveau 2) sur $O(n^2)$ données. Le fait d'utiliser des BLAS de niveau supérieur conduit à une meilleure utilisation des données déjà présentes dans les caches et de ce fait à des performances plus proches des performances en crête.

Dans le tableau 3.1, nous résumons les différents paramètres de granularité des différents niveaux de BLAS. Nous supposons que les matrices sont de tailles $m \times n$, $n \times k$ et $m \times k$.

BLAS	**Nombre de Références mémoires**	**Nombre d'opérations**	**Ratio Refs/Ops pour $n = m = k$**
Niveau 1 (_AXPY)	$3n$	$2n$	$3/2$
Niveau 2 (_GEMV)	$mn + n + 2m$	$2mn$	$1/2$
Niveau 3 (_GEMM)	$2mn + mk + kn$	$2mnk$	$2/n$

TAB. 3.1 - Les effets de l'augmentation de niveau de BLAS

3.1.2 BLAS distribuées

Dans ce paragraphe, nous faisons un tour d'horizon des projets de parallélisation des BLAS de niveau 3. Nous nous intéressons plus particulièrement aux parallélisations sur des machines à mémoire distribuée et surtout aux projets étudiant le traitement de leurs contraintes plutôt que la parallélisation d'algorithmes utilisés dans les BLAS de niveau 3.

De nombreuses parallélisations ont déjà été effectuées sur des machines vectorielles [DDP92, KLL93] ou SIMD [BS93, MJ91]. Dans ce paragraphe, nous passons en revue quelques uns des projets de portage des BLAS sur machines parallèles à mémoire distribuée comme la Delta d'Intel [CDW93b], l'AP1000 de Fujitsu [SB91], un réseau de transputers [BM90] ou bien en utilisant une mémoire partagée virtuelle sur BBN TC2000 [ADDM92].

Le projet PUMMA [CDW93b] a pour but de donner aux utilisateurs de machines parallèles à mémoire distribuée des routines de type BLAS de niveau 3 pour des matrices rangées de manière circulaire par blocs sur une grille de processeurs, transposées ou non (produits $C = AB$, $C = A^T B$, $C = AB^T$ et $C = A^T B^T$). Ces routines sont, pour l'instant, optimisées pour la machine Intel Delta. La bibliothèque PUMMA inclut également une routine de transposition de matrices sur une grille [CDW93a].

Un autre projet a été initié notamment à l'Australian National University. Il concerne l'implémentation de la bibliothèque BLAS de niveau 3 en simple précision sur une machine Fujitsu AP1000 [SB91]. L'AP1000 est une grille de processeurs SPARC utilisant le routage wormhole pour communiquer. Les routines séquentielles ont été optimisées pour les processeurs SPARC et les versions parallèles l'ont été en minimisant le coût des communications grâce au routage wormhole. La distribution choisie est encore circulaire par blocs. Les performances obtenues ont été de 80/90% des performances de crête de la machine. Cette bibliothèque a été ensuite utilisée pour la parallélisation de la factorisation LU sur cette même machine [Bre92].

Un noyau de routines BLAS (`_GEMM` et `_TRSM`) a été implémenté à Toulouse sur un anneau de transputers T800 [BM90]. Cette première version consiste en l'installation définitive de différents processus, chacun chargé d'une opération BLAS. Suivant la nature de l'appel, tel ou tel processus est choisi pour exécuter son code sur les données reçues. Des techniques de rémanence des données sont utilisées pour éviter de ressortir les résultats qui doivent être utilisés pour des calculs futurs ainsi que pour le traitement des matrices dont la taille ne permet pas de les charger entièrement sur la machine. Une autre approche, également effectuée à Toulouse dans l'équipe de Ian Duff, utilise une mémoire partagée virtuelle [ADDM92].

Une parallélisation de sous-ensembles de la bibliothèque BLAS [FSSS92] de niveaux 2 et 3 a été faite au-dessus de l'environnement *Multicomputer Toolbox* et de sa bibliothèque de communication *Zipcode*. Trois classes de bibliothèques CBLAS (Concurrent BLAS) sont définies suivant que les données (objets) sont distribuées ou non :

- *Classe 1* : Objets séquentiels uniquement. Il s'agit alors simplement des BLAS séquentiels, optimisés pour un processeur donné,
- *Classe 2* : Objets dupliqués et objets distribués,
- *Classe 3* : Uniquement des objets distribués.

Il va de soi que les algorithmes diffèrent suivant que certaines données sont dupliquées ou pas. Une grande place est donnée aux distributions de matrices dans le réseau.

Enfin, le Laboratoire des Hautes Performances en Calcul possède dans ses projets la création d'une bibliothèque de calcul pour la future machine à base de T9000. Cette bibliothèque sera, entre autres, constituée de BLAS de niveau 3 parallèles.

3.1.3 LAPACK

LAPACK (Linear Algebra PACKage) est une bibliothèque de haut niveau destinée à résoudre les problèmes principaux d'algèbre linéaire : résolution de systèmes linéaires, solution des moindres carrés linéaires, des problèmes de valeurs propres ainsi que des problèmes de valeurs singulières [DDC+87, DDK91]. Des factorisations de matrices de type : LU, Cholesky, QR, SVD, Schur et Schur généralisée sont également fournies ainsi que des calculs de réordonnancement de factorisations de Schur et d'estimations de nombre condition. Les matrices peuvent être bandes ou bien denses mais les routines ne tiennent pas compte des matrices creuses en général. Et, comme pour les BLAS, les données peuvent être réelles ou complexes, en simple et double précision.

LAPACK provient de la fusion de deux bibliothèques EISPACK et LINPACK et le but original était d'avoir des routines efficaces sur les machines vectorielles à mémoire partagée. EISPACK et LINPACK ont été développées entre les années 70 et 80 pour des supercalculateurs. A la fin des années 80, la librairie LAPACK a été conçue pour des machines vectorielles et parallèles à mémoire partagée. LAPACK utilise de manière intensive les algorithmes par blocs pour utiliser au mieux les hiérarchies de mémoires. A cette fin, LAPACK utilise les BLAS de niveau 3.

ScaLAPACK (Scalable LAPACK) est la bibliothèque LAPACK pour machines parallèles à mémoire distribuée [CDW92, DDGW93]. Celle-ci contient également la version distribuée des BLAS de niveau 3 (PUMMA), la bibliothèque BLACS de communication adaptée à l'algèbre linéaire ainsi que les BLAS séquentiels de niveau 3 en assembleur et des routines de copie de buffers. Les buts principaux lors du portage de LAPACK sur de telles machines est d'avoir la scalabilité, la portabilité, la flexibilité et la facilité d'utilisation. Le paramètre de scalabilité est très important car il se réfère au comportement d'un algorithme lorsque le nombre de processeurs augmente (Chapitre 2, Paragraphe 2.3.2). Ceci est primordial afin de connaître le futur comportement des algorithmes avec les machines massivement parallèles de demain. Les routines de ScaLAPACK disponibles actuellement sont essentiellement des factorisations de matrices en LU, QR et LL^t sur les machines d'Intel iPSC/860, Delta, Paragon et Thinking Machine Corporation CM-5. Une des particularités de la bibliothèque est que tous les appels nécessaires à l'utilisation du parallélisme sont cachés à l'intérieur des BLAS et des BLACS. Plusieurs interfaces d'appels sont envisagées pour ScaLAPACK. La première sera la même que LAPACK originale avec des arguments supplémentaires pour spécifier la distribution des matrices sur les processeurs. Une deuxième version n'aura plus les arguments supplémentaires de rangements de matrices mais ces renseignements seront passés grâce à des routines d'initialisation. Une dernière version concerne le portage pour des langages orientés objets comme C++ (ScaLAPACK++) [DPW93].

3.2 Les bibliothèques de communication

Un des aspects les plus importants des performances des machines parallèles à mémoire distribuée étant leur capacité à communiquer, les bibliothèques de communications sont à la base de la programmation de ces machines. Les buts d'une bibliothèque sont multiples et parfois antinomiques. On peut grossièrement rechercher les performances ou le confort d'utilisation; l'idéal étant, bien entendu, d'avoir les deux à la fois. De gros efforts ont été réalisés depuis quelques années par les constructeurs et les chercheurs pour créer la bibliothèque idéale. Les premières machines parallèles à mémoire distribuée étaient souvent livrées avec des routines de communication de très bas niveau, souvent bloquantes. A présent, l'accent est mis également sur le confort d'utilisation qui permet à des utilisateurs néophytes d'utiliser de telles machines avec un pourcentage de réussite acceptable et surtout des performances raisonnables. Il va de soi que les programmes parallèles doivent rester de taille raisonnable et également être au maximum indépendants de la machine sous-jacente.

Dans ce paragraphe, nous faisons un tour d'horizon des bibliothèques de communication pour machines parallèles à mémoire distribuée. Tous les constructeurs ont proposé une bibliothèque et de nombreuses recherches universitaires ont donné lieu à des produits finis et souvent très efficaces. Notre liste est loin d'être exhaustive, mais représente un ensemble de bibliothèques parmi les plus utilisées ainsi qu'une proposition pour un standard.

3.2.1 PICL

La bibliothèque PICL (Portable Instrumented Communication Library) [GHPW90] a été développée à Oak Ridge National Laboratory et fournit les mêmes routines que la bibliothèque Intel ainsi que des routines pour la mise en place des paramètres du réseau sous-jacent (hypercube, tore, anneau) et le chargement du programme. Une caractéristique intéressante de cette bibliothèque est qu'elle permet la génération de traces au format de Paragraph, l'outil de visualisation de comportement de programmes parallèles, décrit dans un prochain paragraphe (3.3). Elle est bâtie sur trois ensembles de routines : un ensemble de routines de communication de bas niveau, un ensemble de routines de communications globales et enfin, un ensemble de routines utilisées dans le cadre de la génération de traces. Les routines de haut niveau, qui sont programmées à partir des routines de bas niveau, incluent des synchronisations de tous les processeurs ou d'un ensemble de processeurs pré-défini, une routine de diffusion et une routine permettant de décrire quelle topologie virtuelle est utilisée.

3.2.2 BLACS

Les BLACS (Basic Linear Algebra Communication Subroutines) [ABD+91] sont dédiées à des opérations de communication utilisées pour la parallélisation des BLAS de niveau 3 ou de la bibliothèque LAPACK. Celles-ci peuvent, bien entendu, être utilisées pour d'autres applications nécessitant des mouvements de matrices dans un réseau. Il faut noter que les BLACS ne se veulent pas une bibliothèque multi-usages correspondant à toutes les applications parallèles mais plutôt une bibliothèque efficace appliquée au calcul matriciel. De même que dans PICL, on trouve dans les BLACS des routines de bas et de haut niveau. Les routines de bas niveau sont `SD` (envoyer un message), `RV` (recevoir un message), tandis que les routines de haut niveau sont `BS` (diffuser un message) et `BR` (recevoir un message diffusé). On trouve également des opérateurs globaux comme le `MAX` (maximum), `MIN` (minimum) et `SUM` (sommation). Les structures sont essentiellement des matrices rectangulaires et trapézoïdales. Les types sont des entiers, réels (simple et double précision) et des complexes (simple et double précision). Ces routines sont disponibles sur les machines Intel iPSC/860, Delta et Paragon, sur la Thinking machine CM-5 ainsi qu'au-dessus du système pour réseaux hétérogènes PVM.

3.2.3 PVM

Une nouvelle tendance dans le parallélisme est l'utilisation de réseaux de stations de travail pour simuler des machines parallèles de manière économique ou bien pour construire des machines distribuées hétérogènes. On peut, grâce à PVM (Parallel Virtual Machine), avoir un réseau de stations de travail pour traiter des applications en récupérant les heures de CPU inutilisées, ou bien construire une machine constituée de multiples machines parallèles avec, par exemple, des CRAY et des Paragon pour les calculs intensifs et des Silicon Graphics pour les sorties graphiques. La programmation de toutes les applications se fait alors avec les mêmes appels de routines et PVM gère tous les problèmes de formats différents entre les machines. Donc, de ce fait, un programme écrit en PVM pourra tourner sur un réseau de stations SUN, sur un groupe de nœuds d'une machine Paragon et également sur un réseau hétérogène constitué de machines parallèles et de stations de travail distribuées autour du monde. PVM est un ensemble de démons UNIX doublé d'une bibliothèque de communication. Il suffit, pour construire une machine parallèle, d'avoir un compte sur les machines que l'on souhaite utiliser et de lancer les démons de PVM sur ces stations. De plus, PVM permet d'inclure des machines parallèles "classiques" grâce à leur machine frontale sur laquelle est lancé PVM ou mieux à l'intérieur de la machine même sur les nœuds (versions 3.0 et suivantes). PVM a été développé notamment à Oak Ridge National Laboratory et à l'Université du Tennessee dans l'équipe de J.J. Dongarra [DGMJ93] et à l'Université d'Emory par V.S. Sunderam.

D'autres environnements de programmation permettent d'utiliser des réseaux hétérogènes d'ordinateurs comme P4, PARMACS, Trollius LAM, PARFORM, ...

3.2.4 TROLLIUS et TROLLIUS LAM

Plus qu'une bibliothèque de communication, Trollius est un système d'exploitation et un environnement de programmation pour machines parallèles à base de transputers. Il a été développé à Cornell University puis à Ohio State University. En tant qu'environnement de programmation, Trollius comprend un système d'exploitation, une interface ligne de commande avec l'utilisateur, des extensions de langage pour C et FORTRAN et des générations de traces pour Paragraph (décrit par la suite (Paragraphe 3.3)). De plus, une récente version de Trollius, appelée Trollius LAM, permet de développer des programmes sur des réseaux hétérogènes. Trollius supporte plusieurs processus par nœud[1]. Pour de plus amples informations sur Trollius, se référer aux documents [Bur89, LT91].

3.2.5 Intel NX

A la différence des premières machines parallèles, qui ne proposaient à leurs utilisateurs que des primitives de communications de très bas niveau, certes efficaces, mais d'utilisation difficile, Intel propose une bibliothèque assez complète, avec des routines de bas et haut niveau, synchrones et asynchrones [Pie88]. Les routines de haut niveau sont la diffusion et des routines globales (maximum, minimum, somme, OU exclusif, ...) ainsi que plusieurs routines de synchronisation. Elles ont une portabilité ascendante totale sur toutes les machines Intel (iPSC/2, iPSC/860, Delta et Paragon). La Paragon permet en plus à l'utilisateur de définir sa routine de combinaison globale.

3.2.6 MPI

MPI (Message Passing Interface) [DHHW93] est, pour l'instant, la spécification d'un standard d'interface d'échange de messages sur les machines à mémoire distribuée. Le but de ce standard est, comme pour toute bibliothèque, d'avoir accès, sur toutes les machines du même type, à la même interface facile à utiliser. Comme pour les bibliothèques de calcul décrites précédemment, la bibliothèque résultante de MPI pourra être optimisée par chaque constructeur de machines parallèles afin de tenir compte des spécificités en terme de communication de ses machines. Dans ce dernier cas, la bibliothèque sera, en plus, performante. De nombreux groupes de travail réunissant chercheurs, utilisateurs et constructeurs permettent à MPI de couvrir toutes sortes d'applications et de tenir compte des succès et des échecs rencontrés lors de la définition et de la réalisation des autres bibliothèques. Les routines définies correspondent aux routines les plus utilisées dans toutes sortes de codes et pas seulement en algèbre linéaire comme les BLACS précédemment décrites. Les idées proviennent de plusieurs bibliothèques existantes comme Express, NX, Vertex, PARMACS, P4 et PVM. Le standard a été découpé en deux ensembles de routines : MPI1 et MPI2. MPI1 contient toutes les routines de communications locales, de gestion de groupes de processus, de concaténation de messages, de gestion de contextes de communication et des routines utilitaires. MPI2 sera, quant à lui, destiné aux communications globales.

[1] Pour les processeurs mono-tâche comme l'i860 de la Volvox d'ARCHIPEL, cette fonctionnalité n'est, bien sûr, pas disponible.

Nous ne détaillons pas ici le contenu de ce projet ambitieux car de nombreuses caractéristiques de la bibliothèque finale sont encore en discussion. Ce projet représente malgré tout une solution d'avenir, garantissant la portabilité des programmes entre les différentes machines et réseaux de stations.

3.3 Un outil d'analyse de traces : Paragraph

La visualisation graphique est une technique standard pour permettre à l'utilisateur de comprendre les phénomènes complexes. Le comportement des programmes parallèles sur les architectures à mémoire distribuée est souvent extrêmement complexe. Paragraph [HE91] est un système graphique pour visualiser le comportement et les performances des programmes parallèles sur les architectures à passage de messages. L'exécution du programme analysé est rejouée de manière graphique pour permettre une découverte et une interprétation des problèmes du programme. De nombreuses vues existent permettant d'analyser telle ou telle caractéristique du comportement de l'application. Vingt-cinq vues sont disponibles, classées en trois catégories :

- **Des vues liées à l'utilisation des processeurs** Elles présentent l'activité des processeurs et permettent donc d'apprécier l'efficacité.

- **Des vues liées aux communications** Celles-ci permettent de voir les communications entre les processeurs, leur volume et leur fréquence.

- **Des vues liées aux tâches** Grâce à PICL, il est possible de tracer l'exécution de tâches particulières dans le programme et de leur affecter des couleurs différentes dans les vues. Il est ainsi possible, d'apprécier le temps passé dans telle ou telle partie du code.

Les vues `Spacetime`, `Critical Path`, `Gantt` et `Summary` nous intéressent tout particulièrement. Sur la Figure 3.1, ces quatre vues sont présentées pour un problème de factorisation de matrices. La vue `Spacetime` nous présente une vue d'ensemble de l'exécution avec tous les processeurs en ordonnée et le temps en abscisse. Les lignes horizontales donnent l'activité des processeurs ; si une ligne se brise, le processeur est en attente. Les lignes obliques et verticales représentent les communications, une couleur différente est utilisée suivant la taille du message. La vue `Critical Path` est la même que le `Spacetime` mais sur celle-ci le chemin critique du programme est représenté (en rouge). Ce chemin représente la séquence des tâches qui conduisent au temps total d'exécution. Il n'est pas utile d'essayer de gagner sur les autres parties du programme, aucun gain en temps ne sera constaté. Le `Gantt` présente l'activité des processeurs au cours du temps. Les processeurs peuvent être tour à tour en train de travailler (*busy*), en train de communiquer (*overhead*) ou bien en attente (*idle*). Cette vue permet de bien voir les mauvais équilibrages et les pertes d'efficacité. Enfin, `Summary`, est, comme son nom l'indique, un résumé de l'activité des processeurs pour toute l'exécution.

Un des problèmes de Paragraph est son incapacité à visualiser le comportement de plus de 64 processeurs. De nombreuses recherches sont actuellement effectuées afin de trouver des outils permettant d'analyser le comportement de programmes lancés sur des machines massivement parallèles [PBV92].

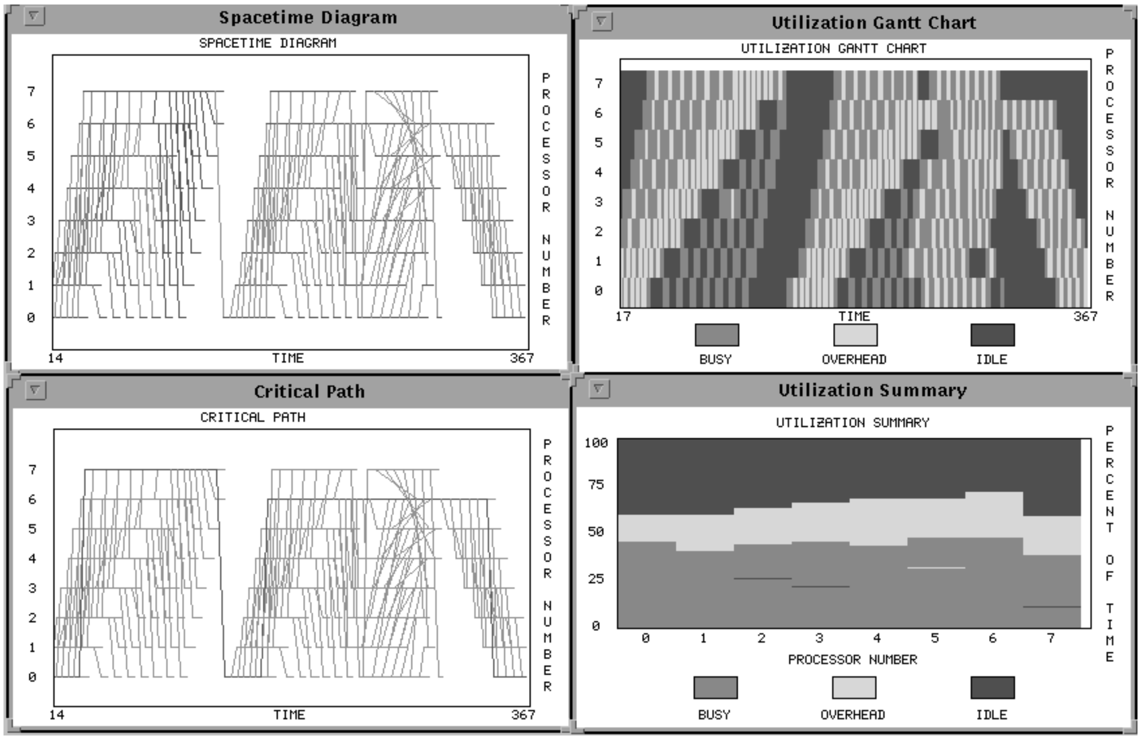

FIG. 3.1 - Vues Paragraph `Spacetime`, `Critical Path`, `Gantt` et `Summary`.

3.4 Conclusion

Le temps où les constructeurs pouvaient vendre une machine avec, comme unique argument, ses performances en crête est révolu et les machines actuelles possèdent, pour la plupart, une bonne partie des "outils" présentés dans ce chapitre [CD94]. Les tendances actuelles en matière de logiciels sont notamment celles des bibliothèques de communication et celles des langages data-parallèles de type High Performance Fortran. Ces deux formes de programmation sont nécessaires et la plupart des constructeurs les proposent. Les bibliothèques de calcul comme les BLAS ou des routines pour la FFT sont également souvent présentes dans les catalogues. En ce qui concerne les bibliothèques de communication, la plupart des constructeurs proposent PVM et font partie du forum MPI. Des systèmes de mémoire partagée virtuelle existent également qui permettent de programmer les machines à mémoire distribuée comme des machines à mémoire partagée [CG90, LP92]. Un constructeur l'implémente au niveau matériel. Il s'agit de Kendall Square Research et des machines KSR1 et 2 [RSRM93]. Les langages eux-mêmes sont actuellement adaptés pour cacher le parallélisme aux utilisateurs. Ce sont les langages data-parallèles, auparavant uniquement disponibles avec les machines SIMD. Un des projets les plus ambitieux est le High Performance Fortran Forum (HPFF) [For93]. Ce langage est basé sur des descriptions de la distribution des matrices et sur quelques directives de compilation. On peut donc enfin constater que la portabilité des programmes et surtout la facilité de développement des codes entrent pour une bonne part dans le choix de telle ou telle machine. Par contre, comme pour les machines séquentielles, on constate un décalage important entre les progrès dans le logiciel et le matériel, et les constructeurs font, maintenant, de plus en plus appel à la collaboration des laboratoires universitaires pour créer les environnements de programmation de demain.

Chapitre 4

Communications globales

Dans ce chapitre, nous présentons nos contributions à l'étude des communications classiques sur des machines parallèles à mémoire distribuée. Nous proposons et analysons plusieurs algorithmes originaux dans le cas d'un nouveau modèle de réseaux reconfigurables et comparons les résultats théoriques avec des expérimentations sur Tnode. Dans une dernière partie, nous proposons un nouvel algorithme de diffusions successives pour la résolution du problème de factorisation de matrices.

4.1 Problèmes de communication classiques

Dans ce paragraphe, nous décrivons les principaux problèmes de communications globales dans les réseaux de processeurs. Ces problèmes ont déjà été très étudiés pour différentes topologies et différents modes de communication. Les algorithmes obtenus font partie des bibliothèques de communications globales comme nous l'avons vu précédemment (Chapitre 3, Paragraphe 3.2) [Rum93, CT93b].

- **La diffusion :** C'est une des opérations les plus utilisées dans les algorithmes parallèles. Appelée également *one-to-all* ou *broadcast*, elle consiste à ce qu'un processeur du réseau envoie un même message à tous les autres [Fra92]. D'autres problèmes peuvent être liés à ce problème comme la diffusion de plusieurs messages d'un même processeur vers tous les autres [BNK93], ou bien des diffusions successives de tous les processeurs du réseau, problème traité par la suite (Paragraphe 4.3). Un autre problème consiste à diffuser un message à un groupe de processeurs. Ce problème est appelé *multicast*.

- **L'échange total :** Tous les processeurs effectuent une diffusion. C'est une procédure de communication globale très utilisée et très coûteuse. Cette opération est également appelée *all-to-all*, *gossiping*, *total-exchange* [Fra92].

- **La diffusion personnalisée :** Cette opération, très proche de la diffusion dans sa définition, consiste à ce qu'un processeur du réseau envoie un message différent à tous les autres processeurs. Il s'agit par exemple de la distribution des données aux processeurs avant le début de l'exécution du programme parallèle. Cette routine est donc aussi appelée distribution, *scattering* ou bien *personalized one-to-all*. L'inverse de cette opération est le rassemblement ou *gathering*. On récupère des données différentes de tous les processeurs sur un seul. Cette dernière opération peut être associée à un calcul, on l'appellera alors réduction ou bien *global combine*. Elle sera traitée de manière pipeline sur un anneau dans un prochain chapitre (Chapitre 5, Paragraphe 5.7.1).

- **La multi-distribution :** Il s'agit d'effectuer une diffusion personnalisée de tous les processeurs du réseau. Cette opération est aussi appelée *multi-scattering*, *complete exchange*, *personalized all-to-all*, ... [Fra92]. La transposition d'une matrice rangée par lignes ou par colonnes peut être réalisée à l'aide d'une routine de multi-distribution [JH89].

- **La transposition par blocs :** Ce problème est très proche de la multi-distribution, mais ici, seuls les processeurs possédant des blocs de la matrice diagonalement opposés échangent leurs données [CT93a].

4.2 Algorithmes avec reconfiguration

Dans ce paragraphe, nous décrivons nos travaux sur les algorithmes permettant de résoudre les problèmes de communication classiques décrits précédemment sur un réseau reconfigurable de processeurs. Notons que les algorithmes décrits par la suite ne sont pas limités à des machines à réseaux optiques ou à base de transputers mais ils peuvent aussi être directement utilisés sur des machines permettant d'émuler un réseau complet avec toutefois un nombre maximum de liens k utilisables en parallèle. Il en va de même pour les machines implémentant le modèle de communication "cut-through" k-port. En effet, même s'il y a contention, l'algorithme peut être exécuté. Le problème consiste donc, dans ce cas, à savoir s'il y a interblocage et à évaluer son surcoût et si possible, modifier l'algorithme pour l'éviter.

4.2.1 Modèle de reconfiguration

Les modèles de communication et de reconfiguration utilisés dans les analyses de complexité correspondent à ceux de notre machine cible, le Tnode. Nous montrerons dans un prochain paragraphe que la portée de cette étude n'est pas limitée à ce type de modèle (Paragraphe 4.2.10).

Environnement C-NET

C-NET est un environnement de programmation pour machines SuperNode qui a été conçu pour tirer avantage des possibilités de reconfiguration de ce type de machine et qui permet le développement de programmes à topologies variables [ABB+91]. L'idée de base des programmes utilisant un tel environnement est qu'un programme peut être divisé en plusieurs phases, chacune séparée par une barrière de synchronisation et possédant une topologie des processeurs propre, cette topologie étant modifiée en cours d'exécution au début de chaque phase. Cela permet donc d'avoir la topologie de processeurs la plus appropriée à la phase courante et d'éviter le routage des messages au maximum, celui-ci conduisant à une baisse de performances, surtout avec le modèle de communication par commutation de message. Le modèle de coût de reconfiguration de cet environnement est donné par la formule suivante (en μs) :

$$T_{reconf} = 20 \left\lceil \frac{c}{8} \right\rceil + 7c + 31 \left\lceil \frac{n}{8} \right\rceil + 7n + 399, \tag{4.1}$$

où c est le nombre de processeurs de la phase courante et n le nombre de processeurs de la phase suivante. Ce modèle a été obtenu expérimentalement par les concepteurs de l'environnement. Il s'explique par le fait que tous les liens des processeurs sont modifiés. Une optimisation de l'environnement permettrait d'avoir un modèle comme celui présenté dans le prochain paragraphe. Avec le système C-NET, il est possible de recouvrir la reconfiguration par des calculs sans communication.

Modèles et hypothèses

Le modèle de reconfiguration de C-NET est peu précis et très lié à l'implémentation du système car il dépend du nombre de processeurs et non pas du nombre de liens modifiés. Par contre, une reconfiguration à l'aide d'un crossbar, tel que le C004 d'INMOS équipant le Tnode, s'effectue en écrivant dans sa mémoire les adresses des liens modifiés. Il s'agit donc d'un coût linéaire en nombre de liens. Nous adopterons donc par la suite le modèle : $\beta_r + N\tau_r$ où N est le nombre de liens modifiés, β_r un coût d'initialisation de la reconfiguration pour chaque phase, et τ_r un coût par lien. τ_r est très petit devant β_r. Il est également fonction du nombre de crossbars utilisés. Par contre, le coût d'initialisation β_r inclut le coût de synchronisation des processeurs et la vérification des connexions sur tous les liens des processeurs et est donc plus important.

Le programme est donc découpé en phases, chacune d'entre elles étant divisée en une sous-phase de reconfiguration où la topologie est construite et une sous-phase d'exécution du programme. Dans la seconde sous-phase, les communications s'effectuent avec le modèle linéaire classique à commutation de message $\beta + L\tau$. Grâce à la reconfiguration, toutes les communications s'effectuent avec des processeurs voisins. Rappelons également que le transputer peut communiquer sur tous ses liens en même temps et en full-duplex. Afin de généraliser cette étude à d'autres modèles, nous supposons que les processeurs ont k liens.

4.2.2 Algorithme de reconfiguration RCP

Dans ce paragraphe, nous décrivons l'algorithme de reconfiguration RCP (Reconfigurating Communication Pattern), un schéma de numérotation où, d'un nœud spécial (supposé par la suite et sans perte de généralité être le nœud 0), des topologies de degré k sont construites à chaque phase. A la fin de son exécution, le RCP garantit que pour n'importe quel nœud dans le système, il y a un chemin (en temps) le connectant au nœud 0. Et ceci, en un nombre minimum de phases. Nous supposons toujours dans un premier temps que le nombre de processeurs P est une puissance de $k+1$, le cas contraire est traité à la fin de chaque paragraphe.

A la première phase, le nœud 0 est connecté à k processeurs. A la deuxième phase, chaque processeur précédemment atteint est capable d'avoir le même comportement que le nœud 0 à la première phase. Ceci est itéré jusqu'à ce que tous les processeurs participant à l'algorithme de communication soient atteints (cf. Alg. 4.1 et un exemple pour $P = 64$ et $k = 3$ dans la Figure 4.1).

On peut voir facilement que le nombre de phases pour atteindre tous les nœuds est $h = \log_{k+1}(P)$. Il y a donc $\log_{k+1}(P)$ sous-phases de reconfiguration. Puisque le RCP peut être vu comme une forêt d'arbres k-aires, le nombre total de liens modifiés est $P-1$. De ce fait, sous le modèle donné dans le paragraphe précédent (Paragraphe 4.2.1), le coût de reconfiguration est donné par :

$$T^{RCP}_{reconf} = \log_{k+1}(P)\beta_r + (P-1)\tau_r. \tag{4.2}$$

Soit l la phase courante et i le numéro d'un processeur déjà atteint. Nous définissons le RCP de telle manière que le nœud i, à la phase l, soit connecté aux nœuds :

$$(k+1)^l + ik,\ (k+1)^l + ik + 1,\ (k+1)^l + ik + 2,\ \cdots,\ (k+1)^l + ik + k - 1$$

Dans les paragraphes suivants, nous noterons $h = \log_{k+1}(P)$.

4.2.3 Algorithme de diffusion personnalisée RCP

Nous supposons qu'au départ, la racine possède P paquets de taille L. L'algorithme de diffusion personnalisée utilisant le RCP est immédiat. A chaque phase de l'algorithme, la racine divise l'ensemble des données en $k+1$ parties égales. Un des paquets résultants reste sur la racine (il contient les données utilisées dans les phases suivantes) tandis que les k autres sont envoyés aux k nœuds connectés à cette phase, donnant un coût total de communication $\beta + \frac{PL}{k+1}\tau$.

A la phase l ($l = 1..h-1$), quand la communication est terminée, une reconfiguration s'effectue et chaque processeur ayant été atteint à la phase précédente se comporte comme la racine à la première phase. A la phase l, nous avons $(k+1)^l$ racines possédant $\frac{PL}{(k+1)^l}$ éléments du message. Tous les nœuds atteints (y compris la racine 0) se connectent à k fils et envoient $1/(k+1)$-ième du

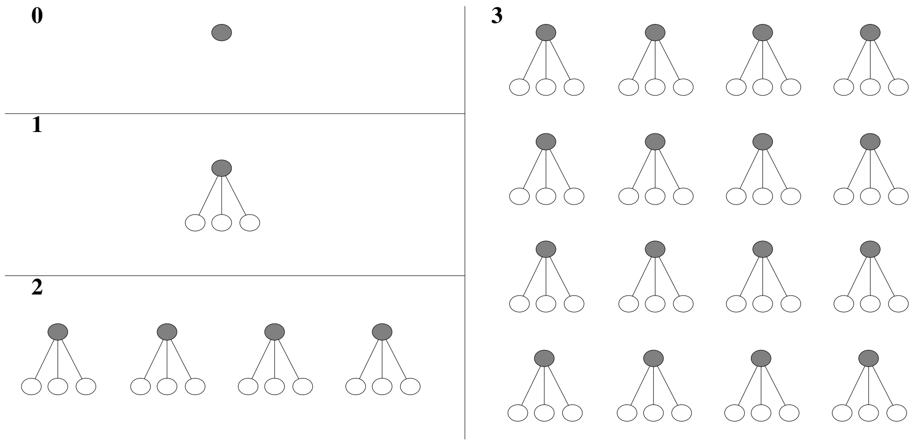

FIG. 4.1 - Algorithme RCP pour $P = 64$ nœuds avec $k = 3$.

```
i = mon_numéro
pour l = 0 à log_{k+1}(P) − 1 faire
   si i < (k + 1)^l /* J'ai été atteint à la phase l − 1 */
      pour j = 0 à k − 1 faire
         me connecter au nœud (k + 1)^l + ik + j
      finpour
   sinon
      si i < (k + 1)^{l+1} Je suis connecté à cette phase
      sinon /* Je ne suis pas concerné par cette phase */
      finsi
   finsi
finpour
```

ALG. 4.1 - Algorithme implémentant le RCP.

message reçu à la phase précédente. L'algorithme s'arrête quand le message sur chaque nœud est de longueur L (la taille du message personnel). A cette phase, tous les nœuds ont été atteints.

Le coût de reconfiguration est donné par le coût de reconfiguration de l'algorithme RCP, soit :

$$\begin{aligned} T^{scat}_{reconf} &= T^{RCP}_{reconf} \\ &= \log_{k+1}(P)\beta_r + (P-1)\tau_r. \end{aligned} \tag{4.3}$$

Le coût total de communication est égal à la somme des coûts à chaque phase, soit :

$$\begin{aligned} T^{scat}_{com} &= \sum_{i=1}^{h} \left(\beta + \frac{PL}{(k+1)^i}\tau\right) \\ &= \log_{k+1}(P)\beta + \frac{P-1}{k}L\tau. \end{aligned} \tag{4.4}$$

Dans le cas où le nombre de processeurs P n'est pas une puissance de $k+1$, le nombre de phases reste le même mais certains processeurs ne travaillent plus à la fin de l'algorithme (ils n'ont plus de fils).

4.2.4 Algorithme de réduction RCP

Nous supposons que les P processeurs possèdent chacun un vecteur de taille L. L'opération de réduction est une opération qui consiste à réduire sur un processeur racine tous les vecteurs à l'aide d'une opération $\bigoplus$. Cette opération est généralement la somme, le minimum, le maximum et les opérations logiques classiques. Une variation de cette opération nécessite que le résultat soit laissé sur tous les processeurs.

L'algorithme RCP de rassemblement, qui est l'inverse de l'algorithme de diffusion personnalisée, peut être utilisé ici. A chaque phase, les processeurs “racines” reçoivent un vecteur de chaque processeur auxquels ils sont connectés et le combinent avec leur propre vecteur. Chaque phase a donc un coût égal à $\beta + L\tau + kL\tau_a$ où τ_a est le coût de l'opération $\bigoplus$ pour un élément. Le coût total de communication est égal à la somme des coûts à chaque phase, soit :

$$\begin{aligned} T_{com}^{red} &= \sum_{i=1}^{h} (\beta + L\tau + kL\tau_a) \\ &= \log_{k+1}(P)\,(\beta + L\tau + kL\tau_a)\,. \end{aligned} \tag{4.5}$$

4.2.5 Algorithme de diffusion RCP

Dans ce paragraphe, nous décrivons deux algorithmes utilisant le RCP pour la diffusion : un algorithme naïf et un algorithme à trois étapes.

Algorithme naïf

Si nous imaginons que, dans le RCP, le nœud 0 doit envoyer un message identique à tous les processeurs, alors un algorithme de diffusion immédiat est obtenu, où à chaque phase, chaque nœud atteint envoie le message le long de ses k liens. Puisque chaque phase coûte $(\beta + L\tau)$ en communication, nous avons un coût total égal à :

$$T_{tot}^{naif} = T_{reconf}^{RCP} + \log_{k+1}(P)(\beta + L\tau). \tag{4.6}$$

Algorithme à trois étapes

L'idée sous-jacente de l'algorithme à trois étapes est d'exécuter dans les deux premières étapes des phases de coûts inférieurs à celles de l'algorithme naïf afin d'augmenter, dans la dernière phase, le nombre de processeurs actifs. Nous allons décrire dans un premier temps une version de l'algorithme dont la première étape consiste à ce que la racine distribue une partie du message à ses fils, suivie d'une étape de diffusion utilisant l'algorithme naïf avec $k + 1$ racines. La dernière étape consistant à regrouper le message distribué. Dans un second temps, nous généraliserons cet algorithme avec l'utilisation d'une première étape basée sur l'algorithme de diffusion personnalisée décrit précédemment.

A la première phase de la première étape de l'algorithme, le message est découpé $(k+1)$ parties. La racine distribue une partie à chacun des processeurs connectés. Le coût de communication de cette phase est de $\beta + (L/(k+1))\tau$. Nous avons alors $k+1$ racines devant diffuser un message de taille $L/(k+1)$. Nous supposons alors que la racine ne diffuse alors qu'une partie du message. L'algorithme naïf s'exécute alors et quand tous les processeurs ont été atteints, nous avons $k+1$ ensembles de processeurs possédant $1/(k+1)$ du message. Le coût de cette seconde étape est égal à $(\log_{k+1}(P)-1)(\beta+(L/(k+1))\tau)$. De ce fait, afin de terminer l'opération de diffusion, le message doit être reconstruit. Pour cela, nous formons des cliques K_{k+1} (réseaux complets) à $k+1$ nœuds (un de chaque ensemble) qui s'échangent, à travers leurs liens de communication, les parties manquantes du message original. Il s'agit alors d'un échange total entre les $k+1$ nœuds de chaque clique. Le coût de cette dernière étape est donc égal à $\beta + (L/(k+1))\tau$. Le coût total en communication de l'algorithme à trois étapes version 1 est donc égal à

$$T_{com}^{3pv1} = (\log_{k+1}(P) + 1)\left(\beta + \frac{L}{k+1}\tau\right). \tag{4.7}$$

Le coût de reconfiguration des deux premières étapes est égal au coût du RCP. Quant à celui de la seconde étape, il est aisé de voir qu'il est égal au coût de la création de $P/(k+1)$ cliques à $k(k+1)/2$ liens soit au total pour tout l'algorithme

$$T_{reconf}^{3pv1} = (\log_{k+1}(P) + 1)\beta_r + (P-1)\tau_r + \frac{Pk}{2}\tau_r. \tag{4.8}$$

Nous avons donc réduit le coût de communication en augmentant le nombre d'étapes. Par contre, cette augmentation se traduit par un surcoût en terme de reconfiguration.

Remarquons que nous pouvons appliquer cette méthode de manière récursive. Nous obtenons donc l'algorithme à trois étapes version 2 avec une première étape, où le message est découpé de manière continue pendant les phases du RCP. Si le découpage est effectué h fois, la seconde étape de diffusion n'est pas exécutée. Nous avons alors une dernière étape, où le message est reconstruit, phase après phase, à l'aide des opérations de reconstruction (échange total) implémentées à l'aide de cliques à $(k+1)$ nœuds. Au départ, nous avons P ensembles à un processeur, puis $P/(k+1)$ ensembles à $k+1$ processeurs, et ainsi de suite jusqu'à ce que tous les processeurs aient le message complet.

Il est clair que le nombre de liens créés est plus grand dans la dernière étape car l'on construit des cliques et non plus des arbres. De ce fait, le coût de reconfiguration est plus grand. En fait, plus le nombre de phases de découpage est grand, plus le coût de la dernière étape est important. Le gain dû au découpage est perdu avec le coût de la reconfiguration. Une solution est de stopper la première étape, à la phase h' $(0 \leq h' \leq h)$ La seconde étape étant constituée de $h - h'$ phases de diffusion naïve. Nous aurons également h' phases de reconstruction du message.

Un exemple avec $P = 64$ et $k = 3$ est donné sur la Figure 4.2. Il y a deux phases dans la première étape $(h' = 2)$, une seule dans la seconde et de ce fait deux phases de reconstruction dans la dernière étape.

Nous donnons à présent la complexité des trois étapes de l'algorithme.

$$\begin{aligned}
T_{com}^{ph.1} &= \sum_{i=1}^{h'} \left(\beta + \frac{L}{(k+1)^i}\tau\right), \\
T_{com}^{ph.2} &= \sum_{i=h'+1}^{h} \left(\beta + \frac{L}{(k+1)^{h'}}\tau\right), \\
T_{com}^{ph.3} &= \sum_{i=1}^{h'} \left(\beta + \frac{L}{(k+1)^i}\tau\right),
\end{aligned}$$

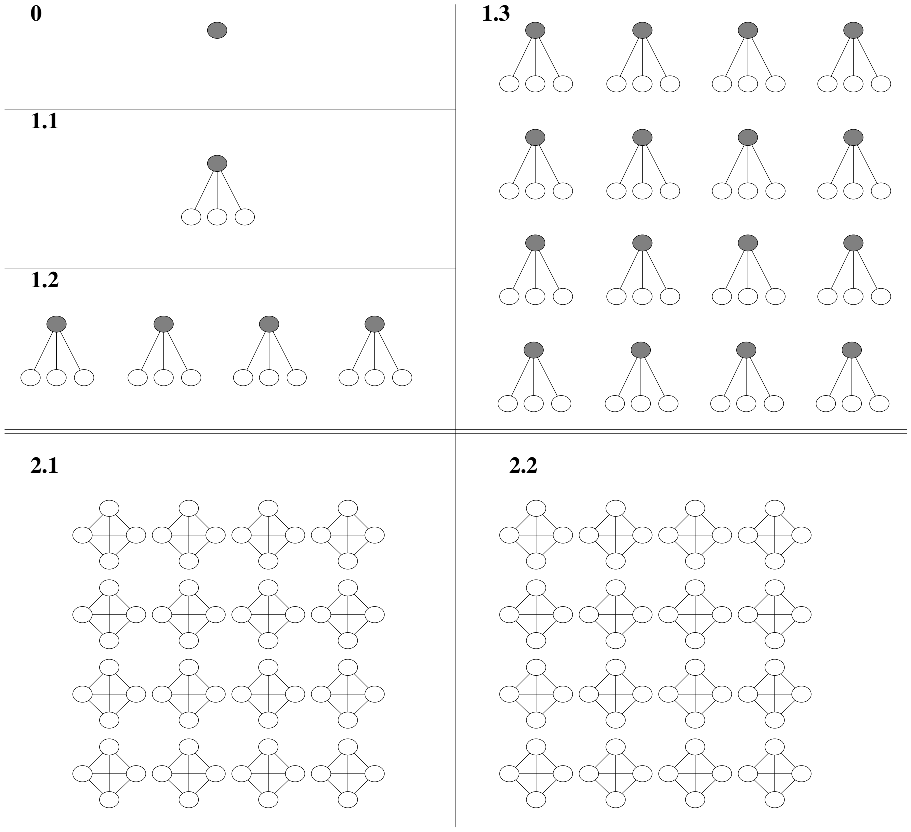

FIG. 4.2 - Algorithme de diffusion utilisant le RCP avec $P = 64$ nœuds et pour $k = 3$ et $h' = 2$.

ce qui donne un coût de communication égal à

$$\begin{aligned}
T_{com}^{3pv2} &= 2\sum_{i=1}^{h'}\left(\beta + \frac{L}{(k+1)^i}\tau\right) + \sum_{i=h'+1}^{h}\left(\beta + \frac{L}{(k+1)^{h'}}\tau\right) \\
&= (h'+h)\beta + \left(\frac{2}{k}\left((k+1)^{h'}-1\right) + h - h'\right)\frac{L\tau}{(k+1)^{h'}} \\
&= \left(h' + \log_{k+1}(P)\right)\beta + \left(\frac{2}{k}\left((k+1)^{h'}-1\right) + \log_{k+1}(P) - h'\right)\frac{L\tau}{(k+1)^{h'}} \\
&\leq \left(h' + \log_{k+1}(P)\right)\beta + \frac{2}{k}L\tau + \frac{(\log_{k+1}(P) - h')}{(k+1)^{h'}}L\tau. \qquad (4.9)
\end{aligned}$$

Les coûts de reconfiguration sont donnés par

$$\begin{aligned}
T_{reconf}^{ph.1,2} &= T_{rec}^{RCP} \\
&= \log_{k+1}(P)\beta_r + (P-1)\tau_r \\
T_{ph.3}^{reconf} &= h'\left(\beta_r + \frac{Pk}{2}\tau_r\right),
\end{aligned}$$

ce qui donne le coût total de reconfiguration :

$$T^{3pv2}_{reconf} = (h' + \log_{k+1}(P))\beta_r + \left[(P-1) + h'\frac{Pk}{2}\right]\tau_r. \tag{4.10}$$

Si nous supposons que l'étape de découpage est effectuée jusqu'à la fin du RCP, c'est-à-dire $h' = h$, le coût de communication est égal à :

$$\begin{aligned} T^{3pv2}_{com} &= 2\left(\log_{k+1}(P)\beta + \frac{P-1}{kP}L\tau\right) \\ &\leq 2\log_{k+1}(P)\beta + \frac{2L\tau}{k}, \end{aligned} \tag{4.11}$$

et le coût de reconfiguration est donné par :

$$T^{3pv2}_{reconf} = 2\log_{k+1}(P)\beta_r + \left[(P-1) + \log_{k+1}(P)\frac{Pk}{2}\right]\tau_r, \tag{4.12}$$

Le coût total de la diffusion RCP T_{3pv2} est donné par la somme des coûts de reconfiguration T^{3pv2}_{reconf} et du coût de communication T^{3pv2}_{com}.

A présent, nous voulons calculer le nombre optimal de phases de découpage (h'_{opt}) qui minimise le temps total d'exécution. Pour l'obtenir, nous dérivons la fonction de coût total par rapport à h'. Nous supposons tout d'abord que le coût de reconfiguration est égal à zéro. Les formules ont été obtenues avec Maple V.

$$\begin{aligned} \frac{\partial T^{tot}_{com}}{\partial h'} &= 0 \\ h' &= \frac{\log(P)k + k - 2\log(k+1) - O\left(\frac{e^{\frac{\log(P)k+k-2\log(k+1)}{k}}\beta}{L\tau}\right)k}{k\log(k+1)}. \end{aligned} \tag{4.13}$$

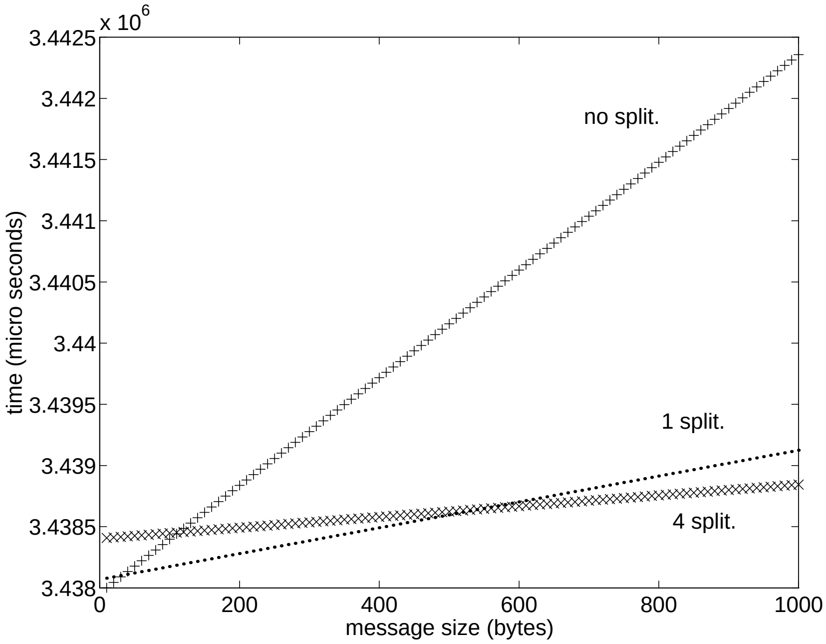

FIG. 4.3 - Diffusion RCP sur 3125 nœuds avec $k = 4$ ($h' = 0$, 1 et 4).

Si nous tenons compte du coût de reconfiguration dans notre minimisation, nous avons :

$$\frac{\partial(T^{3pv2}_{com} + T^{3pv2}_{reconf})}{\partial h'} = 0 \tag{4.14}$$

$$h' = \frac{\log(P)k + k - 2\log(k+1) - O\left(\frac{e^{\frac{\log(P)k+k-2\log(k+1)}{k}}(2\beta+2\beta_r+Pk\tau_r)}{2L\tau}\right)k}{k\log(k+1)} \tag{4.15}$$

Pour la partie communication, il y a un compromis à trouver entre $\log_{k+1}(P)\beta$ et $2\log_{k+1}(P)\beta$ pour le coût d'initialisation et entre $\log_{k+1}(P)L\tau$ et $\frac{2L\tau}{k}$ pour le temps de propagation, comme suit :

h'	**Reconfiguration**	**Communication**
0	$\log_{k+1}(P)\beta_r + (P-1)\tau_r$	$\log_{k+1}(P)(\beta + L\tau)$
$\log_{k+1}(P)$	$2\log_{k+1}(P)\beta_r + \left((P-1) + \log_{k+1}(P)\frac{Pk}{2}\right)\tau_r$	$2\log_{k+1}(P)\beta + \frac{2L\tau}{k}$

La Figure 4.3 montre le temps d'une diffusion RCP de messages de taille 10 à 1000 octets sur une machine cible théorique avec $P = 3125$ processeurs à degré $k = 4$ et avec les paramètres suivants : (ils correspondent à ceux du Tnode de Telmat).

β_r	β	τ
100 μs	11.5 μs	0.88 μs/byte

Nous supposons que h' vaut successivement 0, 1 et 4 ($\log_5(3125) = 5$). Remarquons que, puisque τ_r est le coût d'une affectation en mémoire, il est négligeable devant β_r. Pour des petits messages (≤ 20 octets) l'étape de découpage n'est pas recommandée, un découpage doit être effectué entre 20 et 550 octets, quatre découpages ensuite. Ceci nous donne une idée des temps d'exécution qui pourraient être obtenus avec le Tnode.

Si le nombre de processeurs n'est pas une puissance de $k+1$, l'algorithme naïf peut être utilisé sans aucun problème, les coûts seront identiques et certains processeurs resteront inactifs à la fin de l'algorithme. Si l'on veut utiliser l'algorithme avec découpage, il suffit de rajouter une étape à l'algorithme général: nous appliquerons l'algorithme avec découpage pour la puissance de $k+1$ immédiatement inférieure à P et l'algorithme naïf pour le reste des processeurs.

4.2.6 Algorithme d'échange total

Cette opération peut être implémentée de manière efficace sur un réseau reconfigurable en utilisant plusieurs phases d'opérations d'échange total sur des topologies K_{k+1} comme pour la dernière phase de l'algorithme de diffusion à trois phases version 2. Le nombre de phases reste le même que pour les algorithmes précédents. Le coût de reconfiguration est donné par :

$$\begin{aligned} T_{ata}^{reconf} &= \sum_{i=1}^{h} \left(\beta_r + \frac{Pk}{2}\tau_r\right) \\ &= h\left(\beta_r + \frac{Pk}{2}\tau_r\right) \\ &= \log_{k+1}(P)\left(\beta_r + \frac{Pk}{2}\tau_r\right), \end{aligned} \tag{4.16}$$

et le coût de communication par :

$$\begin{aligned} T_{ata}^{com} &= \sum_{i=1}^{h} \left(\beta + (k+1)^{i-1}L\tau\right) \\ &= \log_{k+1}(P)\beta + \frac{(P-1)L}{k}\tau. \end{aligned} \tag{4.17}$$

4.2.7 Algorithme de multi-distribution

Nous utilisons les mêmes méthodes de reconfiguration que pour l'opération d'échange total, et à chaque phase de l'algorithme, chaque nœud envoie un message de taille $\frac{PL}{k+1}$.

$$\begin{aligned} T_{pata}^{reconf} &= T_{ata}^{reconf} \\ T_{pata}^{com} &= \sum_{i=1}^{h} \left(\beta + \frac{PL\tau}{k+1}\right) \\ &= \log_{k+1}(P)\left(\beta + \frac{PL}{k+1}\tau\right). \end{aligned} \tag{4.18}$$

4.2.8 Résumé des résultats

Dans les tableaux 4.1 et 4.2, nous donnons un résumé des résultats de complexité obtenus et les bornes inférieures en coût d'initialisation et en bande passante pour chacun des algorithmes. La borne inférieure en temps d'initialisation est donnée par le nombre minimum de phases pour atteindre tous les processeurs. Celle en bande passante est donnée par la taille du message sortant (ou entrant) d'un processeur, divisée par son nombre de liens.

L'algorithme de diffusion est divisé en deux cas. Pour celui où aucun découpage n'est effectué (algorithme naïf ($h' = 0$)), la borne inférieure en nombre de phases est atteinte. Pour le second cas, nous supposons que le découpage est effectué h fois ($h' = h$). Nous sommes alors à un facteur deux du nombre de phases et de la bande passante. Les algorithmes intermédiaires se situeront entre les deux ($0 < h' < h$). Nous constatons par ailleurs que les algorithmes de diffusion personnalisée et d'échange total atteignent les bornes inférieures en nombre de phases et en bande passante. L'algorithme de multidistribution atteint la borne inférieure en nombre de phases mais est à un facteur $\frac{(k+1)(P-1)}{Pk\log_{k+1}(P)}$ de la borne inférieure en bande passante.

Algorithme	**Coût de communication**	**Borne inférieure**	
		Init.	**BP**
Diffusion ($h' = 0$)	$\log_{k+1}(P)(\beta + L\tau)$	$\log_{k+1}(P)\beta$	$\frac{L}{k}\tau$
Diffusion ($h' = h$)	$2\log_{k+1}(P)\beta + \frac{2L\tau}{k}$	$\log_{k+1}(P)\beta$	$\frac{L}{k}\tau$
Diff. pers.	$\log_{k+1}(P)\beta + \frac{(P-1)L}{k}\tau$	$\log_{k+1}(P)\beta$	$\frac{(P-1)L}{k}\tau$
Ech. tot.	$\log_{k+1}(P)\beta + \frac{(P-1)L}{k}\tau$	$\log_{k+1}(P)\beta$	$\frac{(P-1)L}{k}\tau$
Multi-dist.	$\log_{k+1}(P)\left(\beta + \frac{PL}{k+1}\tau\right)$	$\log_{k+1}(P)\beta$	$\frac{(P-1)L}{k}\tau$

TAB. 4.1 - Résumé des coûts de communication et des bornes inférieures

Algorithme	**Coût de reconfiguration**
Diffusion ($h' = 0$)	$\log_{k+1}(P)\beta_r + (P-1)\tau_r$
Diffusion ($h' = h$)	$2\log_{k+1}(P)\beta_r + \left((P-1) + \log_{k+1}(P)\frac{Pk}{2}\right)\tau_r$
Diff. pers.	$\log_{k+1}(P)\beta_r + (P-1)\tau_r$
Ech. tot.	$\log_{k+1}(P)\left(\beta_r + \frac{Pk}{2}\tau_r\right)$
Multi-dist.	$\log_{k+1}(P)\left(\beta_r + \frac{Pk}{2}\tau_r\right)$

TAB. 4.2 - Résumé des coûts de reconfiguration

4.2.9 Algorithme de transposition par blocs

Nous supposons que la matrice à transposer est rangée par blocs carrés de taille m^2 sur une grille carrée et l'on souhaite obtenir une matrice transposée rangée de la même manière [CT93a]. Les échanges se font entre les processeurs numérotés (i, j) et (j, i) (avec $i \neq j$). La matrice doit être transposée également localement car nous supposons que sa transposée sera utilisée dans d'autres calculs ultérieurs. Si cela n'est pas le cas, un simple changement d'indices suffit. Nous utilisons un réseau direct de processeurs où chaque processeur diagonalement opposé dans la grille peut être connecté par au moins un lien après une étape de reconfiguration. Ce réseau est donné sur la Figure 4.4.

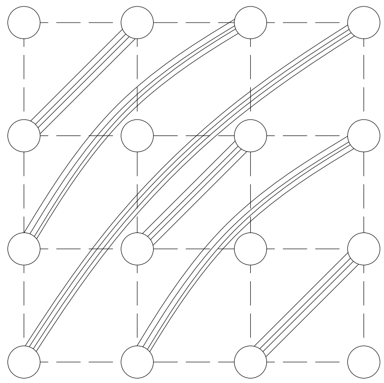

FIG. 4.4 - Réseau direct obtenu par reconfiguration à partir d'une grille ($k = 4$).

Par la suite, nous parlerons de processeurs sous(sur)-diagonaux, ceci n'a, bien entendu, pas de sens physique pour ce réseau. Cependant, comme nous supposons que nous exécutons la transposition d'une matrice rangée par blocs sur une grille, que nous améliorons à l'aide d'une reconfiguration, nous qualifierons de processeur sous(sur)-diagonal un processeur possédant une sous-matrice (i,j) avec $i > j$ ($i < j$). Un processeur "diagonal" possèdera ainsi une matrice d'indices (i,j) avec $i = j$. Nous noterons $T_t(m) = \frac{m(m-1)}{2}\tau_e$, le coût séquentiel de la transposition d'une matrice $m \times m$, où τ_e est le coût d'échange de deux éléments d'une matrice.

Dans toutes nos analyses de complexité, nous prenons en compte le coût de transfert des données dans les mémoires car celui-ci n'est pas négligeable lorsqu'il s'agit d'accéder à des mémoires externes. Les coûts de rangement peuvent être également associés à un calcul si la transposition est utilisée de manière conjointe avec un calcul. Comme nous le verrons par la suite (Chapitre 6, Paragraphe 6.4), la transposition peut être associée à une addition pour résoudre le problème de la mise à jour de rang 2k.

La transposition utilisant un réseau direct de processeurs est immédiate et ne requiert que des communications voisins/voisins. Un premier algorithme "naïf" s'exécute en deux pas. A la première étape, les sous-matrices sont échangées entre les processeurs connectés (les processeurs diagonaux ne font rien). Ensuite, tous les processeurs transposent de manière interne les matrices reçues lors de l'échange. Le coût optimal de transposition utilisant cet algorithme est une fonction du nombre de liens utilisés pour connecter les processeurs. Nous avons vu, dans le paragraphe concernant les topologies, qu'afin d'augmenter la bande passante entre deux processeurs, et si leur degré le permettait, on pouvait connecter les processeurs entre eux par plusieurs liens. Ici, comme nous avons un réseau non connexe et que le degré des processeurs est supposé être égal à k_{max}, le nombre de liens peut aller de 1 à k_{max}. Par contre, si le coût de reconfiguration est une fonction du nombre de liens, alors, lorsque le coût de communication diminue (du fait de l'augmentation de la bande passante), le coût de reconfiguration augmente. Le graphe d'interconnexion optimal dépend donc des caractéristiques physiques de la machine.

Avec nos expérimentations sur la machine Tnode avec le système C-NET, le modèle de reconfiguration est une fonction du nombre de processeurs et de ce fait, nous avons tout intérêt à prendre le plus grand nombre de liens possible entre les processeurs ($k_{max} = 4$ avec les transputers). Malgré tout, nous présentons une analyse de complexité faisant référence au modèle linéaire en nombre de liens modifiés qui est plus réaliste. Le coût de l'algorithme "naïf" est donné par la somme du coût de reconfiguration pour obtenir le réseau direct, le coût de communication et le coût de la transposition locale, soit

$$T_{tr}^1 = T_{reconf} + \left(\beta + \frac{m^2}{k}\tau\right) + T_t(m), \tag{4.19}$$

où $T_{reconf} = \beta_r + \frac{(\sqrt{P})(\sqrt{P}-1)}{2}k\tau_r$ est le coût de l'établissement de k liens entre les processeurs dans le réseau direct. On peut minimiser la fonction de temps T_{tr}^1 en la dérivant par rapport au nombre de liens k. Ce qui nous donne un nombre de liens optimal égal à $k = m\sqrt{\frac{2\tau}{(\sqrt{P})(\sqrt{P}-1)\tau_r}}$ avec comme contrainte que $1 \leq k \leq k_{max}$.

Une optimisation possible de l'algorithme "naïf" consiste à recouvrir la transposition interne des matrices avec les échanges en utilisant un algorithme de type "macro-pipeline". Cependant, cet algorithme nécessite une plus grande place mémoire car il faut pouvoir stocker la matrice initiale et la matrice transposée dans le réseau, ce qui double son coût mémoire. L'algorithme est donné sur la Figure 4.2.

```
en // faire
   recevoir un sous-bloc
   envoyer un sous-bloc
fin
pour i = 1 à μ − 1 en // faire
   recevoir un sous-bloc
   envoyer un sous-bloc
   ranger le sous-bloc précédent
finpour
ranger le dernier sous-bloc
```

ALG. 4.2 - Algorithme utilisant le *macro-pipeline*.

Les sous-matrices sont découpées en μ paquets de taille $\frac{m^2}{\mu}$. Après avoir reçu le premier paquet, les transpositions et les communications s'exécutent en parallèle, en utilisant la même méthode que pour le pipeline utilisé pour améliorer le coût de communication en CM (Paragraphe 2.2.1). Le coût de la transposition est alors donné par :

$$T_{tr}^2 = T_{reconf} + \beta + \frac{m^2}{\mu}\frac{\tau}{k} + (\mu - 1)\max\left(\beta + \frac{m^2}{\mu}\frac{\tau}{k}, T_{irgt}\right) + T_{irgt} \tag{4.20}$$

où $T_{irgt}(m) = \frac{m^2}{\mu}\tau_{aff}$ est le coût de rangement d'un paquet issu d'une sous-matrice de taille $m \times m$ (τ_{aff} est le coût d'une affectation). Ici, le recouvrement des communications est total uniquement si le coût de rangement d'un paquet est plus grand que le coût de l'échange du paquet suivant, c'est-à-dire si l'inégalité :

$$\frac{m^2}{\mu}\tau_{aff} \geq \beta + \frac{m^2}{\mu}\frac{\tau}{k}, \tag{4.21}$$

est vérifiée.

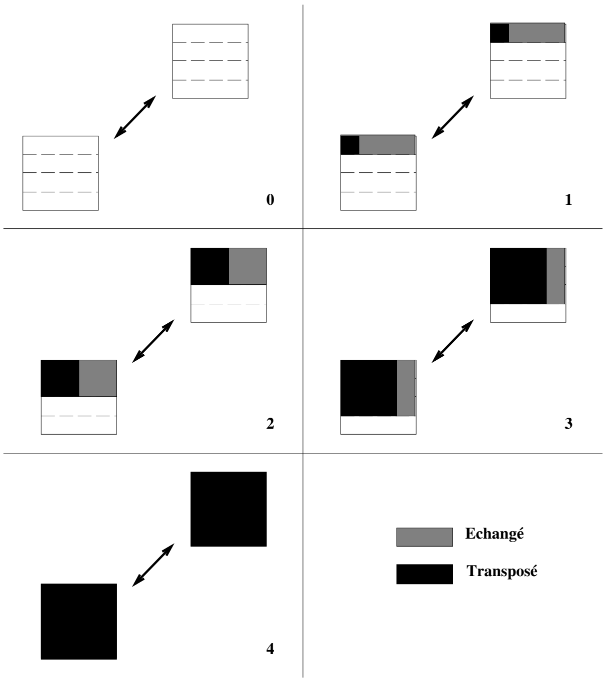

FIG. 4.5 - Problème de la transposition macro-pipeline

Si $\tau_{aff} < \frac{\tau}{k}$, le recouvrement n'est pas possible. Si $\tau_{aff} \geq \frac{\tau}{k}$, le recouvrement total est obtenu pour :

$$\mu \geq \frac{(k\tau_{aff} - \tau)m^2}{k\beta}. \tag{4.22}$$

De plus, nous avons : $1 \leq \mu \leq m^2$. Avec comme machine cible le Tnode, nous avons $\tau = 6,4\mu s$ et $\tau_{aff} = 1,5\mu s$. Même avec quatre liens utilisés, on peut vérifier que l'on a toujours un recouvrement total des rangements par les communications et que le temps total est donc égal à :

$$T_{tr}^2 = T_{reconf} + \mu\beta + \frac{m^2\tau}{k} + \frac{m^2}{\mu}\tau_{aff}. \tag{4.23}$$

On peut minimiser le temps en le dérivant par rapport à μ et en cherchant μ pour lequel il est égal à 0, soit $\mu_{opt} = m\sqrt{\frac{\tau_{aff}}{\beta}}$. Remarquons que sur une machine comme le Tnode, β varie avec le nombre de liens employés en parallèle. Le recouvrement, même s'il ne concerne que les rangements et pas les communications, est bel et bien présent et permet d'obtenir un gain dans le coût total de l'algorithme. Ce dernier est donné par :

$$T_{tr} = T_{reconf} + 2m\sqrt{\beta\tau_{aff}} + m^2\frac{\tau}{k}. \tag{4.24}$$

Si à présent, la taille mémoire de notre machine ne nous permet pas de garder ponctuellement deux copies de la matrice à transposer mais uniquement un buffer de la taille d'un paquet soit : $\frac{m^2}{\mu}$, nous pouvons modifier l'algorithme pour avoir quand même un certain recouvrement des transpositions internes et des communications. Le problème est que maintenant, la transposition interne ne peut pas se faire entièrement à chaque paquet reçu car, dans ce cas, nous aurions un écrasement des données à envoyer par les données déjà reçues.

Une partie croissante de la sous-matrice peut être transposée de manière interne à chaque réception de paquet (Figure 4.5). Ce problème pourrait être résolu si la machine disposait de primitives hardware permettant d'envoyer des messages non contigus. Il suffirait alors de découper la matrice en blocs de taille $\frac{m^2}{\mu^2}$ et d'échanger des blocs diagonalement opposés. Ce genre de primitives sera disponible sur des machines à base de processeurs C40 de chez Texas Instrument. Si nous voulons implémenter cet algorithme avec notre modèle de machine qui ne possède pas de chaînage de blocs mémoires, nous devons avoir un buffer supplémentaire et des recopies pour les processeurs sous-diagonaux. Mais pour rester réaliste avec le modèle de machine étudié, nous avons supposé que ce n'était pas possible et que l'envoi d'un message contenant des données rangées de manière non-contiguës en mémoire nécessitait des recopies dans un buffer. Le coût de l'algorithme 4.5 est donné par l'équation suivante :

$$\begin{aligned} T_{tr} &= T_{reconf} + \beta + \frac{m^2}{\mu}\frac{\tau}{k} + \\ &\sum_{i=0}^{\mu-2}\left(\max\left(\frac{m^2}{\mu}\tau_{aff} + i\frac{m^2}{\mu^2}\tau_{aff}, \beta + \frac{m^2}{\mu}\frac{\tau}{k}\right)\right) + \frac{m^2}{\mu}\tau_{aff} + (\mu-1)\frac{m^2}{\mu^2}\tau_{aff}. \end{aligned} \quad (4.25)$$

Le coût total se trouve donc compris entre deux bornes. Soit le coût des mouvements internes est toujours supérieur au coût de communication d'un paquet, soit le contraire. Le premier n'est bien entendu pas réaliste avec nos machines cibles. Le coût de rangement croissant dans le temps au fur et à mesure que les paquets arrivent, il est possible qu'il existe un i tel que ce coût soit supérieur au coût de communication. Nous allons analyser les deux cas extrêmes.

Coût de rangement > coût de communication Le coût total est alors la somme de tous les coûts de rangement et d'une communication d'un paquet, soit :

$$T_{tr}^1 = T_{reconf} + \beta + \frac{m^2}{\mu}\frac{\tau}{k} + \frac{3\mu-1}{2\mu}m^2\tau_{aff}, \quad (4.26)$$

soit asymptotiquement :

$$T_{tr}^1 \approx T_{reconf} + \beta + \frac{m^2}{\mu}\frac{\tau}{k} + \frac{3m^2\tau_{aff}}{2}. \quad (4.27)$$

Dans ce cas, il faut choisir la taille de paquet la plus petite possible, c'est-à-dire un élément par lien. Ce qui donne un temps total égal à :

$$T_{tr}^1 = T_{reconf} + \beta + \tau + \frac{3m^2\tau_{aff}}{2}. \quad (4.28)$$

Coût de rangement < coût de communication Le coût total est alors la somme de toutes les communications et du dernier rangement, soit :

$$T_{tr}^2 = T_{reconf} + \mu\beta + \frac{m^2\tau}{k} + (2\mu-1)\frac{m^2}{\mu^2}\tau_{aff}, \quad (4.29)$$

soit asymptotiquement :

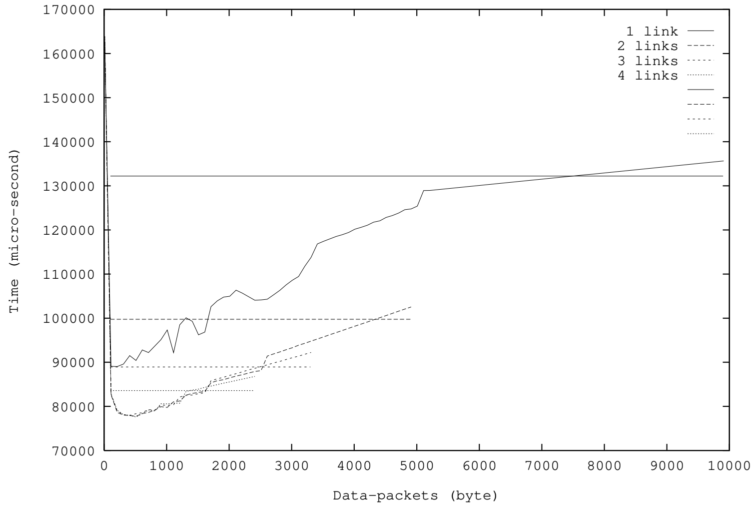

FIG. 4.6 - Calcul expérimental des tailles de paquets optimales pour la transposition de matrice par blocs sur un réseau direct.

$$T_{tr}^2 = T_{reconf} + \mu\beta + \frac{m^2\tau}{k} + \frac{2m^2\tau_{aff}}{\mu}. \tag{4.30}$$

Nous pouvons alors calculer la taille de paquet optimale en minimisant T_{tr}^2, ce qui nous donne $\mu = m\sqrt{\frac{2\tau_{aff}}{\beta}}$.

En remplaçant μ dans l'équation 4.30, le temps total est alors égal à :

$$T_{tr}^2 = T_{reconf} + 2m\sqrt{2\tau_{aff}\beta} + \frac{m^2\tau}{k}. \tag{4.31}$$

Remarquons qu'il serait possible d'avoir une taille de paquet décroissante qui permettrait de maximiser le recouvrement des rangements et des communications. Cette méthode a été appliquée au produit matrice/vecteur par Colombet, Michallon et Trystram [CMT93] mais conduit à des calculs de récurrences complexes.

Expérimentations

Nos expérimentations ont été réalisées sur la machine Tnode de Telmat à 32 processeurs avec l'environnement C-NET. La Figure 4.6 montre la détermination expérimentale de la taille optimale de paquet en utilisant de 1 à 4 liens pour connecter les processeurs entre eux. Les courbes horizontales correspondent à la méthode n'utilisant pas de macro-pipeline. Son coût est sensiblement inférieur à la méthode macro-pipeline n'utilisant qu'un seul paquet car cette dernière nécessite la mise en place de processus gérant les échanges non-bloquants. La Figure 4.7 montre les temps de transposition sur un réseau direct avec ou sans macro-pipeline. Si le macro-pipeline est utilisé, la taille de paquet choisie est celle donnée par la courbe de détermination des tailles optimales de paquets.

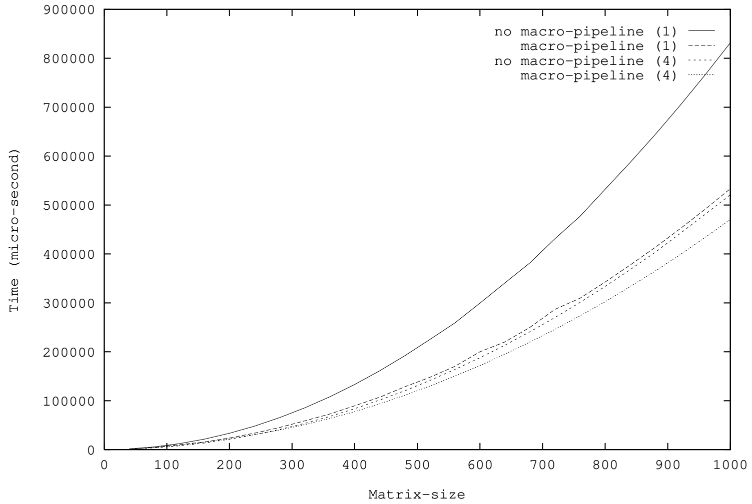

FIG. 4.7 - Macro-pipeline avec 1 et 4 liens avec taille de paquets optimale.

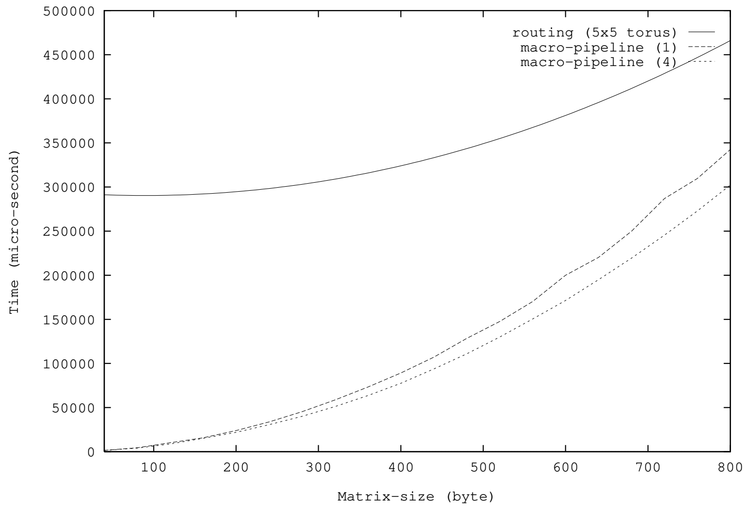

FIG. 4.8 - Comparaisons des algorithmes sur tore et réseau direct pour la transposition de matrice par blocs sur 25 nœuds.

Pour avoir une comparaison avec une topologie fixe (un tore) et avec comme souci d'avoir un temps de programmation réduit, nous avons utilisé les capacités de routage logiciel de l'environnement C-NET. Le coût de transposition donné par l'expérience est : $T_{transpo}^{fixe} = 0.23\,L^2 + 62.54\,L + 2.74\,10^5 \mu s$ où L est la taille de la sous-matrice transférée ($L = m^2$).

La Figure 4.8 montre les temps de transposition par blocs sur un réseau de 25 nœuds. Malgré le coût de reconfiguration, les algorithmes utilisant les réseaux directs sont plus intéressants.

4.2.10 Relations avec d'autres modèles

Nous avons étudié les algorithmes précédents sous un autre modèle de communication adapté aux réseaux optiques. Le modèle est celui des *Optical Passive Star Networks* [?].

D'autres modèles ont des caractéristiques proches du modèle présenté dans ce chapitre. Récemment, Bruck et Ho ont proposé une série d'algorithmes sur réseau complet k-port. Leur modèle ne possède pas la contrainte du coût de reconfiguration présente dans le notre. Les algorithmes étudiés sont la réduction [BH93b], l'échange total [BH93a] et la multi-distribution [BCH93]. Les résultats obtenus sont les mêmes que les nôtres, sans les coûts de reconfiguration. Des algorithmes hybrides sont présentés pour les cas où le nombre de processeurs n'est pas une puissance de $k+1$.

Dans [Sys92], Syska présente un algorithme de diffusion en $\log_5(P)$ sur un tore de processeurs avec un modèle wormhole 4-ports. Cet algorithme évite les contentions grâce à un astucieux découpage récursif du tore en croix.

Un autre algorithme de diffusion sur une grille a été proposé par Barret, Payne et van de Geijn dans [BPG91]. Celui-ci utilise le modèle wormhole 1-port et des expériences ont été réalisées sur la machine Intel Delta. L'algorithme s'exécute en $\log_2(P)$ et tient compte des contentions.

4.3 Les diffusions successives

Comme nous l'avons vu précédemment, la diffusion est une dissémination d'informations pour laquelle l'information présente sur un nœud d'un réseau de communication doit être transmise à tous les autres nœuds le plus rapidement possible [FL91, HHL86]. Dans ce paragraphe, nous considérons le problème pour lequel tous les nœuds d'un réseau doivent, les uns après les autres, diffuser un message distinct que nous appelons problème des *diffusions successives*. C'est un schéma de communication qui apparaît dans plusieurs implémentations d'algorithmes d'algèbre linéaire sur machines à mémoire distribuée, notamment pour les factorisations de matrices.

Il faut noter que ce problème est différent du problème du *commérage* (ou gossiping) [HHL86] pour lequel tous les nœuds doivent faire une diffusion dans n'importe quel ordre, et même simultanément. Nous présentons un algorithme qui résout le problème des diffusions successives sur un hypercube et une borne inférieure sur le temps de n'importe quel algorithme de diffusions successives. Nous montrons que notre algorithme est à un facteur 2 de l'optimalité.

4.3.1 Introduction

Beaucoup d'algorithmes, par exemple en algèbre linéaire, ont la forme suivante. La structure de données est un ensemble S de N données $\mu_1, \ldots, \mu_N$. L'algorithme s'exécute en N pas. Soit $S^{(0)} = S$, c'est-à-dire $\mu_j^{(0)} = \mu_j, j = 1, \ldots, N$. Au pas i, $1 \leq i \leq n$, les algorithmes construisent un ensemble $S^{(i)} = \{\mu_j^{(i)}, j = 1, \ldots, N\}$ par

$$\mu_j^{(i)} = F\left(\mu_j^{(i-1)}, \mu_i^{(i-1)}\right), j = 1, \ldots, N,$$

où F est une fonction de mise-à-jour caractérisant un algorithme particulier. Soit $t_{comp}(F)$ le temps arithmétique nécessaire pour appliquer la fonction F. Un tel algorithme a un coup arithmétique de $N^2 t_{comp}(F)$.

Un tel algorithme peut être implémenté sur une machine à mémoire distribuée de la manière suivante. L'ensemble S est divisé en P blocs, où P est le nombre de processeurs, de telle manière que chaque processeur possède exactement $\frac{N}{P}$ données. Cette allocation étant fixée, chaque processeur, à son tour, calcule une valeur à partir du bloc qu'il possède, et diffuse cette donnée à tous les autres processeurs. Puis, chaque processeur met à jour ses données, un autre processeur calcule la valeur suivante et la diffuse à son tour, et ainsi de suite. Ce processus se termine après N étapes commençant par une diffusion. Ainsi, si $t_{broad}(P)$ est le coût de communication nécessaire pour diffuser une seule donnée d'un processeur vers les $P-1$ autres, alors les communications et les calculs peuvent être facilement ordonnancés en un temps $N(t_{broad}(P) + \frac{N}{P}t_{comp}(F))$ où les diffusions sont exécutées en phases différentes. Le facteur d'accélération *arithmétique* est alors proche de P. Malgré tout, le coût de communication implique que l'efficacité de l'algorithme décroît au fur et à mesure que le nombre de processeurs augmente. Par exemple sur un hypercube, $t_{broad}(P) = O(\log_2(P))$, et de ce fait, le coût total de communication est $O(N \log_2(P))$.

Dans ce paragraphe, nous présentons un ordonnancement des communications qui permet aux diffusions, initialisées par différentes sources, d'être pipelinées sur un hypercube de telle manière que le coût total de communication soit $O(N + \log_2(P))$. Notre résultat est basé sur le fait que nos différentes diffusions à la suite dans un hypercube sont sans conflits et assurent la marche correcte de l'algorithme (chaque valeur est reçue au bon pas).

Nous avons choisi l'hypercube afin de pouvoir implémenter une factorisation de matrice plus efficacement en entrelaçant au mieux les diffusions nécessaires à la dissémination du pivot à chaque étape. Ce problème peut être également étudié sur d'autres topologies et d'autres modèles de communication.

4.3.2 Présentation du problème

La distribution des données dépend d'une fonction `alloc` : le processeur `alloc`(j) possède la donnée μ_j. Nous considérons l'algorithme 4.3 devant être ordonnancé sur une machine parallèle. L'algorithme est donné en terme d'instructions exécutées sur tous les processeurs (la fonction `mynode` retournant le numéro du processeur courant).

```
début
   pour i = 1 à P faire
      si mynode = alloc(i) alors x ← μ_i
      diffuser(alloc(i), x)                  (1)
      ranger(x)                              (2)
   finpour
fin
```

ALG. 4.3 - Algorithme utilisant des diffusions successives.

L'algorithme de diffusions successives est constitué de P itérations (on peut le répéter jusqu'à obtenir N itérations). Chaque itération travaillant sur une donnée différente $\mu_i, i = 1, \ldots, p$. A l'itération i, le processeur `alloc`(i), possédant μ_i, diffuse μ_i à tous les autres processeurs (instruction (1)). Quand un processeur différent de `alloc`(i) rencontre l'instruction "`diffuser`(`alloc`$(i), x$)", cela signifie qu'il attend jusqu'à réception d'une valeur x d'un de ses voisins et que, éventuellement, il participera à la diffusion en renvoyant cette valeur à certains de ses voisins en suivant le protocole choisi pour la diffusion depuis le nœud alloc(i). Ensuite, chaque processeur range cette valeur (instruction (2)).

Notre but est d'organiser les communications avec le respect de l'ordre des tâches de l'Algorithme de Diffusions Successives. Par exemple, deux valeurs μ_{i_0} et μ_{i_1} ne doivent pas arriver dans un processeur dans un ordre inverse (éventuellement, si cette situation apparaît, nous devons être capables de savoir par avance l'ordre de réception de telle manière que chaque processeur puisse ordonner correctement le rangement de ses données). Remarquons que l'organisation des communications dépend de la découverte d'une allocation adaptée pour l'élément μ_i et, cette allocation étant fixée, trouver un ordonnancement des opérations de diffusion.

Le moyen simple, mais peu efficace ($O(P \log(P))$), d'implémenter l'Algorithme de Diffusions Successives, réalise P diffusions séparées. La bande-passante globale du réseau sera certainement utilisée de manière inefficace pendant chaque diffusion (particulièrement si la taille des données est petite). De ce fait, il serait plus intéressant de recouvrir plusieurs diffusions, toujours en conservant l'ordre des diffusions de l'algorithme et en respectant les contraintes de communication.

Notations

Soit Q_d, le graphe de $P = 2^d$ nœuds numérotés de 0 à $2^d - 1$ tel qu'il y a une arête entre deux nœuds x et y si et seulement si leur numérotation binaire diffère exactement d'un bit. Pour un nœud x de représentation binaire sur d bits $x_{d-1}x_{d-2}\dots x_1x_0$, chaque indice i dénote une dimension différente. Pour deux nœuds x et y, $\delta(x,y)$ est la distance de Hamming entre x et y ($(x \oplus y)_i = x_i + y_i \pmod 2, i = 0, \dots, d-1$), et $|x|$ est le nombre de bits à 1 dans x, alors $\delta(x,y) = |x \oplus y|$ (où $\oplus$ est l'opération XOR bit à bit). Pour n'importe quelle valeur de bit x_i, $\overline{x_i}$ est son complément.

Contraintes de communications

Chaque processeur est supposé capable de travailler avec un seul message atomique à un pas donné. En particulier, un processeur ne peut pas simultanément recevoir plus d'un message. Mais, nous supposons que n'importe quel processeur peut envoyer simultanément un message à tous ses voisins. De plus, un processeur ne peut, à la fois, recevoir et émettre. Finalement, nous supposons que le coût de l'envoi d'un message atomique de n'importe quel processeur vers ses voisins est 1. Même si les algorithmes restent valables dans le cas d'un modèle k-port classique, le modèle présenté est utilisé pour le calcul de la borne inférieure.

4.3.3 Diffusions successives sur l'hypercube

Notre but est de trouver une fonction "alloc" et d'organiser les communications de telle manière que, pour chaque processeur, les rangements de l'instruction (2) seront exécutés dans un ordre correct, c'est-à-dire $\mu_1, \mu_2, \dots, \mu_P$. Une implémentation de l'Algorithme des Diffusions Successives satisfaisant cette condition sur l'ordre des rangements est dite *valide*.

Proposition 4.3.1
Il existe une implémentation valide de l'Algorithme de Diffusions Successives s'exécutant en $2P + \log_2 P - 2$ étapes sous les contraintes de communication spécifiées.

Pour prouver cette proposition, nous avons besoin de rappeler la définition d'un arbre de recouvrement binomial dans un hypercube, et ses différentes rotations [JH89].

Définition *L'*arbre de recouvrement binomial *dont la racine est x de Q_d est noté $SBT(x)$. Pour n'importe quel nœud $u = u_{d-1}\dots u_1u_0$ de Q_d, soit k la dimension satisfaisant $(u \oplus x)_k = 1$ et $(u \oplus x)_i = 0, \forall i < k$ ($k = -1$ si $u \oplus x = 0$). Soit $M_u = \{k-1, \dots, 0\}$ (éventuellement vide si $k \leq 0$).*

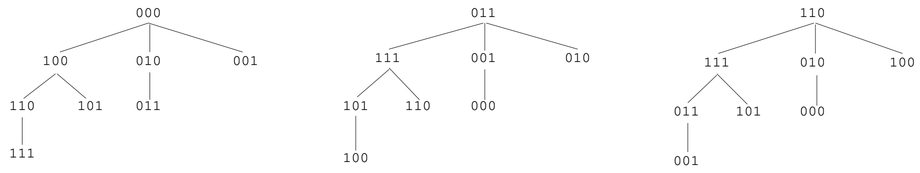

FIG. 4.9 - $SBT(000), SBT(011)$ et $R^1(SBT(011))$ dans Q_3.

Alors le père de u dans $SBT(x)$ est

$$\begin{cases} u_{d-1}\dots u_{k+1}0u_{k-1}\dots u_0 & si\ k \neq -1; \\ \emptyset & si\ k = -1; \end{cases}$$

et ses k fils sont

$$\begin{cases} u_{d-1}\dots u_{k+1}1u_{k-1}\dots \overline{u_j}\dots u_0, \forall j \in M_u & si\ k \neq -1; \\ u_{d-1}\dots \overline{u_j}\dots u_0, \forall j \in \{0,\dots,d-1\} & si\ k = -1. \end{cases}$$

Notons que $SBT(x) = x \oplus SBT(0)$ où l'opération de $\oplus$ s'applique à chaque noeud de $SBT(0)$.

Pour chaque nœud x de Q_d, $R(x)$ est sa rotation des bits à gauche, c'est-à-dire

$$R(x_{d-1}x_{d-2}\dots x_1x_0) = x_{d-2}\dots x_1x_0x_{d-1}.$$

Pour $j > 1$, la j-ème rotation à gauche est définie comme $R^j = R \circ R^{j-1}$ ($R^0 = R^d = Id$). Cet opérateur peut être appliqué sur tous les nœuds des arbres de recouvrement binomiaux puisqu'il préserve l'adjacence. Notez qu'un nœud u n'a aucun fils dans $R^j(SBT(x)), j \in \{0,\dots,d-1\}$ si et seulement si $u_j = \overline{x_j}$. Cette propriété est la clé de la preuve du Lemme 4.3.4. La Figure 4.9 montre $SBT(000), SBT(011)$ et $R^1(SBT(011))$ dans Q_3.

N'importe quel arbre $R^j(SBT(x)), j = 0,\dots,d-1$ peut être utilisé pour diffuser un message de x [JH89] : à chaque pas, le message est envoyé simultanément de chacun des nœuds d'un même niveau à tous leurs fils dans l'arbre. Les niveaux sont numérotés de 0 à d, la racine étant le seul nœud au niveau 0. Une telle diffusion prend d pas car la profondeur d'un tel arbre est d. Plus précisément, si u est à une distance $\delta(u,x)$ de la source x, u recevra la donnée après $\delta(u,x)$ pas, car ces arbres contiennent un seul plus court chemin entre n'importe quel nœud et ses fils dans l'arbre.

Nous sommes maintenant prêts à donner notre allocation et notre stratégie de diffusion.

Stratégie d'allocation

Pour pouvoir commencer les diffusions au plus tôt, il faut prendre la donnée suivante dans un processeur fils de la racine. Notre allocation utilise une séquence de Code de Gray Binaire Réfléchi $\mathrm{BRGC}(d) = \{x^{(1)}, x^{(2)}, \dots, x^{(2^d)}\}$ (voir par exemple [Joh87]). Rappelons que

- $\mathrm{BRGC}(1) = \{0, 1\}$ et
- $\mathrm{BRGC}(d) = \{0\mathrm{BRGC}(d-1), 1\overline{\mathrm{BRGC}(d-1)}\}$ où $\overline{\mathrm{BRGC}(d)}$ est $\mathrm{BRGC}(d)$ en ordre inverse.

Par exemple, $\mathrm{BRGC}(2) = \{00, 01, 11, 10\}$ et $\mathrm{BRGC}(3) = \{000, 001, 011, 010, 110, 111, 101, 100\}$. Nous définissons $\mathrm{alloc}(j) = x^{(j)}$, $1 \leq j \leq 2^d$ c'est-à-dire le j-ème élément de la séquence BRGC.

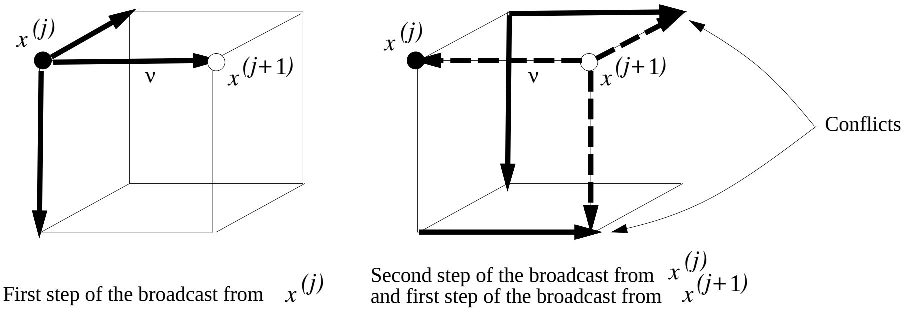

FIG. 4.10 - Un conflit apparaît quand deux SBT sont démarrés à chaque pas.

Stratégie de diffusion

Premièrement, nous décrivons l'algorithme de diffusion du processeur $x^{(j)}, j \in \{1, \ldots, 2^d\}$. Soit ν tel que $x^{(j)} \oplus x^{(j+1)} = 2^\nu$ (avec $x^{(2^d+1)} = x^{(1)}$), c'est-à-dire que $x^{(j)}$ et $x^{(j+1)}$ sont différents dans la dimension ν. Alors, $x^{(j)}$ utilise $R^\nu(SBT(x^{(j)}))$ pour diffuser μ_j. Notons que, lorsque $x^{(j)}$ exécute ses diffusions, $x^{(j+1)}$ recevra μ_j au premier pas. De plus, par construction, $x^{(j+1)}$ est une feuille dans $R^\nu(SBT(x^{(j)}))$.

Maintenant, nous devons spécifier comment ces diffusions peuvent être ordonnancées successivement. Puisque $x^{(j+1)}$ est une feuille dans $R^\nu(SBT(x^{(j)}))$, après la réception de μ_j, et supposant que toutes les valeurs précédentes ont été reçues, $x^{(j+1)}$ est prêt à commencer sa diffusion de μ_{j+1}. Si $x^{(j+1)}$ exécute le premier pas de sa diffusion en simultanéité avec le second pas de la diffusion à partir de $x^{(j)}$, un conflit aura lieu (voir par exemple la Figure 4.10). Notre ordonnancement des diffusions est donc comme suit : après réception de μ_j, le processeur $x^{(j+1)}$ attend un pas, puis commence la diffusion de μ_{j+1}. Il n'y a aucune modification des diffusions qui sont toutes exécutées niveau par niveau. Le pas auquel les différentes diffusions sont initiées est uniquement spécifié ici : tous les deux pas, une nouvelle diffusion débute.

Lemme 4.3.2 *La stratégie d'allocation et l'ordonnancement des diffusions spécifié précédemment donne une implémentation* valide *de l'Algorithme des Diffusions Successives.*

Preuve : Nous devons montrer que l'ordre de réception des séquences $\mu_j, j = 1, \ldots, P$ est correct. Supposons qu'il ne l'est pas, et soit i le plus petit entier tel que $x^{(i)}$ reçoit deux valeurs dans un mauvais ordre, c'est-à-dire qu'il reçoit $\mu_{j''}$ avant $\mu_{j'}$ pour j' et j'' satisfaisant $j'' > j'$.

Soit $t_{j'}$ et $t_{j''}$, respectivement le top d'émission de $\mu_{j'}$ par $x^{(j')}$ et le top d'émission de $\mu_{j''}$ par $x^{(j'')}$. Puisqu'il y a $j''-j'-1$ nœuds entre $x^{(j')}$ et $x^{(j'')}$ dans la séquence de Gray, $t_{j''} \geq t_{j'}+2(j''-j')$, et $\delta(x^{(j')}, x^{(j'')}) \leq j'' - j'$.

A présent, puisque $x^{(i)}$ reçoit $\mu_{j''}$ avant $\mu_{j'}$, $t_{j''} + \delta(x^{(j'')}, x^{(i)}) \leq t_{j'} + \delta(x^{(j')}, x^{(i)})$. Donc $2(j'' - j')+\delta(x^{(j'')}, x^{(i)}) \leq \delta(x^{(j')}, x^{(j'')})+\delta(x^{(j'')}, x^{(i)})$. De ce fait $\delta(x^{(j')}, x^{(j'')}) \geq 2(j''-j')$, et donc $j' = j''$, soit une contradiction.

Lemme 4.3.3 *Supposant qu'il n'y a pas de conflits, l'ordonnancement des diffusions spécifié précédemment permet d'effectuer l'Algorithme de Diffusions Successives en temps* $2P + \log_2(P) - 2$.

Preuve : Une nouvelle diffusion est initiée tous les deux pas. Après $2(P - 1)$ pas, la dernière diffusion débute et se termine en $\log_2(P)$ pas.

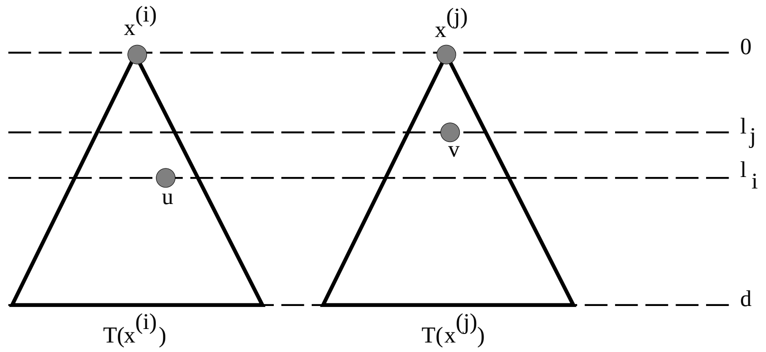

FIG. 4.11 - Nœuds actifs à deux niveaux pour deux arbres de diffusions.

Lemme 4.3.4 *L'ordonnancement des diffusions spécifié précédemment exécute l'Algorithme des Diffusions Successives sans conflit.*

Preuve: Soit $T(x^{(i)})$ et $T(x^{(j)})$ les arbres de diffusion, comme spécifié précédemment, de $x^{(i)}$ et $x^{(j)}$ respectivement, $i < j$. Soit u un nœud de Q_d au niveau l_i dans $T(x^{(i)})$, et v un autre nœud de Q_d au niveau l_j dans $T(x^{(j)})$. Supposons, qu'au temps t, u est actif dans la diffusion de $x^{(i)}$ (Figure 4.11). Remarquons que cette supposition implique que $l_j < l_i$. Est-il possible que $u = v$?

Il y a deux cas:

1. Au temps t, u et v *envoient* respectivement μ_i et μ_j. Dans ce cas, $l_i - l_j = 2(j-i)$. D'un autre coté, $x^{(i)}$ et $x^{(j)}$ diffèrent d'au moins $j-i$ bits. Maintenant, par construction des arbres de diffusion,

$$\begin{aligned} u &= x^{(i)} \oplus \hat{u} \quad \text{où } \hat{u} \in R^r(SBT(0)) \text{ au niveau } l_i, \\ \text{et} \quad v &= x^{(j)} \oplus \hat{v} \quad \text{où } \hat{v} \in R^s(SBT(0)) \text{ au niveau } l_j, \end{aligned}$$

   pour les entiers r et s. De ce fait

$$u = v \Rightarrow \hat{u} \oplus \hat{v} = x^{(i)} \oplus x^{(j)}.$$

   De nouveau par construction des arbres, $|\hat{u} \oplus \hat{v}| \geq l_i - l_j = 2(j-i)$ et $|x^{(i)} \oplus x^{(j)}| \leq j - i$. Donc $u \neq v$, sinon $i = j$.

   Le cas où, au temps t, u et v *reçoivent* respectivement μ_i et μ_j, peut être traité de la même façon.

2. Au temps t, u *envoie* μ_i et v *reçoit* μ_j. Dans ce cas, $l_i - l_j = 2(j-i) - 1$. Donc, si $j - i > 1$, un argument similaire au cas 1 montre que $u \neq v$.

   Supposons que $j = i+1$ et notons ν la dimension dans laquelle $x^{(i)}$ et $x^{(i+1)}$ diffèrent. Le nœud u est au niveau l_i dans $T(x^{(i)})$ et v est au niveau $l_i - 1$ dans $T(x^{(i+1)})$. Donc, si $\underline{u = v}$ alors $u \oplus x^{(i)}$ a exactement l_i bits 1, et $u \oplus x^{(i+1)}$ a exactement $l_i - 1$ bits 1. De ce fait, $u_\nu = \overline{(x^{(i)})_\nu}$, et u est une feuille dans $T(x^{(i)}) = x^{(i)} \oplus R^\nu(SBT(0))$: une contradiction avec le fait que u *envoie* μ_i. De ce fait, $u \neq v$.

Preuve de la Proposition 4.3.1: suit directement les Lemmes 1, 2 and 3.

Borne inférieure Une borne inférieure du problème des Diffusions Successives est donnée par :

Proposition 4.3.5 *Soit $\alpha(d)$ le maximum, sur tous les arbres de recouvrement T de Q_d, du nombre de feuilles de T. N'importe quelle implémentation de l'Algorithme des diffusions successives sans conflits s'exécute en un temps au moins égal à $2P - \alpha(d) - 1$ où $P = 2^d$.*

Preuve : Dans n'importe quelle implémentation de l'Algorithme des diffusions successives sans conflit, chaque processeur doit recevoir $P - 1$ messages, et envoyer son propre message. De plus, soit $T(x)$ l'arbre de diffusion du processeur $x \in \{0, \ldots, P - 1\}$. Le nombre de feuilles de $T(x)$ est plus petit que $\alpha(d)$. De ce fait, durant la diffusion de x, au moins $P - \alpha(d) - 1$ nœuds ont à renvoyer le message après réception. Donc, pendant l'implémentation complète de l'Algorithme, il y a au moins $P(P - \alpha(d) - 1)$ opérations de renvoi. De ce fait, il existe un nœud x_0 qui renvoie au moins $P - \alpha(d) - 1$ messages. Donc, n'importe quelle implémentation de l'Algorithme des Diffusions Successives nécessite un temps au moins égal à $(P - 1) + 1 + (P - \alpha(d) - 1)$.

Notons que $\alpha(d) \geq \frac{P}{2}$. Cette borne est atteinte par l'Arbre de recouvrement Binomial. Malgré tout, pour $d \geq 4$, il est possible de faire mieux "à la main" : $\alpha(4) \geq 10$. De ce fait, il est peut être possible d'exécuter l'algorithme plus rapidement que $\frac{3}{2}P + o(P)$. D'un autre coté, il est impossible de l'exécuter en $P + o(P)$.

4.3.4 Simulations

Nous avons effectué des simulations de différents algorithmes des diffusions successives. La Figure 4.12 présente une simulation de l'algorithme sur un 4-cube consistant à démarrer une diffusion à la fin de la précédente. La Figure 4.13 présente une méthode utilisant les arbres de recouvrements binomiaux sans rotations avec des décalages des tops de départ adaptés à une exécution sans conflit. Enfin la Figure 4.14 présente la simulation des diffusions successives à partir des arbres rotatifs. Toutes ces simulations utilisent un programme séquentiel générant des traces lisibles par le logiciel Paragraph. Remarquons que les échelles sont les mêmes pour tous les diagrammes. Le gain obtenu grâce à nos schémas apparaît ici de manière évidente.

4.3.5 Conclusion

Nous avons proposé un schéma de diffusions successives efficace permettant d'avoir un gain important par rapport à une méthode utilisant des arbres binomiaux "classiques". Nous sommes proches de la borne inférieure qui est très serrée et inatteignable. Si le modèle est un peu particulier, le schéma de communication reste valable pour un modèle k-port.

En plus de ses applications en algèbre linéaire parallèle, le problème des diffusions successives est intéressant par lui-même. En particulier, il serait intéressant d'étudier ce problème sur d'autres topologies et sous d'autres contraintes comme spécifiées dans [FL91].

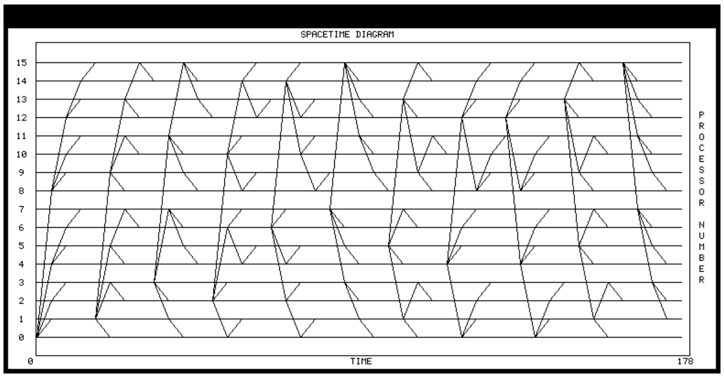

FIG. 4.12 - Utilisation des arbres binomiaux classiques et départ d'une diffusion à la fin de la précédente.

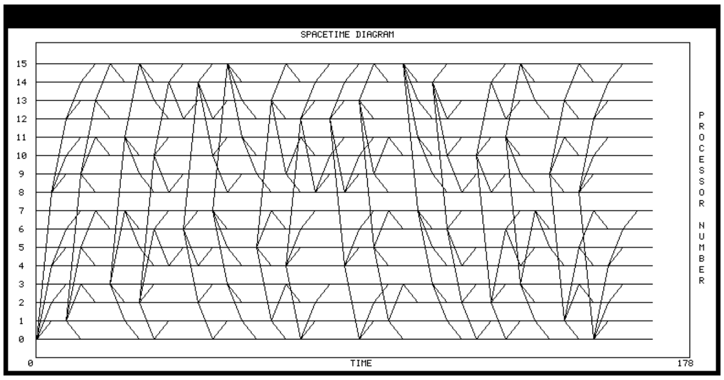

FIG. 4.13 - Utilisation des arbres binomiaux classiques et départ d'une diffusion selon le schéma 2-3-2-3.

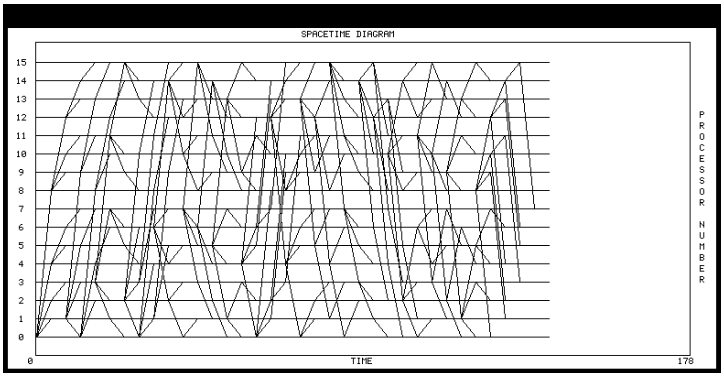

FIG. 4.14 - Mélange de différents arbres de diffusion pendant l'implémentation des Diffusions Successives sur un 4-cube (schéma proposé).

Chapitre 5

Les LOCCS

5.1 Pourquoi les LOCCS?

La manière la plus naturelle de programmer les machines parallèles est le data-parallélisme où les programmes alternent entre les phases de calcul pur et les phases de communication bloquantes. Cette approche possède les avantages d'être facile à comprendre et de donner des programmes plus faciles à prouver, déboguer et maintenir. Par contre, les performances ne sont pas forcément au rendez-vous car les communications ne sont pas recouvertes. De plus, les recouvrements calculs/communications et calculs/calculs sur des processeurs différents ne sont pas toujours réalisables à cause des dépendances existant entre les différentes tâches de calcul et de communication. Comme nous l'avons vu dans le chapitre consacré aux bibliothèques et environnements, la plupart des bibliothèques de communication disponibles sur les machines parallèles actuelles possèdent des routines de communication non-bloquantes qui permettent des recouvrements calculs/communications. Le problème du recouvrement est résolu à l'aide de méthodes de macro-pipelines difficiles à programmer, à déboguer et à maintenir. Les codes résultants, même s'ils sont efficaces, ont grossi en taille et en complexité, ce qui n'est pas acceptable en terme de lisibilité et de portabilité. De plus, la tendance qui consiste à cacher au maximum au programmeur l'utilisation du parallélisme n'est plus suivie si l'on rajoute des communications de bas niveau dans le code. Notre but est donc de proposer aux utilisateurs de machines parallèles une bibliothèque portable de routines implémentant le macro-pipeline et permettant de recouvrir au maximum calculs et communications et calculs entre eux sur des processeurs différents. Nous avons baptisé cette bibliothèque LOCCS pour **L**ow **O**verhead **C**ommunication and **C**omputation **S**ubroutines.

Après une introduction au problème posé, nous faisons un état de l'art des études générales sur le macro-pipeline et sur des bibliothèques permettant de résoudre efficacement le problème du recouvrement. Ensuite, nous présentons les routines LOCCS et un exemple de spécification. Puis, après une série d'expériences sur iPSC/860 et Paragon et un exemple de calcul de taille optimale de paquet, nous présenterons des exemples d'applications du macro-pipeline avec l'utilisation des routines LOCCS. Des méthodes de calcul général de la taille de paquet optimale sont données avant quelques conclusions.

5.2 Description du problème

Le temps d'exécution des algorithmes parallèles sans recouvrement des communications est toujours égal à : $T_{total} = T_{calculs} + T_{communications}$. Pour obtenir un maximum d'efficacité, le programmeur doit optimiser la répartition des calculs sur les processeurs, et augmenter ainsi le ratio calculs/communications pour obtenir $T_{calculs} \geq T_{communications}$. Le réseau de communication doit être suffisamment performant, mais même ainsi la majeure partie du temps est passée en attente! Si les communications sont recouverts, la situation est différente, nous avons : $T_{total} = \max(T_{calculs}, T_{communications})$. Il suffit alors de trouver l'algorithme qui permet d'avoir un recouvrement total des communications par les calculs et l'efficacité sera maximale. Même si le recouvrement n'est pas total, il y aura un gain par rapport à la méthode sans recouvrement. On a souvent dit que le principal problème des réseaux de communication était leur latence. Ceci n'est pas entièrement correct, le problème principal est l'incapacité du système à recouvrir calculs et communications. Si c'est possible, alors la latence du réseau pourra être masquée par des calculs si la communication a été initiée suffisamment tôt [ECGS92a].

Du fait de la structure même de certains algorithmes et de leur graphe de dépendance, leur implémentation sur des machines parallèles à mémoire distribuée peut parfois conduire à de mauvais équilibrages de charge et à des communications non recouvertes. Nous allons nous appliquer à

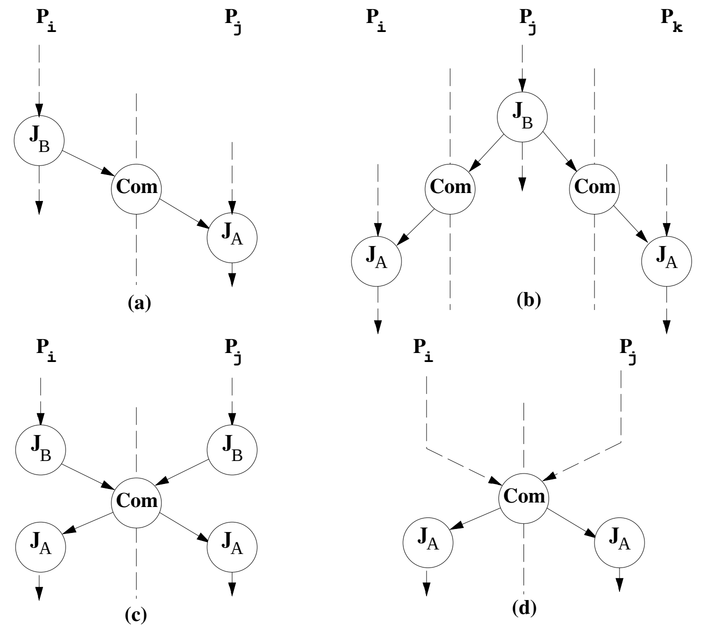

FIG. 5.1 - Différents graphes de dépendance.

réduire ce déséquilibrage et à recouvrir au maximum calculs et communications. Sur la Figure 5.1, nous présentons quelques parties de graphes de dépendance pouvant conduire à des surcoûts en terme de communications et à des mauvais équilibrages de charge entre les processeurs. Dans un programme parfaitement parallèle, tous les processus peuvent s'exécuter de manière concurrente, se communiquant des données lorsque cela est nécessaire. Nous définissons les ***opérations pipelines***, comme étant celles pour lesquelles un processeur ne peut commencer son exécution que lorsqu'il a reçu les données calculées par un autre processeur [Kin88].

C'est le cas par exemple sur la Figure 5.1 pour les différents graphes. Par exemple, en (a), un processeur P_i effectue un calcul (J_B) et envoie le résultat à un processeur P_j qui effectue, à son tour, un calcul (J_A). Nous constatons que le processeur P_i ne peut envoyer sa donnée au processeur P_j si elle n'a pas été calculée. Le processeur P_j ne peut commencer son calcul que lorsqu'il aura reçu la donnée. Nous sommes donc dans l'incapacité de recouvrir calculs et communications. En (b), le processeur P_j diffuse une donnée à deux processeurs P_i et P_k après un calcul J_B. En (c), les deux processeurs ont le même comportement et effectuent des calculs avant et après communication.

L'idée de base du macro-pipeline est, comme nous l'avons vu précédemment avec la transposition sur un réseau reconfigurable (Chapitre 4, Paragraphe 4.2.9), de découper l'espace des données en paquets de taille ν et d'envoyer les paquets dès que possible. Les communications des prochains paquets seront recouvertes par les calculs et, de plus, les processeurs destinataires pourront commencer à travailler plus tôt. Nous aurons donc un recouvrement des calculs entre les processeurs. Bien évidemment, il ne doit pas y avoir de dépendance à l'intérieur des tâches J_B et J_A qui empêcherait de découper l'espace des données.

J_B(`A, taille_de_A`)	`recevoir(`A`, taille_de_`A`)`
`envoyer(`A`, taille_de_`A`,`P_j`)`	J_A`(`A`, taille_de_`A`)`

ALG. 5.1 - Programmes des processeurs P_i et P_j.

```
jb (0, ν, sous-bloc 0 de A)                          } Initialisation
pour i = 0 à μ − 1 en // faire                       }
   envoyer(sous-bloc i de A, ν, Pj)                  } Régime normal
   jb(i + 1, ν, sous-bloc i + 1 de A)                }
finpour                                              }
envoyer(sous-bloc μ − 1 de A, ν, Pj)                 } Terminaison
fin                                                  }
```

ALG. 5.2 - Macro-pipeline sur le processeur P_i.

5.3 Exemple simple

Nous allons décrire dans ce paragraphe la résolution d'un problème possédant un graphe de dépendance de type (a) sur la Figure 5.1 à l'aide d'un macro-pipeline. Nous supposons que le processeur P_i doit effectuer une tâche J_B avant de transmettre le résultat au processeur P_j qui, à son tour, exécute une tâche J_A. L'algorithme n'utilisant pas le macro-pipeline est l'algorithme 5.1 (nous supposons que les deux processeurs travaillent sur une matrice A locale).

Afin d'exécuter au plus tôt la tâche J_A, si les dépendances internes aux deux tâches le permettent, l'espace de données sera découpé en μ paquets de taille ν et des micro-tâches j_b seront exécutées séquentiellement. La communication correspondant à une micro-tâche précédemment exécutée sera effectuée au plus tôt, ce qui permettra non seulement un recouvrement des calculs et des communications, mais également un recouvrement des calculs entre les deux processeurs. Le programme macro-pipeline du processeur P_i est donné par l'algorithme 5.2. Celui du processeur P_j est symétrique. Le code résultant est plus compliqué et peu portable et l'obtention de performances correctes nécessite une très bonne connaissance de la machine cible.

L'exécution macro-pipeline d'un tel programme sur iPSC/860 avec des travaux de tailles égales est illustrée à l'aide de Paragraph sur la Figure 5.2 où les processeurs i et j sont respectivement 0 et 1. Les deux vues *Spacetime* montrent les communications entre les processeurs. Nous constatons que, à l'aide de la routine LOCCS utilisée, le gain obtenu est un facteur environ égal à deux sur le temps total. Les diagrammes de Gantt montrent le découpage des tâches en micro-tâches, qui recouvrent les communications.

Le paramètre ν peut être modifié théoriquement et expérimentalement. Une courbe du temps d'exécution par rapport à la taille des paquets est donnée sur la Figure 5.3. Nous voyons clairement sur cette figure qu'il existe une taille de paquet optimale qui permet d'avoir le meilleur recouvrement possible. Le début de la courbe, qui correspond aux petites tailles de paquets, montre que si le nombre de paquets est trop important, le temps d'exécution peut être plus grand que celui de l'algorithme sans macro-pipeline, et ceci à cause du trop grand nombre de latences. Ensuite, pour les grandes tailles de paquets, le recouvrement n'est que partiel et le gain moins important. La

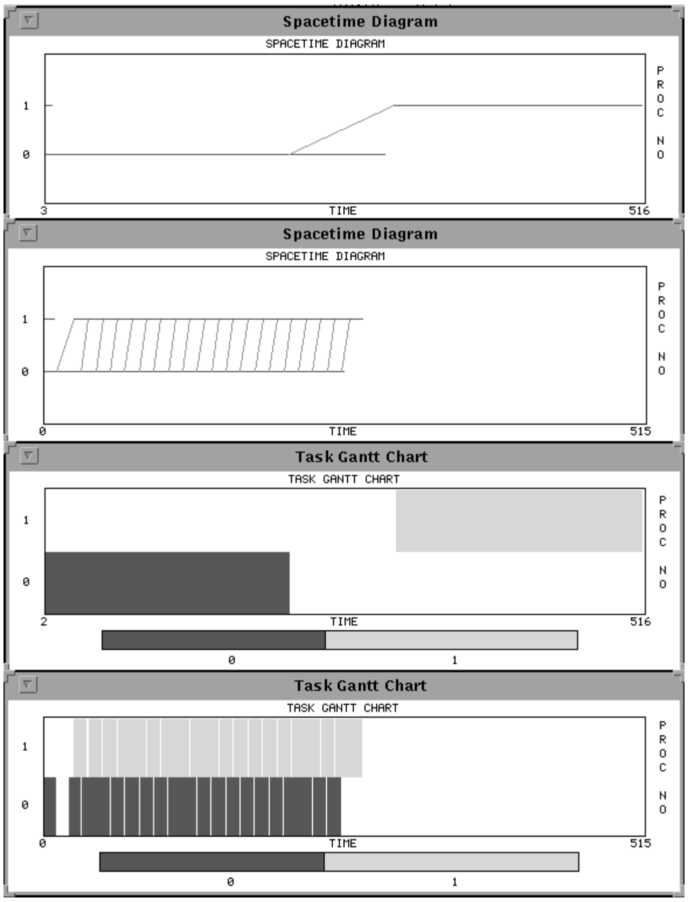

FIG. 5.2 - Exécution d'un macro-pipeline classique sur iPSC/860.

taille de paquet théorique correspondant à cet exemple sera calculée de manière précise dans un prochain paragraphe (Paragraphe 5.5.3).

Nous verrons dans les études théoriques et expérimentales que les bonnes propriétés des machines parallèles pour une implémentation des LOCCS sont des routines de communications non-bloquantes (locales et globales) et des temps de latence faibles.

5.4 Travaux précédents

Différentes approches ont été utilisées qui donnent des solutions différentes au problème crucial du recouvrement des calculs et des communications. Nous ne parlerons pas des études spécifiques pour des problèmes bien particuliers mais des systèmes ou des bibliothèques utilisables par les programmeurs de machines parallèles. Dans ce paragraphe, nous passons en revue les projets de recherche "concurrents" en soulignant leurs qualités et leurs défauts.

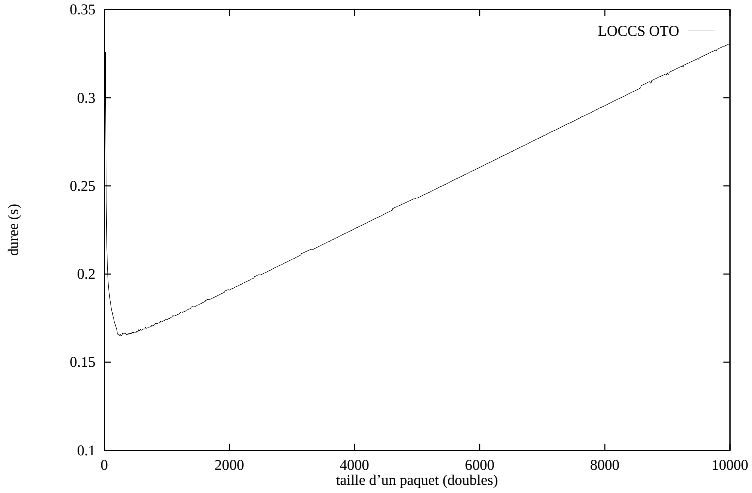

FIG. 5.3 - Temps d'exécution en fonction de la taille de paquet sur iPSC/860.

5.4.1 Les Messages Actifs

Les Messages Actifs (Active Messages) ont été développés à Berkeley par von Eicken, Culler, Goldstein et Schauser [ECGS92b, ECGS92a]. L'idée principale est de reprendre les avantages des machines *message-driven*. Un système logiciel est réalisé qui permet de mettre dans l'entête du message l'adresse d'une séquence d'instructions, définie par l'utilisateur, qui sera extraite du réseau et intégrée dans les calculs en cours avec comme argument le corps du message lui-même. Ceci est réalisé sur des machines parallèles à mémoire distribuée traditionnelles. Ce passage du réseau aux calculs doit s'effectuer le plus rapidement possible. Sous les Messages Actifs, le réseau de communication est vu comme un pipeline opérant à une vitesse déterminée par le surcoût des communications et par une latence reliée à la taille du message et à la longueur du pipeline. Un émetteur peut initier plusieurs messages à la suite pour garder le pipeline rempli et continuer ses calculs. Les calculs reliés aux messages seront effectués dès réception. Les Messages Actifs ne sont pas bufferisés afin de garder le surcoût des communications à son minimum. En mélangeant calcul et communication dans une même entité, leur recouvrement sera aisé et l'efficacité augmentée. Les Messages Actifs ont été utilisés pour augmenter les performances du langage data-parallèle SPLIT-C. Quand un processeur a besoin d'une donnée, il effectue un `GET` suivi d'un calcul sur la donnée reçue. S'il doit envoyer une donnée à un autre processeur, il effectue alors un `PUT` suivi d'un calcul à la réception. Grâce aux Messages Actifs, ceci peut être effectué de manière asynchrone en recouvrant les calculs [ECGS92b]. La principale différence entre les Messages Actifs et les architectures "message-driven" est que pour les premiers, les tâches dont les adresses sont intégrées aux messages sont exécutées immédiatement à la réception et ne peuvent pas être interrompues alors que pour les secondes, le processeur de calcul récupère des tâches à effectuer d'une file créée avec les messages reçus et peut être interrompu. Dans ces dernières architectures, les messages sont bufferisés et le surcoût est, de ce fait, plus important. Le système des messages actifs est notamment disponible sur la CM-5 [EK92] et la nCUBE/2 [ECGS92a].

Avantages

- Bon recouvrement des calculs et des communications,
- intégration dans SPLIT-C,
- implémentable efficacement sur des machines classiques,
- peut être utilisé pour implémenter les LOCCS.

Inconvénients

- Pas général (pas de travail avant envoi),
- nécessité au programmeur d'écrire le pipeline "à la main",
- pas d'opérations de communications globales.

5.4.2 Les multi-threads

Ces études, effectuées à l'Université de Washington par E.W. Felten et D.Mc Namee, ont eu pour objet de recouvrir les calculs et les communications en utilisant une technique issue de la programmation des machines parallèles à mémoire partagée appelée *multithreading* [FN92]. Cette technique, basée sur un système appelé NewThreads, consiste à diviser le programme parallèle en différents flots, et à passer d'un flot à l'autre à chaque communication, en recouvrant les communications d'un flot par les calculs d'un autre. Le système offre à l'utilisateur des routines de gestion des flots (création, destruction, synchronisation) et permet de communiquer entre les flots à l'aide de primitives de communication classiques, bloquantes pour un flot donné, mais qui peuvent être recouvertes par les calculs d'un autre flot. Des tests ont été effectués avec un programme parallèle de résolution de systèmes d'équations aux dérivées partielles [FN92]. Il s'agit d'un problème nécessitant des échanges entre voisins et des calculs intérieurs et aux bords. La solution consiste à recouvrir les échanges de données sur les bords avec les calculs intérieurs. Ces expériences ont montré un léger gain pour des matrices de grande taille en utilisant deux flots. Par contre, pour les matrices de petite taille, le temps du programme utilisant les flots est supérieur au temps du programme sans recouvrement. De plus, des expériences avec un programme écrit spécialement et utilisant des recouvrements très fins et particuliers à l'algorithme ont montré un gain avoisinant les 50%. NewThreads est écrit en C++ [FN92]. Ce système est disponible pour l'Intel iPSC/2, des réseaux de stations de travail et des machines parallèles à mémoire partagée.

Avantages

- Programmation simple proche du séquentiel et du modèle SPMD,
- recouvrement des calculs et des communications.

Inconvénients

- Lourd en terme de système (recopies inutiles, création de processus),

– parallélisme non caché,

– nécessité au programmeur d'écrire le pipeline "à la main",

– pas d'opérations de communications globales.

5.4.3 Les travaux de King

Dans sa thèse [Kin88] et dans ses articles en collaboration avec L.N. Ni et W.H. Chu [KCN88, KN88], C.T. King a décrit une méthode d'analyse des algorithmes macro-pipelines. Remarquons qu'il s'agit d'une étude théorique des performances de ces algorithmes et d'expérimentations sur un produit de matrices et pas une bibliothèque générale. Les temps d'exécution ont été modélisés à l'aide d'une variante des réseaux de Pétri. Les temps d'exécution pour un grain fixé sont ainsi calculés. Cette étude pourrait être adaptée à nos problèmes pour le calcul du temps d'exécution. Malheureusement, King ne donne pas la "formule magique" de la taille de paquet optimale.

Avantages

– Etude précise du temps d'exécution à l'aide de réseaux de Pétri,

– étude d'une application (produit de matrices),

Inconvénients

– Ce n'est pas une bibliothèque,

– absence de calcul de la taille optimale de paquet.

5.4.4 Optimisation des communications dans la compilation de Fortran_D

Fortran_D est la réalisation d'un prototype de compilateur pour le langage data-parallèle pour machines à mémoire distribuée HPF. Nous nous attachons dans ce paragraphe à décrire les optimisations effectuées dans la réalisation du compilateur Fortran_D sur iPSC/860 décrites dans la thèse de Tseng [Tse93] (Chapitres 5 et 6). Les problèmes rencontrés dans la réalisation du compilateur sont très proches de ceux qui nous concernent et la résolution est similaire en de nombreux points aux études décrites dans ce chapitre.

Les optimisations des communications dans Fortran_D sont assez nombreuses et concernent principalement la réduction du nombre d'initialisations par vectorisation des messages et le recouvrement des calculs et des communications grâce à la technique du macro-pipeline. En ce qui concerne la réduction du surcoût des communications, la vectorisation des messages est utilisée pour réduire le nombre de messages envoyés. Ensuite, des messages différents à envoyer à un même processeur peuvent être mis dans un même buffer et envoyés en un bloc. Des communications globales peuvent être également extraites du code pour une plus grande efficacité. La deuxième classe d'optimisations concerne la réduction du surcoût grâce aux recouvrements calculs/communications. Il s'agit de déplacer les échanges de messages afin qu'ils soient recouverts par des calculs sur d'autres données. Enfin, des optimisations, concernant le problème évoqué dans ce chapitre, sont effectuées sur les codes parallélisés par Fortran_D. Il s'agit alors de pipeline à grain fin et à gros grain. Le pipeline à grain fin permet d'envoyer les données non-locales dès que possible, c'est-à-dire, dès qu'elles ont été calculées. Cette méthode permet aux processeurs attendant les données de commencer dès que possible. Le nombre d'initialisations est alors trop important. Il est égal au nombre de données calculées. Afin de minimiser ce surcoût tout en essayant de faire démarrer le processeur en attente le plus tôt possible, une optimisation consiste à augmenter le grain des données envoyées en rajoutant un pas de boucle égal à la taille du paquet. Pour arriver à cette optimisation, Fortran_D, partant du programme original, passe par une étape intermédiaire utilisant le pipeline à grain fin pour arriver, grâce à la vectorisation de message, à un pipeline à gros grain. La taille du grain est déterminée à l'aide du ratio calculs/communications. Une analyse de complexité sommaire est effectuée pour des cas simples de calculs en n^2 et n^3 mais avec un coût de calcul constant par élément. Des études plus précises sont prévues pour des versions ultérieures du compilateur.

Avantages

- Etude en relation avec l'analyse des dépendances à la compilation,
- étude précise des optimisations possibles en fonction des différents types de dépendances,
- génération par le compilateur du code qui recouvre les communications.

Inconvénients

- Calcul de la taille de paquet très approximative (pour cette version du compilateur).

Remarquons qu'un tel compilateur pourrait faire appel aux routines décrites dans ce chapitre au lieu d'écrire le code du macro-pipeline dans le langage intermédiaire. Cela permet d'obtenir plus rapidement un compilateur efficace pour une machine donnée.

5.5 Les routines LOCCS

Dans ce paragraphe, nous présentons les différentes routines créées et détaillons la spécification de l'une d'entre elles. Nous présentons également leur implémentation sur l'iPSC/860 et la Paragon d'Intel. Une spécification plus précise des autres routines déjà implémentées et des routines d'initialisation associées est donnée dans l'annexe A.

5.5.1 Présentation des différentes routines

Les routines spécifiées sont au nombre de 8. Ces routines concernent une bonne partie des schémas de communication classiques. La routine de communication utilisée par paquet est une routine de communication globale, ce qui permet une parfaite portabilité des codes sur différentes machines. En effet, si la routine de communication ne s'effectue pas pour chaque paquet mais globalement pour le message, l'utilisateur doit connaître l'algorithme utilisé pour écrire le code des travaux effectués sur les paquets reçus. Il est inconcevable de réaliser une routine LOCCS par algorithme de communication. Le tableau 5.1 donne la liste des routines et le schéma de communication associé.

Avec la routine `loccs_shift`, nous proposons une généralisation des schémas de communication de type shift. La Figure 5.4 donne la forme d'un shift sur un processeur. Celui-ci reçoit des messages de certains processeurs et doit les envoyer à d'autres processeurs. Nous avons donc un ensemble de processeurs envoyant un message de façon macro-pipeline à un processeur qui doit à son tour les renvoyer. Les deux ensembles sont liés. Des tâches doivent être effectuées sur les messages passant sur le processeur appelant la routine LOCCS. A chaque pas de l'opération macro-pipeline, une fonction peut être appliquée à l'un des tableaux contenant les numéros des processeurs invoqués.

Routine	Type de communication
`loccs_oto`	Un vers un
`loccs_exchange`	Echange entre deux processeurs
`loccs_shift`	Shift entre plusieurs processeurs
`loccs_ota`	Diffusion
`loccs_pota`	Diffusion personnalisée
`loccs_ato`	Rassemblement
`loccs_ata`	Echange total
`loccs_pata`	Multidistribution

TAB. 5.1 - Liste des routines LOCCS

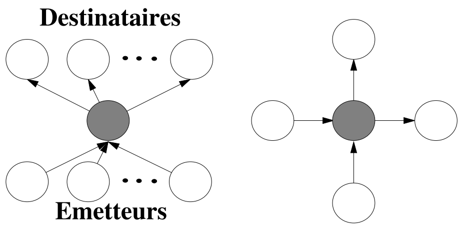

FIG. 5.4 - Deux schémas de communication de type shift.

Par exemple, cette fonction peut être celle qui consiste à parcourir toutes les dimensions d'un hypercube dans le cas d'un schéma de communication de type "papillon" comme pour la résolution de la FFT parallèle (Chapitre 7).

5.5.2 Spécification de la routine `loccs_oto`

Etant donné deux processeurs, adjacents ou pas, la routine `loccs_oto` envoie un message d'un processeur vers un autre. Le processeur émetteur (destinataire) doit effectuer une tâche avant envoi (réception) job_b et une après job_a. Cela correspond directement aux schémas (a) ou (b) de la Figure 5.1.

Séquence d'appel

Un appel à la routine `loccs_oto` est réalisé par :

- `loccs_oto(me, oto, buffer,` ν`, size,` job_b`,` job_a`)`
```
int         me;
sOto        oto;
sBufferOto  buffer;
int         ν, size;
sJob        job_b, job_a;
```

Paramètres

Les différentes structures utilisées par les routines LOCCS pour définir l'environnement et les buffers utilisés sont décrites en annexe. Nous avons:

`me`:	numéro logique de l'appelant de `loccs_oto`

`oto`:	`typedef struct {`
	`int sender;     /* Numéro logique de l'envoyeur */`
	`int receiver;   /* Numéro logique du receveur */`
	`} sOto;`

`buffer`:	Buffers pour les messages
ν:	Taille d'un paquet
`size`:	Taille totale du message
job_b, job_a:	Tâches devant être faites sur un paquet avant et après une étape de communication

Remarques concernant l'implémentation

Suivant les machines cibles, différentes stratégies d'acheminement du message pourront être utilisées comme l'utilisation de chemins disjoints et découpage du message supplémentaire [SS89].

5.5.3 Implémentation

Un des intérêts de cette bibliothèque est sa portabilité. Nous avons donc choisi d'implémenter des sous-ensembles des LOCCS sur différentes plateformes et avec différents environnements de programmation. La première version des LOCCS est écrite en C et une version C++ sera développée pour l'utilisation avec ScaLAPACK++. Des études sont également en cours pour permettre l'appel des routines LOCCS depuis Fortran. Le tableau 5.2 donne la liste des routines implémentées sur les différentes plateformes au 1/11/93. Pour la Paragon, il sera possible d'implémenter d'autres routines grâce au deuxième processeur dédié aux communications qui pourrait réaliser des schémas de communication globaux non-bloquants. Ceci sera implémenté dès que le système permettra de lancer deux programmes différents sur chacun des deux nœuds de la machine [Haw93].

Plateforme	**Routines disponibles**	**Logiciel utilisé**	**Routines implémentables**
Intel iPSC/860	`loccs_oto`, `loccs_ota`, `loccs_exchange`, `loccs_shift`	NX/PICL	Aucune autre
Intel Paragon	`loccs_oto`, `loccs_ota` `loccs_exchange`, `loccs_shift`	NX	toutes
ARCHIPEL Volvox	Aucune	Volcom	toutes
TMC CM-5	Aucune	Messages actifs	toutes
Réseaux de stations	`loccs_oto`	PVM 3.*	toutes
	Aucune	Trollius LAM	toutes

TAB. 5.2 - Différentes implémentations des LOCCS et faisabilité des autres.

Nous allons maintenant décrire quelques expériences réalisées sur iPSC/860 et Paragon qui donnent une idée des exécutions de deux routines LOCCS sur ces machines et du calcul de la

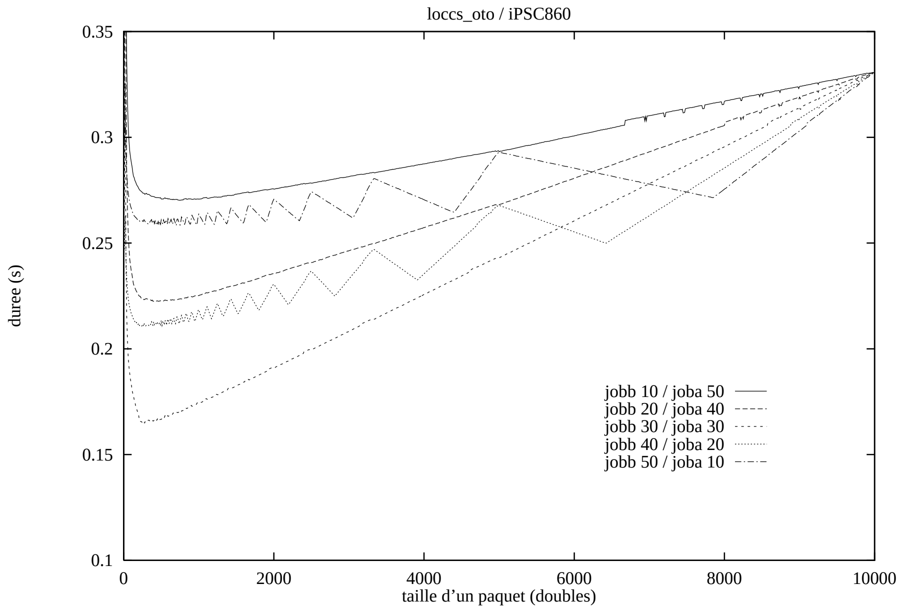

FIG. 5.5 - Différentes exécutions de `loccs_oto` sur iPSC/860.

taille optimale du paquet. Les deux routines testées sont `loccs_oto` et `loccs_ota`. Les tâches sont constituées de boucles dont nous avons fait varier la taille pour apprécier l'importance des différents paramètres de la machine et de l'algorithme. Ceci nous permettra de prévoir le comportement du programme utilisant les LOCCS et de calculer une taille de paquet la plus proche possible de la taille optimale. Sur la Figure 5.5, nous avons fait varier le rapport entre les tâches job_b (Job_{before}) et job_a (Job_{after}) pour la routine `loccs_oto` entre deux nœuds de l'iPSC/860. La taille totale des tâches est de 60 unités, une unité correspondant à une addition sur chaque élément du paquet courant. Nous avons exécuté la routine pour les rapports job_b/job_a égaux à 10/50, 20/40, 30/30, 40/20 et enfin 50/10 et pour différentes tailles de paquets ν. Ceci nous donne cinq courbes de temps différentes. Les dents de scie correspondent à des exécutions avec les tailles des derniers paquets différentes de la taille des autres paquets. Nous constatons que l'exécution la meilleure concerne le cas où les tâches sont parfaitement équilibrées. D'autre part, la taille optimale des paquets se situe toujours environ dans la même fourchette ([200..500] pour un message initial de taille 10000).

La Figure 5.6 présente deux séries d'expériences avec la routine `loccs_oto` sur iPSC/860 et Paragon pour des tailles différentes de paquets et des rapports job_b et job_a différents. Les performances des communications de la Paragon sont bien meilleures, ce qui explique les meilleures performances pour des rapports égaux entre les tâches. La pente de la courbe est également moins importante. Nous constatons toutefois les mêmes comportements. La forme des courbes Paragon est moins précise et cela est dû aux appels systèmes et à la gestion multitâches qui n'existent pas sur l'iPSC/860. Dans le cas où du rapport 10/20 sur la Paragon, nous constatons une partie "plate" sur la courbe. Ce comportement est pour l'instant inexpliqué et donne un minimum pratiquement constant entre des tailles de paquets variant de 500 à 2500 doubles. Le code étant exactement le même que celui exécuté sur l'iPSC/860, nous sommes tentés d'attribuer ce comportement au système.

La Figure 5.7 concerne différentes exécutions de la routine `loccs_ota` sur iPSC/860 avec quatre

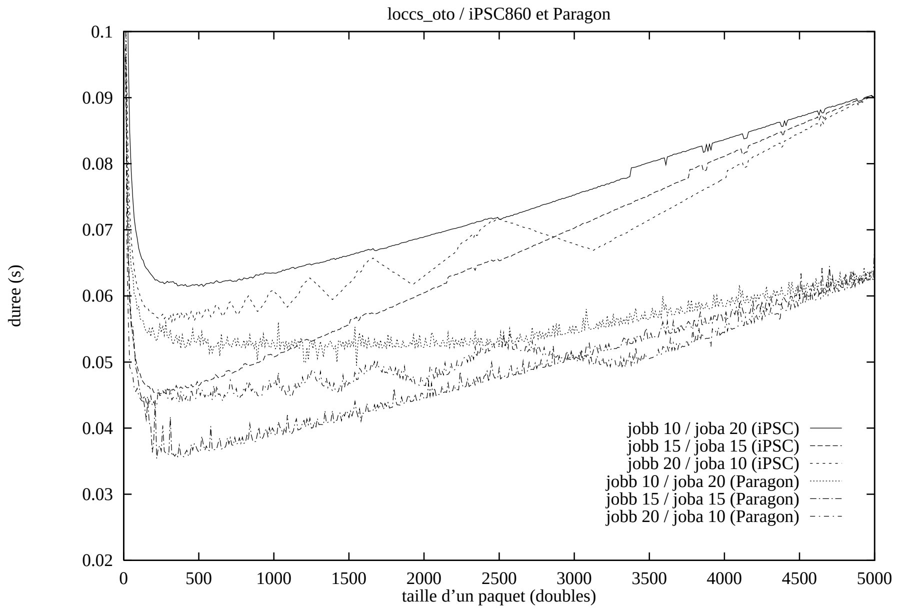

FIG. 5.6 - Différentes exécutions de la routine `loccs_oto` sur iPSC/860 et Paragon.

processeurs. Nous constatons exactement les mêmes phénomènes que pour l'exécution de la routine `loccs_oto`.

5.6 Analyse d'une exécution

Dans ce paragraphe, nous allons analyser précisément un exemple d'exécution de la routine `loccs_oto` sur iPSC/860 et calculer la taille de paquet optimale de manière théorique.

Nous pouvons expliquer plus précisément les dents de scie grâce à la courbe 5.8 et aux diagrammes de Paragraph `Spacetime` de la Figure 5.9. Ce phénomène se répète à chaque fois qu'une taille de paquets divise la taille totale du message. Nous avons effectué des traçages avec PICL et Paragraph pour certains points particuliers de la courbe 5.8 correspondant au rapport 20/10, soit un coût du travail avant la communication plus long que celui du travail après (coupes de 1 à 5). La coupe 1 est positionnée sur un pic (taille de paquet $\nu = 2520$). Ceci correspond au cas où la taille du dernier paquet est la même que celle des autres. La coupe 2 correspond à une partie en descente ($\nu = 3000$), la coupe 3, à un creux ($\nu = 3160$), la coupe 4, à une montée ($\nu = 4040$) et enfin la coupe 5 correspond au cas où le message original n'est pas découpé ($\nu = L = 5040$). Nous constatons très nettement sur ces diagrammes que les pics et les creux sont dûs aux différences entre la taille du dernier paquet et celle des autres. Un élément qui influe également sur ces dents de scie est la différence de coût pour la communication du premier paquet. Ceci est dû à la gestion des buffers sur l'iPSC/860. Ce phénomène est diminué si le nombre de paquets est grand.

Nous allons à présent analyser cette courbe de manière théorique pour retrouver les comportements observés. Dans un premier temps, nous allons étudier le phénomène de dents de scie et donner une prédiction du temps d'exécution. Puis nous calculerons le nombre de paquets optimal. Remarquons que les travaux avant et après ont des coûts linéaires en L.

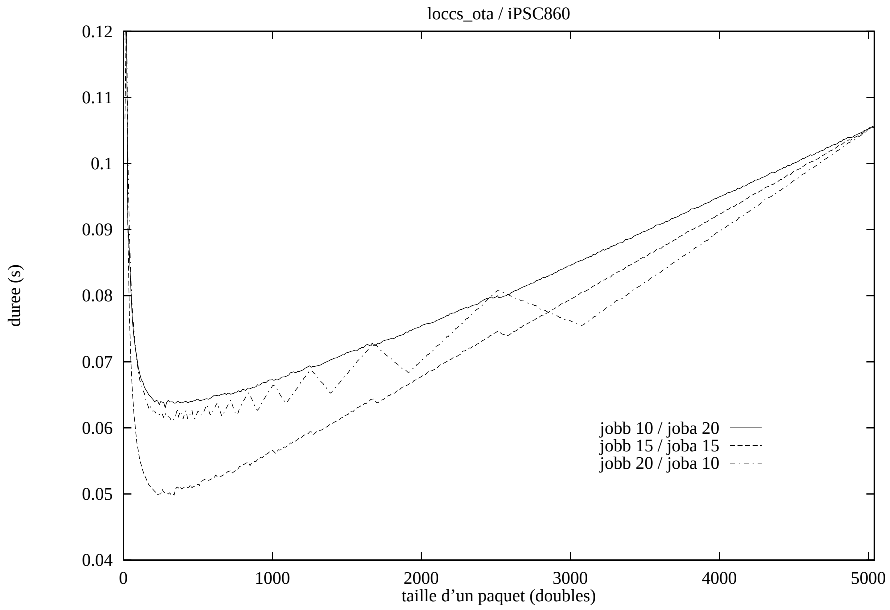

FIG. 5.7 - Exécution de `loccs_ota` sur iPSC/860 avec quatre processeurs.

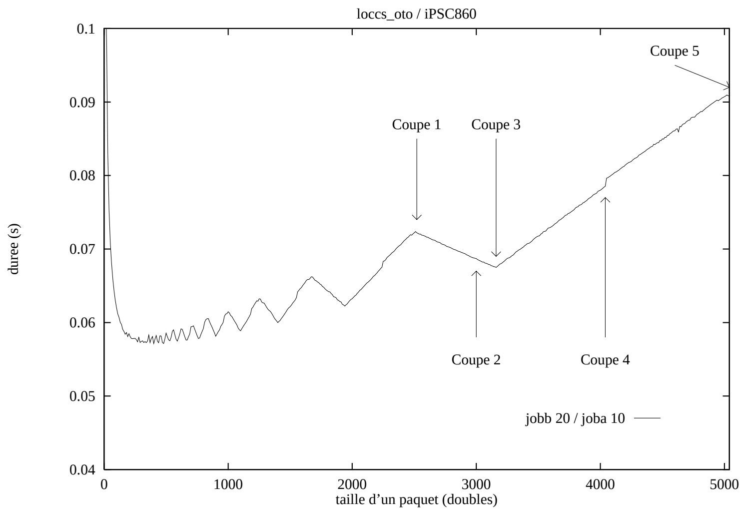

FIG. 5.8 - Une exécution de `loccs_oto` sur iPSC/860 (rapport 20/10).

5.6.1 Analyse des dents de scie

Nous supposons que nous avons μ paquets et un message de taille L. Un calcul élémentaire est modélisé avec τ_a, les communications par $\beta + L\tau$. Deux tailles sont utilisées : ν la taille de tous les paquets sauf le dernier et ν_{last} la taille du dernier paquet. Par la suite, nous noterons job_b, le

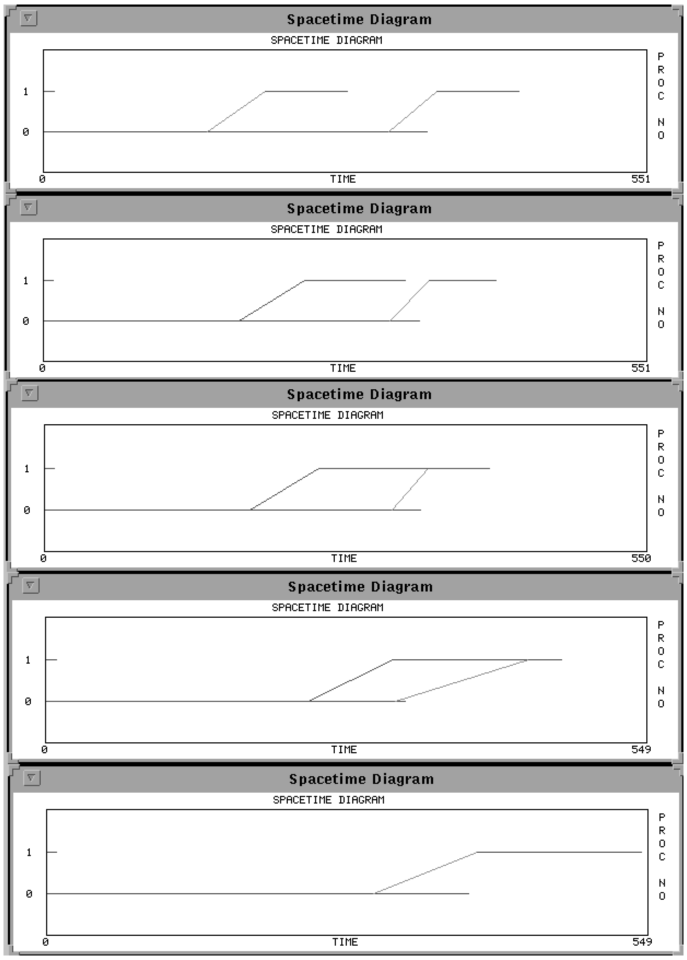

FIG. 5.9 - Diagrammes `Spacetime` correspondant aux coupes de 1 à 5.

travail effectué sur l'émetteur avant de communiquer. Son coût en fonction d'une taille de paquet ν est donné par $Tjob_b(\nu) = R_1\nu\tau_a$. Sur notre exemple $R_1 = 20$ et $\tau_a = 0.502\mu s$. job_a est le travail effectué sur le destinataire après réception du message. Son coût en fonction d'une taille de paquet ν est donné par $Tjob_a(\nu) = R_2\nu\tau_a$. Ici $R_2 = 10$. Nous avons donc $R_1 > R_2$. Ces travaux sont les travaux atomiques sur un paquet du macro-pipeline. Nous supposons également que le coût d'une communication d'un paquet de taille ν est donné par $Tcom(\nu) = \beta + \nu\tau$. Sur l'iPSC/860, nous avons $\beta = 136\mu s$ et $\tau = 0.384\mu s/o$.

Grâce au diagramme `Spacetime` donné sur la Figure 5.9, nous pouvons analyser plus facilement le problème des dents de scie. Pour les deux diagrammes supérieurs, correspondant aux coupes 1 et 2, le temps du dernier job_b est plus important que la somme du temps de l'avant dernier job_a et de la communication qui le précède. Sous l'hypothèse que $Tjob_a(\nu) + Tcom(\nu) < Tjob_b(\nu_{last})$, le chemin critique est donc constitué de $\mu - 1$ job_b sur des paquets de taille ν, du dernier job_b, d'une communication du dernier paquet et du dernier job_a, soit un temps total égal à :

$$T^1_{oto} \quad = \quad (\mu - 1)Tjob_b(\nu) + Tjob_b(\nu_{last}) + Tcom(\nu_{last}) + Tjob_a(\nu_{last})$$

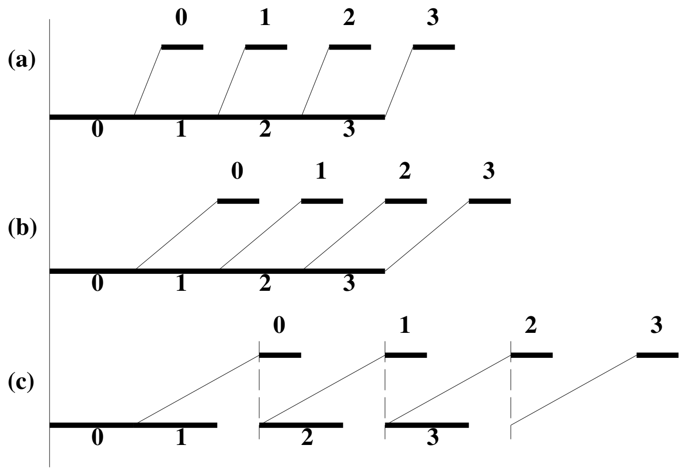

FIG. 5.10 - Trois diagrammes de Gantt suivant différents coûts de communication.

$$= \quad (R_1 L + R_2 \nu_{last})\tau_a + \beta + \nu_{last}\tau \tag{5.1}$$

Pour la coupe 3 (diagramme Spacetime central), les temps sont équilibrés entre le travail avant et la somme du travail après et de la communication. Le temps total est donc toujours égal à T^1_{oto}. Par contre, pour la coupe 4, le temps du dernier job_b est inférieur au temps de l'avant dernier job_a plus la communication précédente. Le chemin critique change donc pour les derniers termes, nous avons alors un temps total égal à :

$$\begin{aligned} T^2_{oto} &= (\mu - 1)Tjob_b(\nu) + Tcom(\nu) + Tjob_a(\nu) + Tjob_a(\nu_{last}) \\ &= ((\mu - 1)R_1\nu + R_2(\nu + \nu_{last}))\tau_a + \beta + \nu\tau \end{aligned} \tag{5.2}$$

Nous constatons que si ν augmente, ν_{last} diminue et le temps total T^1_{oto} décroît (pente négative sur la courbe) jusqu'à atteindre l'équilibre (coupe 2) et ensuite le temps T^2_{oto} augmente. Le temps de la coupe 2 est obtenu quand $T^1_{oto} = T^2_{oto}$. Avec les paramètres de notre machine cible, nous trouvons que cet équilibre est réalisé en théorie pour une taille de paquet de $\nu = 3129$ octets (3160 en pratique). Le phénomène se répète au gré des différentes tailles de paquets. Il sera moins visible si le nombre de paquets est très grand.

5.6.2 Calcul de la taille de paquet optimale

Nous allons à présent déterminer la taille de paquet optimale pour ce problème. Nous savons déjà que $Tjob_b(\nu) > Tjob_a(\nu)$. Nous pouvons donc avoir trois cas suivant le coût de communication. Ces trois cas sont résumés sur la Figure 5.10 avec trois diagrammes de Gantt. Nous supposons que tous les paquets ont la même taille.

Soit le coût de communication est inférieur au coût du job_b ((a) sur la Figure 5.10), soit les deux coûts sont égaux ((b) sur la Figure 5.10) et enfin le coût de communication peut être supérieur au coût de calcul ((c) sur la Figure 5.10). Nous allons détailler les deux cas extrêmes, le second donnant la même complexité que le premier. Le temps total peut être donné par

$$T_{oto} = Tjob_b(\nu) + (\mu - 1)\max(Tjob_b(\nu), Tcom(\nu)) + Tcom(\nu) + Tjob_a(\nu) \quad (5.3)$$

Le temps minimum sera obtenu si les calculs et les communications sont parfaitement équilibrés (cas (b) sur la Figure 5.10). Si nous suppposons que les communications peuvent être de coût supérieur à celui des calculs, nous aurons alors ($\nu = L/\mu$)

$$\begin{aligned} T_{oto} &= Tjob_b(\nu) + \mu Tcom(\nu) + Tjob_a(\nu) \\ &= \frac{R_1 L\tau_a}{\mu} + \mu\beta + L\tau + \frac{R_2 L\tau_a}{\mu} \end{aligned} \quad (5.4)$$

Il suffit de dériver cette fonction par rapport à μ et de trouver le nombre de paquets où elle est égale à zéro pour minimiser le temps total, soit

$$\frac{\delta T_{oto}}{\delta\mu} = \beta - \frac{L}{\mu^2}(R_1 + R_2)\tau_a.$$

Ce qui nous donne un nombre de paquets optimal égal à

$$\mu_{opt} = \sqrt{\frac{L(R_1 + R_2)\tau_a}{\beta}}$$

Il doit être par ailleurs compris entre 1 et L. Nous constatons donc que plus le temps de latence ou startup β est petit, plus il est possible de découper le message initial en petits paquets et ainsi permettre au processeur recevant de commencer son travail plus tôt. En remplaçant dans la formule du μ_{opt} les paramètres machines par ceux de l'iPSC/860 et la taille du message par $5040 * 8$ octets, nous obtenons une taille de paquet optimale $\nu_{opt} = 252$ doubles (250 en pratique). Nous pouvons donc calculer de manière très précise le nombre de paquets optimal et de ce fait leur taille.

5.7 Exemples d'utilisation

Dans ce paragraphe, nous donnons quelques exemples d'utilisation des routines LOCCS. Pour chaque exemple, après une présentation succincte du problème, nous décrivons un algorithme n'utilisant pas les recouvrements calculs/communications. Après quoi, nous présentons une amélioration de l'algorithme grâce au macro-pipeline et son écriture à l'aide de routines LOCCS. Pour l'un des exemples, nous donnons le résultat d'expériences.

5.7.1 Réduction sur un anneau

Dans ce paragraphe, nous présentons une méthode permettant d'effectuer une opération de réduction sur un réseau linéaire à l'aide de la routine `loccs_shift`.

La réduction est une opération globale qui consiste à réduire un ensemble de vecteurs élément par élément sur un processeur avec une opération que nous noterons $\bigoplus$. Des opérations classiques sont la somme, le produit, le minimum, le maximum et les opérations booléennes. On peut également envisager n'importe quelle opération pourvu qu'elle soit associative et commutative. Si P est le nombre de processeurs et j un élément sur le processeur i, alors une opération de réduction sur la racine k évalue :

$$D_{k,j} = D_{0,j} \bigoplus D_{1,j} \bigoplus \ldots \bigoplus D_{P-1,j}$$

L'élément j peut, bien entendu, être l'élément d'indice k d'un vecteur, auquel cas, l'opération est effectuée pour tous les éléments des vecteurs. Des variations de cette opération nécessitent que le résultat soit stocké sur tous les processeurs à la fin de l'exécution. Malgré tout, nous ne traiterons ici que l'opération qui consiste à effectuer une réduction sur un processeur particulier, appelé racine.

Nous supposons dans la suite que nous avons un réseau linéaire de P processeurs et ce nombre P est impair pour faciliter l'analyse. Celle-ci sera aisément adaptable à un nombre de processeurs pair. Chaque processeur possède un vecteur de taille m éléments.

Résolution sans pipeline

Un algorithme simple consiste à ce que les processeurs aux extrémités envoient leur vecteur à leur voisin qui, après réception combinent le vecteur reçu avec leur propre vecteur et envoient le résultat à leur voisin vers la racine. Et ainsi de suite jusqu'à ce que la racine reçoive les vecteurs de droite et de gauche et les combine à son tour pour obtenir le vecteur résultat. Si τ est le coût de communication par élément et τ_a le coût de l'opération de réduction entre deux éléments, le coût total de cet algorithme est donné par :

$$T_{red}^{naf} = \frac{P-1}{2}(\beta + m\tau) + \frac{P+1}{2} m\tau_a \tag{5.5}$$

Résolution avec pipeline

Comme dans toutes les méthodes pipeline décrites précédemment, le vecteur est découpé en μ paquets de taille m/μ. Nous pouvons alors procéder par vagues en faisant partir les paquets les uns à la suite des autres et en utilisant le même algorithme que celui décrit pour le vecteur entier. Nous pouvons alors obtenir une utilisation maximum des liens de communication et des processeurs. La Figure 5.11 donne quelques pas d'un exemple d'exécution pour $P = 7$ et $\mu = 5$. D'autres méthodes existent également qui utilisent des recouvrements calculs/communications [BLPG93]. La méthode décrite dans ce paragraphe, même si elle n'est pas la meilleure pour ce type de topologie, permet d'obtenir de bonnes performances avec un code très simple.

Pour analyser cet algorithme, nous découpons son exécution en deux phases principales, celle de remplissage du pipeline qui dure jusqu'à l'arrivée du premier paquet à la racine et la suite où la racine combine et reçoit les paquets. Dès que la racine combine, le pipeline est ralenti car celle-ci doit faire le double d'opérations pour combiner les vecteurs venant de droite et de gauche. Le temps total est donné par :

$$\begin{aligned} T_{red}^{pipe} &= \left(\beta + \frac{m}{\mu}\tau\right) + (P-3)\max\left(\beta + \frac{m}{\mu}\tau, \frac{m}{\mu}\tau_a\right) + \\ & (\mu - 1)\max\left(\beta + \frac{m}{\mu}\tau, 2\frac{m}{\mu}\tau_a\right) + 2\frac{m}{\mu}\tau_a. \end{aligned} \tag{5.6}$$

Nous devons maintenant analyser les différents cas donnés par les termes max.

- **Cas où** $\beta + \frac{m}{\mu}\tau \leq \frac{m}{\mu}\tau_a$

  Nous avons alors $\mu \leq \frac{m(\tau_a - \tau)}{\beta}$. Le coût total de l'algorithme devient alors :

$$T_{red}^{1} = \beta + \frac{m}{\mu}\tau + (P - 3 + 2\mu)\frac{m}{\mu}\tau_a. \tag{5.7}$$

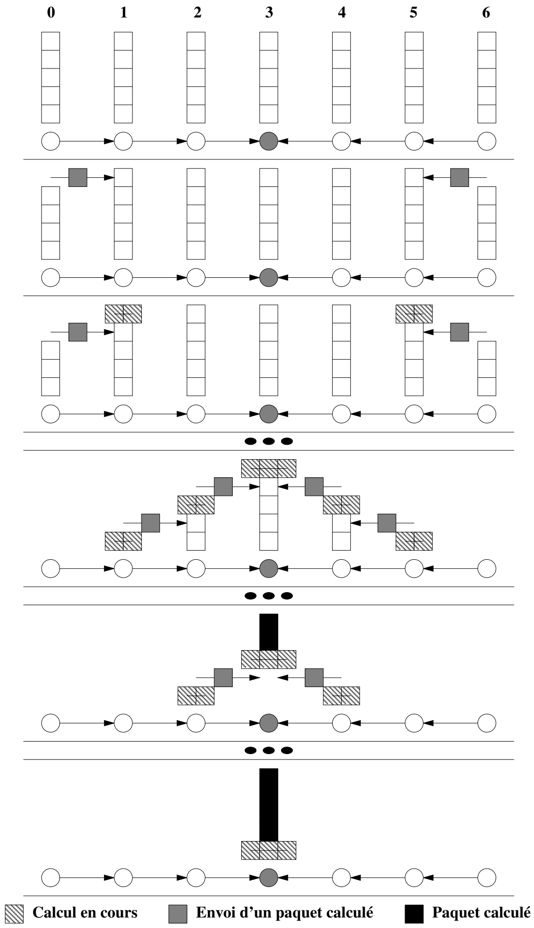

FIG. 5.11 - Réduction pipeline sur un anneau.

La minimisation à l'aide de la dérivée du temps par rapport à μ nous indique qu'il faut choisir μ le plus grand possible, nous choisissons alors $\mu = \frac{m(\tau_a - \tau)}{\beta}$.

- **Cas où** $\frac{m}{\mu}\tau_a < \beta + \frac{m}{\mu}\tau \leq 2\frac{m}{\mu}\tau_a$
  Ceci correspond à $\frac{m(\tau_a-\tau)}{\beta} < \mu \leq \frac{m(2\tau_a-\tau)}{\beta}$. Le coût total de l'algorithme devient alors :

$$T^2_{red} = (P-2)\left(\beta + \frac{m}{\mu}\tau\right) + 2m\tau_a. \tag{5.8}$$

  Il suffit de choisir $\mu = \frac{m(2\tau_a-\tau)}{\beta}$.

- **Cas où** $\beta + \frac{m}{\mu}\tau > 2\frac{m}{\mu}\tau_a$
  Ceci correspond à $\mu > \frac{m(2\tau_a-\tau)}{\beta}$. Le coût total de l'algorithme devient alors :

$$T^3_{red} = (P-3+\mu)\left(\beta + \frac{m}{\mu}\tau\right) + \frac{2m}{\mu}\tau_a. \tag{5.9}$$

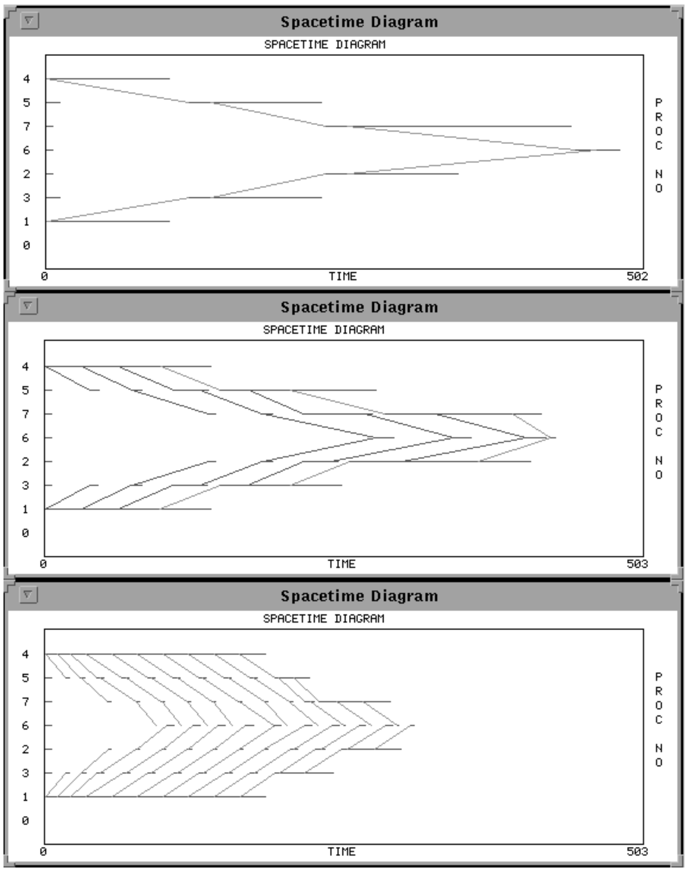

FIG. 5.12 - Exécution de la réduction à l'aide de `loccs_shift` sur iPSC/860.

Nous pouvons alors minimiser le temps en le dérivant par rapport à μ, ce qui nous donne une taille de paquet optimale égale à : $\mu = \sqrt{\frac{m((P-3)\tau+2\tau_a)}{\beta}}$.

Soit un temps total égal à :

$$T^3_{red} = (P-3)\beta + m\tau + 2\sqrt{\beta((P-3)\tau + 2\tau_a)m} \tag{5.10}$$

Si l'opération est une addition, avec nos machines cibles, seul le troisième cas sera valable. Ceci n'est pas évident dans le cas d'une opération définie par l'utilisateur. Néanmoins, nous ne considérons ici que les opérations linéaires en m. Le temps final est asymptotiquement en $O(m\tau)$.

Ecriture à l'aide des LOCCS et expériences

La réduction pipeline a été écrite à l'aide de la routine `loccs_shift` sur iPSC/860 et de nombreux tests de tailles de paquets ont été effectués. Cette réduction peut s'écrire très simplement en chaînant des appels à la routine. Le programme résultant est donné dans l'annexe B. La Figure 5.12 montre une exécution de la réduction de matrices 100×100 sur sept processeurs. Les trois diagrammes Spacetime correspondent à des exécutions avec 1, 4 et 10 paquets.

Conclusion

L'utilisation des routines LOCCS pour la réduction donne donc de bons résultats en gardant le code simple et transparent. Le même type d'optimisation pourrait être effectué dans le cas d'une routine de préfixe parallèle.

5.7.2 Factorisation de Cholesky

Soit une matrice symétrique définie positive A, il est possible de la factoriser comme le produit de deux matrices triangulaires LL^t où L est une matrice triangulaire inférieure. Si la matrice $A(N, N)$ est rangée sur un anneau de processeurs de manière circulaire par colonnes, un algorithme parallèle a été proposé par Geist et Heath [GH85].

```
pour j = 0 à N − 1 faire
   si col_j est une de mes colonnes alors
      cdiv(j)
   finsi
   diffuser(col_j)
   pour toutes mes colonnes k > j faire
      cmod(k,j)
   finpour
finpour
```

ALG. 5.3 - Algorithme de factorisation de Cholesky par colonnes.

Dans l'algorithme 5.3, `cmod(k,j)` effectue la $(j+1)$ème actualisation des colonnes k et `cdiv(j)` calcule la version finale de la colonne j en la divisant par la racine carrée de son élément diagonal. La version de l'algorithme pipeline pour Cholesky par colonnes est triviale, et a été d'ailleurs suggérée par les auteurs [GH85]. Pour l'implémenter, nous utilisons la routine `loccs_ota`. `CMOD` est la tâche correspondant à l'opération `cmod` sur toutes les colonnes restantes de la matrice (Algorithme 5.4). Nous n'avons pas détaillé ici le remplissage des structures `ota`, `bufferota`, `cdiv` et `CMOD`.

```
pour j = 0 à N − 1
   loccs_ota(me, ota, bufferota, ν, N-j-1, cdiv, CMOD)
endfor
```

ALG. 5.4 - Factorisation de Cholesky par colonnes utilisant la routine `loccs_ota`.

5.7.3 Conclusion

De nombreuses autres applications existent et sont en cours de développement comme l'élimination de Gauss, le filtrage en imagerie, le préfixe parallèle et la multi-réduction. Toutes ces applications se prêtent très facilement à une implémentation à l'aide des routines présentées dans ce chapitre. Nous verrons dans les deux prochains chapitres d'autres applications des LOCCS comme la mise à jour de rang 2k (Chapitre 6, Paragraphe 6.4.4) et la FFT bi-dimensionnelle (Chapitre 7, Paragraphe 7.4).

5.8 Calcul de la taille de paquet optimale

Nous avons vu précédemment que la taille du paquet est le point clé de l'obtention de performances à l'aide des routines LOCCS. S'il est trop petit, le nombre d'initialisations est trop grand et les performances plus mauvaises que si l'on n'utilisait pas le recouvrement (à gauche sur la Figure 5.3). Par contre, si la taille de paquet est trop grande, le recouvrement n'est pas assez important (à droite sur la Figure 5.3). Il existe donc un compromis, fonction de la taille des données traitées, du coût des calculs et du coût des communications. Dans ce paragraphe, nous donnons quelques idées concernant le calcul de la taille de paquet optimale. Ce problème est crucial et ne doit pas être laissé à la charge de l'utilisateur. Deux options seront expérimentées sur les routines, nous les présentons dans les deux paragraphes suivants.

5.8.1 Routine de calcul de la taille de paquet

Nous pourrions donner à l'utilisateur une routine de calcul de la taille de paquet optimale ν_{opt}. Cette routine serait accompagnée d'une routine d'initialisation des paramètres de la machine cible (coûts de communication, coûts de calculs). Celle-ci existe déjà dans le compilateur Fortran-D.

Les problèmes majeurs sont au nombre de deux. Premièrement, comment l'utilisateur peut-il passer à la routine ses différents coûts de calculs, fonctions de la taille des données? Une constante devant la taille ne suffit pas, car alors nous ne sommes pas capables de savoir si les calculs sont en $O(n^2)$, $O(n^3)$, ...Une autre solution consiste à passer, à l'aide d'une chaîne de caractères, la complexité, même approximative, des calculs utilisés. Le coût du traitement de la chaîne risque de faire perdre l'intérêt d'une telle bibliothèque. Une bonne solution pour les problèmes d'algèbre linéaire consiste à passer un polynôme de degré 3. Remarquons qu'un compilateur de type HPF peut donner ces informations à la routine. Par contre, la taille totale du message ne peut n'être connue que lors de l'exécution. Un deuxième problème se pose qui est que, au niveau du processeur, nous n'avons pas une connaissance globale de l'algorithme. C'est à dire que même si nous sommes capables de calculer le coût des calculs locaux, des contraintes liées à l'algorithme peuvent faire que ce calcul est tout simplement faux. Un bon exemple de ce type de problème est la réduction où la racine ralentit les autres processeurs. Un calcul au niveau d'un processeur ne convient pas dans ce cas.

Ce calcul est donc très compliqué dans le cas général et le choix de la taille de paquet est laissé à l'utilisateur pour cette première version de la bibliothèque. Des expériences peuvent être faites pour déterminer la meilleure taille.

5.8.2 Taille de paquet adaptative

La taille de paquet pourrait être calculée de manière totalement dynamique, évoluant au cours du temps en fonction de l'algorithme. Un calcul de différence de temps entre les calculs et les communications à l'étape i pourrait permettre un "réglage" dynamique de la taille du paquet de l'étape $i+2$. Cette modification peut avoir lieu avec un cycle de pas régulier, pas forcément égal à un, pour éviter un surcoût nuisible aux performances. Il faut rajouter la taille du prochain paquet dans le message suivant ($i+1$). Il suffit de partir avec une taille de paquet fonction des paramètres de la machine et de tenir compte du nombre d'initialisations et de leur coût.

Avec une telle solution, nous supprimons tous les inconvénients énoncés dans le paragraphe précédent. Cette voie semble donc très prometteuse et sera implémentée dans la prochaine version de la bibliothèque.

5.9 Conclusion

Nous avons présenté dans ce chapitre une bibliothèque permettant à l'utilisateur de machines parallèles à mémoire distribuée d'utiliser des méthodes de type *macro-pipeline* de manière efficace et relativement transparente. Les avantages et inconvénients de cette bibliothèque, dans son état actuel, sont les suivants :

Avantages

- Programmation simple proche du séquentiel et du modèle SPMD,
- recouvrement des calculs et des communications,
- recouvrement calculs/calculs entre les processeurs,
- macro-pipeline transparent pour l'utilisateur,
- pas d'augmentation de la taille du code source parallèle.

Inconvénients

- Nécessite des communications globales non-bloquantes,
- calcul de la taille du paquet optimale complexe pour l'utilisateur,
- écriture des travaux par paquets par l'utilisateur.

Le premier inconvénient est dû à la forme de nos routines qui utilisent des routines de haut niveau pour chaque paquet. Une autre version des LOCCS avec recouvrement au-dessus des différentes étapes d'une routine de haut niveau est à l'étude. Il se trouve que la version présentée dans ce chapitre est complètement portable puisque indépendante de l'algorithme de communication utilisé. Dans le second cas, l'utilisateur doit avoir la connaissance du fonctionnement de l'algorithme de communication sur la machine cible choisie pour savoir sur quelles données il travaille et quelle est la taille des messages reçus à chaque étape. Le second inconvénient est inhérent au macro-pipeline lui-même. Cela sera résolu dans la prochaine version de la bibliothèque à l'aide d'une méthode dynamique de calcul de la taille : la taille adaptative. Le dernier, nécessitant du programmeur une connaissance du découpage externe ne peut être résolu qu'au niveau de la compilation comme dans les compilateurs de langages data-parallèles où le compilateur lui-même se charge d'écrire les codes correspondant aux Job_{before} et Job_{after}. De ce fait, une telle bibliothèque peut être avantageusement utilisée par un compilateur de type HPF.

Chapitre 6

Algèbre linéaire parallèle

Dans ce chapitre, nous présentons nos travaux sur la parallélisation de routines d'algèbre linéaire de la bibliothèque BLAS de niveau 3 et de LAPACK. Le choix de ce type de BLAS est dû à la granularité importante de ce type d'opérations et à leur utilité pour la parallélisation de la bibliothèque LAPACK utilisant des méthodes par blocs. Il faut noter que la parallélisation de BLAS de niveaux inférieurs est également importante et a déjà été très étudiée par ailleurs. Notre but immédiat n'est pas de fournir une bibliothèque complète "clés en main" mais plutôt d'en étudier la faisabilité et de guider le programmeur vers le choix de tel ou tel algorithme suivant la machine cible. Les programmeurs et les vendeurs de machines parallèles devront tenir compte de ces paramètres pour implémenter de manière efficace la bibliothèque BLAS de niveau 3 sur telle ou telle machine parallèle. Néanmoins, les algorithmes parallèles implémentés ici peuvent très bien être utilisés dans un programme mais ils ne traitent pas tous les cas possibles comme différents rangements et différents types de données.

Dans ce chapitre, cinq sujets sont abordés. Le nombre et l'étendue des communications étant directement liés à la répartition des données sur le réseau, nous présentons une allocation générale des matrices sur une grille ou bien des anneaux. Cette répartition sera utilisée par la suite pour la résolution de problèmes d'algèbre linéaire. Ensuite, nous présentons une liste des contraintes liées à l'implémentation d'algorithmes d'algèbre linéaire dans une bibliothèque parallèle. Dans un troisième paragraphe, nous faisons une étude complète du produit de matrices. Après un tour d'horizon rapide des algorithmes existant sur machines parallèles à mémoire distribuée, nous effectuons une comparaison théorique et expérimentale de trois d'entre eux, choisis pour leurs qualités. Dans le quatrième paragraphe, nous présentons une étude de la mise à jour de rang 2k sur une architecture reconfigurable. Enfin, avant une conclusion, nous présentons deux méthodes de factorisation LU, sur un anneau et sur une grille.

6.1 Allocation des données sur les processeurs

La distribution des données sur les processeurs est l'une des clés de l'obtention de bonnes performances. En effet, un mauvais équilibrage des charges conduit à une mauvaise efficacité. Les processeurs doivent tous rester actifs le plus longtemps possible et le coût des calculs dépend du nombre de données présentes. De plus, sur les machines parallèles à mémoire distribuée, le coût de l'accès aux données rangées localement dans le processeur est bien inférieur au coût de l'accès à des données rangées dans la mémoire d'un autre processeur. Les données nécessaires à un calcul doivent donc se trouver au maximum sur le processeur sur lequel il s'exécute. Avec les environnements actuels, c'est l'application qui est responsable du chargement des données sur les processeurs. Avec les langages data-parallèles de type High Performance Fortran [For93] ou Fortran-D [Tse93], la distribution des données est spécifiée au moment de la déclaration des tableaux.

Dans ce paragraphe, nous nous intéresserons plus particulièrement aux rangements de données propres à l'algèbre linéaire dense. De nombreux problèmes tels que ceux liés à l'imagerie peuvent nécessiter des rangements équivalents [Ube93]. Supposons que nous ayons à distribuer un vecteur de M données sur P processeurs. Nous pouvons décrire cette allocation comme le mapping d'un indice global d'une donnée m sur une paire d'indices (p,i) où p est le processeur sur lequel la donnée est déposée et i, l'indice local dans ce processeur. Nous avons $0 \leq m < M$ et $0 \leq p < P$.

Deux décompositions classiques de matrices dans une grille sont la décomposition par blocs et la décomposition circulaire ou *scattered* [DW93]. Les algorithmes par blocs pour l'algèbre linéaire sont de plus en plus utilisés afin d'augmenter le grain de calcul et d'améliorer l'utilisation des mémoires hiérarchiques. Nous avons vu dans un précédent chapitre l'effet de l'augmentation du grain de calcul sur les performances (Chapitre 3, Paragraphe 3.1.1). Pour la distribution par blocs,

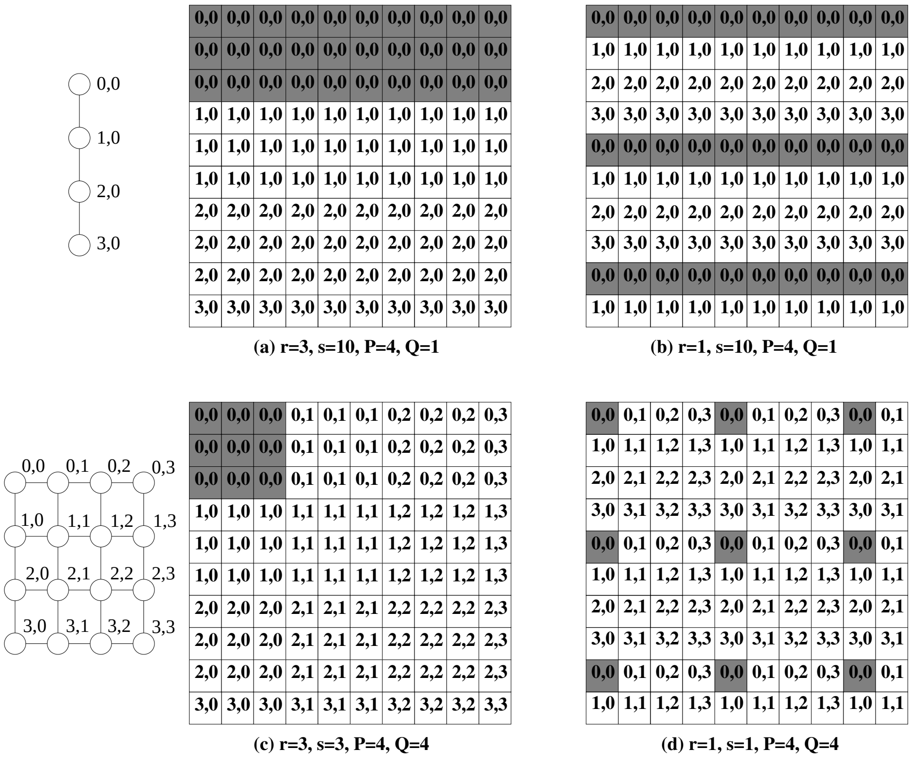

FIG. 6.1 - Différentes allocations d'une matrice sur un réseau linéaire et une grille.

nous avons $m \mapsto (\lfloor m/L \rfloor, mmodL)$, où $L = \lceil M/P \rceil$. Et pour la distribution circulaire, nous avons $m \mapsto (mmodP, \lfloor m/P \rfloor)$. Les distributions par blocs complets ou circulaires sont des cas particuliers de la décomposition circulaire par blocs. Dans ce cas, la distribution est donnée par un triplet (p, b, i) où p est le numéro de processeur, b le numéro de bloc et i l'index dans le bloc. Cette distribution est donnée par $m \mapsto (\lfloor \frac{mmodT}{r} \rfloor, \lfloor \frac{m}{T} \rfloor, mmodr)$ où r est la taille de bloc et $T = rP$. Notons que cette distribution suppose que l'élément 0 est placé sur le processeur 0, ce qui est souvent le cas. Pour une généralisation, se référer au rapport de Dongarra et Walker [DW93]. Supposons à présent, que nous ayons à décomposer une matrice de taille $M \times N$ sur une grille de $P \times Q$ (P lignes et Q colonnes de processeurs). Le nombre de processeurs est donc de $N_p = PQ$ [DW93]. Les décompositions précédemment décrites s'appliquent également aux matrices où toutes les variations de tailles de blocs sont possibles. La décomposition d'une matrice donne un triplet de paires d'indices $(m, n) \mapsto ((p, q), (b, d), (i, j))$. La Figure 6.1 donne différents exemples de décomposition d'une matrice 10×10 (les numéros dans la matrice sont les indices des processeurs où sont rangés les éléments). Sur la gauche de la figure, nous donnons les numérotations des processeurs et les topologies virtuelles utilisées. Les deux décompositions supérieures concernent un réseau linéaire de $P = 4$ processeurs, les deux décompositions inférieures s'effectuent sur une grille 4×4. La taille verticale des blocs est donnée par le paramètre r et la taille horizontale des blocs par le paramètre s.

Nous utiliserons dans un prochain chapitre un rangement de matrice dans un hypercube adapté à la transformée de Fourier rapide. Celle-ci sera décrite dans le chapitre concernant la FFT (Chapitre 7). Notons qu'il est parfois nécessaire de changer l'allocation des données en cours de programme afin de garder une bonne efficacité. Des routines de communications spécifiques sont alors nécessaires. Un bon exemple est la transposition de matrice qui est utilisée notamment dans le produit de matrices et la FFT bi-dimensionnelle. Des changements de tailles de blocs sont également parfois rendus nécessaires par les algorithmes parallèles. Pour des exemples d'analyse théorique des permutations, se référer entre autres aux articles de Van de Velde [Vel90] et de Cosnard, Loi et Tourancheau [CLT92].

Par la suite, nous supposerons que la taille des blocs divise la taille des données.

6.2 Contraintes liées à l'implémentation dans une bibliothèque

De nombreux problèmes apparaissent avec l'utilisation de machines parallèles à mémoire distribuée pour l'implémentation de bibliothèques d'algèbre linéaire. Ceux-ci n'existaient pas avec les machines à mémoire partagée. Tous ces problèmes sont, bien entendu, liés au fait que les données sont distribuées dans le réseau. Dans ce paragraphe, nous présentons les problèmes et proposons quelques solutions.

6.2.1 Interface extérieure

Le problème de l'implémentation de routines d'algèbre linéaire sur une machine parallèle à mémoire distribuée ne réside pas uniquement dans le choix du meilleur algorithme pour une routine de calcul donnée. L'interface avec l'extérieur a une extrême importance. Plusieurs cas peuvent se présenter [SB91] :

1. **Hôte uniquement** : Les routines sont appelées de l'hôte avec les matrices sur l'hôte. Ceci nécessite un chargement du réseau avec distribution adéquate des matrices et exécution sur le réseau.

2. **Hôte et processeurs** : Les routines sont appelées de l'hôte mais les matrices sont déjà rangées dans le réseau.

3. **Processeurs uniquement** : Les routines sont appelées de l'intérieur du réseau avec des matrices déjà rangées sur le réseau.

Remarquons que pour l'option 1, la distribution des matrices peut être faite en tenant compte très précisément du rangement le plus adapté au calcul courant. Par exemple, pour un produit AB^T, la matrice B se doit d'être rangée de manière transposée dans le réseau. De plus, afin de limiter le nombre d'initialisations, les différentes matrices nécessaires au calcul peuvent être groupées en une seule et même matrice. Si l'option 1 permet de distribuer les matrices comme on le souhaite et, de ce fait, de la manière la plus appropriée pour le calcul courant, les options 2 et 3 peuvent survenir à la suite d'autres calculs et les matrices peuvent donc être rangées d'une manière qui n'est pas appropriée pour la routine courante, ce qui peut alors conduire à des temps d'exécution prohibitifs. Il peut alors être nécessaire d'utiliser une routine de redistribution de matrice. De plus, même si un rangement circulaire par blocs est utilisé, il se peut que la taille des blocs ne soit pas adaptée au calcul courant. Il y a donc des compromis entre le *coût de redistribution + coût de l'algorithme avec rangement adapté* et le *coût de l'algorithme avec rangement non adapté*. Deux autres options peuvent alors être choisies, suivant que les données doivent être distribuées ou dupliquées. En effet, suivant la taille des données, il peut être parfois plus intéressant de dupliquer une des matrices pour réduire le nombre de mouvements de données.

Un autre problème, soulevé par P. Berger et P. Morere, lors de l'implémentation de la bibliothèque BLAS de niveau 3 sur un anneau de transputers [BM90], concerne la taille des matrices. Il se peut que les matrices nécessaires au calcul courant ne tiennent pas entièrement sur le réseau. On pourra alors utiliser une méthode par blocs, en appelant séquentiellement de l'hôte plusieurs fois la même routine BLAS avec des blocs différents. Une réutilisation des calculs précédents est, là aussi, parfois nécessaire. Le problème alors soulevé concerne la rémanence des données sur le réseau. Il est inconcevable que toutes les matrices soient chargées et déchargées à chaque appel de routine. Deux solutions ont été envisagées [BM90] :

1. **Solution explicite** : En passant des paramètres supplémentaires aux routines pour leur signifier que les matrices sont déjà sur le réseau (ou pas) et doivent rester (ou pas) après exécution de la routine,

2. **Solution implicite** : Le processeur interface connaît la position des matrices sur le réseau et dans les mémoires locales.

Nous pensons qu'une solution hybride serait idéale pour une première version : les routines supposent toujours que les données sont sur le réseau et y restent, d'autres routines sont appelées pour charger et décharger le réseau quand c'est utile. Ainsi, les routines sont codées de manière indépendante de la topologie du réseau, seules les routines de chargement/déchargement ont une connaissance précise des possibilités de la machine en matière de rangement de données (disques multiples, position des matrices dans tel ou tel groupe de processeurs, ...).

Une dernière remarque tient à l'utilisation d'une interface orientée objet. Dans [DPW93], une interface orientée objet pour ScaLAPACK, appelée ScaLAPACK++ est décrite dans laquelle les matrices distribuées sur le réseau sont des objets. L'interface avec les BLAS distribués est la même que pour les BLAS séquentiels, le parallélisme apparaît à l'intérieur de ces routines sous forme d'appel à des routines de communication (BLACS). ScaLAPACK++ utilise une classe abstraite appelée `LaDistGenMat` (**La**pack **Dist**ributed **Gen**eral **Mat**rix) de laquelle les différentes distributions de matrices peuvent être dérivées. L'utilisateur d'un programme codé avec ScaLAPACK++ ne verra rien de l'utilisation du parallélisme. Le code, identique au séquentiel, sera pourtant optimisé en son sein, grâce à des routines BLAS parallèles optimisées.

6.2.2 Caractéristiques des algorithmes

Dans ce paragraphe, nous passons en revue les caractéristiques des algorithmes, nécessaires pour être de bons candidats dans le cadre de la parallélisation de routines d'algèbre linéaire. Ces caractéristiques principales sont :

- **la capacité de s'adapter à des distributions générales :** Une distribution circulaire par blocs peut être utilisée afin d'améliorer l'équilibrage des charges comme, par exemple, pour la factorisation LU (Paragraphe 6.5). Des modifications des algorithmes doivent être effectuées pour supporter ce type de distribution. Ces modifications ne sont pas triviales et ont été étudiées dans le cadre de ScaLAPACK pour le produit de matrices par Choi, Dongarra et Walker [CDW93b]. Quoi qu'il en soit, ces modifications ne changent pas de manière importante le comportement général des algorithmes.

- **la possibilité de traiter des matrices rectangulaires :** Dans ce cas, des solutions simples existent comme mettre la matrice rectangulaire dans une matrice carrée et combler avec des 1 sur la diagonale et des 0 ailleurs. Cette solution n'est pas idéale car elle tend à augmenter la taille des calculs avec des opérations inutiles mais permet d'utiliser les algorithmes classiques.

- **l'application efficace à des formes de réseaux quelconques :** Nous supposons pour les trois premiers algorithmes de produit de matrices étudiés que le réseau sous-jacent est un tore (ou une grille) carré à deux dimensions. Cette hypothèse a son importance car ces algorithmes fonctionnent au maximum de leur efficacité pour ces topologies [FJL+88]. Les autres algorithmes nécessitent des réseaux plus particuliers comme des hypercubes ou des réseaux de tailles fixées. Grâce au routage présent sur la plupart des nouvelles machines, on peut simuler un réseau "virtuel" sur un réseau physique. Il est bien entendu évident que des problèmes de contention peuvent alors apparaître.

- **la possibilité de traiter des problèmes utilisant des matrices transposées :** Comme nous l'avons vu dans le chapitre concernant les bibliothèques, les routines BLAS de niveau 3 peuvent porter sur des matrices transposées (par exemple pour les opérations `_GEMM`, produits A^TB, AB^T et A^TB^T).

6.3 Le produit de matrices

Dans ce paragraphe, nous présentons quelques algorithmes de produit de matrices sur machines à mémoire distribuée. Les trois premiers algorithmes utilisent comme réseau sous-jacent un tore à deux dimensions, les deux suivants des hypercubes et enfin pour les derniers, des réseaux en anneaux ou en hypertores. Ces derniers offrent l'intérêt de réduire le nombre de multiplications effectuées.

Nous supposons que nous avons trois matrices A, B et C de taille $N \times N$. Nous avons à calculer le produit $C = C + AB$. Les trois premiers algorithmes sont destinés à un tore ou bien une grille 2D. Par souci de simplicité, nous supposons que le nombre P de processeurs est un carré parfait $(\sqrt{P} \times \sqrt{P})$.

6.3.1 Six algorithmes de produit de matrices sur machine parallèle à mémoire distribuée

Nous donnerons par la suite une analyse de complexité précise et des expériences pour trois d'entre eux, choisis pour leur capacité à s'adapter à des rangements de matrices circulaires par blocs et à des tailles de réseaux non carrés. Les analyses de complexité sont faites pour nos machines cibles Intel Paragon (grille 2D et communications avec protocole wormhole 1-port) et Volvox IS-860 d'ARCHIPEL (tore 2D et protocole à commutation de message 4-ports). Nous partons d'une distribution des matrices par blocs de telle manière que les sous-matrices A_{ij}, B_{ij} et C_{ij} soient disposées sur le processeur dont les coordonnées dans la grille sont (i, j). Après le calcul du produit, les matrices doivent se retrouver à leur place de départ afin de pouvoir être utilisées pour d'autres calculs. Remarquons que cette possibilité serait laissée au choix de l'utilisateur de la routine parallèle de produit de matrices dans le cas de son implémentation dans le cadre d'une bibliothèque.

Algorithme de Cannon

La version blocs de l'algorithme de Cannon [Can69] a été décrite entre autres dans [Ber89]. Cet algorithme nécessite une redistribution de deux des matrices avant le début des calculs et le rangement inverse ensuite. Les sous-matrices C_{ij} sont rangées de manière naturelle sur le tore (sous-matrice C_{ij} rangée sur le nœud (i, j)). Mais pour les matrices A et B, de manière à avoir des communications voisins/voisins, la matrice A est "tournée" horizontalement de telle sorte que

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} = \begin{matrix} A00 & A01 & A02 & A03 \\ A11 & A12 & A13 & A10 \\ A22 & A23 & A20 & A21 \\ A33 & A30 & A31 & A32 \end{matrix} \times \begin{matrix} B00 & B11 & B22 & B33 \\ B10 & B21 & B32 & B03 \\ B20 & B31 & B02 & B13 \\ B30 & B01 & B12 & B23 \end{matrix}$$

FIG. 6.2 - Rangement des sous-matrices pour l'algorithme de Cannon.

sa diagonale soit rangée sur la première colonne de processeurs du tore, la diagonale de B étant distribuée sur la première ligne (Figure 6.2). Pour obtenir ce rangement, les sous-matrices sont déplacées sur les lignes et les colonnes de processeurs suivant leur indice de ligne ou de colonne (A_{ij} sur le processeur $(i, (i+j) mod \sqrt{P})$), B_{ij} sur le processeur $((i+j) mod \sqrt{P}, j)$. Ces rotations sont souvent appelées *preskewing* et *postskewing* dans la littérature [BMSV92].

Cet algorithme ne requiert que des communications voisins/voisins. Pendant la phase de calculs, les sous-matrices A_{ij} se déplacent de droite à gauche sur les lignes de processeurs (rotation horizontale) tandis que les sous-matrices B_{ij} se déplacent de haut en bas (rotation verticale), les matrices C_{ij} restant sur place. De plus, si une bufferisation est possible, communications et calculs de deux étapes différentes peuvent être aisément recouverts (Algorithme 6.1).

```
/* mouvements de données avant les calculs */
Rotations de A et B

/* calcul du produit de matrice */
pour k = 1 à √P en // faire
   GEMM(A, B, C)
   Rotation verticale de B
   Rotation horizontale de A
finpour

/* mouvements de données après les calculs */
Rotations de A et B
```

ALG. 6.1 - Algorithme de Cannon.

Algorithme de Fox

Cet algorithme, décrit par Fox dans [FJL+88, FO87] a été développé en premier pour l'hypercube du CalTech. Le réseau sous-jacent est le tore à deux dimensions. Cet algorithme nécessite la diffusion des sous-matrices A_{ij} durant son exécution. Il est d'ailleurs souvent appelé dans la littérature algorithme *broadcast-multiply-roll* [CDW93b, LS92].

Les données sont réparties de manière naturelle sur le tore, c'est-à-dire que les sous-matrices $[A, B, C]_{ij}$ sont rangées sur les processeurs dont les coordonnées sont (i, j). Cet algorithme nécessite des diffusions horizontales des diagonales de A. Les communications des sous-matrices B sont des

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} \mathrel{+}= \begin{matrix} A00 & A00 & A00 & A00 \\ A11 & A11 & A11 & A11 \\ A22 & A22 & A22 & A22 \\ A33 & A33 & A33 & A33 \end{matrix} \; x \; \begin{matrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{matrix}$$

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} \mathrel{+}= \begin{matrix} A01 & A01 & A01 & A01 \\ A12 & A12 & A12 & A12 \\ A23 & A23 & A23 & A23 \\ A30 & A30 & A30 & A30 \end{matrix} \; x \; \begin{matrix} B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \\ B00 & B01 & B02 & B03 \end{matrix}$$

FIG. 6.3 - Deux premiers pas de l'algorithme de Fox.

rotations verticales (Algorithme 6.2). Les deux premiers pas de l'algorithme sont donnés sur la Figure 6.3.

```
/* pas de mouvements de données avant les calculs */

/* calcul du produit de matrices */
Diffuser la première diagonale de A
pour k = 1 à √P − 1 en // faire
   GEMM(A, B, C)
   Rotation verticale de B
   Diffuser la diagonale suivante de A
finpour
en // faire
   GEMM(A, B, C)
   Rotation verticale de B
fin

/* pas de mouvements de données après les calculs */
```

ALG. 6.2 - Algorithme de Fox.

Algorithme de Snyder

Cet algorithme utilise également une distribution des matrices par blocs sur un tore à deux dimensions. Le schéma de communications est différent car il utilise une transposition préalable de la matrice B avant exécution de l'algorithme ainsi que des sommes globales sur les lignes de processeurs des produits calculés à chaque étape [LS92]. Les matrices A_{ij} et C_{ij} sont rangées sur les processeurs de manière naturelle. Pour la matrice B, il s'agit d'une transposition par blocs (B_{ij}

sur le processeur (j, i)).

Les deux premiers pas de cet algorithme sont donnés sur la Figure 6.4. Nous avons mis en gras sur cette figure le processeur où s'effectue l'accumulation des sous-matrices C. Remarquons qu'il correspond aux différentes diagonales du tore, dans le même ordre que pour les diffusions de Fox (Algorithme 6.3).

/* mouvements de données avant les calculs */
transposer la matrice B

/* calcul du produit de matrices */
en // faire
   GEMM(A, B, C)
   Rotation verticale de B
finpour
pour $k = 1$ à $\sqrt{P} - 1$ en // faire
   Somme globale sur les lignes pour $C_{i,(i+k-1) mod \sqrt{P}}$
   GEMM(A, B, C)
   Rotation verticale de B
finpour
Somme globale sur les lignes pour $C_{i,(i+\sqrt{P}-1) mod \sqrt{P}}$

/* mouvements de données après les calculs */
transposer la matrice B

ALG. 6.3 - Algorithme de Snyder pour le nœud (i, j).

$$\begin{array}{cccc} \mathbf{C00} & C01 & C02 & C03 \\ C10 & \mathbf{C11} & C12 & C13 \\ C20 & C21 & \mathbf{C22} & C23 \\ C30 & C31 & C32 & \mathbf{C33} \end{array} = \begin{array}{cccc} A00 & A01 & A02 & A03 \\ A10 & A11 & C12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{array} \times \begin{array}{cccc} B00 & B10 & B20 & B30 \\ B01 & B11 & B21 & B31 \\ B02 & B12 & B22 & B32 \\ B03 & B13 & B23 & B33 \end{array}$$

$$\begin{array}{cccc} C00 & \mathbf{C01} & C02 & C03 \\ C10 & C11 & \mathbf{C12} & C13 \\ C20 & C21 & C22 & \mathbf{C23} \\ \mathbf{C30} & C31 & C32 & C33 \end{array} = \begin{array}{cccc} A00 & A01 & A02 & A03 \\ A10 & A11 & C12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{array} \times \begin{array}{cccc} B01 & B11 & B21 & B31 \\ B02 & B12 & B22 & B32 \\ B03 & B13 & B23 & B33 \\ B00 & B10 & B20 & B30 \end{array}$$

FIG. 6.4 - Les deux premiers pas de l'algorithme de Snyder.

Algorithme de Berntsen

L'algorithme de Berntsen présenté dans [Ber89] et analysé en terme de scalabilité dans [GK91b] utilise la connectivité de l'hypercube. Il nécessite $P = 2^{3q}$ nœuds avec comme restriction que $P \leq N^{3/2}$ pour multiplier deux matrices $N \times N$ et un rangement par lignes pour la matrice A et un rangement par colonnes pour la matrice B. L'hypercube est alors découpé en 2^q sous-cubes, chacun effectuant une multiplication de type Cannon entre les sous-matrices A de taille $\frac{N}{2^q} \times \frac{N}{2^{2q}}$ et B de taille $\frac{N}{2^{2q}} \times \frac{N}{2^q}$. Il est montré dans [Ber89] que, grâce à l'hypercube et à l'algorithme en "somme cascadée", le coût des communications est réduit.

On peut alors noter que si la topologie en hypercube n'est pas disponible, l'algorithme est équivalent à l'algorithme de Cannon. De plus, grâce aux recouvrements calculs/communications et au ratio de leur coût ($O(N^3)$ contre $O(N^2)$), l'intérêt de cet algorithme se réduit. Il est également important de voir qu'une modification des paramètres "taille de matrice" ou "taille du réseau" conduit à des modifications non triviales de l'algorithme (si on veut garder une efficacité maximum bien entendu).

Variante de l'algorithme DNS

Un algorithme de produit de matrices, proposé par Dekel, Nassimi et Sahni dans [DNS81] utilise un hypercube à $P = N^3 = 2^{3q}$ éléments avec donc un élément par processeur pour donner un temps total de $O(\log(N))$. Cet algorithme est très intéressant d'un point de vue théorique, mais inutilisable avec des machines à gros grain comme celles utilisées dans cette thèse. Dans [GK91b], Gupta et Kumar ont proposé une variante par blocs de cet algorithme, utilisant un nombre de processeurs $P = 2^{3q}$ compris entre 1 et N^3. La topologie adéquate est encore un hypercube et l'algorithme tire partie de sa connectivité. Utilisé sur un tore, il sera équivalent à l'algorithme de Fox.

Algorithmes de Strassen et Winograd

Le premier algorithme rapide de produit de matrices a été découvert par Strassen en 1969 [Str69]. La méthode consiste à réduire le nombre de multiplications en les remplaçant par des additions. Le nombre de multiplications passe de $O(n^3)$ à $O(n^{\log_2 7})$. En 1973, Winograd améliora la méthode de Strassen pour arriver à un nombre de multiplications égal à $O(n^{2.37})$ [Win73]. Les implémentations de ces algorithmes ont été surtout faites sur des machines à mémoire partagée ou vectorielles [Bai88, Hig89, KHJS93] ou SIMD [LA91, BMSV92]. Récemment, Dumitrescu, Roch et Trystram [DRT92] ont proposé des implémentations efficaces de ces algorithmes en séquentiel, et en parallèle sur des anneaux et des hypertores. En partant du graphe des tâches des deux algorithmes et grâce à un placement approprié, il est possible d'obtenir une implémentation efficace des deux algorithmes sur machine à mémoire distribuée. Concernant les algorithmes rapides en séquentiel, ils ont montré que suivant la taille des matrices, on pouvait utiliser tel ou tel algorithme. Dans [Hig89], Higham a traité le cas des matrices de formes quelconques. Pour les implémentations parallèles, le résultat est que le meilleur algorithme est celui de Strassen parallèle pour ses performances, sa simplicité et sa possible utilisation avec des anneaux dont le nombre de processeurs n'est pas une puissance de 7, mais un multiple [DRT92].

Il faut noter toutefois que de tels algorithmes ne sont pas faciles à implémenter tels quels dans le cadre d'une bibliothèque BLAS 3 distribuée. En effet, la distribution des matrices en début de calcul est particulière et ne s'accommode pas facilement d'une distribution circulaire par blocs et à des tailles de réseau quelconques. Nous pensons plutôt que ces algorithmes très performants peuvent être utilisés à l'intérieur de produits de matrices "classiques", soit en séquentiel, soit sur des groupes de processeurs, avec une redistribution préalable des matrices sur ces groupes.

6.3.2 Analyse et expérimentations des algorithmes sur un tore

Dans ce paragraphe, nous présentons une analyse de complexité et une série d'expériences pour les trois algorithmes sur un tore (Cannon, Fox et Snyder). Le choix de ces algorithmes réside dans le fait qu'ils s'adaptent mieux que les autres à une implémentation des BLAS de niveau 3 distribués. Ce choix est confirmé par les études et les expériences de Choi, Dongarra et Walker [CDW93b]. Ces trois algorithmes ont deux points communs qui sont une distribution des matrices par blocs et une rotation de la matrice B sur les colonnes de processeurs.

Nous supposons que l'une des machines utilise un protocole de communication de type à commutation de messages 4-port (SF4P par la suite), c'est à dire que des échanges peuvent s'effectuer entre 4 processeurs voisins pour le coût d'une communication. La seconde machine possède un protocole wormhole 1-port (WH1P par la suite), c'est à dire qu'une seule communication (ou un échange) peut être traitée à la fois. Les deux machines sont full-duplex et possèdent les mêmes processeurs de calculs. Pour nos expériences, nous utilisons une Volvox IS-860 dont le modèle correspond à celui de la première machine et une Paragon dont le modèle correspond à la seconde machine. Nous supposons que les deux machines sont utilisées avec des tores 2D. En effet, même si les connexions entre les processeurs sur les bords n'existent pas sur la Paragon, grâce aux liens full-duplex et au modèle de communication wormhole, on peut utiliser des connexions latérales virtuelles [CDW93b]. Les paramètres des machines cibles sont donnés dans le tableau 6.1.

Mode de com.	**paramètres**	**signification**
SF4P	$\beta_{sf} + L\tau_{sf}$	coût d'une com. msg de taille L entre 2 voisins
WH1P	$\beta_{wh} + L\tau_{wh}$	coût d'une com. msg de taille L entre 2 proc. quelconques
/	τ_a	coût moyen d'une multiplication et d'une addition

TAB. 6.1 - Paramètres des machines cibles pour le produit de matrices.

Pour chaque algorithme, nous donnons les coûts des pré et post-traitements ($T_{algo}^{red\ modele}$) et des produits parallèles eux-mêmes ($T_{algo}^{comp\ modele}$). La somme de ces deux coûts donne le coût total de l'algorithme. Les caractéristiques des trois algorithmes sont résumées dans le tableau 6.2.

Algorithme	**Cannon**	**Fox**	**Snyder**
Pré(post)-traitements	rotations de A et B	aucun	transpo. de B
Calculs des produits	sur place	sur place	sommes globales sur les lignes
Mouvements de A	rotations hor.	diffusions hor.	aucun
Mouvements de B	rotations vert.	rotations vert.	rotations vert.

TAB. 6.2 - Résumé des caractéristiques des trois algorithmes de produit de matrices par blocs.

Algorithme de Cannon

Dans le tableau 6.3, nous donnons les résultats de complexité pour une étape de l'algorithme de Cannon pour les deux modèles. Le coût de calcul correspond aux trois boucles du produit de matrices de taille m, soit $2m^3\tau_a$.

Nous allons analyser en premier le modèle SF4P. En ce qui concerne les pré et post-traitements pour les matrices A et B, une solution consiste à limiter le nombre de rotations jusqu'à $\left\lfloor \sqrt{P}/2 \right\rfloor$. En effet, si l'on prend comme exemple la dernière ligne de processeurs, pour les rotations de A, au lieu de tourner $\sqrt{P}$ fois vers la gauche, il suffit de tourner une fois vers la droite. Le plus grand nombre de rotations se trouve alors au milieu du tore ou l'on doit tourner $\left\lfloor \sqrt{P}/2 \right\rfloor$ fois. Remarquons que, grâce aux communications sur tous les liens à la fois, les coûts de pré(post)-traitements pour A et B ne s'additionnent pas car ils sont exécutés en parallèle. Les rotations de A et de B pendant le calcul se font également en parallèle.

Etape	Modèle	
	SF4P	**WH1P**
Pré(post)trait.	$\left\lfloor \frac{\sqrt{P}}{2} \right\rfloor (\beta_{sf} + m^2\tau_{sf})$	$2 \left\lfloor \frac{\sqrt{P}}{2} \right\rfloor (\beta_{wh} + m^2\tau_{wh})$
Prod. de ss-mat	$2m^3\tau_a$	$2m^3\tau_a$
Com. de A	$\beta_{sf} + m^2\tau_{sf}$	$2(\beta_{wh} + m^2\tau_{wh})$
Com. de B	$\beta_{sf} + m^2\tau_{sf}$	$2(\beta_{wh} + m^2\tau_{wh})$

TAB. 6.3 - Coût des différentes parties de l'algorithme de Cannon pour un pas pour les deux modèles.

Pour le modèle SF4P, en doublant l'espace mémoire pour le stockage des matrices A et B, on peut recouvrir les communications et les calculs, ce qui donne les résultats de complexité pour le calcul du produit de matrices (sans les pré et post-traitements)

$$T_{Cannon}^{comp\ SF4P} = \sqrt{P} \max\left(2m^3\tau_a, \beta_{sf} + m^2\tau_{sf}\right), \tag{6.1}$$

auquel il faut ajouter le coût des pré et post-traitements

$$T_{Cannon}^{red\ SF4P} = 2\left\lfloor \sqrt{P}/2 \right\rfloor (\beta_{sf} + m^2\tau_{sf}). \tag{6.2}$$

Pour la machine utilisant le modèle WH1P et à cause du port unique utilisable, les coûts de communication s'additionnent. Malgré tout, les communications peuvent être recouvertes par les calculs.

Les pré et post-traitements nécessitent une attention toute particulière. En effet, un algorithme naïf utilisera le même algorithme que pour le modèle précédent, avec un temps total (pour un traitement) de $4\left\lfloor \sqrt{P}/2 \right\rfloor (\beta_{wh} + m^2\tau_{wh})$. On peut remarquer que, grâce au modèle de communication wormhole, les processeurs peuvent envoyer leurs données directement au processeur concerné sans passer par tous les processeurs intermédiaires. Par contre, ceci n'est pas si immédiat si l'on tient compte des problèmes de contention, inévitables sur un anneau. La Figure 6.5 montre des exécutions de pré(post)rotations avec le modèle WH1P pour un anneau 8 processeurs et pour des rotations de 1, 2, 3 et 4. On constate alors que l'on peut réduire le nombre de pas en tenant compte des contentions. Le temps total est donc compris entre $2(\beta_{wh} + m^2\tau_{wh})$ et $4\left\lfloor \sqrt{P}/2 \right\rfloor (\beta_{wh} + m^2\tau_{wh})$. Et

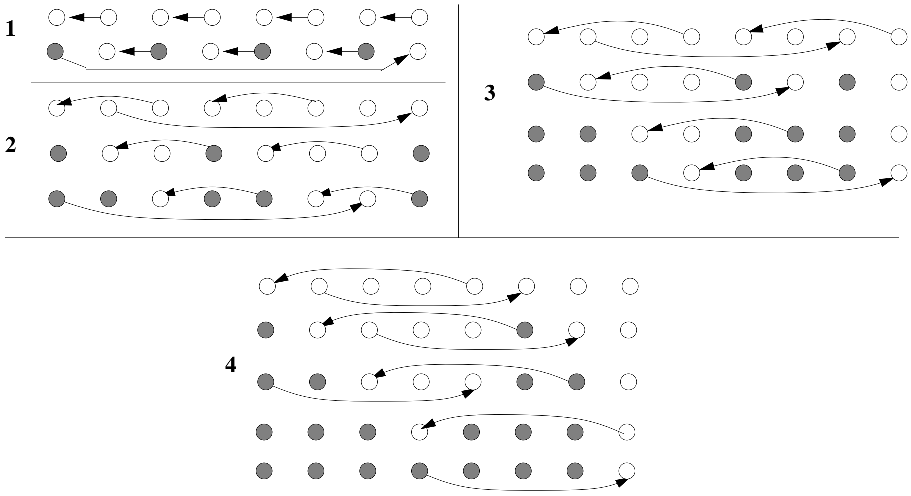

FIG. 6.5 - Pré(post)rotations WH1P de la matrice A pour l'algorithme de Cannon.

plus précisément, si l'on considère que l'on peut faire en moyenne deux communications par pas, il est approximativement égal à $2\left\lfloor\sqrt{P}/2\right\rfloor(\beta_{wh}+m^2\tau_{wh})$ car on doit additionner le temps pour le pré-traitement de la matrice A et celui de la matrice B.

Le coût nécessaire au calcul est donné pour ce modèle par

$$T_{Cannon}^{comp\ WH1P} = \sqrt{P}\max\left(2m^3\tau_a, 4(\beta_{wh}+m^2\tau_{wh})\right), \tag{6.3}$$

auquel il faut ajouter le coût des pré et post-traitements

$$T_{Cannon}^{red\ WH1P} = 2\left\lfloor\sqrt{P}/2\right\rfloor(\beta_{wh}+m^2\tau_{wh}). \tag{6.4}$$

Algorithme de Fox

Comme nous avons $\sqrt{P}$ étapes pour les calculs, nous avons uniquement à multiplier le coût total d'un pas par le nombre de pas. Le surcoût le plus important de l'algorithme de Fox est le coût des diffusions des sous-matrices A. A la différence de l'algorithme précédent, suivant le modèle de communication, des algorithmes différents sont utilisés. Le problème est la diffusion dans un anneau de processeurs. Pour le modèle SF4P, on peut utiliser ou non le pipeline, suivant la taille du message à diffuser. Si le pipeline est utilisé, la taille optimale des paquets est donnée par $\sqrt{\frac{m^2\beta_{sf}}{\left\lfloor\frac{\sqrt{P}}{2}\right\rfloor\tau_{sf}}}$. Remarquons que, avec le modèle SF4P, les rotations de B sont complètement recouvertes par les diffusions de A, elles n'apparaissent donc pas dans le calcul du temps total.

Sans utiliser de pipeline pour les diffusions (SP), le coût nécessaire au calcul est donné pour ce modèle par

$$\begin{aligned} T_{Fox}^{compSP\ SF4P} \quad = \quad & \left\lfloor\sqrt{P}/2\right\rfloor(\beta_{sf}+m^2\tau_{sf}) + \left(\sqrt{P}-1\right)\max\left(2m^3\tau_a, \left\lfloor\sqrt{P}/2\right\rfloor(\beta_{sf}+m^2\tau_{sf})\right) + \\ & \max\left(2m^3\tau_a, \beta_{sf}+m^2\tau_{sf}\right) \end{aligned} \tag{6.5}$$

Etape	Modèle	
	SF4P	**WH1P**
Pré(post)trait.	0	0
Prod. de ss-mat	$2m^3\tau_a$	$2m^3\tau_a$
Com. de A (ss. pipeline)	$\left\lfloor\frac{\sqrt{P}}{2}\right\rfloor(\beta_{sf}+m^2\tau_{sf})$	$\log(\sqrt{P})(\beta_{wh}+m^2\tau_{wh})$
Com. de A (av. pipeline)	$\left(\sqrt{\beta_{sf}\left(\left\lfloor\frac{\sqrt{P}}{2}\right\rfloor-1\right)}+\sqrt{m^2\tau_{sf}}\right)^2$	/
Com. de B	$\beta_{sf}+m^2\tau_{sf}$	$2(\beta_{wh}+m^2\tau_{wh})$

TAB. 6.4 - Coût des différentes parties de l'algorithme de Fox pour un pas pour les deux modèles.

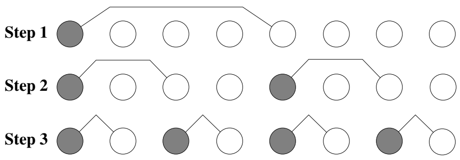

FIG. 6.6 - Méthode de diffusion sur un anneau avec modèle WH1P.

et si on utilise un pipeline pour les diffusions (AP), le coût nécessaire au calcul est donné pour ce modèle par

$$\begin{aligned} T_{Fox}^{compAP\ SF4P} &= \left(\sqrt{\beta_{sf}\left(\left\lfloor\sqrt{P}/2\right\rfloor-1\right)}+\sqrt{m^2\tau_{sf}}\right)^2 + \\ &\left(\sqrt{P}-1\right)\max\left(2m^3\tau_a, \left(\sqrt{\beta_{sf}\left(\left\lfloor\sqrt{P}/2\right\rfloor-1\right)}+\sqrt{m^2\tau_{sf}}\right)^2\right) + \\ &\max\left(2m^3\tau_a, \beta_{sf}+m^2\tau_{sf}\right). \end{aligned} \tag{6.6}$$

Pour le WH1P, la différence principale entre les deux modèles est la diffusion. On utilise ici une variante de l'algorithme *recursive doubling* de diffusion dans les hypercubes. La variation consiste à envoyer au milieu de l'anneau à la première étape et continuer ainsi en doublant à chaque étape le nombre de nœuds informés [BPG91]. Le coût nécessaire au calcul est donné pour ce modèle par

$$\begin{aligned} T_{Fox}^{comp\ WH1P} &= \log(\sqrt{P})(\beta_{wh}+m^2\tau_{wh}) + \\ &\left(\sqrt{P}-1\right)\max\left(2m^3\tau_a, (\log(\sqrt{P})+2)(\beta_{wh}+m^2\tau_{wh})\right) + \\ &\max\left(2m^3\tau_a, 2(\beta_{wh}+m^2\tau_{wh})\right). \end{aligned} \tag{6.7}$$

Algorithme de Snyder

Les différences majeures entre l'algorithme de Snyder et les précédents est l'utilisation d'une pré et post-transposition de la matrice B et les sommes globales pour accumuler les sous-matrices

C_{ij}. Remarquons que dans le tableau 6.5 résumant les divers coûts pour le calcul des produits de matrices, nous avons séparé en deux le calcul du produit lui-même (AB) et l'accumulation des matrices C qui nécessite ici des communications sur les lignes de processeurs. Nous avons, par ailleurs, distingué la méthode avec et sans pipeline. Pour la transposition sur un tore 2D en SF4P, nous pouvons utiliser l'algorithme décrit dans [CT93a]. Celui-ci utilise une méthode pipeline sur deux chemins disjoints dans le tore 2D. Pour les accumulations de sous-matrices C sur les lignes de processeurs, nous pouvons utiliser la méthode pipeline décrite dans le chapitre sur les LOCCS (Chapitre 5, Paragraphe 5.7.1). Les complexités sont données pour $\sqrt{P}$ impair. Pour la transposition, nous utilisons deux algorithmes différents suivant les modes de routage. Pour la commutation de message, nous utilisons l'agorithme décrit dans [CT93a]. Pour le modèle wormhole, nous utilisons l'algorithme récursif décrit dans [Ho93].

Etape	**Modèle**	
	SF4P	**WH1P**
Pré(post)trait.	$\lfloor\sqrt{P}/2\rfloor\left(\sqrt{\beta_{sf}}+\sqrt{\frac{N^2}{2P}\tau_{sf}}\right)^2$	$log_2(\sqrt{P})\beta_{wh}+\left(\frac{\sqrt{P}}{2}+1\right)m^2\tau_{wh}$
Prod. de ss-mat	$2m^3\tau_a$	$2m^3\tau_a$
Acc. des sous-mat. C sans pipeline	$\lfloor(\sqrt{P}-2)/2\rfloor(\beta_{sf}+m^2\tau_{sf})$ $+\lfloor(\sqrt{P}+1)/2)\rfloor m^2\tau_a$	$\log(\sqrt{P})(\beta_{wh}+m^2\tau_{wh}+m^2\tau_a)$
Acc. des sous-mat. C avec pipeline	$(\sqrt{P}-3)\beta_{sf}+m^2\tau_{sf}$ $+2m\sqrt{\beta_{sf}((\sqrt{P}-3)\tau_{sf}+2\tau_a)}$	/
Com. de A	0	0
Com. de B	$\beta_{sf}+m^2\tau_{sf}$	$2(\beta_{wh}+m^2\tau_{wh})$

TAB. 6.5 - Coût des différentes parties de l'algorithme de Snyder pour un pas pour les deux modèles.

Le coût total de cet algorithme pour le modèle SF4P s'obtient donc en additionnant le coût nécessaire au calcul du produit au coût des pré et post-transpositions (nous ne considérons que la version pipeline pour les accumulations).

$$\begin{aligned}
T_{Snyder}^{comp\ SF4P} &= \max(2m^3\tau_a, \beta_{sf}+m^2\tau_{sf}) + \\
&\quad (\sqrt{P}-1)\max\left(2m^3\tau_a, (\sqrt{P}-3)\beta_{sf}+m^2\tau+2m\sqrt{\beta_{sf}((\sqrt{P}-3)\tau_{sf}+2\tau_a)}\right) + \\
&\quad (\sqrt{P}-3)\beta_{sf}+m^2\tau_{sf}+2m\sqrt{\beta_{sf}((\sqrt{P}-3)\tau_{sf}+2\tau_a)}, \qquad (6.8)
\end{aligned}$$

et le coût des pré et post-traitements :

$$T_{Snyder}^{red\ SF4P} = \lfloor\sqrt{P}/2\rfloor\left(\sqrt{\beta_{sf}}+\sqrt{\frac{N^2}{2P}\tau_{sf}}\right)^2. \qquad (6.9)$$

Pour le WH1P, les accumulations sont effectuées avec une méthode proche de celle utilisée auparavant pour la diffusion des sous-matrices A_{ij} dans l'algorithme de Fox. L'algorithme s'exécute de manière inverse de la diffusion, et après chaque réception, les processeurs additionnent les sous-matrices C reçues avec la sous-matrice C calculée à l'étape courante et renvoie le résultat si nécessaire. Le nombre d'étapes est bien entendu le même que pour la diffusion.

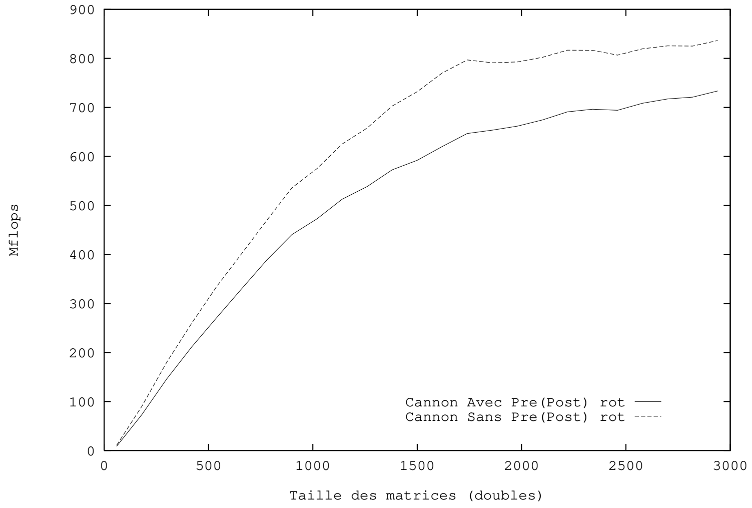

FIG. 6.7 - Performances de l'algorithme de Cannon sur 36 processeurs avec ou sans (pré)post-traitements sur la Volvox IS-860.

Le coût total de cet algorithme pour le modèle WH1P s'obtient donc en additionnant le coût nécessaire au calcul du produit

$$\begin{aligned} T_{Snyder}^{comp\ WH1P} &= \max\left(2m^3\tau_a, 2(\beta_{wh} + m^2\tau_{wh})\right) + \\ &\left(\sqrt{P}-1\right)\max\left(2m^3\tau_a, \log(\sqrt{P})(\beta_{wh} + m^2\tau_{wh} + m^2\tau_a) + 2(\beta_{wh} + m^2\tau_{wh})\right) + \\ &\log(\sqrt{P})(\beta_{wh} + m^2\tau_{wh} + m^2\tau_a), \end{aligned} \tag{6.10}$$

et le coût des pré et post-transpositions

$$T_{Snyder}^{red\ WH1P} = log_2(\sqrt{P})\beta_{wh} + \left(\frac{\sqrt{P}}{2} + 1\right) m^2\tau_{wh}. \tag{6.11}$$

Expérimentations et comparaisons

Les expérimentations ont été faites sur la Volvox IS-860. Les communications ont été recouvertes au maximum et les techniques pipelines présentées dans les paragraphes précédents pour le modèle à commutation de message ont été implémentées. Les Figures 6.7 et 6.8 donnent une comparaison entre une version sans (pré)post-traitements et une version avec pour les algorithmes de Cannon et Snyder. La perte de performances due aux redistributions des sous-matrices avant et après les calculs est significative. Elle est plus importante pour Cannon que pour Snyder. En effet, la redistribution pour l'algorithme de Snyder est effectuée à l'aide d'une reconfiguration et est donc peu coûteuse.

Les Figures 6.9 et 6.10 donnent des comparaisons de performances pour les trois algorithmes et avec le produit de matrices séquentiel. Les algorithmes présentés sur la première courbe n'utilisent pas de redistributions des matrices, à l'inverse de ceux présentés sur la seconde courbe. Nous constatons que, du fait du recouvrement des communications par les calculs plus facile à implémenter pour l'algorithme de Cannon, ses performances sont bien meilleures que celles des deux autres

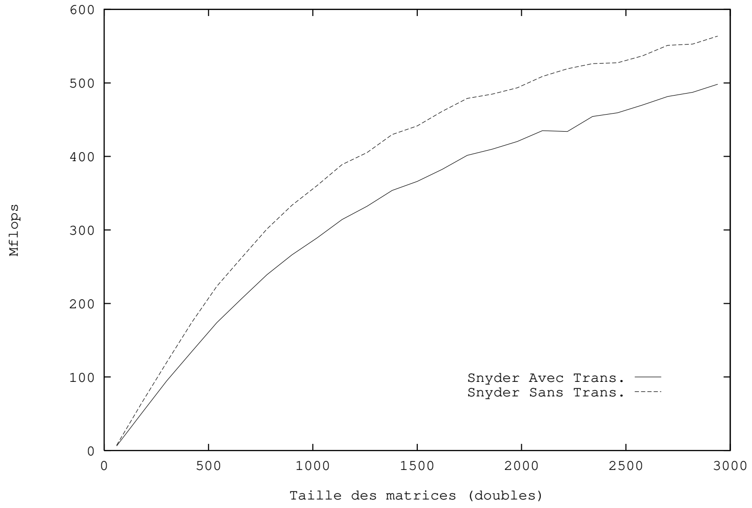

FIG. 6.8 - Performances de l'algorithme de Snyder sur 36 processeurs avec ou sans (pré)post-traitements sur la Volvox IS-860.

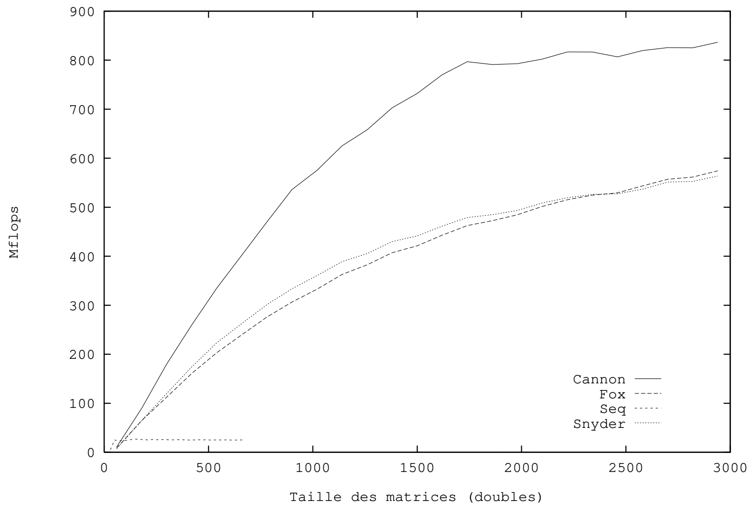

FIG. 6.9 - Comparaison des performances des trois algorithmes sans (pré)post-traitements sur la Volvox IS-860.

algorithmes. Remarquons également que sans les redistributions, les performances des algorithmes de Fox et Snyder sont équivalentes. Par contre, avec les redistributions, uniquement nécessaires pour l'algorithme de Snyder, les performances sont légèrement amoindries. Les performances de la routine BLAS de produit de matrices séquentiel sont d'environ $25 Mflops$ soit un total de 820 Mflops sur 36 processeurs avec Cannon sans pré et post-traitements.

Des optimisations pourraient être faites sur l'implémentation du pipeline des diffusions ou des

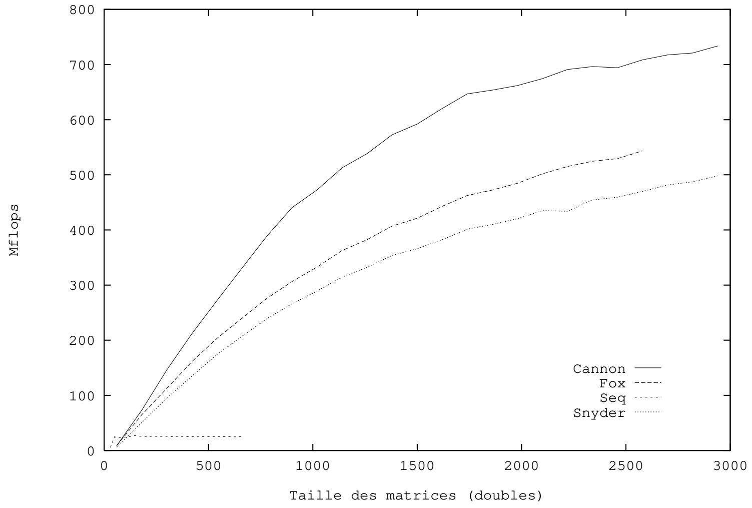

FIG. 6.10 - Comparaison des performances des trois algorithmes avec (pré)post-traitements sur la Volvox IS-860.

sommes globales, pour les algorithmes de Fox et Snyder. En effet, dans cette version du code, seuls les mouvements des sous-matrices B_{ij} sont recouverts.

6.3.3 Traitement des produits avec des matrices transposées

La routine _GEMM des BLAS de niveau 3 doit pouvoir traiter des produits de matrices dont une ou plusieurs matrices sont transposées (A^TB, AB^T et A^TB^T). Si la routine de produit est appelée juste après un chargement de matrices, il suffit de tenir compte, dans ce chargement, de cette particularité et ensuite d'appeler un produit de matrices classique. Par contre, si un tel produit intervient sur des matrices déjà en place, il faut pouvoir travailler sur les bons éléments. Une solution consiste à transposer une ou deux matrices avant appel d'un algorithme classique. Le surcoût engendré par cette communication peut être évité. Pour les produits AB^T et A^TB, il suffit d'utiliser une variante de l'algorithme de Snyder avec des transpositions internes des matrices. Suivant le produit à effectuer, les sommes pour les sous-matrices C_{ij} se feront sur les lignes ou bien les colonnes de processeurs. Pour le produit A^TB^T, deux solutions sont possibles. Après transposition locale des sous-matrices A_{ij} et B_{ij}, on peut utiliser une variation d'un algorithme classique. Par contre, la matrice C résultante doit être alors transposée à son tour (physiquement sur le réseau). Ou bien, on utilise l'algorithme classique et on transpose la matrice C résultat sur le réseau et localement.

Il est toujours possible de calculer efficacement ces produits de matrices transposées avec les trois algorithmes sur un tore, en jouant sur les communications de A et de B et (ou) les sommes de C. Cela a été également confirmé avec un rangement circulaire par blocs sur la Delta dans [CDW93b].

6.4 Les mises à jour de rang 2k

La routine BLAS de niveau 3 _SYR2K peut être aisément parallélisée grâce aux routines de transposition et de produit de matrices. Rappelons que cette routine effectue les opérations :

$$C = \alpha AB^T + \alpha BA^T + \beta C \quad \text{et} \quad C = \alpha A^T B + \alpha B^T A + \beta C. \tag{6.12}$$

Elle constitue un exemple très intéressant de l'utilisation de certaines routines de calcul et de communication présentées précédemment. Des produits de matrices de la forme $A^T B$ ou AB^T sont nécessaires. Remarquons que, comme nous avons $AB^T = (BA^T)^T$, l'opération de mise à jour de rang 2k peut être calculée avec un produit de matrices, une transposition et une addition [Els90]. L'algorithme général pour calculer la mise à jour de rang 2k $C = \alpha AB^T + \alpha BA^T + \beta C$ sur une machine parallèle à mémoire distribuée est donné en 6.4. La sous-matrice B_{ij} doit être chargée sur les processeurs dont les coordonnées sont (j, i), pour ne pas avoir à transposer cette matrice pour le premier produit. La transposition interne est faite de manière implicite.

`(I)`	`charger les matrices` A, B^T `et` C
`(II)`	`calculer` $T = \alpha AB^T$
`(III)`	`calculer` $C = \beta C + T$
`(IV)`	`transposer` T
`(V)`	`calculer` $C = C + T^T$
`(VI)`	`décharger la matrice` C

ALG. 6.4 - Algorithme général pour la mise à jour de rang 2k d'une matrice symétrique.

6.4.1 Algorithme sur un tore

Nous utilisons l'algorithme décrit précédemment (Algorithme 6.4) pour effectuer cette mise à jour. Notons que l'on pourrait, bien entendu, trouver un meilleur algorithme en tenant compte notamment de la symétrie de la matrice et surtout de la largeur des matrices A et B. Nous découpons l'algorithme en quatre phases :

La phase 1 correspond à (I) : Nous chargeons le réseau avec les trois matrices A, B^T et C.

La phase 2 correspond à (`II` et `III`) : Il s'agit d'un produit de matrices $T = \alpha AB^T$.

La phase 3 correspond à (`IV`) et (`V`) : Nous recouvrons la transposition de la matrice T avec son addition avec C.

La phase 4 correspond à (VI) : Il s'agit de la récupération de la matrice résultat sur l'hôte.

6.4.2 Algorithmes utilisant un réseau reconfigurable

Sur un réseau reconfigurable, nous découpons également l'algorithme en quatre phases (Algorithme 6.5), chacune d'entre elles correspondant à une phase de reconfiguration : deux communications globales utilisant le RCP, une phase utilisant un tore 2D et un réseau direct de transposition par blocs. La différence avec l'algorithme sur un tore décrit précédemment est que nous recouvrons la transposition et la seconde addition en utilisant une méthode macro-pipeline.

Premièrement, les matrices sont chargées en utilisant l'algorithme de diffusion personnalisée avec RCP. Ensuite, $T = \alpha AB^T$ et $C = \beta C + T$ sont calculées sur un tore 2D. Puis, on dispose

```
Phase 1 : [Algorithme RCP de diffusion personnalisée]
   Distribution des matrices A, B^T et C
Phase 2 : [Topologie = tore 2D]
   Calculer T = αAB^T avec algorithme de Cannon
   Calculer C = βC + T
Phase 3 : [Topologie = réseau direct]
   en parallèle faire
      échanger les sous-matrices T
      calculer C = C + T^T
   fin
Phase 4 : [Algorithm RCP de rassemblement]
   Récupération de la matrice C sur l'hôte
```

ALG. 6.5 - Mise à jour de rang 2k sur un réseau reconfigurable

le réseau de processeurs en un réseau direct pour la transposition. Grâce à une méthode macro-pipeline, nous recouvrons la transposition de T et l'addition avec C ($C = C + T^T$). La dernière opération consiste à regrouper la matrice sur l'hôte en utilisant l'algorithme inverse de la diffusion personnalisée, toujours avec l'algorithme RCP.

Etude de complexité

Nous donnons à présent une analyse de complexité pour chaque phase. Nous supposons que nous avons P processeurs et des matrices $N \times N$. Pour les phases 1, 2 et 4, nous utilisons les complexités décrites dans les chapitres précédents pour ces mêmes algorithmes. La somme de deux matrices distribuées par blocs est évidente.

Phase 1 Nous utilisons l'algorithme de distribution RCP d'un message de taille N^2/P.

Phase 2 Le coût est le même que pour le produit de matrices de Cannon sans pré-traitement sur un tore 2D.

Phase 3 Pour cette phase, nous utilisons une variante de l'algorithme de transposition macro-pipeline décrit précédemment (Chapitre 4, Paragraphe 4.2.9). Le coût total de l'opération $C = C + T^T$ est de $(N^2/P)\tau_{add}$.

Nous découpons les sous-matrices en μ paquets de taille $\frac{N^2}{P\mu}$. Ce qui nous donne :

$$T_{ph\ 3} = \beta + \frac{N^2}{P\mu}\frac{\tau}{k} + (\mu - 1)\max\left(\beta + \frac{N^2}{P\mu}\frac{\tau}{k}, \frac{N^2}{P\mu}\tau_{add}\right) + \frac{N^2}{P\mu}\tau_{add}. \tag{6.13}$$

Nous pouvons à présent comparer les coûts de communication et de calcul dans le terme en max

$$\beta + \frac{N^2}{P\mu}\frac{\tau}{k} \geq \frac{N^2}{P\mu}\tau_{add} \Rightarrow \mu \geq \frac{N^2\,(k\tau_{add} - \tau)}{kP\beta}. \tag{6.14}$$

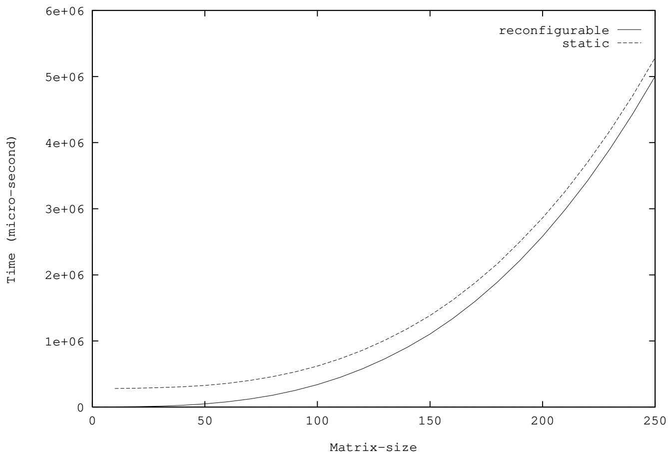

FIG. 6.11 - Mise à jour de rang 2k sur 25 processeurs (tore 2D et réseau reconfigurable) sur Tnode.

Sur notre machine cible, nous avons un recouvrement total des calculs par les communications, ce qui nous donne un coût pour cette phase égal à :

$$T_{ph\ 3} = \mu\beta + \frac{N^2}{P}\frac{\tau}{k} + \frac{N^2}{P\mu}\tau_{add}. \tag{6.15}$$

Nous avons $\mu_{opt} = \sqrt{\frac{N^2\tau_{add}}{P\beta}}$, soit pour cette phase

$$T_{ph\ 3} = 2\frac{N}{\sqrt{P}}\sqrt{\tau_{add}\beta} + \frac{N^2}{P}\frac{\tau}{k}. \tag{6.16}$$

Phase 4 Nous avons le même coût pour la distribution mais avec une taille de message égale à la taille d'une sous-matrice.

6.4.3 Expériences sur le Tnode

Les Figures 6.11 et 6.12 donnent les temps et efficacité de l'algorithme de mise à jour de rang 2k sur le Tnode. La solution utilisant la reconfiguration reste plus intéressante grâce au gain dans la transposition. Asymptotiquement, le coût du produit de matrices, égal dans les deux cas, l'emporte. La modularité de cet algorithme fait qu'il se prête tout naturellement à une décomposition par phases et à une implémentation sur un réseau avec reconfiguration quasi-dynamique. Remarquons que le coût de reconfiguration peut être recouvert en partie dans la phase 2.

6.4.4 Utilisation des routines LOCCS

Le recouvrement de la transposition et de l'addition peuvent être écrits à l'aide de la routine LOCCS d'échange entre deux processeurs `loccs_exchange`. La tâche job_a qui doit être exécutée après réception est le rangement de la matrice en mémoire. La partie 3 de ces routines s'écrirait alors comme sur l'algorithme 6.6.

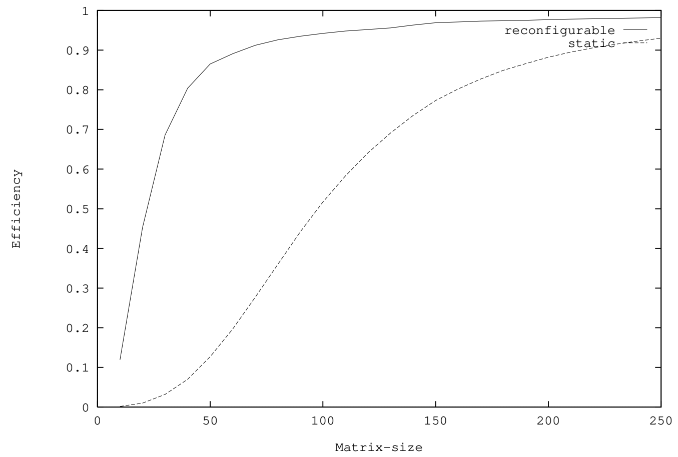

FIG. 6.12 - Efficacité de la mise à jour de rang 2k sur 25 processeurs (tore 2D et réseau reconfigurable) sur Tnode.

```
exchange.partner1 = me
exchange.partner2 = autre
job_b.job = NULL
job_a.job = rangement
job_a.param = NULL
loccs_exchange(me, exchange, buffer, ν, job_b, job_a)
```

ALG. 6.6 - Phase 3 de la mise à jour de rang 2k utilisant les LOCCS

6.5 Factorisation LU

Dans cette section, nous effectuons une analyse très précise de la factorisation LU sur une machine parallèle à mémoire distribuée. Nous étudions dans un premier temps une distribution des données de type entrelacée par colonnes et l'algorithme de type kji ou "right looking" avec pivotage partiel. Dans un second temps, nous étudions la même factorisation avec, cette fois, une distribution circulaire par blocs. Pour la première distribution, nous présentons premièrement l'algorithme avec des communications synchrones et nous comparons les temps d'exécution sur un anneau et un réseau direct. Dans un second temps, nous introduisons la possibilité de recouvrements calculs/communications grâce à l'utilisation de communications asynchrones. Nous présentons une analyse très fine des temps d'attente et étudions leur influence. Nous introduisons différents cas en fonction des paramètres de la machine qui correspondent à des chemins critiques différents. Les complexités sont corroborées par des expériences sur iPSC/860 et Paragon. Pour la seconde distribution, nous présentons une amélioration de la méthode par blocs qui permet, grâce à un pipeline du pivotage, de réduire le coût des communications tout en gardant un grain de calcul élevé. Ce résultat est utilisé dans l'implémentation du benchmark LINPACK sur Paragon.

6.5.1 Elimination de Gauss

L'élimination de Gauss peut être utilisée pour la résolution d'un système d'équations du type $Ax = b$. Ce processus transforme la matrice A en une matrice triangulaire avec une mise à jour de la partie droite de l'équation, de telle manière que la solution du système est immédiate à l'aide d'une résolution d'un système triangulaire. La factorisation LU utilise le même algorithme et transforme la matrice A en deux matrices L et U, où $A = LU$ et L et U sont deux matrices triangulaires respectivement inférieure et supérieure. De ce fait, plusieurs systèmes peuvent être résolus à l'aide de deux résolutions de systèmes triangulaires, $Ly = b$ et $Ux = y$.

Il y a plusieurs versions de factorisation LU selon le rangement des boucles internes [Rob90, GL89]. Nous étudions la forme kji où "Right-Looking", qui est la plus facile d'implémentation sur une machine parallèle à mémoire distribuée (puisque la matrice est distribuée par colonne, le pivotage partiel est effectué simplement à l'intérieur de chaque processeur sans communication [PBKP92]).

```
pour k = 0 à N − 2
   Scale: exécuter la tâche S_k
   pour toutes les colonnes j ≥ k + 1
      Mise à jour (Update): exécute la tâche U_kj
   finpour
finpour
```

ALG. 6.7 - Algorithme séquentiel kji pour l'élimination de Gauss.

- La tâche S_k représente la recherche de pivot, l'échange des éléments dans une colonne et la mise à jour de la colonne d'élimination.
- La tâche U_{kj} consiste dans l'échange des éléments pivots et la mise à jour de la colonne restante j.

Tâche S_k	Tâche U_{kj}
recherche_ pivot(k, $ipvt(k)$)	échanger(k, $ipvt(k)$)
échanger(k, $ipvt(k)$)	pour $i = k + 1$ to $N - 1$
$c := \frac{1}{a_{kk}}$	$a_{ij} = a_{ij} - a_{ik} * a_{kj}$
pour $i := k + 1$ à $N - 1$	finpour
$a_{ik} := a_{ik} * c$	
finpour	

ALG. 6.8 - Algorithmes pour les tâches S_k et U_{kj} de l'élimination de Gauss.

(Remarque: les multiplieurs L sont sauvés dans le tableau A à la place des éléments qui deviendraient zéro dans la tâche S_k.)

6.5.2 Algorithme parallèle pour la factorisation LU avec une distribution entrelacée par colonnes

Des versions parallèles de l'algorithme de factorisation LU sont décrites dans [Saa86a, RTV89, Rob90]. En rapport avec la topologie sous-jacente, la différence entre les méthodes est la manière avec laquelle la colonne pivot est envoyée à tous les processeurs. Deux méthodes très connues pour diffuser sont la diffusion à l'aide d'arbres de recouvrement minimaux et l'algorithme anneau pipeline (diffusion unidirectionnelle dans l'anneau) [Saa86a, RTV89, Rob90].

```
me = my_id()
pour k = 0 à N − 2
   si (alloc(k) == me)                /* Je possède la col. pivot */
      S_k                                  /* scale la col. pivot */
   C_k                     /* diffusion pipeline de la col. pivot k */
   pour tous les j ≥ k et alloc(j) == i
      U_kj                        /* mise à jour des col. internes j */
   finpour
finpour
```

ALG. 6.9 - Algorithme parallèle de factorisation LU sur anneau pipeline.

Pour la décomposition LU parallèle à l'aide de l'algorithme kji, nous avons sélectionné l'ensemble de tâches de calcul suivant : S_k est le scaling de la colonne pivot, U_{kj}^p est la mise à jour des colonnes présentes sur un processeur p (suivant la décomposition de type circulaire), et C_k^p est la diffusion de la colonne k sur le processeur P utilisant les échanges de messages (signalons que C_k^p peut être un envoi, une réception suivie d'un envoi ou d'une réception, suivant la stratégie de diffusion utilisée et le numéro de processeur). De ce fait, la factorisation LU d'une matrice $N*N$ est constituée de $N-1$ tâches S (S_0 à S_{N-2}), $(N-P)(P-1)+\frac{P(P-1)}{2}$ tâches C_j^p et $(N-P)P+\frac{P(P-1)}{2}$ tâches U_j^p.

```
Tâche C_k
   message=concaténation(ipvt(k),A[k,k+1 : N − 1])
   diffusion_pipeline(message,n − k)
```

ALG. 6.10 - Algorithme de la tâche C_k de l'élimination de Gauss.

Modèles

Dans la suite du paragraphe, β_{scal} et τ_{scal} sont les paramètres pour l'inversion d'un élément pivot, la recherche du maximum des vecteurs, l'échange des deux éléments et la multiplication du vecteur en utilisant la routine des BLAS de niveau 1 DSCAL [LHKK79].

Nous définissons β_{upd} et τ_{upd} comme les paramètres de coût des échanges des éléments pivots et la multiplication et additions entre des vecteurs (routine DAXPY des BLAS de niveau 1). Ces paramètres de la machine cible sont déterminés expérimentalement sur nos machines et les performances en

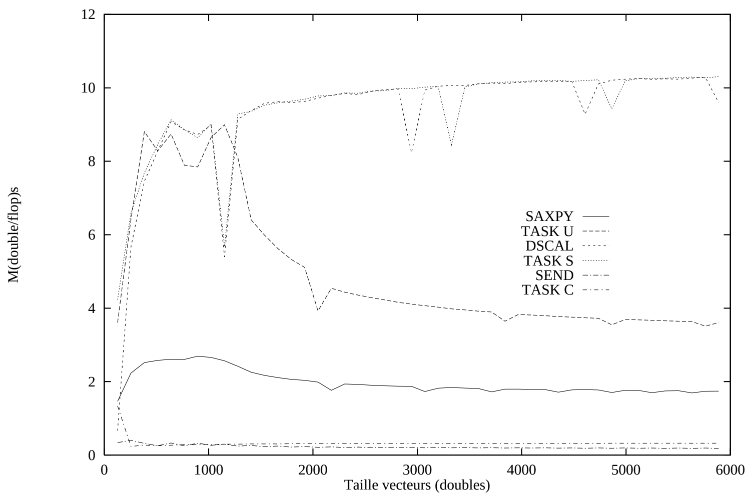

FIG. 6.13 - Performances des tâches sur la machine Intel iPSC/860 en fonction de la taille des vecteurs pour les tâches (U (opérations `DAXPY` et échanges) en Mflops, S (opérations `DSCAL`, recherche du maximum, échange) en Mflops et C (opération d'envoi/réception bloquante) en Mmots par seconde) comparées aux performances des routines `DAXPY` et `DSCAL` et de communication.

Mflops et Mdoubles sont données pour l'iPSC/860 sur la Figure 6.13 et pour la Paragon sur la Figure 6.14. Remarquons que la recherche du pivot et les échanges sont très coûteux (la tâche S est très loin des performances en crête de la routine `DSCAL`) et que les machines cibles donnent leurs meilleures performances pour des tailles de vecteurs autour de 1000 pour l'iPSC/860 et 1800 pour la Paragon (probablement à cause des gestions mémoire et caches).

Les paramètres des deux machines cibles sont donnés dans le tableau 6.6.

Machine	τ_{upd} $(\mu s/double)$	τ_{scal} $(\mu s/double)$	τ_{com} $(\mu s/double)$
iPSC/860	0.097	0.57	5
Paragon	0.058	0.32	2

TAB. 6.6 - Valeurs des paramètres machines pour la factorisation LU.

Suivant la notation introduite dans [LKK83], où une tâche est une unité indivisible de calcul ou de communication, la relation de précédence binaire entre les tâches est notée $<$ (Cette relation est irréflexive, antisymétrique et transitive). Si A et B appartiennent à l'ensemble des tâches, alors $A < B$ veut dire que la tâche A doit terminer son exécution avant le début de la tâche B.

Travaux reliés

De nombreuses méthodes ont été proposées pour la factorisation LU (voir [PBKP92] et les travaux de [Cap87, CRT89, CTV87, CT93b, Rob90, Saa86b, Tou89]). Par exemple, [CG87] présente un pivotage partiel et l'équilibrage des charges dans une méthode par ligne avec une résolution de

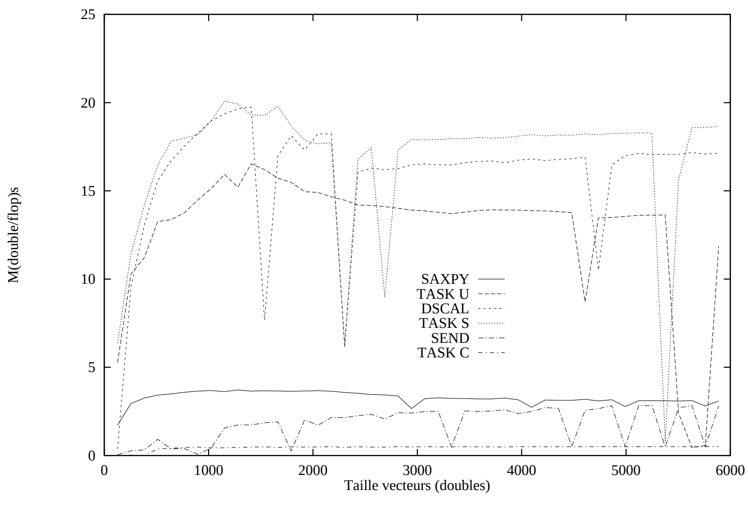

FIG. 6.14 - Performances des tâches sur la machine Intel Paragon en fonction de la taille des vecteurs.

système immédiate associée. Mais [LC89] montre que la résolution de systèmes triangulaires peut avoir les mêmes performances avec un rangement par colonnes. Dans [DO90, RTV89], la distribution par panneaux de colonnes est prouvée être très efficace car elle produit un bon équilibrage de la charge entre les calculs et les communications et, en pratique, cette méthode a donné de bons résultats [DO90, RT88]. La méthode par blocs est introduite dans [CDW92], et [BDL91, DGW92] montrent que les coûts de communication peuvent être réduits en utilisant une distribution circulaire par blocs.

Pour comprendre ces différentes méthodes et être capables de prédire les performances sur une machine donnée, nous devons effectuer une analyse précise. Dans ce premier paragraphe, nous nous intéressons plus particulièrement à des méthodes par colonnes. Nous montrons qu'un recouvrement pratiquement total des communications peut être obtenu. De ce fait, nous obtenons des performances pratiquement optimales avec une décomposition simple circulaire par colonnes. Cette méthode de recouvrement des communications peut être étendue aux méthodes par blocs.

Communications synchrones

Nous commençons par une analyse de l'algorithme sans recouvrement. Deux topologies sont comparées : un réseau complet et un anneau. L'algorithme sur anneau sera noté PR (*Pipeline Ring*), et celui sur le tore PC (*Pipeline Complete*).

Sur le réseau complet, avec notre modèle quand aucun recouvrement n'est autorisé, le chemin critique de la décomposition LU est donné par les contraintes de précédence de l'exécution séquentielle et est décrit par le diagramme de Gantt de la Figure 6.15, où les arcs du chemin représentent la relation $<$ entre les tâches.

Le chemin critique suit la séquence suivante :

$$S_0 < C_0^0 < U_0^1 S_1 < C_1^1 < U_1^2 S_2 < C_2^2 < U_2^3 \ldots S_{N-2} < C_{N-2}^{P-2} < U_{N-2}^{P-1}$$

En suivant ce chemin critique, nous calculons le temps d'exécution.

Proposition 1 *Le temps total d'exécution de l'algorithme PC sans recouvrements des communications et des calculs est donné par*

$$T_{pipe}^{complet} = T_{scal}^{c} + T_{com}^{c} + T_{upd}^{c}. \tag{6.17}$$

Où,

$$T_{scal}^{c} = \sum_{i=0}^{N-2} (\beta_{scal} + (N-i-1)\tau_{scal}) = (N-1)\beta_{scal} + \frac{N(N-1)}{2}\tau_{scal} \tag{6.18}$$

$$T_{com}^{c} = \sum_{i=0}^{N-2} (\beta_{com} + (N-i)\tau_{com}) = (N-1)\beta_{com} + \frac{(N+2)(N-1)}{2}\tau_{com} \tag{6.19}$$

$$\begin{aligned} T_{upd}^{c} &= \sum_{j=0}^{\frac{N}{P}-1} \left[\sum_{k=1}^{P} \left(\beta_{upd} + (\frac{N}{P} - j)(N - (j \times (P-1) + k))\tau_{upd} \right) - (\beta_{upd} + \frac{N^2}{P}\tau_{upd}) \right] \\ &= (N-1)\beta_{upd} + \left(\frac{N^3}{3P} + \frac{N^2}{4} - \frac{3N^2}{4P} + \frac{N}{4} - \frac{NP}{12} \right) \tau_{upd} \end{aligned}$$

De ce fait,

$$\begin{aligned} T_{pipe}^{complet} &= (N-1)(\beta_{upd} + \beta_{scal} + \beta_{com}) + \frac{N(N-1)}{2}(\tau_{scal} + \tau_{com}) \\ &+ \left(\frac{N^3}{3P} + \frac{N^2}{4} - \frac{3N^2}{4P} + \frac{N}{4} - \frac{NP}{12} \right) \tau_{upd} \end{aligned} \tag{6.20}$$

Remarquons que pour les autres topologies, le chemin critique doit suivre la décomposition *alloc* de $C_j^{alloc(j)}$ et $U_j^{alloc(j)}$ entre les tâches S_j et S_{j+1}.

La Figure 6.15 montre le chemin critique de l'algorithme pipeline de factorisation LU correspondant à l'anneau. Cette topologie introduit de nouveaux arcs de précédence entre les tâches C_j^{i-1} (envoi) et C_j^{i} (réception). Suivant les paramètres de la machine cible, ces dépendances introduisent des temps d'attente dans le chemin critique. Si nous examinons les grossissements dans la Figure 6.17 et 6.18, on peut distinguer deux cas : $T_{scal(k)} > T_{com(k-2)}$ (cas 1) et son inverse (cas 2) :

A l'étape k, on peut déterminer dans quel cas on se trouve :

$$\begin{aligned} T_{scal(k)} &> T_{com(k-2)} \\ \beta_{scal} + (N-(k-1))\tau_{scal} &> \beta_{com} + (N-(k-2))\tau_{com} \\ k \quad &> ou < \quad \frac{\beta_{com} - \beta_{scal} + (N+2)\tau_{com} - (N-1)\tau_{scal}}{\tau_{com} - \tau_{scal}} \end{aligned}$$

suivant le signe de $\tau_{com} - \tau_{scal}$. Généralement, et c'est vrai sur nos machines cibles, $\tau_{com} > \tau_{scal}$. De ce fait, la formule ci-dessus, est équivalente à $k > (N+2) + o(1)$, ce qui est toujours faux. De ce fait, sur la plupart des machines, l'analyse de complexité correspondra au cas 2.

Le temps d'exécution de l'algorithme PR est calculé en suivant le chemin critique et en prenant en compte les temps d'attente (remarquons que nous ne trouvons pas le même temps total que dans [PBKP92] puisque nous tenons compte des temps d'attente introduits dans la stratégie pipeline). Comme le pivot a besoin d'être transmis au processeur suivant (qui contient la nouvelle

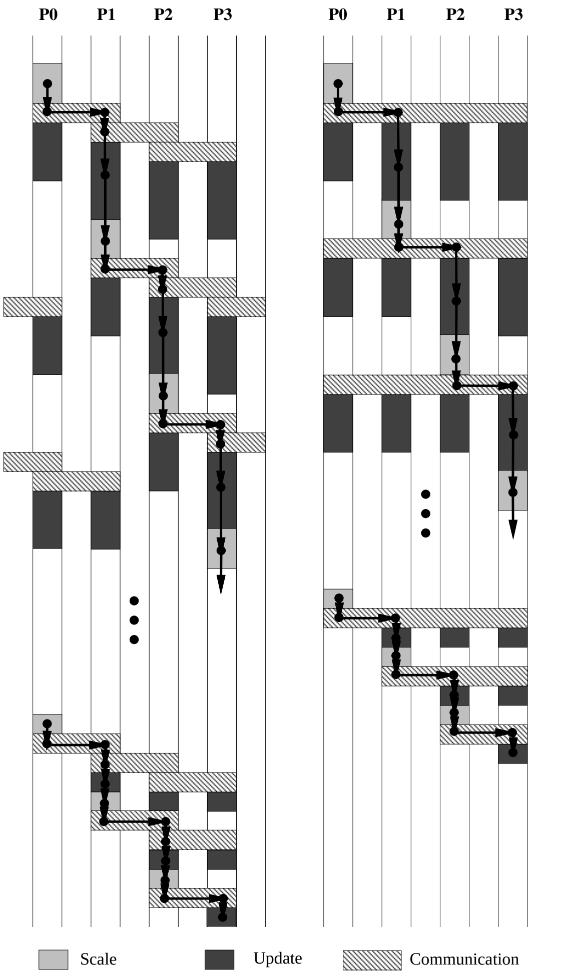

FIG. 6.15 - Diagramme de temps pour les algorithmes synchrones (à gauche sur l'anneau, à droite sur le réseau complet).

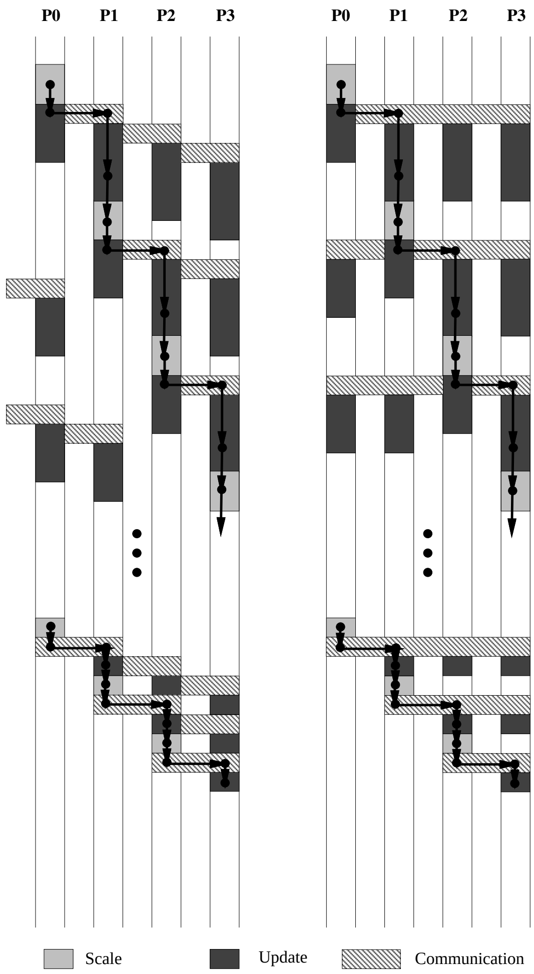

FIG. 6.16 - Diagramme de temps des algorithmes avec communications asynchrones (anneau sur la gauche, réseau complet sur la droite).

colonne pivot grâce à notre distribution circulaire) le temps T_{com} est augmenté d'un facteur deux. Le temps d'attente T_{idle} est exprimé comme une fonction de T_{scal} et T_{com}.

Proposition 2 *Le coût total de l'algorithme PR sans recouvrement des communications et des calculs est donné par l'équation 6.21.*

$$T_{pipe}^{ring} = T_{scal}^{r} + T_{com}^{r} + T_{upd}^{r} + T_{idle}^{r} \tag{6.21}$$

Où,

$$T_{scal}^{r} = T_{scal}^{c} \tag{6.22}$$

$$T_{com}^{r} \;\; = \;\; \sum_{i=0}^{N-2} 2(\beta_{com} + (N-i)\tau_{com}) = 2 * T_{com}^{c}$$

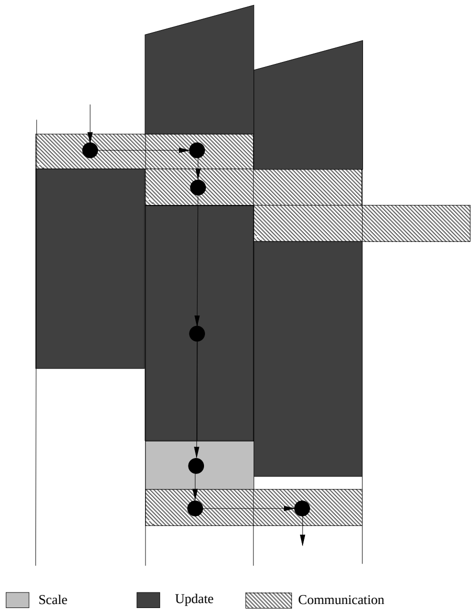

FIG. 6.17 - Grossissement sur les temps d'attente dans le chemin critique (cas 1).

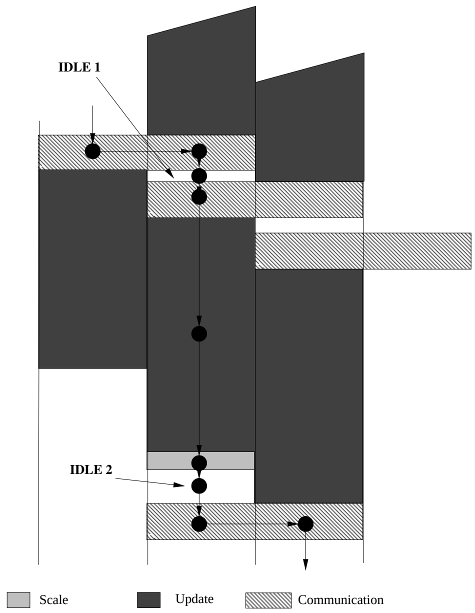

FIG. 6.18 - Grossissement sur les temps d'attente dans le chemin critique (cas 2).

$$T^r_{upd} = T^c_{upd} \tag{6.23}$$

$$\begin{aligned}
T^r_{idle} &= \sum_{i=1}^{N-3} \left((C^{i-1} - S^i)^+ + C^{i-1} - C^i\right) \\
&= \sum_{i=1}^{N-3} \left((C^{i-2} - S^i)^+\right) \\
&= \sum_{i=1}^{N-3} \left((\beta_{com} + (N-i)\tau_{com} - (\beta_{scal} + (N-i-1)\tau_{scal}))^+ + \tau_{com}\right) \\
&= (N-3) * ((\beta_{com} - \beta_{scal})^+ + \tau_{com}) + \frac{(N+2)(N-3)}{2}\tau_{com} - \frac{N(N-3)}{2}\tau_{scal}
\end{aligned} \tag{6.24}$$

où $(x)^+$ est la fonction qui retourne x si $x > 0$ et 0 sinon. Nous obtenons donc :
Cas 1 *$(T_{scal(k)} > T_{com(k-2)})$:*

$$T^{ring}_{pipe} = T^{complete}_{pipe} + (N-1)\beta_{com} + \frac{N(N-1)}{2}\tau_{com} \tag{6.25}$$

Cas 2: *$(T_{scal(k)} \leq T_{com(k-2)})$:*

$$T^{ring}_{pipe} = T^{complete}_{pipe} + (2N-4)\beta_{com} + (N^2 - N - 3)\tau_{com} - (N-3)\beta_{scal} - \frac{N(N-3)}{2}\tau_{scal} \tag{6.26}$$

$$= T^{ring}_{pipe}(case1) + (N-3)\beta_{com} + (N^2/2 + 3N/2 - 3)\tau_{com} - (N-3)\beta_{scal} - \frac{N(N-3)}{2}\tau_{scal}$$

Communications Asynchrones

Dans ce paragraphe, nous présentons une amélioration du temps d'exécution, basée sur la réduction du temps de communication grâce à son recouvrement partiel par des calculs. Remarquons que ce recouvrement est réalisé à l'intérieur même d'un processeur; cela ne correspond pas aux recouvrements entre les processeurs décrits dans [PBKP92] qui sont l'effet même de l'algorithme pipeline sur anneau. Nous verrons que le temps d'exécution sur un réseau complet peut être obtenu avec l'algorithme pipeline sur un anneau. De ce fait, le temps d'exécution reste le même quelle que soit la topologie complexe utilisée (l'anneau, le réseau complet, l'hypercube, ...).

La Figure 6.16 montre que le chemin critique sur le réseau complet n'est pas changé en utilisant les communications asynchrones. La légère amélioration qui apparaît au début de la tâche U^k_k sur le processeur pivot n'affecte pas le temps d'exécution du chemin critique. Cela est dû au fait que la tâche C doit attendre que la réception soit terminée pour pouvoir commencer les tâches U.

Proposition 3 *Le coût total de l'algorithme PC avec des envois asynchrones (recouvrement des calculs et des communications) est donné par*

$$T^{complete}_{pipe-async} = T^c_{scal} + T^c_{com} + T^c_{upd} \qquad (6.27)$$

Où $T^c_{scal}, T^c_{com} et T^c_{upd}$ sont les mêmes que dans la section synchrone.

Quand les communications asynchrones sont disponibles, le processeur qui contient la colonne pivot suivante n'est pas retardé par l'envoi de la colonne pivot courante. De ce fait, il commence ses mises à jour dès que possible (tâches U). Comme les données sont équidistribuées grâce à une distribution circulaire, les tâches U_{kj} sont pratiquement de la même durée, de ce fait, les communications sur l'anneau ne peuvent pas perturber le temps d'exécution du chemin critique, comme montré sur la Figure 6.19.

Nous suivons le chemin critique de la Figure 6.16 pour déterminer le temps total d'exécution de l'algorithme. Cela donne les résultats suivants :

Proposition 4 *Le coût total de l'algorithme PR avec des envois asynchrones (recouvrement des calculs et des communications) est donné par l'équation 6.28.*

$$T^{ring}_{pipe,asynchrone} = T^r_{scal} + T^r_{com} + T^r_{upd} \qquad (6.28)$$

où T^r_{scal} et T^r_{upd} sont les mêmes que dans le cas synchrone et T^r_{com} est la moitié de celle synchrone. De ce fait,

$$T^{ring}_{pipe,asynchrone} = T^{complete}_{pipe,asynchrone} \qquad (6.29)$$

Notons, que nous supposons que $T^r_{com} << T^r_{upd}$ car $T^r_{com} = O(N^2)$ tandis que $T^r_{upd} = O(N^3)$. Remarquons que, sur notre machine cible, les performances sont telles que pour une taille $N > 3$ $T^r_{com} << T^r_{upd}$.

Il est remarquable que, d'un point de vue analytique, dès que l'architecture cible est capable d'effectuer des communications asynchrones, l'algorithme classique de décomposition LU sur un anneau s'exécute aussi bien que le meilleur temps d'exécution de cet algorithme sur n'importe quelle topologie.

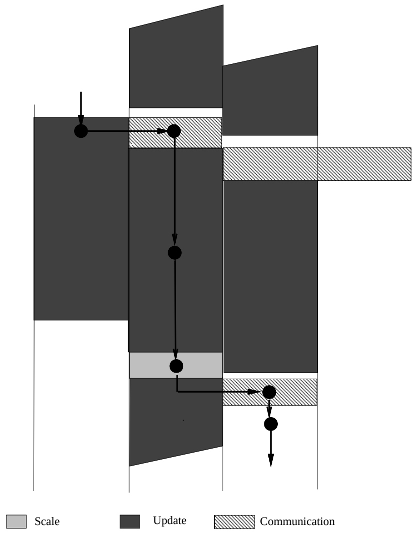

FIG. 6.19 - Zoom sur le chemin critique.

Expérimentations

Les expériences ont été effectuées sur les machines Intel iPSC/860 et Paragon en utilisant jusqu'à 64 processeurs. Pour les tests sur anneau, nous avons utilisé un anneau plongé dans l'hypercube. De manière évidente, nous ne sommes pas capables d'effectuer des expériences sur un réseau complet de processeurs. Nous l'avons simulé en utilisant les capacités de routage Circuit-Switching de la machine. Les problèmes de contention ainsi que le modèle 1-port de la machine ont augmenté les surcoûts des temps d'exécution et, de ce fait, les résultats sur le réseau complet ne sont pas satisfaisants.

Afin d'avoir de vraies communications synchrones, nous avons utilisé la fonction `sendrecv` de la bibliothèque Intel qui autorise un acquittement de la réception des messages.

Sur la Figure 6.20, nous donnons les temps d'exécution théoriques et expérimentaux de l'algorithme PR sur iPSC/860 avec des communications synchrones et sur la courbe 6.21 les même expériences avec des communications asynchrones. Les courbes expérimentales ont été obtenues à partir de nos formules de calcul de temps et des paramètres machine donnés par les tests des différentes tâches. Nos courbes théoriques sont très proches des courbes expérimentales, ce qui montre la validité de notre analyse. Remarquons également que pour les communications synchrones, nos temps théoriques sont pessimistes et que pour les communications asynchrones, ils sont plutôt optimistes.

Sur la courbe 6.22, nous avons comparé les expériences avec communications synchrones et asynchrones sur 8 et 16 processeurs. Le gain obtenu grâce au recouvrement des communications est léger mais bien présent.

La Figure 6.23 donne une comparaison entre les temps théoriques et expérimentaux sur 16 processeurs de la Paragon et la Figure 6.24 donne les temps d'exécution avec les communications asynchrones jusqu'à 64 processeurs. Les pics sont dus à des problèmes du système de la machine testée.

Nous pensons qu'une telle analyse peut prédire le comportement des algorithmes sur des machines stables pour d'autres nombres de processeurs. En utilisant les BLAS de niveau 1 assembleurs,

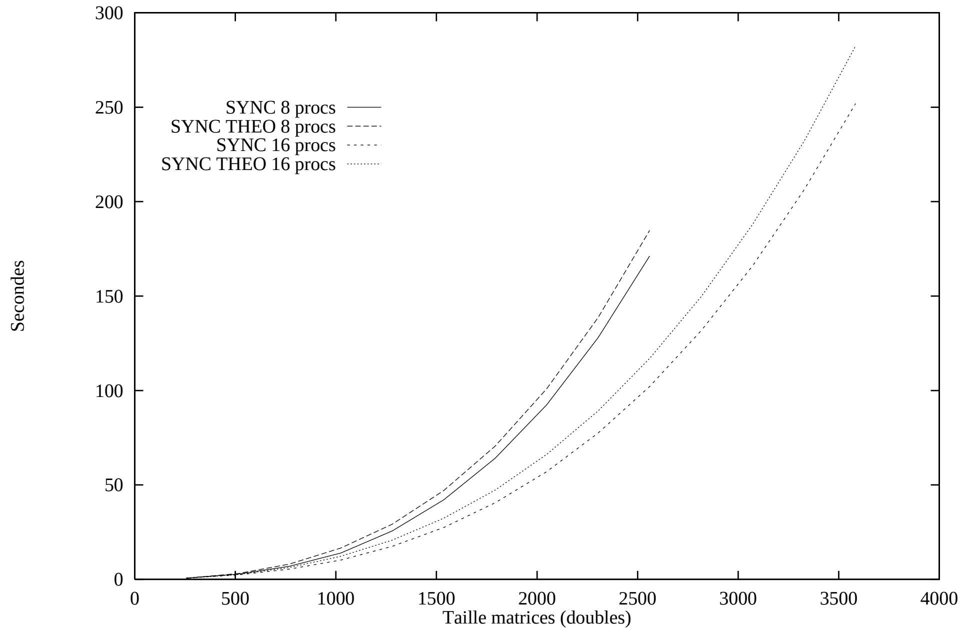

FIG. 6.20 - Temps d'exécution théoriques et expérimentaux sur iPSC/860 pour 8 et 16 processeurs avec communications synchrones.

les performances totales correspondent aux performances obtenues avec les tests sur les noyaux de calcul. Les meilleures performances obtenues ont été de 419 Mflops avec une matrice de $N = 7000$ sur 64 processeurs sur l'iPSC/860 et 471 sur la Paragon.

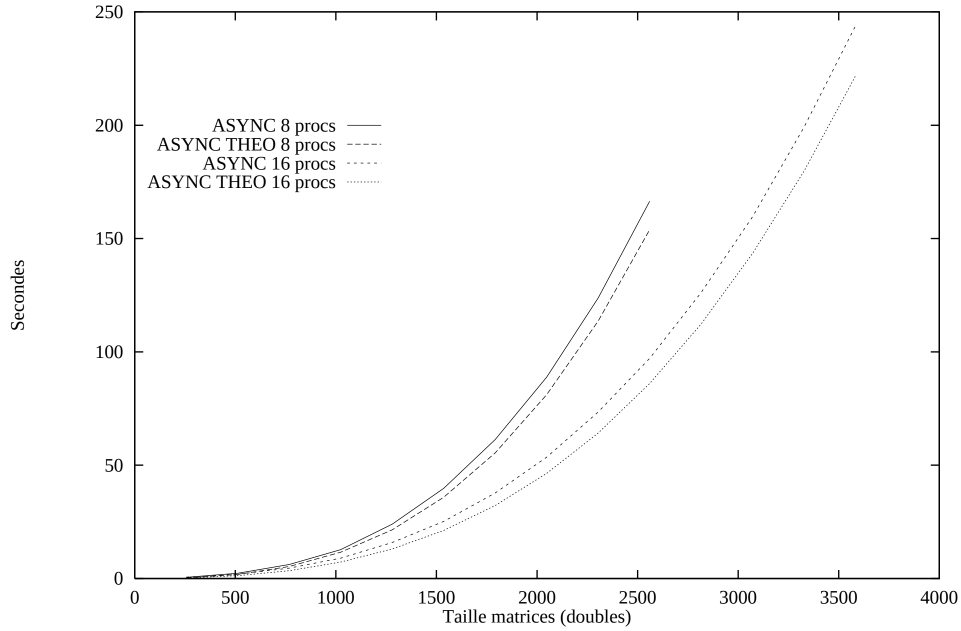

FIG. 6.21 - Temps d'exécution théoriques et expérimentaux sur iPSC/860 pour 8 et 16 processeurs avec communications asynchrones.

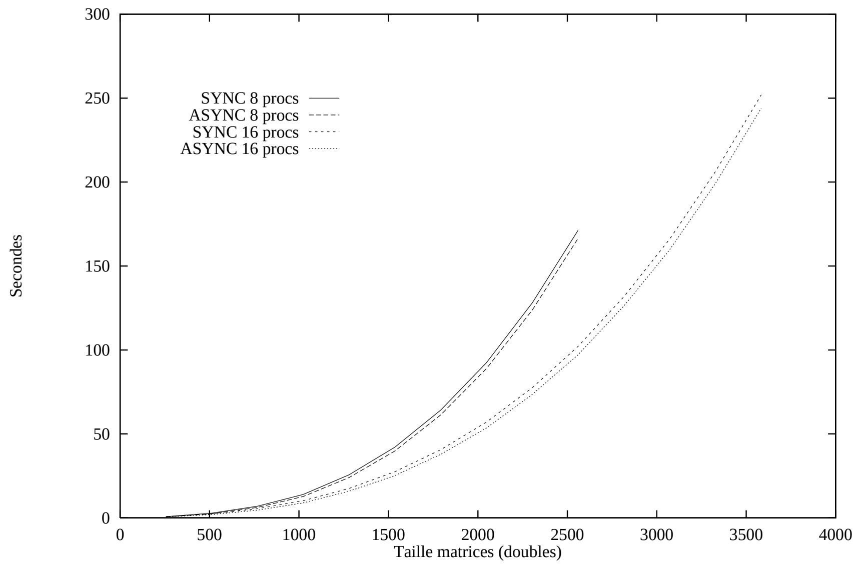

FIG. 6.22 - Temps d'exécution expérimentaux sur iPSC/860 pour 8 et 16 processeurs avec communications synchrones et asynchrones.

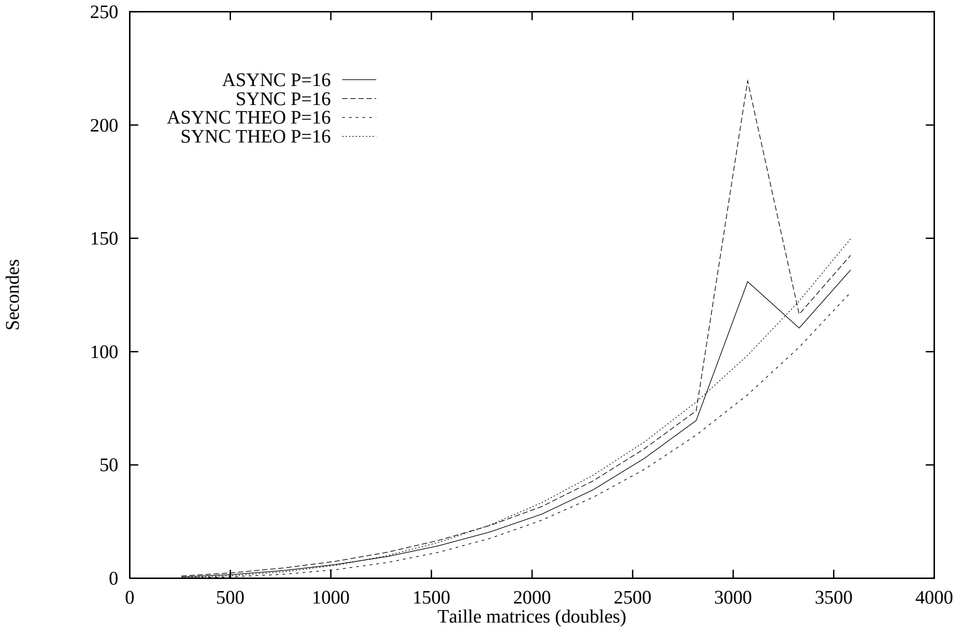

FIG. 6.23 - Temps d'exécution théoriques et expérimentaux sur Paragon jusqu'à 16 processeurs avec communications synchrones et asynchrones.

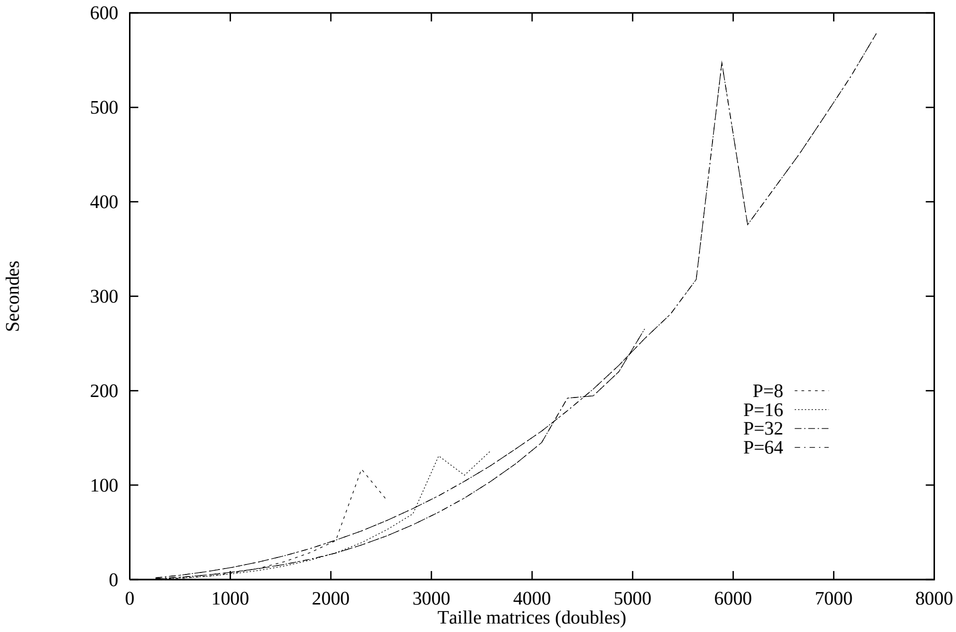

FIG. 6.24 - Temps d'exécution sur Paragon jusqu'à 64 processeurs.

6.5.3 Algorithme par blocs sur une grille

Le but de cette étude est d'améliorer un code existant de ScaLAPACK grâce à un pipeline sur le pivotage.

Nous allons préalablement décrire un algorithme séquentiel de factorisation LU par blocs. Comme nous l'avons vu en présentant les BLAS, les performances des BLAS de niveau 3 sont plus importantes que celles des BLAS de niveau 2 à cause de la mauvaise utilisation des mémoires hiérarchiques pour les seconds. La bibliothèque LAPACK a donc été codée entièrement en BLAS de niveau 3 et plus particulièrement en maximisant le nombre d'appels à la routine de produit de matrices, très efficace sur la plupart des nouveaux processeurs. La factorisation LU n'échappe pas à ce “recodage”.

$$\begin{array}{|c|c|}\hline \mathbf{A}_{00} & \mathbf{A}_{01} \\ \hline \mathbf{A}_{10} & \mathbf{A}_{11} \\ \hline \end{array} = \begin{array}{|c|c|}\hline \mathbf{L}_{00} & \mathbf{0} \\ \hline \mathbf{L}_{10} & \mathbf{L}_{11} \\ \hline \end{array} \bullet \begin{array}{|c|c|}\hline \mathbf{U}_{00} & \mathbf{U}_{01} \\ \hline \mathbf{0} & \mathbf{U}_{11} \\ \hline \end{array}$$

FIG. 6.25 - Factorisation LU de la matrice A par blocs.

Nous pouvons traiter certains blocs ensemble en les regroupant. Ainsi, sur la Figure 6.25, nous avons le nouveau découpage des matrices A, L et U [DW93]. Sur cette figure, si A est une matrice de taille $M \times N$ et que la taille des blocs carrés est r, A_{00} est une sous-matrice de taille $r \times r$, A_{01} est de taille $r \times (N-r)$, A_{10} de taille $(M-r) \times r$ et enfin A_{11} est de taille $(M-r) \times (N-r)$. L_{00} et L_{11} sont des matrices triangulaires inférieures avec des 1 sur la diagonale principale, et U_{00} et U_{11} des matrices triangulaires supérieures. Nous pouvons alors écrire :

$$\begin{aligned} L_{00}U_{00} &= A_{00} && (6.30)\\ L_{10}U_{00} &= A_{10} && (6.31)\\ L_{00}U_{01} &= A_{01} && (6.32)\\ L_{10}U_{01} + L_{11}U_{11} &= A_{11} && (6.33) \end{aligned}$$

On peut utiliser les équations 6.30 et 6.31 pour exécuter une factorisation LU sur le premier panneau de taille $M \times r$ (A_{00} et A_{10}). Ensuite, nous pouvons résoudre le système triangulaire de l'équation 6.32, ce qui nous donne U_{01}. Enfin, l'équation 6.33 est réécrite pour donner :

$$A'_{11} = A_{11} - L_{10}U_{01} \tag{6.34}$$

De ce fait, pour trouver L_{11} et U_{11}, il suffit de factoriser la matrice A'_{11} en réutilisant l'algorithme précédent [DW93]. Le nombre d'étapes total est donné par $K = \min(\lceil M/r \rceil, \lceil N/r \rceil)$. Après k étapes de l'algorithme précédent, nous nous retrouvons dans la configuration donnée par la Figure 6.26.

Les kr premières colonnes de la matrice A ont été factorisées et donc les kr premières colonnes de L et U sont connues. Nous allons donc nous occuper des parties B, C et E de la matrice A. Tout d'abord, le panneau B est factorisé avec pivotage pour donner le nouveau panneau de L (étape 1). Ensuite, à l'aide de la routine BLAS de niveau 3 `_TRSM` de résolution de système triangulaire, on résout le système $L_0U_1 = C$ pour donner U_1 (étape 2). Enfin, grâce à la routine BLAS de niveau

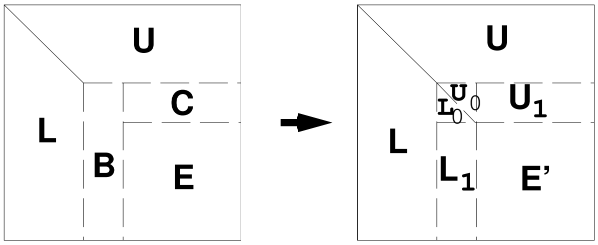

FIG. 6.26 - Etape $k+1$ de la factorisation LU de la matrice A par blocs.

3 _GEMM, on effectue une mise à jour de rang r sur la sous-matrice E ce qui donne $E' = E - L_1U_1$ (étape 3).

Nous allons maintenant décrire la parallélisation de cette opération sur une grille de processeurs telle qu'elle a été effectuée pour la bibliothèque ScaLAPACK [CDW92]. Elle utilise un pivotage partiel sur les lignes de la matrice et les échanges sont effectués explicitement pour garder une cohérence dans le rangement de la matrice finale. La distribution des matrices est circulaire par blocs. Notre but est d'optimiser l'algorithme existant pour atteindre les performances les plus intéressantes pour des matrices de petites tailles. Les performances pour les matrices de grandes tailles étant atteintes grâce au produit de matrices qui constitue la plus grande part du travail. L'algorithme est donné sur l'algorithme 6.11.

L'étape 1 ne nécessite le travail que d'une colonne de processeurs. Tout d'abord, le pivotage doit être effectué pour chacune des r colonnes (Figure 6.27).

Dans le cas de la distribution par colonnes que nous avons étudié [DDT93], l'étape 1 ne nécessite pas de communication car le processeur "pivot" possède toute la colonne pivot. Avec le rangement par blocs, ce n'est plus le cas, et des communications sont nécessaires pour la recherche du pivot, sa diffusion "verticale" et les échanges. De même, à la seconde étape, la résolution du système triangulaire nécessite des communications entre tous les processeurs d'une même ligne de la grille (diffusion de L_0). Si la distribution de la matrice était par lignes, aucune communication ne serait nécessaire. Par contre, une fois la sous-matrice triangulaire reçue, chaque processeur exécute sa propre résolution. A la troisième et dernière étape, deux diffusions sont nécessaires. Une diffusion sur toutes les lignes de processeurs pour L_1 et une diffusion sur toutes les colonnes pour U_1. Remarquons que les diffusions de L_1 et L_0 peuvent être groupées. Après ces deux étapes de diffusion, les mises à jour peuvent être effectuées en parallèle sur tous les processeurs sans communications.

Nous avons effectué une optimisation afin de réduire le coût des communications. Celle-ci permet de pipeliner les diffusions des pivots et des données du panneau L_1 en les découpant en blocs et en diffusant les blocs dès que possible. C'est-à-dire que, tout en gardant le grain de calcul des opérations BLAS de niveau 3 à son maximum, le grain du pivotage et réduit afin de profiter d'un pipeline des communications. Remarquons que cette optimisation peut s'écrire à l'aide des routines LOCCS.

6.5.4 Expérimentation sur Paragon

Nous avons utilisé deux codes différents pour cette implémentation sur Paragon. Le premier code est donné avec la Paragon et le second est celui du LINPACK parallèle. Nous avons comparé ces deux codes, ainsi que la version améliorée du code de LINPACK utilisant le pivotage pipeline.

```
pcol = q0
plig = p0
pour k = 0 à min(Mb, Nb) - 1 faire
    pour i = 0 à r - 1 faire                                        \
       si (q = pcol) alors                                          |
          trouver la valeur du pivot et sa position                 |
       finsi                                                        |
       diffuser les deux valeurs à tous les processeurs             |
       échanger les lignes pivots                                   > étape 1
       si (q = pcol) alors                                          |
         diviser les elts. sous-diag. de la col. r par le pivot     |
       finsi                                                        |
    finpour                                                         /

    si (p = plig) alors                                             \
       diffuser L0 à tous les procs de la ligne                     |
       résoudre L0U1 = C  /* _TRSM */                               > étape 2
    finsi                                                           /

    diffuser L1 à tous les procs de la ligne pivot                  \
    diffuser U1 à tous les procs de la colonne pivot                > étape 3
    mettre à jour E <- E - L1U1  /* _GEMM */                        /

    pcol = (pcol + 1)modQ
    plig = (plig + 1)modP
finpour
```

Alg. 6.11 - Algorithme parallèle pour la factorisation LU d'une matrice rangée de manière circulaire par blocs.

La Figure 6.28 donne les performances du benchmark LINPACK et de la factorisation LU associée pour différentes tailles de réseaux (1×4, 2×4, 4×4) et différentes tailles de matrices. Remarquons que la résolution des systèmes triangulaires ne coûte presque rien face à la factorisation. Asymptotiquement, l'utilisation d'un réseau carré de processeurs est la plus efficace. Pour des petites tailles de matrices (jusqu'à 700), une factorisation sur un anneau de 4 processeurs en utilisant des panneaux de colonnes est très performante. Cela est dû à une réduction des coûts de communication tout en gardant une utilisation de routines BLAS de niveau 3 très efficaces.

La Figure 6.29 montre l'importance de la forme du réseau sur les performances. Nous avons testé deux grilles de processeurs (2×4 et 4×2) et les performances les plus intéressantes sont obtenues pour la première grille. On peut attribuer cette différence significative au pivotage sur les colonnes qui est plus coûteux si la colonne de processeurs est grande.

Enfin, la Figure 6.30 montre le résultat de notre optimisation sur l'étape de pivotage/scaling sur les performances. Sur le benchmark LINPACK total, cette différence est significative puisqu'elle permet de gagner environ 10% sur les performances en Mflops.

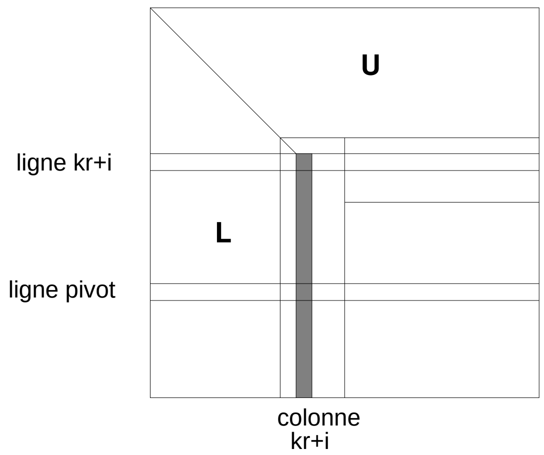

FIG. 6.27 - Pivotage à l'étape $k+1$ de la factorisation LU de la matrice A par blocs.

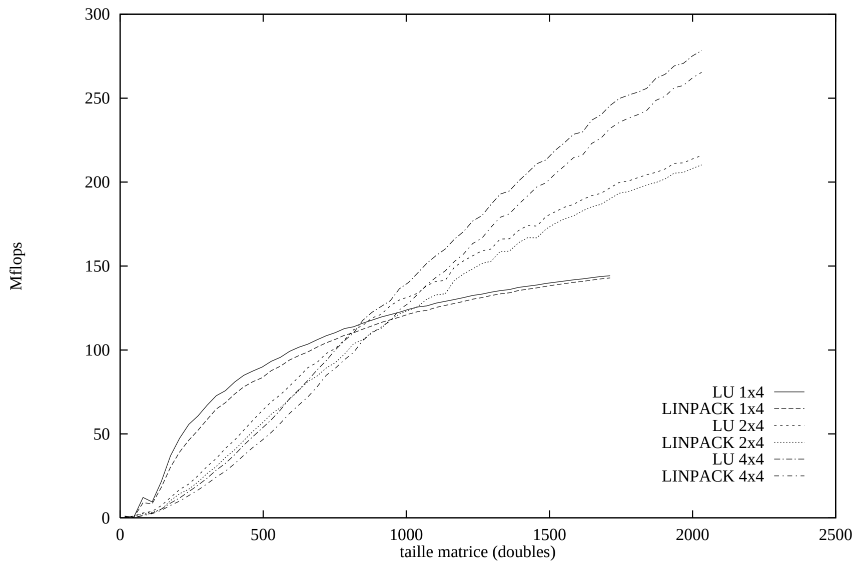

FIG. 6.28 - Factorisation LU et benchmark LINPACK sur Paragon pour différentes tailles de réseaux.

6.5.5 Conclusion

Nous avons vu que l'utilisation de routines de calculs locaux par blocs sont nécessaires pour l'obtention des meilleures performances. Néanmoins, l'étude des algorithmes sur des réseaux et avec des distributions de données plus simples permettent de déterminer des optimisations facilement applicables à des algorithmes plus complexes. Le gain obtenu grâce à l'utilisation du pivotage pipeline est significatif. Le code obtenu est toutefois très simple. L'utilisation des LOCCS pour une telle optimisation rendra le code portable.

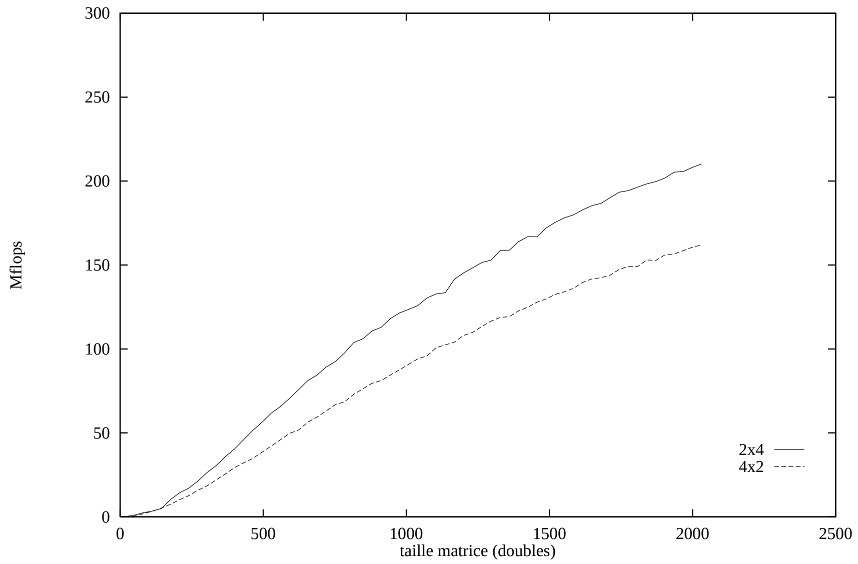

Fig. 6.29 - Benchmark LINPACK sur Paragon avec 8 processeurs.

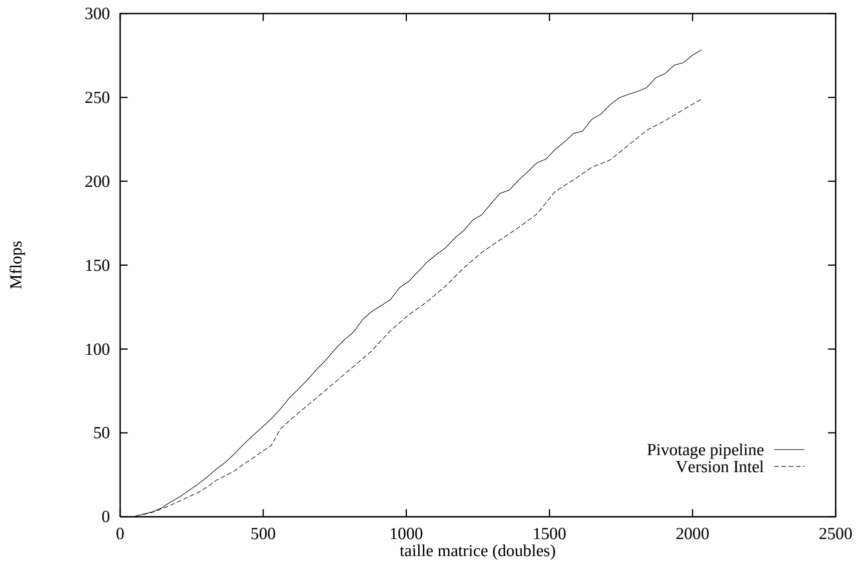

Fig. 6.30 - Optimisation du pivotage pour LINPACK sur la Paragon.

Chapitre 7

FFT multidimensionnelles

La transformée de Fourier est utilisée dans de nombreux domaines comme le traitement du signal, les mathématiques appliquées, le traitement d'images, ...En particulier, elle est utilisée pour calculer des convolutions, analyser des spectres, identifier les particularités d'une image [FJL+88], et par exemple calculer la transformée spectrale en résolvant des équations différentielles [FW93]. Les transformées de Fourier multi-dimensionnelles sont utilisées dans l'analyse d'images avec des techniques de filtrage, la dynamique des fluides, la résolution d'équations de Poisson [BP86, SBOP91], ...La transformée de Fourier discrète peut être calculée rapidement grâce à l'algorithme rapide (FFT). Le calcul des FFT mono et multi-dimensionnelles a été très étudié sur toutes sortes d'architectures parallèles comme par exemple des supercalculateurs à mémoire partagée [AGGM90] et vectoriels [AL88], des machines SIMD [BP86, JK92, ZRB+90] et MIMD [Ber92, Cha88, Chu88, DT89, GK90, HP91, SBOP91, Wal88, WWD91].

Dans ce chapitre, nous présentons notre contribution à la résolution de la Transformée de Fourier Rapide mono et multi-dimensionnelle à l'aide de méthodes de recouvrements efficaces des communications par les calculs [CD93]. Ce travail est basé principalement sur les études de Chu [Chu88] et Walker [Wal88, WWD91]. Dans ses articles, Chu présente des implémentations de FFT bi-dimensionnelles sur hypercube et compare ces algorithmes en terme de ratio calculs/communications. Il montre notamment que l'intérêt des différentes méthodes dépend de la taille des matrices à traiter et des capacités de la machine cible. Il insiste sur le fait que les recouvrements calculs/communications changent les données du problème et peuvent donner des résultats différents et des algorithmes plus performants. Walker utilise la FFT pour résoudre le problème de la transformée spectrale qui sera ensuite utilisée dans une application plus importante. Il présente pour la FFT bi-dimensionnelle des méthodes par blocs permettant de recouvrir calculs et communications et de réduire les coûts d'initialisation des échanges de messages.

Après une présentation du problème et la parallélisation classique de la FFT1D sur un hypercube, nous présentons tout d'abord nos travaux sur le calcul de plusieurs FFT1D et ensuite leur application pour la parallélisation d'algorithmes de FFT2D.

7.1 Présentation du problème

La Transformée de Fourier Discrète (DFT) d'un vecteur réel (ou complexe) x de taille N est le vecteur X de taille n, défini par :

$$X_j = \sum_{k=0}^{N-1} x_k \omega^{jk} \text{ où } \omega^{jk} = e^{-\frac{2i\pi jk}{N}} \text{ et } i = \sqrt{-1}.$$

Le premier algorithme de transformée de Fourier Rapide (FFT) a été proposé par Cooley et Tuckey [CT65]. Depuis celui-ci, de nombreuses variantes ont été proposées [Swa87]. Elles ne diffèrent, pour la plupart, que dans la façon de stocker les données intermédiaires. L'idée de base de ces algorithmes consiste à diviser les données x en entrée en deux sous-ensembles à chaque étape de l'algorithme, et de les combiner en utilisant un schéma "papillon". La Figure 7.1 présente un exemple de l'algorithme pour $N = 8$. Remarquons que nous nous intéressons plus particulièrement à l'algorithme "classique" de Cooley et Tuckey. D'autres algorithmes existent et une étude similaire pourrait être effectuée avec eux. Malgré tout, ces algorithmes nécessitent sensiblement toujours les mêmes schémas de communications.

Si nous supposons que $N = 2^u$, que nous notons $(i_0 \dots i_{u-1})$ la représentation binaire de i, alors l'algorithme peut être exprimé de la manière donnée dans l'algorithme 7.1 [AHU74, GK92]. x est le vecteur d'entrée, X le vecteur résultat et N leur taille.

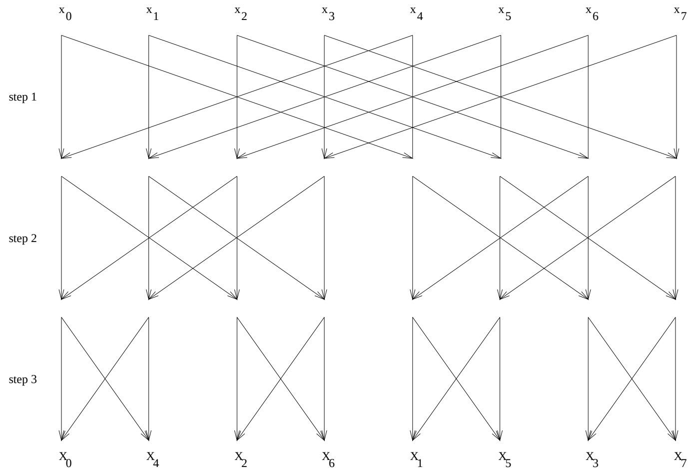

FIG. 7.1 - Schéma “papillon” pour l'algorithme de Cooley-Tuckey pour $N = 8$.

`routine fft1d_seq(x,X,N)`
`début`
    `pour` $i = 0$ `à` $N-1$ `faire` $R[i] = x[i]$ `finpour`
    `pour` $l = 0$ `à` $u-1$ `faire`
        `pour` $i = 0$ `à` $N-1$ `faire` $S[i] = R[i]$ `finpour`
        `pour` $i = 0$ `à` $N-1$ `faire`
            $R[(i_0 \dots i_{u-1})] = S[(i_0 \dots i_{l-1} 0 i_{l+1} \dots i_{u-1}]$
                $+\omega^{(i_l i_{l-1} \dots i_0 0 \dots 0)} \times S[(i_0 \dots i_{l-1} 1 i_{l+1} \dots i_{u-1}]$
        `finpour`
    `finpour`
`fin`

ALG. 7.1 - Algorithme séquentiel de FFT mono-dimensionnelle.

Le coût de l'algorithme séquentiel est donné par $T^{seq}_{fft1d} = N \log(N) \tau_a$, où τ_a est le coût d'une opération de multiplication et d'addition sur des complexes.

7.1.1 FFT bi-dimensionnelle

Le calcul de la Transformée de Fourier Discrète bi-dimensionnelle est donnée par l'équation suivante [FJL+88] :

$$X(j_1, j_2) = \sum_{k_1=0}^{n-1} \left(\sum_{k_2=0}^{n-1} x(k_1, k_2) \times \exp\left(\frac{-2i\pi(k_1.j_1 + k_2.j_2)}{n} \right) \right).$$

Cette équation peut être transformée pour se ramener au calcul de deux FFT1D classiques [BP86]:

$$X(j_1, j_2) = \sum_{k_1=0}^{n-1} \exp\left(\frac{-2i\pi k_1.j_1}{n} \right) \times Y_{j_2}(k_1).$$

où $Y_{j_2}(k_1)$ est la FFT1D en fonction de la variable k_1. L'algorithme est alors immédiat : calculer une FFT1D suivant chaque dimension. Le coût séquentiel d'une FFT2D est donné par $T^{seq}_{fft2d} = 2N^2 \log(N)\tau_a$.

7.2 FFT1D parallèle

Dans ce paragraphe, nous rappelons la méthode classique de parallélisation de l'algorithme de Cooley et Tuckey sur un hypercube.

7.2.1 Allocation des données pour la FFT1D parallèle

Pour la FFT, l'allocation des données classique décrite précédemment (Chapitre 6, Paragraphe 6.1) ne donne pas des résultats très intéressants à cause de problèmes de contentions. Sur un hypercube, nous pouvons tenir compte de la connectivité afin de les éviter et communiquer uniquement avec des processeurs voisins. Supposons que nous ayons un vecteur x à distribuer sur l'hypercube. Nous supposons que la dimension de l'hypercube est d avec $P = 2^d$ nœuds. Nous définissons la fonction *"Bit Reverse"* (*BR*) telle que : $BR(i_0 i_1 \ldots i_{k-1}) = (i_{k-1} i_{k-2} \ldots i_0)$. Nous découpons le vecteur x en $\frac{N}{P} = u - d$ blocs. La fonction d'allocation est donnée par : $Alloc(i) = BR((i-1)_2)$ $i = 1 \ldots N$, où i est l'indice d'un bloc et $(i)_2$ est la représentation binaire de i. Cette allocation implique que toutes les communications nécessaires à l'algorithme de FFT1D s'effectuent entre des nœuds voisins dans l'hypercube [Cha88].

7.2.2 Parallélisation de la FFT1D

La parallélisation de la FFT1D sur hypercube a déjà été présentée de nombreuses fois avec différents modèles. Nous présentons l'algorithme "classique" comme introduction à nos méthodes de réduction des coûts de communications. Nous décrivons les différentes étapes de l'algorithme parallèle pour $N = P = 8$ sur la Figure 7.2.

```
routine fft1d_par(x,X,N,P)
début
   calculer la FFT1D sur mes données (taille N/P)
   pour l = 0 à u − 1 faire
      échanger le bloc avec le processeur q ⊕ l
      calculer un papillon(mon_bloc,bloc_reçu)
   finpour
fin
```

ALG. 7.2 - Algorithme de FFT1D parallèle.

Nous notons $\bigoplus$, l'opération OU exclusif bit-à-bit et une opération papillon$(a, b) = a \pm \omega b$. L'algorithme du processeur q est donné par l'algorithme 7.2. Son coût est donné par :

$$T^{par}_{fft1d} = \frac{N}{P} \log\left(\frac{N}{P}\right)\tau_a + \log(P)\left(\beta + \frac{N}{P}\tau + \frac{N}{P}\tau_a\right)$$

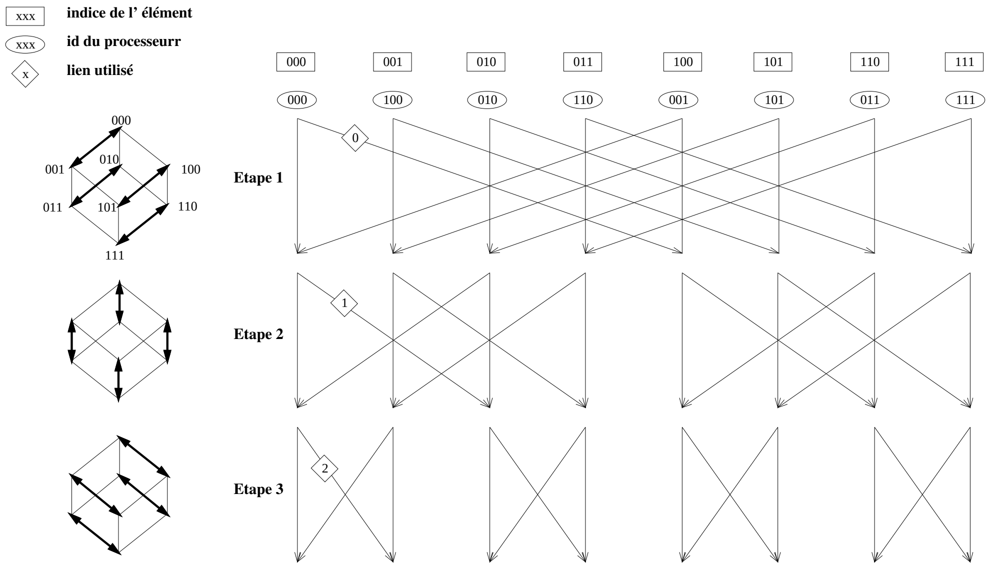

FIG. 7.2 - Exécution de l'algorithme de FFT1D pour $N = 8$ sur un 3-cube.

$$= \frac{N}{P}\log(N)\tau_a + \log(P)\left(\beta + \frac{N}{P}\tau\right). \quad (7.1)$$

7.3 Parallélisation du calcul de plusieurs FFT1D

Dans ce paragraphe, nous décrivons la parallélisation du calcul d'un ensemble de FFT1D distribuées. Nous supposons donc, dans un premier temps, que les données utilisées par chaque FFT1D sont distribuées sur l'hypercube à l'aide d'une fonction BR et nous avons à calculer M FFT1D de taille N distribuées sur P processeurs. Le coût séquentiel de l'algorithme est donné par : $MN\log(N)\tau_a$. Un premier algorithme "naïf" consiste à utiliser M fois l'algorithme parallèle de calcul de FFT1D, ce qui nous donne un coût de :

$$T^{naif}_{Mfft1d} = M\frac{N}{P}\log(N)\tau_a + M\log(P)\left(\beta + \frac{N}{P}\tau\right). \quad (7.2)$$

Nous allons présenter différentes méthodes permettant d'obtenir des recouvrements calculs communications maximaux grâce à l'utilisation de la bande passante totale de l'hypercube et à la réduction du nombre d'initialisations.

7.3.1 Routines utilisées

La parallélisation du calcul de plusieurs FFT1D nécessite l'appel à plusieurs routines de base de calcul et de communication. La table 7.1 donne le nom et la description des routines utilisées et la Figure 7.3 deux exemples d'utilisation des routines de communications développées pour ces algorithmes. Sur cette figure, à gauche nous avons un échange bloquant (BExchange) ou non bloquant (NBExchange) de deux blocs de `r` lignes de longueur `L` entre deux processeurs. Si l'échange est non bloquant, alors un entier j est renvoyé par la routine qui permet, grâce à une fonction `wait(j)` de savoir si le message a bien été échangé. Sur la droite de la figure, nous donnons un

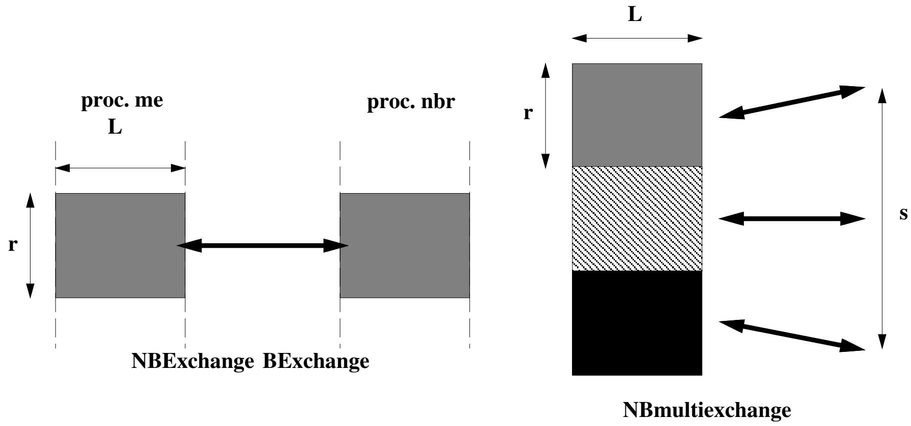

Fig. 7.3 - Routines de communications pour la FFT.

exemple d'échange de `s` blocs de `r` lignes de taille `L` avec `s` processeurs en parallèle. Dans ce cas, nous utilisons `s` liens en parallèle. Remarquons que si notre machine cible est k-port, il faut nécessairement avoir $k \geq s$.

Routine	**Description**
`FFT1D(frow,L,r)`	FFT1D locale sur un paquet de `r` lignes de longueur `L` débutant à la ligne `frow`
`Update(frow,L,r,k)`	Mise à jour sur un paquet de lignes reçu à l'étape `k`
`Bexchange(nbr,frow,r,L,k)`	Echange bloquant à l'étape `k` avec le proc. `nbr` de `r` lignes de longueur `L`
j`=NBexchange(nbr,frow,r,L,k)`	Echange non-bloquant
j`=NBmultiexchange(s,frow,r,L,k)`	Echange non-bloquant de `s` paquets de `r` lignes en parallèle sur s liens
`wait(`j`)`	Routine de complétion d'une communication

Tab. 7.1 - Différentes routines utilisées pour la FFT.

Remarquons que les routines de calculs par blocs ont le même effet sur les accès mémoire que les routines BLAS de niveau 2. Il est possible de les coder de manière très efficace pour un processeur donné en tenant compte de la hiérarchie mémoire (réutilisation des données, optimisation des accès mémoire).

7.3.2 Réduction du nombre d'initialisations

Afin de réduire le nombre d'initialisations pour les communications, il convient de traiter **en une communication** les échanges nécessaires au calcul de plusieurs lignes. Les échanges "papillons" de r lignes peuvent être groupés. Si l'on n'utilise pas le recouvrement des calculs et des communications, nous obtenons une méthode de type SPMD où les phases de calculs alternent avec les phases de communication. Nous appelons cet algorithme SPMD1 (Algorithme 7.3).

```
routine fft1d_spmd1(f,r)
début
   FFT1D(f + (i − 1)r, N/P, r)
   Bexchange(nbr, f + (i − 1)r, r, N/P, 0)

   pour k = 1 à log(P) − 1 faire
      Update(f + (i − 1)r, N/P, r, k)
      Bexchange(nbr, f + (i − 1)r, r, N/P, k)
   finpour

   Update(f + (i − 1)r, N/P, r, log(P))
fin
```

ALG. 7.3 - Algorithme SPMD1.

Soit r le nombre de lignes d'un bloc. Le coût total de l'algorithme SPMD1 est donné par :

$$\begin{aligned} T_{mfft1d}^{SPMD1} &= \frac{M}{r}\left[r\frac{N}{P}\log\left(\frac{N}{P}\right)\tau_a + \log(P)\left(\beta + r\frac{N}{P}\tau + r\frac{N}{P}\tau_a\right)\right] \\ &= \frac{MN}{P}\log(N)\tau_a + \frac{M}{r}\log(P)\beta + \frac{NM}{P}\log(P)\tau. \end{aligned} \tag{7.3}$$

Le défaut de cet algorithme est que les liens des processeurs ne sont pas utilisés de manière optimale. En effet, même si les communications peuvent se faire sur k liens à la fois, l'algorithme 7.3 n'en utilise qu'un à chaque étape.

7.3.3 Allocation optimisée des données

Nous cherchons à minimiser encore plus le coût des communications en utilisant la connectivité de l'hypercube. Pour cela, comme nous l'avons vu précédemment, le nombre d'initialisations doit être réduit et l'utilisation de la bande passante de l'hypercube doit être maximisée. Pour limiter le nombre d'initialisations, nous venons de voir dans le paragraphe précédent, qu'il suffit de calculer plusieurs FFT1D avant d'échanger les données correspondantes en une seule communication. Pour utiliser la bande passante totale de l'hypercube, il faut pouvoir utiliser tous les liens à chaque étape. Nous utilisons une distribution des données permettant ce type d'optimisations. s est le nombre de liens utilisés en parallèle et r la taille des blocs. La distribution pour utilisation de $s = 2$ liens et r lignes sur un hypercube de dimension 2 est donnée sur la Figure 7.4. Un autre exemple pour $s = 3$ sur un hypercube de dimension 3 est donné sur la Figure 7.5.

7.3.4 Algorithme ASYNC

Le but de l'algorithme asynchrone ASYNC est de maximiser le recouvrement des calculs et des communications en calculant plusieurs paquets de lignes en même temps et en tenant compte des possibilités d'utilisation de k ports à la fois. Dans [WWD91], Walker donne quelques remarques pour utiliser le recouvrement et le calcul par blocs de lignes pour améliorer les performances. Dans ce

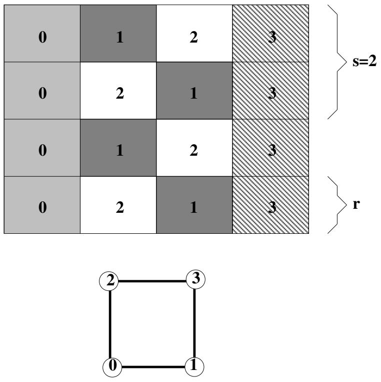

FIG. 7.4 - Répartition de la matrice en utilisant les paramètres $s = 2$ et r.

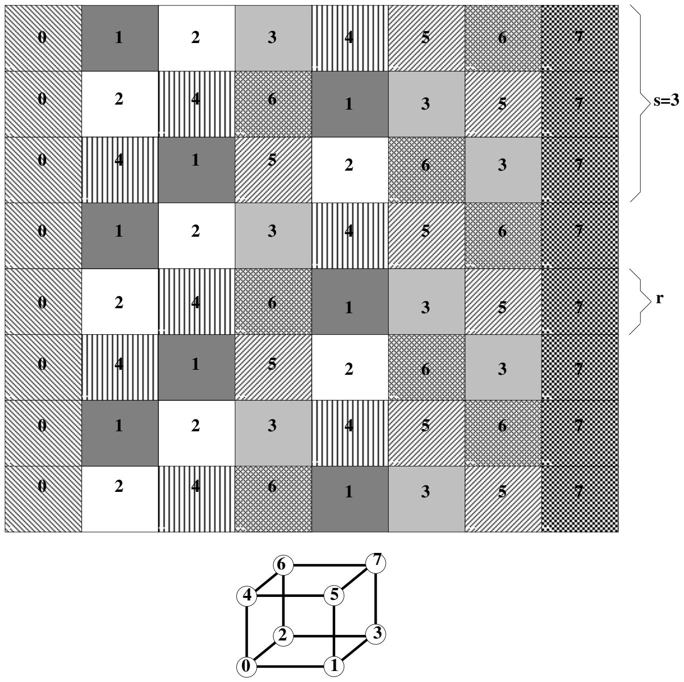

FIG. 7.5 - Répartition de la matrice en utilisant les paramètres $s = 3$ et r.

premier paragraphe, nous donnons une généralisation de la méthode décrite par Walker [WWD91] utilisant un nombre de liens s utilisés supérieur à un.

Ce premier algorithme utilise une méthode "échange au plus tôt", c'est-à-dire, dès qu'un bloc de r lignes est calculé on initie l'échange correspondant au pas de calcul courant et on passe au bloc suivant (Algorithme 7.4). Remarquons que cette méthode peut s'utiliser avec une machine 1-port en travaillant avec uniquement deux paquets de r lignes à la fois ($s = 2$).

Nous allons maintenant décrire des méthodes utilisant des schémas de type SPMD, permettant d'avoir également un recouvrement calculs communications et des méthodes par blocs.

```
routine fft1d_async(f,s,r)
début
   pour i = 1 à s faire
      FFT1D(f + (i − 1)r, N/P, r)
      j=NBexchange(nbr, f + (i − 1)r, r, N/P, 0)
   finpour

   pour k = 1 à log(P) − 1 faire
      pour i = 1 à s faire
         wait(j)
         Update(f + (i − 1)r, N/P, r, k)
         j=NBexchange(nbr, f + (i − 1)r, r, N/P, k)
      finpour
   finpour

   pour i = 1 à s faire
      wait(j)
      Update(f + (i − 1)r, N/P, r, log(P))
   finpour
fin
```

ALG. 7.4 - Algorithme ASYNC.

7.3.5 Algorithmes de type SPMD avec recouvrements

En utilisant le rangement de la matrice des éléments à calculer décrit dans le paragraphe 7.3.3, et une variante de l'algorithme SPMD1, il est possible d'utiliser tous les liens de communication en même temps. Nous appellerons cet algorithme SPMD2. Grâce à la distribution de la matrice, les communications nécessaires au calcul de plusieurs blocs se font sur des liens différents. Soit s le nombre de blocs de r lignes calculés à la même étape et échangés en parallèle. s est borné par le nombre de liens d'un processeur, soit k (où k est la dimension de l'hypercube). Donc, à chaque appel de la routine `fft1d_spmd2`, rs lignes sont calculées. Nous supposons que rs divise M (Algorithme 7.5).

Le coût total de l'algorithme SPMD2 est donné par :

$$\begin{aligned} T_{mfft1d}^{SPMD2} &= \frac{M}{rs}\left[rs\frac{N}{P}\log\left(\frac{N}{P}\right)\tau_a + \log(P)\left(\beta + r\frac{N}{P}\tau + rs\frac{N}{P}\tau_a\right)\right] \\ &= \frac{MN}{P}\log(N)\tau_a + \frac{M}{rs}\log(P)\beta + \frac{MN}{sP}\log(P)\tau. \end{aligned} \tag{7.4}$$

Remarquons que le coût des communications est réduit par un facteur s en terme de bande passante par rapport à l'algorithme SPMD1 (equ. 7.3). Par contre, nous n'avons ici aucun recouvrements calculs/communications. En utilisant deux schémas SPMD et en débutant le second schéma au début de la première communication du premier, nous sommes capables de recouvrir les communications du premier schéma par les calculs du second. Nous appelons par la suite cet algorithme SPMDO (O pour optimisé). Notons que cet algorithme est une déviation de la définition

```
routine fft1d_spmd2(f,r,s)
début
   pour i = 1 à s faire
      FFT1D(f + (i − 1)r, N/P, r)
   finpour
   Bexchange(nbr, f + (i − 1)r, rs, N/P, 0)

   pour k = 1 à log(P) − 1 faire
      pour i = 1 à s faire
         Update(f + (i − 1)r, N/P, r, k)
      finpour
      Bexchange(nbr, f + (i − 1)r, rs, N/P, k)
   finpour

   pour i = 1 à s faire
      Update(f + (i − 1)r, N/P, r, log(P))
   finpour
fin
```

ALG. 7.5 - Algorithme SPMD2.

du SPMD où aucun recouvrement n'est autorisé. Afin de maximiser l'utilisation des liens de communication, nous utilisons l'algorithme SPMD2 comme schéma SPMD pour l'algorithme SPMDO. L'algorithme est donné par l'algorithme 7.6 et un exemple est donné sur la Figure 7.6.

Nous donnons maintenant une analyse de complexité de cet algorithme. Afin de calculer le coût total de l'algorithme SPMDO, nous devons connaître le ratio calculs/communications. Remarquons que, même si les communications ne sont pas totalement recouvertes, nous avons un gain sur l'algorithme SPMD2. Le coût total est donné par :

$$\begin{aligned}
T_{mfft1d}^{SPMDO} &= \frac{M}{2rs}(rs\frac{N}{P}\log\left(\frac{N}{P}\right)\tau_a + \max\left(\beta + r\frac{N}{P}\tau, rs\frac{N}{P}\log\left(\frac{N}{P}\right)\tau_a\right) + \\
&\quad (2\log(P)-1)\max\left(\beta + r\frac{N}{P}\tau, rs\frac{N}{P}\tau_a\right) + rs\frac{N}{P}\tau_a) \\
&= \frac{M}{2rs}(rs\frac{N}{P}\left(\log\left(\frac{N}{P}\right)+1\right)\tau_a + \max\left(\beta + r\frac{N}{P}\tau, rs\frac{N}{P}\log\left(\frac{N}{P}\right)\tau_a\right) + \\
&\quad (2\log(P)-1)\max\left(\beta + r\frac{N}{P}\tau, rs\frac{N}{P}\tau_a\right).) \qquad (7.5)
\end{aligned}$$

Nous devons effectuer la comparaison d'une étape de communication avec le calcul d'un bloc de FFT1D sur un paquet de taille $\frac{N}{P}$ et la même chose avec la mise à jour d'un paquet.

$$\beta + r\tfrac{N}{P}\tau < rs\tfrac{N}{P}\log\left(\tfrac{N}{P}\right)\tau_a \quad \text{et} \quad \beta + r\tfrac{N}{P}\tau < rs\tfrac{N}{P}\tau_a$$

Ceci nous donne deux valeurs de r qui permettent un recouvrement des calculs par les communications.

$$r > \frac{P\beta}{N(s\log(N/P)\tau_a - \tau)} \quad \text{et} \quad r > \frac{P\beta}{N(s\tau_a - \tau)}$$

```
routine fft1d_spmd_opt(f,s,r)
début
   pour i = 1 à s faire
      FFT1D(f + (i − 1)r, N/P, r)
   finpour
   i1=NBmulti_exchange(s, f + (i − 1)r, rs, N/P, 0)

   pour i = 1 à s faire
      FFT1D(f + rs + (i − 1)r, N/P, r)
   finpour

   pour k = 1 à log(P) − 1 faire
      i2=NBmulti_exchange(s, f + rs + (i − 1)r, rs, N/P, k − 1)
      Wait(i1)
      pour i = 1 à s faire
         Update(f + (i − 1)r, N/P, r, k)
      finpour
      i1=NBmulti_exchange(s, f + (i − 1)r, rs, N/P, k)
      Wait(i2)
      pour i = 1 à s faire
         Update(f + rs + (i − 1)r, N/P, r, k − 1)
      finpour
   finpour

   i2=NBmulti_exchange(s, f + rs + (i − 1)r, rs, N/P, log(P))
   Wait(i1)
   pour i = 1 à s faire
      Update(f + (i − 1)r, rs, N/P, log(P))
   finpour

   Wait(i2)
   pour i = 1 à s faire
      Update(f + rs + (i − 1)r, rs, N/P, log(P))
   finpour
fin
```

ALG. 7.6 - Algorithme SPMDO.

Le terme concernant les FFT1D locales est bien sûr plus grand mais celui concernant les mises à jour est utilisé $2\log(P)$ fois. Nous constatons donc qu'il y a des compromis sur la taille des blocs à utiliser pour avoir le meilleur recouvrement possible. Le nombre de liens s utilisés en parallèle sera choisi égal à la dimension de l'hypercube pour minimiser le surcoût des communications.

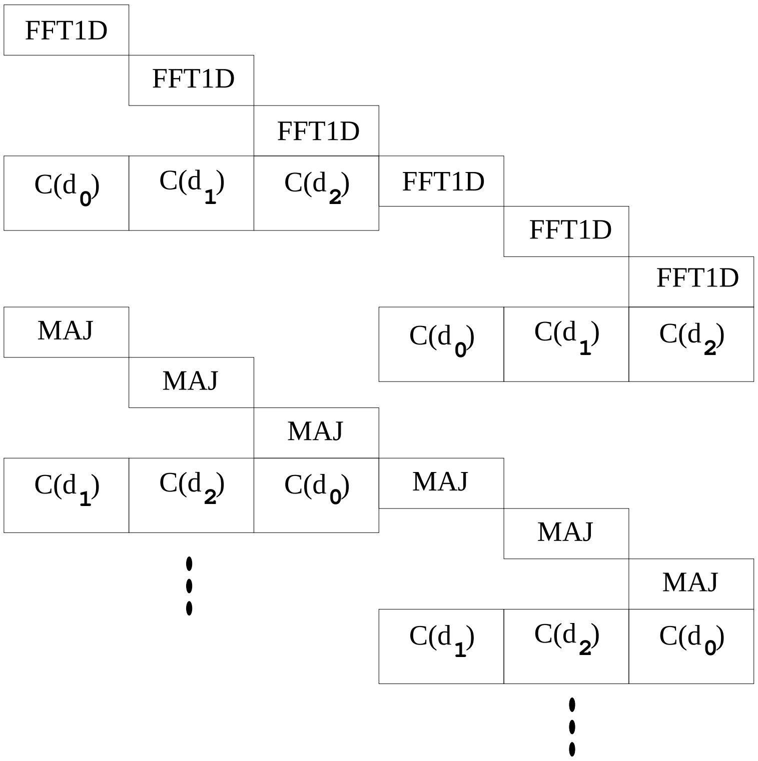

FIG. 7.6 - Exemples de calcul de plusieurs FFT1D avec l'algorithme SPMDO.

7.4 Parallélisation de la FFT2D

Comme nous l'avons vu précédemment, la FFT2D peut se récrire à l'aide de la FFT1D. De la même manière, des algorithmes parallèles de FFT1D peuvent être utilisés pour la FFT2D parallèle. Les trois algorithmes présentés sont ceux utilisés dans l'article de Chu. Ceux-ci sont améliorés grâce à nos méthodes de recouvrement calculs/communications. Pour chaque méthode, nous présentons en premier la méthode sans recouvrement. Les différentes méthodes pour le calcul de la FFT2D sont données sur la Figure 7.7. Tous ces algorithmes utilisent deux phases de calcul et de communication.

Le premier algorithme, appelé ici Transpose (*Transpose Split (TS)*), est une adaptation directe de l'algorithme séquentiel : après avoir calculé de manière totalement parallèle des FFT1D sur chaque ligne et sur chaque processeur, une transposition de la matrice est effectuée. Ensuite, des FFT1D sont de nouveau calculées sur toutes les lignes ((a) sur la Figure 7.7). Il peut être parfois nécessaire de transposer la matrice à nouveau pour retourner au rangement original, afin d'être à même de réutiliser le résultat du calcul. Le second algorithme, appelé ici Local Distribué (*Local Distributed (LD)*), utilise la même distribution par lignes que l'algorithme TS. L'algorithme s'exécute également en deux phases, la première étant la même que pour l'algorithme TS. Mais pour cet algorithme, nous ne transposons pas la matrice entre les deux phases. La seconde phase est donc le calcul distribué d'une série de FFT1D sur les colonnes ((b) sur la Figure 7.7). Le dernier algorithme présenté ici, l'algorithme Blocs (*Block (Bl)*), consiste en deux phases de FFT1D distribuées. La matrice est cette fois rangée par blocs.

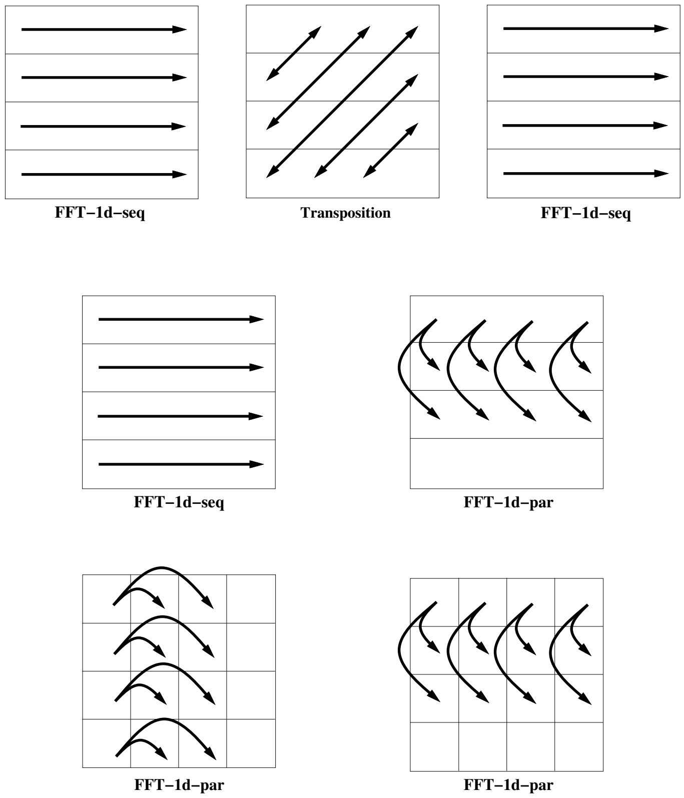

FIG. 7.7 - Algorithmes TS, LD et Bl pour la FFT2D.

7.4.1 Allocation des données

Les deux premiers algorithmes utilisent des rangements par blocs de lignes. Ces blocs de lignes peuvent être alors rangés n'importe comment pour l'algorithme TS. Pour le second algorithme (LD), le rangement devra se faire de manière adaptée à une résolution à l'aide d'un algorithme de type SMPD. Pour le dernier algorithme, nous utilisons directement le rangement décrit pour le calcul de multiples FFT1D distribuées.

7.4.2 Algorithme Transpose-Split (TS)

L'algorithme TS est donné en 7.7.

Comme nous l'avons vu précédemment, la transposition d'une matrice rangée par lignes (ou par colonnes) est un schéma de communication de type multi-distribution. Si l'on ne mélange pas les messages, son coût est donné par [JH87] : $\log(P)\left(\beta + \frac{N^2}{2P}\tau\right)$. Si l'on néglige le coût de la transposition interne, le coût total de l'algorithme TS est donné par :

$$\begin{aligned} T^{TS}_{fft2d} &= 2T^{seq}_{fft1d} + 2T^{hyp}_{tranp} \\ &= 2\frac{N^2}{P}\log(N)\tau_a + 2\log(P)\left(\beta + \frac{N^2}{2P}\tau\right). \end{aligned} \tag{7.6}$$

```
routine TS
début
   /* première phase */
   calculer les FFT1D sur les lignes (en utilisant fft1d_seq)
   transposer la matrice

   /* seconde phase */
   calculer les FFT1D sur les colonnes (en utilisant fft1d_seq)
   [transposer la matrice de nouveau]
fin
```

Alg. 7.7 - Algorithme TS.

Si la transposition n'est pas recouverte, le surcoût dû aux communications donne des efficacités peu intéressantes. Nous pouvons améliorer l'algorithme précédent en recouvrant le calcul de FFT1D locales avec la transposition de la matrice. On peut transposer un ensemble de r lignes dès qu'il a été calculé (Algorithme 7.8).

```
routine TS_opt(N,N,r,P)
début
   /* première phase */
   FFT1D(i,N,r)
   pour i = 1 à N/P pas r en // faire
      FFT1D(i,N,r)
      NB_transp(N,r)
   finpour
   NB_transp(N,r)

   /* seconde phase */
   si (pas de transposition arrière) alors
      FFT1D(i,N,N/P)
   sinon
      FFT1D(i,N,r)
      pour i = 1 à N/P pas r en // faire
         FFT1D(i,N,r)
         NB_transp(N,r)
      finpour
      NB_transp(N,r)
   finsi
fin
```

Alg. 7.8 - Algorithme TS optimisé.

Nous donnons, à présent, l'analyse de complexité de la première phase de l'algorithme TS avec recouvrements. S'il est nécessaire de faire une transposition arrière, le coût total est donné en multipliant par deux le coût de la première phase. Par contre, si l'on ne transpose pas la matrice dans la seconde phase, le coût de cette dernière est donné par le calcul de $\frac{N}{P}$ FFT1D.

Le coût de la première phase est donné par les équations suivantes :

$$\begin{aligned} T_{fft2d}^{TS\ opt\ ph.\ 1} &= rT_{fft1d}^{seq}(N) + \left(\frac{N}{r} - 1\right) \max\left(rT_{fft1d}^{seq}(N), T_{transp}^{hyp}(Nr)\right) + T_{transp}^{hyp}(Nr) \\ &= Nr\log(N)\tau_a + \left(\frac{N}{r} - 1\right) \max\left(Nr\log(N)\tau_a, \log(P)\left(\beta + \frac{Nr}{2P}\tau\right)\right) + \\ &\quad \log(P)\left(\beta + \frac{Nr}{2P}\tau\right). \end{aligned} \tag{7.7}$$

Nous pouvons, à présent, calculer la taille r des blocs pour laquelle nous avons un recouvrement maximum.

$$\begin{aligned} Nr\log(N)\tau_a &\geq \log(P)\left(\beta + \frac{Nr}{2P}\tau\right) \\ r &\geq \frac{\log(P)\beta}{N\left(2P\log(N)\tau_a - \log(P)\tau\right)}. \end{aligned} \tag{7.8}$$

Il existe donc des compromis sur le nombre de lignes r calculées à la fois. Remarquons que cet algorithme peut s'écrire de manière immédiate à l'aide de la routine LOCCS `loccs_pata`. Le travail effectué avant la communication étant le calcul des FFT1D sur r lignes. L'algorithme est donné en 7.9.

```
routine TS_opt(N,N,r,P)
début
   /* première phase */
   loccs_pata(me, pata, buffer,rN, FFT1D,NOJOB)
   /* seconde phase */
   si (pas de transposition arrière) alors
      FFT1D(i,N,N/P)
   sinon
      loccs_pata(me, pata, buffer,rN, FFT1D,NOJOB)
   finsi
fin
```

ALG. 7.9 - Algorithme TS utilisant la routine `loccs_pata`.

7.4.3 Algorithme Local-Distribué (LD)

L'algorithme LD est donné en 7.10.

Si l'on envoie un bloc de taille $\frac{N^2}{P}$ à chaque étape de la routine `fft1d_par` au lieu d'envoyer des messages de taille $\frac{N}{P}$, afin de minimiser le coût des initialisations, le coût de l'algorithme est donné par :

```
routine LD
début
   /* première phase */
   calcul des FFT1D sur les lignes (en utilisant fft1d_seq)

   /* seconde phase */
   calcul des FFT1D distribuées sur les col. (avec fft1d_par)
fin
```

ALG. 7.10 - Algorithme Local Distribué.

$$\begin{aligned} T_{LD} &= \frac{N^2}{P}\log_2(N)\tau_a + \left[N\frac{N}{P}\log(N)\tau_a + \log(P)(\beta + \frac{N^2}{P}\tau)\right] \\ &= \frac{2N^2}{P}\log_2(N)\tau_a + \log(P)(\beta + \frac{N^2}{P}\tau). \end{aligned} \quad (7.9)$$

Afin d'utiliser les recouvrements calculs/communications dans l'algorithme, on peut utiliser les algorithmes ASYNC et SPMDO décrits précédemment. Ces algorithmes seront utilisés dans la seconde phase de l'algorithme LD. Dans le cas où $s > 1$, une redistribution de la matrice est nécessaire qui permet d'utiliser un algorithme avec recouvrements sur plusieurs liens. Cette redistribution s'apparente à une transposition mais avec moins de blocs échangés (un bloc sur s reste sur un processeur, les autres partent chez les voisins). Malgré tout, cette redistribution risque d'être pénalisante pour cet algorithme.

7.4.4 Algorithme par blocs

L'algorithme Bl est donné en 7.11.

```
routine Bl
début
   /* première phase */
   calcul des FFT1D distribuées sur les lignes (avec fft1d_par)

   /* seconde phase */
   calcul des FFT1D distribuées sur les colonnes (avec fft1d_par)
fin
```

ALG. 7.11 - Algorithme Bloc.

Le coût de cet algorithme est donné par :

$$T_{Bl} = \frac{2N^2}{P}\log(N)\tau_a + 2\log(\sqrt{P})\left(\beta + \frac{N^2}{P}\tau\right). \quad (7.10)$$

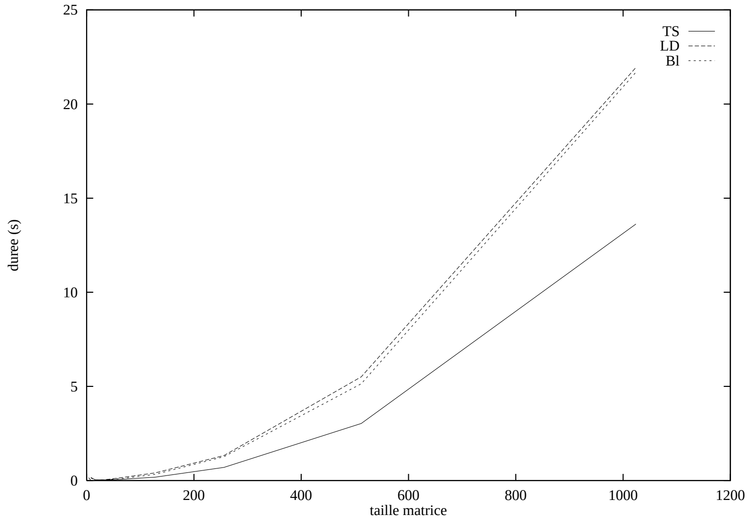

FIG. 7.8 - Comparaison des temps de FFT2D sur 8 processeurs.

Nous pouvons alors recouvrir les communications en utilisant un algorithme ASYNC ou SPMDO pour chaque phase.

7.4.5 Expériences et comparaisons

Nous avons effectué des expériences sur iPSC/860 avec 8 et 16 processeurs. Les Figures 7.8 et 7.9 montrent des comparaisons de différents algorithmes sans recouvrement pour différentes tailles de matrices. L'algorithme le plus performant est toujours TS grâce à la faible taille du réseau et à une grande réduction de coût des initialisations. En effet, dans cette version, la transposition est effectuée en une étape. Si la taille du réseau augmente, la différence se fait moins importante. De plus, la différence entre les algorithmes LD et Bl, qui n'existait pratiquement pas avec 8 processeurs, est plus significative. En effet, les communications, même si elles sont plus nombreuses, sont de tailles moins importantes et sont effectuées par plus de processeurs en parallèle.

La Figure 7.10 montre les temps d'exécution en fonction de la taille des blocs pour la FFT2D d'une matrice de taille 128. Remarquons que le temps décroît jusqu'à atteindre un minimum. A partir d'une taille de bloc égale à 128, aucun découpage n'est effectué. On peut ainsi apprécier le gain obtenu grâce aux recouvrements des calculs et des communications. A gauche de la courbe, on constate comme pour les LOCCS que le nombre d'initialisations étant trop important, le temps est supérieur au temps sans découpage.

Sur la Figure 7.11, nous comparons les temps d'exécution avec et sans recouvrements en fonction de la taille de la matrice pour la méthode LD sur 8 processeurs. Notons que le gain n'est que d'un quart du temps total. Ceci est idéal car la première phase de l'algorithme LD consiste en des FFT1D locales sur tous les processeurs. Nous ne pouvons donc gagner du temps que dans la seconde phase. Nous divisons ici son temps par deux.

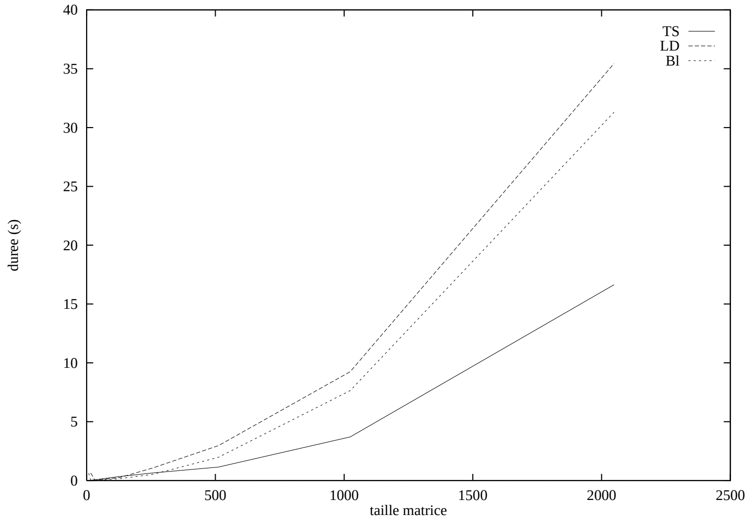

FIG. 7.9 - Comparaison des temps de FFT2D sur 16 processeurs.

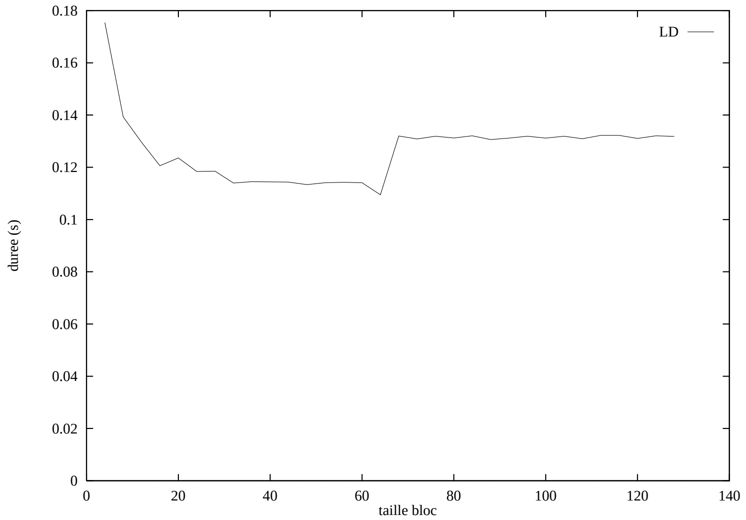

FIG. 7.10 - Temps d'exécution en fonction de la taille de bloc pour l'algorithme LD sur 8 processeurs ($n = 128$).

7.5 Conclusion

Nous avons présenté dans ce chapitre des méthodes efficaces de recouvrements des calculs et des communications pour le calcul de plusieurs FFT1D parallèles avec une application directe dans le calcul de la FFT2D à l'aide des méthodes LD et Bl. Une méthode originale de recouvrement des transpositions pour l'algorithme Transpose de FFT2D a également été proposée. Ces méthodes et routines résultantes feront l'objet d'une bibliothèque sur iPSC/860 et Paragon. Elles peuvent être

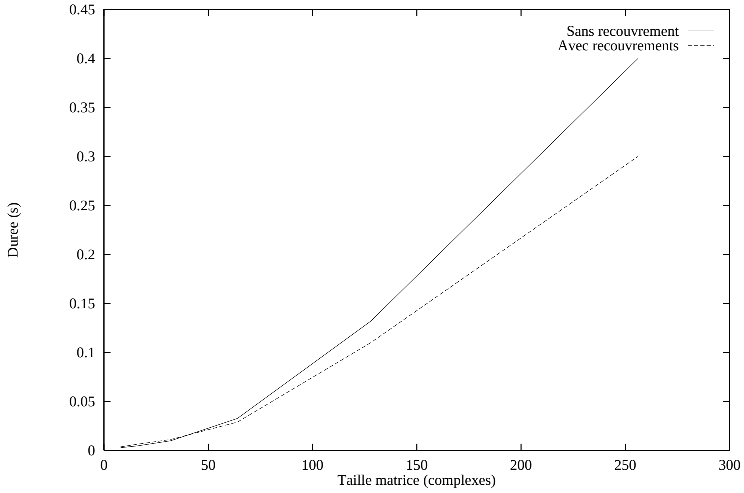

FIG. 7.11 - Temps d'exécution en fonction de la taille de la matrice pour l'algorithme LD avec et sans recouvrement sur 8 processeurs.

également directement utilisées pour le calcul de la FFT3D qui peut se réécrire comme l'assemblage d'une série de FFT1D et d'une FFT2D. D'autres méthodes de FFT doivent être encore étudiées comme la méthode Vector Radix [Chu88] qui permet de réduire le coût des calculs. D'autres rangements sur une grille devraient être étudiés afin de minimiser les contentions, dont la plupart sont inévitables avec le calcul de la FFT.

Chapitre 8

Application à la simulation numérique

Le but de ce chapitre est de montrer qu'une décomposition de domaine appropriée offre un algorithme efficace pour résoudre un modèle de combustion à réaction/diffusion sur une machine parallèle à mémoire distribuée.

Le modèle de combustion considéré est un exemple typique de problème multi-échelle fortement non linéaire. Dans les problèmes multi-échelles, on peut clairement tirer avantage de la structure particulière de la solution dans la définition de la méthode de décomposition de domaine [Gar94]. Comme nous devons obtenir une représentation précise de la solution dans les couches de transition où la solution est raide, les problèmes à échelles multiples donnent de très gros calculs. De plus, les instabilités du front de combustion se forment dans les couches limites. Il est donc également important de résoudre précisément cette zone raide. Nous utilisons un schéma adaptatif car le ratio de l'échelle d'espace dans les couches limites, divisé par l'échelle d'espace dans les régions où la solution n'est pas raide, est un nombre très petit. Nous avons choisi d'utiliser une méthode adaptative pseudospectrale qui a été prouvée très précise pour résoudre des problèmes d'échelles multiples [BBHM92]. L'une des caractéristiques principales de la méthode est que l'on peut garder une structure de donnée régulière en terme de transformation et de décomposition de domaine pour une méthode adaptative qui résout un problème raide à deux dimensions avec une structure de front complexe. Il est, de ce fait, plus facile d'implémenter un tel algorithme sur une machine parallèle. Malgré tout, nous faisons face à deux difficultés dans l'implémentation de notre algorithme parallèle : premièrement, nous utilisons un schéma numérique d'ordre élevé pour discrétiser l'équation en espace et malheureusement, nous pouvons avoir une taille limite quant à la taille du problème à cause du mauvais conditionnement. Ceci n'est pas à priori un bon moyen d'exploiter une machine massivement parallèle car nous ne sommes pas capables d'appliquer la loi de Gustafson (Chapitre 2, Paragraphe 2.3.2). Pour mesurer les performances de notre algorithme, nous comparons nos résultats en terme de temps d'exécution et de précision avec ceux du meilleur algorithme séquentiel à notre disposition. Ce code séquentiel résout le même problème et a été optimisé pour une machine vectorielle.

8.1 Présentation du problème

Nous considérons un modèle de combustion avec le mécanisme de réaction séquentiel :

$$A \rightarrow \mu B \rightarrow C,$$

où μ est le coefficient stochiométrique pour la première réaction. Le réactif A est converti en un produit intermédiaire B avant d'être brûlé et converti dans le produit final C. Ce modèle très simplifié a été utilisé avec succès pour modéliser la combustion des hydrocarbures [WD81]. Dans ce chapitre, nous considérons le cas d'une flamme cylindrique avec une injection ponctuelle de carburant. Le problème possède trois inconnues, la température : θ, la concentration du réactif A : C_1 et la concentration du produit intermédiaire B : C_2, toutes fonctions des coordonnées polaires r et ψ, et qui correspondent au modèle thermodiffusif pour une réaction à deux pas d'Arrhenius [Pel87]. Les équations satisfaites par θ, C_1 et C_2 sont :

$$\begin{aligned}
\frac{\partial \theta}{\partial t} &= \Delta\theta - \frac{K}{r}\frac{\partial \theta}{\partial r} + \alpha Da_1 R_1(\theta) + (1-\alpha) Da_2 R_2(\theta), \\
\frac{\partial C_1}{\partial t} &= \frac{1}{L_1}\Delta C_1 - \frac{K}{r}\frac{\partial C_1}{\partial r} - Da_1 R_1(\theta), \\
\frac{\partial C_2}{\partial t} &= \frac{1}{L_2}\Delta C_2 - \frac{K}{r}\frac{\partial C_2}{\partial r} + Da_1 R_1(\theta) - Da_2 R_2(\theta),
\end{aligned}$$

où R_1 et R_2 sont des termes sources non linéaires :

$$R_1(\theta) = e^{\frac{\beta_1(\theta-\theta_0)}{1+\gamma_1(\theta-\theta_0))}}, R_2(\theta) = e^{\frac{\beta_2(\theta-1)}{1+\gamma_2(\theta-1))}},$$

Les paramètres sont : L_1 et L_2, les nombres de Lewis; Da_1 et Da_2, les nombres de Daemker; N_1 et N_2, l'énergie d'activation de chaque réaction chimique; α, la température de la première réaction; K, une mesure de la puissance de l'injection de carburant; $\theta_0 \in [0,1]$, est une température pondérée proche de la température à laquelle la première réaction apparaît; β_i, est proportionnel à l'énergie d'activation N_i. Les coefficients γ_i sont fonction de la température de la mixture non brûlée, de la température finale adiabatique et de θ_0. L'énergie d'activation de chaque réaction chimique est grande et par conséquent β_i $(i = 1, 2)$, de même que l'un des nombres de Daemker, sont grands. Une dérivation de ce modèle pour une géométrie cartésienne a été effectuée dans [Pel87].

θ, C_1 et C_2 sont des fonctions périodiques de ψ. Les conditions limites satisfaites par θ, C_1 et C_2 sont :

$$\theta \to 0,\ C_1 \to 1,\ C_2 \to 0,\ \text{quand}\ r \to 0; \quad \theta \to 1,\ C_1 \to 0,\ C_2 \to 0,\ \text{quand}\ r \to \infty.$$

Pour le domaine de calcul, nous prenons $(r, \psi) \in (r_0, r_1) \times (0, 2\pi)$, où $0 < r_0 < r_1 < \infty$; r_0 est suffisamment petit et r_1 suffisamment grand pour que les conditions aux limites soient remplacées par les conditions de Dirichlet,

$$\theta(r_0, \psi) = 0,\ C_1(r_0, \psi) = 1,\ C_2(r_0, \psi) = 0;$$

$$\theta(r_1, \psi) = 1,\ C_1(r_1, \psi) = 0,\ C_2(r_1, \psi) = 0.$$

Parce que l'activation d'énergie de la réaction chimique est grande, la flamme est une couche de transition fine, et le problème de combustion est de type perturbation singulière. Puisque nous avons deux réactions chimiques, nous avons deux fronts de combustion. Suivant la valeur des paramètres, ces deux fronts peuvent être superposés ou bien être complètement séparés. Nous sommes principalement intéressés par les interactions non linéaires entre ces deux fronts et la formation de flammes cellulaires. L'analyse asymptotique d'un modèle thermodiffusif avec une réaction chimique à deux pas peut être trouvée dans [BMM87, Pel87]. Dans ces articles, des régimes différents ont été considérés . Une analyse de stabilité linéaire de flammes planaires se propageant librement suggère la possibilité de flammes cellulaires quand le premier nombre de Lewis L_1 est suffisamment petit par rapport à 1. Il s'agit d'un cas analogue au cas du mécanisme de réaction unique. Le but de ce travail est de paralléliser la simulation directe d'une telle flamme cellulaire sur une machine parallèle à mémoire distribuée.

8.2 Les méthodes numériques

Notre méthode numérique est construite sur une décomposition de domaine pseudo-spectrale adaptative. Nous décrivons tout d'abord la méthode pseudo-spectrale standard [CHQZ87, GO77]. Considérons le modèle unidimensionnel simple :

$$u_t = u_{xx} + R(u),\ -1 \leq x \leq 1,$$

La solution est approximée en développant u comme une suite finie de polynômes de Chebyshev

$$u \sim u_J = \sum_{j=0}^{J} a_j T_j(x).$$

où les coefficients a_j sont obtenus par collocation, c'est-à-dire que u_J satisfait l'équation sur un ensemble de $J+1$ points x_j, $j = 0..J$. Nous utilisons les points de Gauss-Lobatto :

$$x_j \;=\; cos\left(\frac{j\Pi}{J}\right),\; j = 0..J\;.$$

L'avantage principal de la méthode pseudo-spectrale comparée à une méthode par éléments finis est l'amélioration de la précision : l'approximation pseudo-spectrale est de précision d'ordre exponentiel pourvu que u soit suffisamment différentiable. Comme inconvénient, l'approximation pseudospectrale est imprécise lorsque la fonction à approximer est raide.

Les principales difficultés pour calculer la solution de notre problème de combustion sont les zones de variations rapides de la température et de la concentration des réactifs. Plus précisément, nous avons la possibilité d'avoir deux couches qui correspondent au(x) front(s) de combustion : les termes non linéaires $Da_1C_1R_1(\theta)$ et $Da_2C_2R_2(\theta)$ sont d'ordre un *uniquement* dans ces couches où la solution est raide. La dynamique de ce processus de combustion est dirigée par ces deux couches de transition minces; par conséquent, il est important de calculer de manière très précise ces couches.

De manière à tirer avantage de la haute précision de la méthode pseudo-spectrale standard, nous devons introduire un schéma adaptatif. Il est maintenant classique d'utiliser une transformation de coordonnées qui dilate la couche comme on le fait dans la construction d'un développement asymptotique [BGMM89, Eck79, GP88]. De manière concrète, nous considérons le problème de perturbation singulière :

$$u_t \;=\; u_{xx} \;+\; NR(u),\; -1 \leq x \leq 1,$$

où N est un paramètre grand. Nous supposons que nous avons une couche de transition en x_0. Nous introduisons alors une famille de transformations à un paramètre :

$$s \in [-1,1] \rightarrow y = f(s,\varepsilon) \in [-1,1]$$

où ε est un petit paramètre qui décrit comment on concentre les points de collocation dans l'espace physique. Par exemple, si x_0 est un bout d'un intervalle, nous utilisons une transformation de type couche limite telle que :

$$f(s,\varepsilon) = \mp\left(\frac{4}{\pi}\,atan\left(\varepsilon\,tan\left(\frac{\pi}{4}\,(\pm s-1)\right)\right)\right). \tag{8.1}$$

Nous utilisons un signe positif (resp. négatif) pour $x_0 = 1$ (resp. $x_0 = -1$). Si $x_0 = 0$, nous utilisons une transformation de type couche de transition telle que :

$$f(s,\varepsilon) = \varepsilon\,tan(\;s\;atan(\;\varepsilon^{-1}\;)). \tag{8.2}$$

Si e_J représente l'erreur numérique, nous pouvons utiliser une estimation à priori [BMM87, BGMM89] :

$$|e_J| \;\leq \frac{C^t}{N^p}||u||_\varepsilon,$$

le meilleur paramètre de dilatation ε s'obtient lorsque $||u||_\varepsilon$ est minimum. On peut également montrer que ε dépend de l'épaisseur de la couche limite [Gar94].

Si notre problème de perturbation singulière possède deux couches différentes localisées en $x_0 \neq x_1$, le changement de coordonnées précédent ne peut résoudre les deux couches. Donc, pour résoudre de manière précise les deux couches, nous introduisons un degré de liberté supplémentaire dans notre méthode d'adaptivité avec la méthode de décomposition de domaines [EGP89, GH86]. Nous pouvons découper le domaine de calcul $[-1, 1]$ en deux sous-domaines de telle sorte que chaque sous-domaine contienne uniquement une couche.

En fait, même dans le cas où l'on souhaite résoudre une seule couche, une décomposition de domaine peut être utile. Un certain nombre d'expérimentations numériques suggèrent qu'il est même plus intéressant de résoudre une couche de transition interne localisée en x_0 avec deux sous-domaines et une distribution de type couche limite (8.1) dans chaque sous-domaine plutôt qu'avec un domaine unique et une distribution de type couche de transition (8.2) [Gar94]. Pour le problème de combustion considéré, la précision semble être au moins équivalente, mais aussi la concentration naturelle des points de Chebyshev aux extrémités de l'intervalle fait que la stratégie de décomposition de domaine adaptative est moins sensible à l'erreur sur la position de la couche interne. De même, dans le régime où nous avons deux fronts bien séparés $x_0 \neq x_1$, nous utilisons quatre sous-domaines, deux pour la résolution de chaque couche interne.

La manière dont l'interface peut être trouvée dans l'algorithme n'a pas été encore définie. Comme suggéré dans [BBHM92], on peut utiliser une estimation à priori [BMM87, BGMM89] paramétrée par la position de l'interface et les paramètres d'étirement de chaque sous-domaine. Cependant, le problème de minimisation résultant est relativement complexe à résoudre car il dépend de plusieurs paramètres. Puisque l'interface correspond à la position des couches, dans l'esprit de l'analyse asymptotique, nous avons choisi la position des interfaces correspondant aux maximum de chaque terme de la réaction $Da_i C_i R_i$. Nous avons vérifié que ce choix est en parfaite adéquation avec le critère d'adaptivité dans [BGM93].

Parce que nous calculons des flammes cellulaires, la position du front de combustion peut dépendre de l'angle. Nous devons donc moyenner la position des fronts définie précédemment en fonction de l'angle. La transformation de type couche limite fait que cette décomposition de domaine adaptative fonctionne bien, même si le front oscille comme sur la Figure 8.5. En particulier, le cas $L_1 = 0.37$ et $L_2 = 0.9$ correspond à une flamme à quatre cellules avec un seul front; nous avons testé la précision de la solution avec plusieurs grilles et nombre de points. Nous avons également comparé cette solution obtenue avec deux domaines et une stratégie de transformation de type couche limite avec un code à un seul domaine utilisant une transformation de type couche de transition. La précision de notre calcul dans la norme maximum avec 12000 degrés de liberté est d'ordre 10^{-4}. La précision est analogue à la précision de la solution donnée par le code séquentiel.

Finalement, comme nous allons le voir, la décomposition de domaine augmente le potentiel de parallélisme de notre algorithme.

8.3 Parallélisation

Par la suite, nous appellerons n_a^g le nombre total d'angles, n_a le nombre d'angles par processeur ($n_a = n_a^g/P$ si nous avons P processeurs et $\psi_i = \frac{i}{n_a} 2\pi$), N le nombre de points de Chebyshev dans la direction radiale et n_d le nombre de domaines.

Nous allons tout d'abord décrire l'algorithme avec l'équation scalaire :

$$\frac{\partial u}{\partial t} = \Delta u + F(u); \quad (r, \psi) \in [r_o, r_1] \times [0, 2\pi],$$

où Δ est le laplacien en coordonnées cylindriques. Nous considérons le schéma d'Euler suivant :

$$\frac{u^{n+1} - u^n}{t^{n+1} - t^n} = D_{rr} u^{n+1} + \frac{1}{r^2} D_{\psi\psi} u^n + F(u^n), \tag{8.3}$$

où D_{rr} est l'opérateur de différenciation par rapport au rayon r et $D_{\psi\psi}$ est l'opérateur de différenciation par rapport à l'angle ψ.

Soit $\{\psi_i, i = 1..n_a\}$, la discrétisation régulière par rapport à l'angle de pas $h = \frac{2\pi}{n_a - 1}$ et u_i la restriction de la solution à $\psi = \psi_i$. On peut ré-écrire (8.3) comme :

$$\hat{D}u_i^{n+1} \quad = \frac{1}{r^2} \quad D_{\psi\psi} u_i^n \ + \ F(u_i^n), \ pour \ i = 1..N. \tag{8.4}$$

Nous utilisons premièrement une dépendance explicite sur l'angle dans ce schéma afin d'avoir un algorithme parallèle. Ce choix sera justifié plus tard. Nous considérons une différence finie d'ordre 6 de $\frac{1}{r^2}\frac{\partial^2 u}{\partial \psi^2}$. Par conséquent, $D_{\psi\psi} u_i^n$ dépend uniquement de u_j pour $j = i-3, ..., i+3$. Pour éviter les contraintes sur le pas de temps dues au traitement explicite de $\frac{\partial^2 u}{\partial \psi^2}$, nous utilisons une formule de différences finies d'ordre deux du pas $2h$ dans le sous-domaine Ω^* de Ω restreint au disque $r < r^*$. r^* sera défini plus tard. De ce fait, chaque itération dans le temps peut être résumée comme sur l'algorithme 8.1.

`pour chaque angle` $\psi = \psi_i$ `faire`
   `tâche 1 : calculer` $F(u_i^n)$ `et lire` u_j `pour` $j = i-3, ..., i+3$, $j \neq i$
   `tâche 2 : calculer` $D_{\psi\psi} u_i^n$
   `tâche 3 : calculer la solution du système linéaire (8.4)`
`finpour`

ALG. 8.1 - Un pas d'itération pour la solution du problème de combustion

L'algorithme 8.1 peut être appliqué de la même manière à notre système d'équations aux dérivées partielles non linéaires avec $u_i^n = (\Theta_i^n, C_{1,i}^n, C_{2,i}^n)$. En plus du schéma Eulérien, nous avons utilisé un schéma prédicteur-correcteur d'ordre deux pour le schéma à progression en temps, mais cela n'ajoute pas de difficultés pour obtenir un algorithme parallèle.

Nous pouvons justifier le fait que le schéma est explicite en ψ de la manière suivante : le front de combustion est principalement raide dans la direction radiale et, par conséquent, nous avons pour la température $\frac{\partial^2 \theta}{\partial \psi^2} \prec\prec \frac{\partial^2 \theta}{\partial r^2}$ de même que pour C_1 et C_2. De plus, le pas de temps utilisé dans notre expérience était principalement restreint par le traitement explicite des termes de réaction non-linéaires quand $r^* = 4$ and $n_a = 64$ au lieu du traitement explicite de $\frac{\partial^2 \theta}{\partial \psi^2}$. Dans le disque $r < r^*$, la dépendance entre les inconnues est extrêmement faible car l'instabilité cellulaire démarre près du front de combustion et est rapidement amortie par le processus de diffusion. De même que la convection du combustible limite la propagation de l'instabilité vers le point source d'injection du combustible. De ce fait, une approximation d'ordre faible de la dérivé de second ordre par rapport à l'angle dans le sous-domaine Ω^* n'affecte pas la qualité globale du résultat.

En plus du parallélisme fondé sur les termes explicites vus précédemment, deux autres niveaux de parallélisme doivent être considérés. Premièrement, nous pouvons utiliser la décomposition de domaine le long des rayons. Pour chaque angle, le système linéaire (8.4) peut être résolu avec n_d processeurs mis en contact uniquement pour la résolution du problème d'interface qui est un système linéaire tridiagonal de taille $n_d - 1$. Deuxièmement, notre système d'équations aux dérivées partielles est découplé à l'intérieur de chaque étape d'itération et le calcul peut être ainsi distribué sur trois sous-blocs de processeurs, correspondant aux différentes équations.

Nous décrivons à présent l'utilisation de la dépendance explicite sur l'angle et sa parallélisation (premier niveau de parallélisme). L'utilisation de la parallélisation de la décomposition de domaine

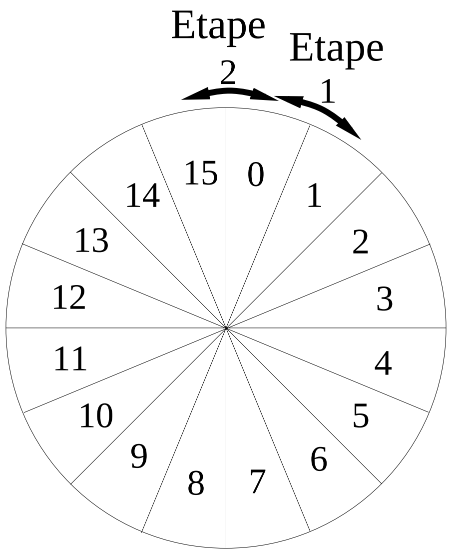

FIG. 8.1 - Distribution des rayons sur un anneau de 16 processeurs.

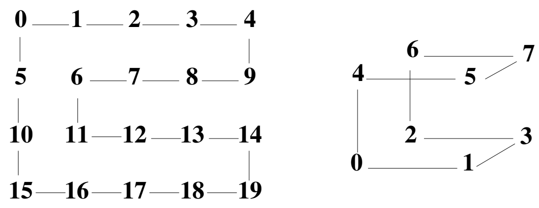

FIG. 8.2 - Exemples d'anneaux dans la grille et dans l'hypercube.

(second niveau de parallélisme) et du dernier niveau qui concerne l'utilisation de trois groupes de processeurs sont en cours d'étude.

Le premier niveau de parallélisation concerne la distribution des angles sur un anneau. La Figure 8.1 donne la distribution des données sur un anneau. L'anneau est ensuite plongé dans la topologie choisie (hypercube ou grille) en fonction de la machine cible utilisée (iPSC/860 ou Paragon). La Figure 8.2 donne des exemples de plongement d'un anneau de 20 processeurs dans une grille et d'un anneau de 8 processeurs dans l'hypercube. L'algorithme parallèle du premier niveau nécessite de nombreux échanges entre des processeurs possédant des rayons voisins. Afin d'éviter les contentions et d'optimiser les performances, une attention toute particulière est portée à la localité des communications. Sur l'hypercube, une méthode classique est utilisée grâce au code de Gray. Par contre, dans la grille, nous dessinons un "serpent". Si nous utilisons P processeurs, l'ensemble des rayons est découpé en blocs de taille n_a^g/P. Nous distinguons le numéro logique de processeur `my_id_log` qui est le numéro de bloc de rayons et le numéro physique du processeur `my_id`. Nous devons avoir une fonction `my_id=f(my_id_log)` qui, à un numéro de bloc de rayons `my_id_log`, donne un numéro de processeur `my_id`.

L'utilisation de différences finies d'ordre 6 permet de maximiser le recouvrement des calculs et des communications. En utilisant la distribution sur l'anneau, à l'étape k, chaque processeur peut calculer un ensemble de rayons internes sans communication (en grisé sur la Figure 8.3). Mais le calcul des angles frontières nécessite l'accès à des données situées sur les processeurs voisins à droite et à gauche.

Afin de réduire le nombre d'initialisations, les différentes données de Teta et des concentrations aux frontières sont rangées ensemble dans deux buffers (B_{left} et B_{right} dans l'Algorithme 8.2) et envoyées de manière asynchrone. Le coût du rangement dans les buffers est négligeable face aux coûts d'initialisation. Nos machines cibles étant des machines 1-port, nous devons adopter une stratégie pair/impair. C'est-à-dire que pour une première étape, tous les processeurs de numéro logique pair échangent leurs données avec les processeurs de droite (et les processeurs impairs

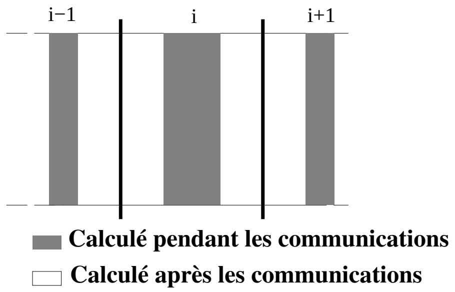

FIG. 8.3 - Calcul des différences finies

échangent avec leur voisin de gauche). Dans une seconde étape de communication, l'inverse est effectué. Les communications peuvent être faites dès que possible et recouvertes par d'autres calculs (calcul des dérivées en angle des points intérieurs, calcul des dérivées en espace et calcul des termes source). La partie critique du programme est donnée dans l'algorithme 8.2. `NBexchange` est une routine d'échange non-bloquante.

Nous donnons à présent une analyse de complexité du recouvrement des calculs et des communications pour ce premier niveau de parallélisation. Nous supposons que nous avons un coût de communication linéaire ($\beta + L\tau$) et que le coût moyen d'une opération arithmétique est de τ_a.

Typiquement, dans notre application, nous aurons $64 \leq n_a \leq 256$, $20 \leq N \leq 50$ et $2 \leq n_d \leq 6$. A chaque étape de calcul, nous avons à échanger les valeurs de θ, $C1$ et $C2$ pour 3 angles dans chaque direction de l'anneau, ce qui nous donne le coût suivant :

$$T_{com} = \beta + 3n_a N n_d \tau. \tag{8.5}$$

Le coût des différences finies pour les points intérieurs qui ne dépendent pas des données présentes sur les processeurs voisins est donnée par :

$$T_{int} = 24 n_d N (n_a - 6)\tau_a. \tag{8.6}$$

Le coût du calcul des dérivées en espace d'ordre un par rapport à l'angle, utilisant des produits de matrice, est donné par :

$$T_{SD} = n_d N^2 n_a \tau_a. \tag{8.7}$$

Et enfin, si τ_{exp} est le coût d'une exponentielle sur un réel double précision, alors le coût du calcul des termes source est égal à :

$$T_{ST} = 12 n_d N n_a \tau_a + 2 n_d N n_a \tau_{exp}. \tag{8.8}$$

Le coût total est donné par la sommation des coûts 8.6, 8.7, et 8.8 :

$$\begin{aligned} T_{overl.} &= 24 n_d N (n_a - 6)\tau_a + n_d N^2 n_a \tau_a + 12 n_d N n_a \tau_a + 2 n_d N n_a \tau_{exp} \\ &\equiv O(n_d N^2 n_a). \end{aligned} \tag{8.9}$$

Les communications seront entièrement recouvertes par les calculs ($T_{overl.} \geq T_{com}$), pour des N asymptotiquement supérieurs à $3\tau/\tau_a$. Sur nos machines cibles, nous avons un rapport de 20 entre τ_{exp} et τ_a. Le recouvrement est donc total pour plus de 30 points.

```
début
   my_id_log = f^{-1}(my_id)
   ⋮
   ranger les données frontières de θ, C1 et C2 dans B_left pour la gauche
   ranger les données frontières de θ, C1 et C2 dans B_right pour la droite
   si (my_id_log est pair) alors
      NBexchange B_left avec la gauche
   sinon
      NBexchange B_right avec la droite
   finsi
   si (my_id_log est pair) alors
      NBexchange B_right avec la droite
   else
      NBexchange B_left avec la gauche
   finsi
   Calculer ∂²θ/∂ψ², ∂²C1/∂ψ² et ∂²C2/∂ψ² pour les points intérieurs (indépendants)
   Calculer les dérivées en espace ∂θ/∂r, ∂C1/∂r et ∂C2/∂r
   Calculer les termes source
   Vérifier que les échanges sont terminés
   Calculer ∂²θ/∂ψ², ∂²C1/∂ψ² et ∂²C2/∂ψ² pour les pts intérieurs (dépendants des voisins)
   Calculer les parties droites des systèmes linéaires
   ⋮
fin
```

ALG. 8.2 - Partie critique de l'algorithme sur un anneau (premier niveau de parallélisme).

8.4 Expériences sur iPSC/860 et Paragon

Les expériences ont été effectuées sur des machines Intel iPSC/860 à 128 nœuds et Paragon à 66 nœuds. Sur la Figure 8.4, nous donnons les différentes vues de Paragraph, correspondant à une exécution sur 8 processeurs de l'iPSC/860. Sur le diagramme `Spacetime`, les traits obliques sont les échanges entre voisins dans l'anneau à chaque pas de l'algorithme et les lignes verticales des synchronisations qui garantissent une maximisation de l'utilisation de la bande passante grâce à l'utilisation des liens en full-duplex. Que ce soit sur l'iPSC/860 ou la Paragon, une synchronisation préalable des processeurs garantit que l'échange se fera en full-duplex [SLF91]. Cette synchronisation est obtenue à l'aide d'un message de taille nulle. Le surcoût entraîné par cet échange est largement "remboursé" par le gain obtenu en temps de communication. Afin de minimiser le nombre d'initialisations, et de ce fait le coût des communications, les données pour θ, C_1 et C_2 sont échangées en même temps. L'utilisation de Paragraph permet de constater l'efficacité de l'optimisation du code et du recouvrement des communications.

Les temps d'exécution, pour 100 pas de temps et les paramètres ($N = 24$, $n_d = 4$) et n_a^g variant de 32 à 256, de la première version de l'algorithme parallèle sont donnés dans les tables 8.1 et 8.2. Les entrées "/" du tableau correspondent à des tailles de problèmes trop importantes pour la mémoire des processeurs ou bien à des nombres d'angles trop petits pour le nombre de processeurs

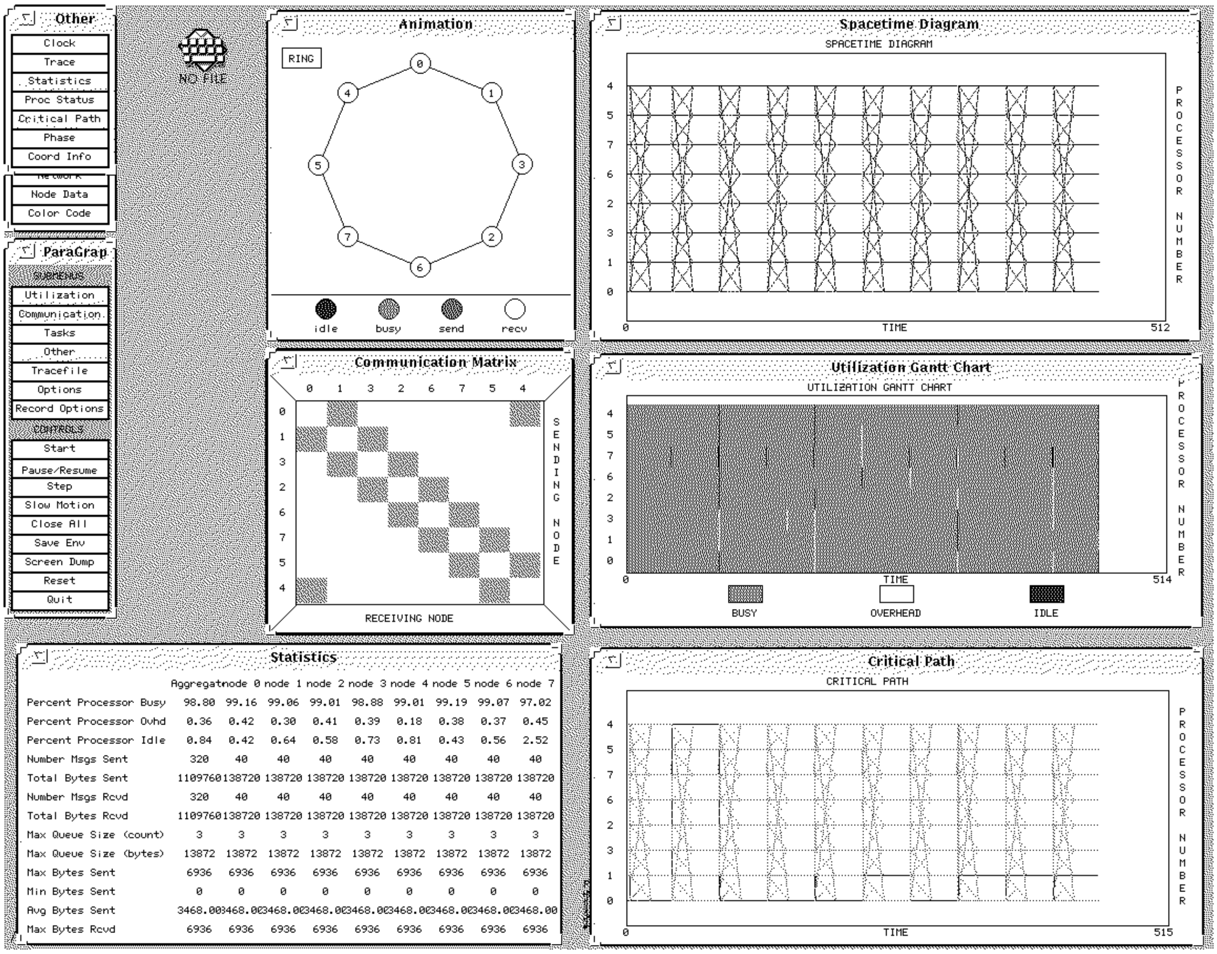

FIG. 8.4 - Ecran Paragraph pour l'exécution du problème de simulation.

dans les anneaux.

Remarquons que les facteurs d'accélération obtenus sont très bons (environ 56 sur 64 processeurs pour l'iPSC) et que l'algorithme scale très bien. En effet, si on augmente le nombre de processeurs et la taille du problème avec le même facteur, le temps reste le même. Cela est dû au fait que les communications sont parfaitement recouvertes grâce à la réorganisation du programme.

Si, d'autre part, nous comparons les temps obtenus sur l'iPSC/860 et la Paragon, nous constatons un gain de 20%. Cela est dû à la différence de vitesse entre les i860 des deux machines (XP contre XR). De plus, même si les temps de calcul sont réduits de 20% sur la Paragon, les communications s'effectuent cinq fois plus rapidement sur cette dernière machine que sur l'iPSC/860 (bande passante de $12.8Mo/s$ pour la Paragon contre $2.8Mo/s$ pour l'iPSC/860 sur les machines utilisées).

Nous donnons dans le tableau 8.3, une comparaison des temps obtenus sur diverses machines du commerce pour les mêmes tailles de problème. Remarquons que les temps obtenus pour le CRAY correspondent à un code optimisé spécialement pour cette machine en tenant compte de la vectorisation, ce qui n'est pas le cas pour les autres machines.

Les Figures 8.5, 8.7 et 8.9 donnent des représentations 3D des différents champs (température,

Nombre de proc.	n_a^g			
	32	64	128	256
1	38.29	76.021	152.07	/
2	22.64	44.36	85.45	175.0
4	11.80	22.66	44.36	88.46
8	6.19	11.78	22.66	44.38
16	/	6.21	11.82	22.76
32	/	/	6.218	11.83
64	/	/	/	6.21

TAB. 8.1 - Résultats sur iPSC/860 en temps (secondes) jusqu'à 64 processeurs

Nbr of proc.	n_a			
	32	64	128	256
1	26.87	53.61	107.03	/
2	16.47	31.83	62.71	125.00
4	8.91	16.60	32.03	63.10
8	5.04	8.95	16.65	32.13
16	/	5.06	8.96	16.67
32	/	/	5.08	8.99
64	/	/	/	5.10

TAB. 8.2 - Résultats sur Paragon en temps (secondes) jusqu'à 64 processeurs

concentration de la première espèce et concentration de la seconde espèce) pour une solution stationnaire. Pour chaque représentation, le centre du cercle correspond à l'endroit où est envoyée la première espèce (le combustible). La température augmente quand on s'éloigne du centre. Remarquons également, la présence très nette de quatre cellules. Les Figures 8.6, 8.8 et 8.10 donnent une représentation des mêmes variables pour une solution avec onde qui correspond à des nombres de

Machine	P	**Temps (s)**
CRAY YMP	1	≈ 10
HP 720	4	51
Archipel Volvox IS/860	8	13
Intel iPSC/860	8	12
nCUBE/2	64	11
Archipel Volvox IS/860	16	7
Intel iPSC/860	16	6
Intel Paragon	16	5

TAB. 8.3 - Résultats en temps pour diverses machines.

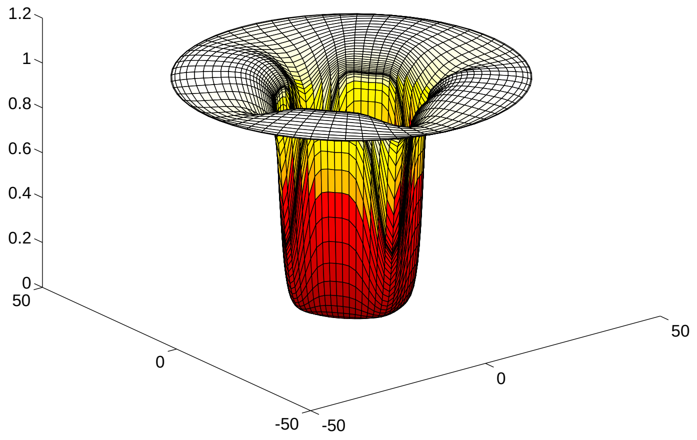

FIG. 8.5 - Champ de température pour une solution stationnaire à 4 cellules.

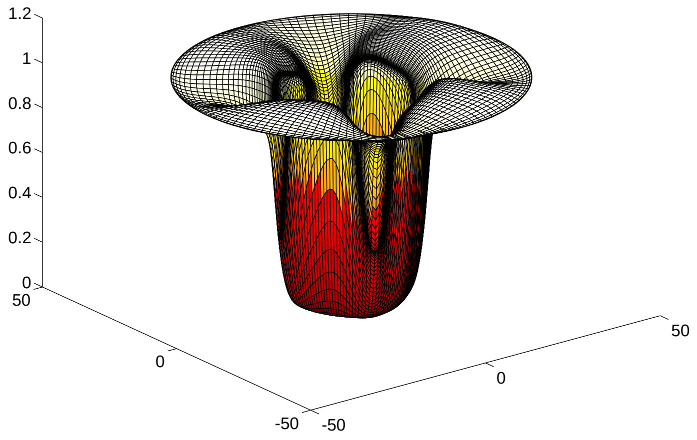

FIG. 8.6 - Champ de température pour une solution (onde à 4 cellules).

Lewis différents. Remarquons la présence d'une rotation sur les rayons.

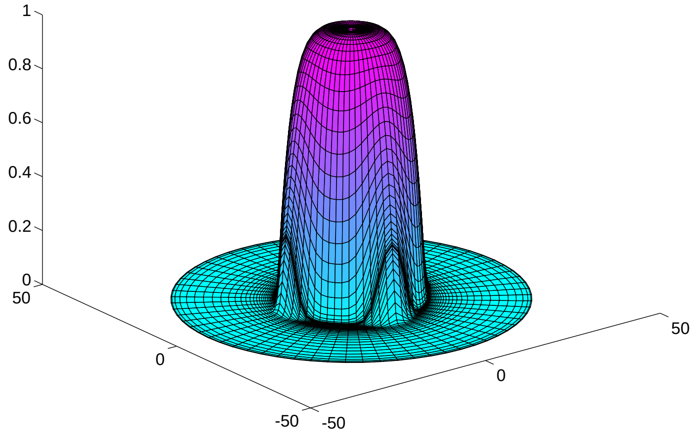

FIG. 8.7 - Champ de concentration pour la 1ère espèce A (solution stationnaire à 4 cellules).

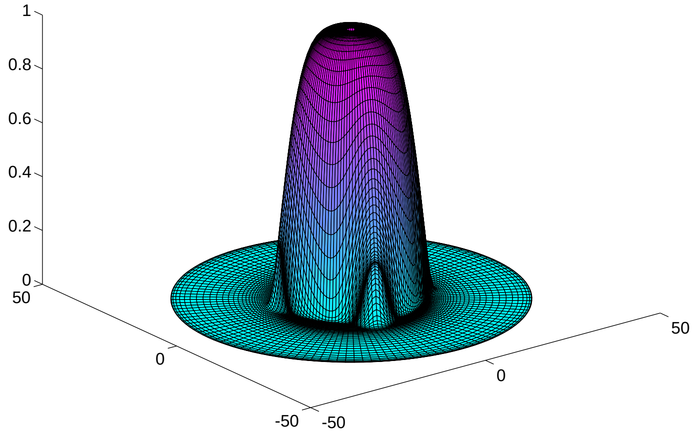

FIG. 8.8 - Champ de concentration pour la 1ère espèce A (onde à 4 cellules).

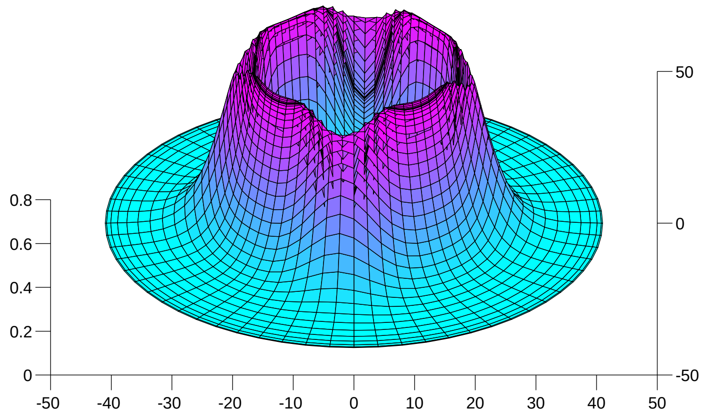

FIG. 8.9 - Champ de concentration pour la 2ème espèce B (solution stationnaire à 4 cellules).

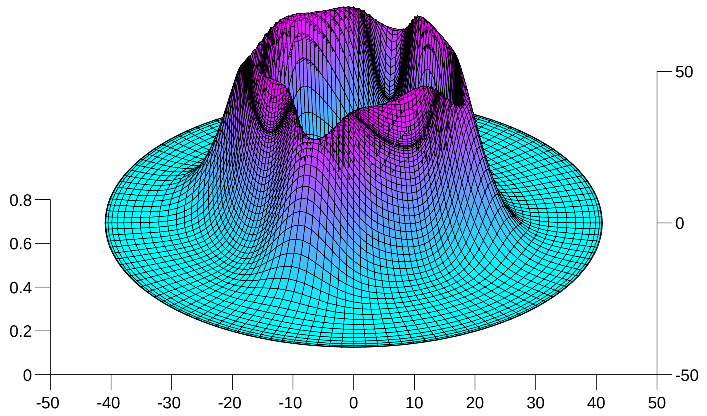

FIG. 8.10 - Champ de concentration pour la 2ème espèce B (onde à 4 cellules).

Chapitre 9

Conclusion et perspectives

Nous avons présenté dans cette thèse une série d'études de bibliothèques de communications, de recouvrements calculs/communications, d'algèbre linéaire parallèle et de transformée de Fourier multidimensionnelle et enfin une application pour la résolution sur une machine parallèle de la simulation de front de flammes. Nous donnons des travaux futurs sur les points abordés dans la thèse et des perspectives d'avenir sur le sujet.

9.1 Travaux futurs

Il est bien entendu évident que de nombreux travaux restent encore à faire autour des sujets abordés dans cette thèse. Nous allons à présent passer en revue chaque chapitre et suggérer quels travaux peuvent être effectués par la suite.

Les routines de communication

Il s'agit ici d'un sujet en plein développement, avec les changements récents de modèles de routage des machines du commerce ainsi que les perspectives que nous offrent de nouveaux modèles comme l'optique. Les routines étudiées dans ce chapitre peuvent être transformées pour tenir compte de contraintes technologiques liées aux nouveaux modèles. Les réseaux optiques offrent de nouvelles possibilités très intéressantes et les algorithmes utilisant la reconfiguration présentés ici peuvent être utilisés avec de tels composants. De nouvelles études de complexité pourraient être étudiées qui donneraient de nouveaux paramètres pour les algorithmes. Comme nous l'avons montré dans ce chapitre, les liens entre les méthodes utilisant la reconfiguration et le routage wormhole sont plus qu'étroits, les techniques de communications présentées peuvent subir des transformations (notamment dans la numérotation des processeurs), permettant une adaptation à des topologies fixes et à de tels routages. De plus, avec l'apparition du processeur T9000 et de son circuit de routage C104, les algorithmes présentés pourront également être étudiés avec le modèle de communication correspondant. Il serait donc intéressant d'offrir à l'utilisateur une bibliothèque de communications basée sur les algorithmes présentés mais avec un modèle wormhole. Ceci sera fait avec comme machine cible la Paragon d'Intel.

En ce qui concerne les diffusions enchaînées, des algorithmes utilisant d'autres modèles et d'autres topologies sont en cours d'étude. Il serait également intéressant d'ajouter une contrainte avec un coût de calcul ajouté. La borne inférieure est trop forte et sera peut être affinée.

Les LOCCS

Cette bibliothèque, dont l'intérêt n'est plus à démontrer, nécessite encore de nombreuses études, notamment du côté du calcul de la taille de paquet optimale et d'une interface utilisateur orientée objet. L'utilisation du langage Fortran n'a pas été abordée ici et des études sont en cours pour permettre aux nombreux utilisateurs de ce langage d'avoir accès aux méthodes macro-pipelines transparentes. Des implémentations de cette bibliothèque sur d'autres plate-formes comme PVM sont également en cours afin d'avoir un produit fini, disponible via le réseau. Des applications utilisant les LOCCS sont par ailleurs étudiées, notamment pour la simulation et l'analyse d'images. Les interactions avec un compilateur de type HPF devraient être étudiées plus en profondeur avec leurs concepteurs pour permettre un développement plus rapide des parties du compilateur optimisant les communications.

BLAS 3 distribuées

Un projet LHPC pour EUROTOPS concerne l'étude et l'implémentation d'une bibliothèque d'algèbre linéaire de type BLAS et LAPACK sur une machine parallèle à base de T9000. Certaines études sont encore nécessaires pour définir l'interface utilisateur, régler les problèmes de distribution et de rémanence des données dans le réseau, enfin et surtout le développement d'une bibliothèque complète testée et optimisée représente un travail très important. L'étude de la bibliothèque la plus indépendante de la topologie reste à faire. Concernant le produit de matrices, il serait intéressant d'étudier des algorithmes hybrides, reprenant les meilleures caractéristiques de chacun des algorithmes présentés. Il serait également possible d'avoir des algorithmes "adaptatifs", le choix de tel ou tel algorithme se faisant suivant les caractéristiques de l'appel courant comme la taille et la forme du réseau, la taille et la forme de la matrice, ... Les BLAS séquentiels pourraient également tirer avantageusement partie des algorithmes rapides tels que Strassen ou Winograd, en gardant bien entendu une stabilité acceptable.

LAPACK distribué

Les algorithmes présentés dans cette thèse ne représentent qu'une infime partie de la bibliothèque LAPACK. D'autres algorithmes doivent encore être parallélisés et de nombreuses études doivent être initiées. Ce sujet est en plein développement et fait également partie du projet LHPC EUROTOPS. Des études (et implémentations) restent à faire en ce qui concerne les interfaces orientées objets des routines de ScaLAPACK. De plus, nous avons commencé l'étude de méthodes permettant un déséquilibrage de la distribution de la matrice permettant une plus grande efficacité lors de l'exécution de l'algorithme de factorisation LU. Cet équilibrage des charges pourrait permettre une meilleure efficacité des processeurs.

La FFT

Les nouveaux modèles de communications offrent de nouvelles possibilités en permettant d'accéder à n'importe quel processeur du réseau avec le même coût que pour ses voisins. Par contre, les contraintes liées aux conflits sur les liens empêchent d'utiliser efficacement une topologie de processeurs comme si on avait affaire à un réseau complètement connecté. La quasi-disparition de l'hypercube nous conduit également à revoir les algorithmes en tenant compte de la topologie sous-jacente pour une plus grande efficacité. Les algorithmes présentés dans cette thèse peuvent être modifiés pour tenir compte de nouvelles distributions de matrices et de nouvelles topologies. Il existe par ailleurs de nouveaux algorithmes de FFT qui nous permettrons peut être d'obtenir des temps d'exécution plus courts.

Un problème réel

Si les algorithmes parallèles d'algèbre linéaire sont maintenant bien connus, leur utilisation dans le cadre de la parallélisation d'applications est un sujet totalement nouveau. Avec la sortie de machines efficaces et équipées de nombreux logiciels, de plus en plus de chercheurs d'autres disciplines chercheront à paralléliser leurs codes sur des machines à mémoire distribuée. Toutes les bibliothèques présentées dans cette thèse ont leur utilité dans de telles applications et de nouvelles routines devront être créées pour répondre à ces nouveaux besoins.

Concernant la simulation de front de flammes, certaines optimisations peuvent encore être faites, notamment sur le plongement d'anneaux d'anneaux dans une grille quelconque. Le second niveau de parallélisation utilisant la décomposition de domaine est en cours de développement et promet des résultats intéressants. Le troisième niveau de parallélisme, qui concerne le traitement des trois composantes Teta et les deux concentrations sur des groupes de processeurs différents n'a pas été abordé dans cette thèse à cause d'une limitation dans le nombre de processeurs utilisables. Une étude est en cours pour ajouter cet autre niveau de parallélisme qui pose, de nouveau, le problème du plongement de topologies adaptées dans la grille. Par ailleurs, de nombreux autres problèmes du même type restent encore à paralléliser.

9.2 Perspectives

Si à présent, il n'y a personne pour douter de l'avenir des machines parallèles à mémoire distribuée, le douloureux problème de leur programmation reste un sujet en pleine lumière. Les langages de type HPF sont très prometteurs quant à leur facilité d'utilisation, mais leur effective performance sur les machines actuelles dans un futur proche semble plus qu'incertaine, notamment pour les problèmes non réguliers. Nos bibliothèques peuvent aider le développeur d'un compilateur pour un tel langage à obtenir facilement des performances intéressantes sur des noyaux de calculs, de communications ou de calculs et communications recouverts. De plus, comme nous l'avons montré pour les différents algorithmes de calculs présentés, sans l'utilisation de compilateurs paralléliseurs et avec l'aide de bibliothèques et d'outils appropriés, des performances très intéressantes peuvent être obtenues avec un code qui reste simple et portable.

Les bibliothèques présentées dans la thèse seront utilisées de manière intensive sur machines parallèles à mémoire distribuée dans un futur très proche. Une étude est en cours pour installer des routines LOCCS dans la bibliothèque ScaLAPACK afin d'obtenir des meilleures performances grâce au macro-pipeline.

La facilité d'utilisation et les performances sont les clés du développement de ces machines et de leur entrée dans le monde industriel. Les performances des processeurs sont là, mais les moyens de les obtenir pour n'importe quel type de problème n'existent pas encore. Les bibliothèques représentent une solution à moyen terme pour les utilisateurs mais à long terme pour les compilateurs. En effet, un compilateur paralléliseur aura tout profit à être capable de détecter des parties de codes pouvant être remplacées par un appel à une bibliothèque optimisée par le constructeur pour sa machine cible. Plus les bibliothèques seront portables (et portées), plus le développement des compilateurs sera simple et le résultat performant.

Annexe A

Spécification des routines LOCCS

Dans cette annexe, nous donnons une spécification des différentes routines LOCCS et des routines annexes telles qu'elles ont été implémentées sur nos machines cibles. Cette présentation des routines n'est pas complète, un rapport technique suivra, qui servira de guide utilisateur précis. Le but de cette annexe est de montrer la portée des routines et de donner une idée de leur utilisation.

A.1 Les paramètres

Dans ce paragraphe, nous décrivons les paramètres que l'on retrouve dans toutes les routines. Si des paramètres particuliers sont nécessaires, ils seront donnés avec la description de la routine. Les routines de création et de destruction des buffers sont également décrites.

A.1.1 Le contexte

Ces paramètres font référence à tous les éléments généraux nécessaires à une routine LOCCS. Ce sont les paramètres :

- `me` : numéro de processeur de l'appelant,
- **une structure :** particulière suivant la routine (`oto`, `ota`, `exchange`, `shift`, `ato`, `ata`, `pata`). Son type est `sOto`, `sOta`, `sExchange`, `sShift`, `sAto`, `sAta`, `sPata`. Elle contient les numéros des participants pour les routines n'utilisant pas tous les processeurs (`loccs_oto`, `loccs_exchange`, `loccs_shift`) et la racine pour les routines globales `loccs_ota` et `loccs_ato`. Pour les deux dernières routines globales (`loccs_ata` et `loccs_pata`), elle n'est pas utilisée.

Pour suivre le principe "SPMD", tous les participants à une routine LOCCS possèdent les mêmes paramètres. Seul le numéro de processeur `me` distingue les processeurs entre eux et permet à la routine d'exécuter les bonnes parties de code, de manière transparente pour l'utilisateur.

A.1.2 Les buffers

Afin de simplifier l'utilisation des routines, nous avons donné à l'utilisateur, s'il le souhaite, la possibilité d'allouer automatiquement les buffers nécessaires au macro-pipeline avant la routine. Pour cela, des routines ont été créées, une par routine LOCCS. Ces routines ont le format suivant :

- `loccs_init_XXX(paramètres)`

où `XXX` est la fin du nom de la routine LOCCS appelée (`OTO, OTA, ...`). En effet, l'allocation des buffers n'est pas la même suivant la routine utilisée. Les paramètres diffèrent également suivant la routine mais quatre paramètres sont présents sur toutes les routines :

- `sBufferXXX` : est un pointeur sur une structure qui contiendra les adresses des buffers (Les buffers alloués sont de type `void`).
- ν est la taille des paquets en octets.
- `sizeDATA` taille en octets des données manipulées.
- `type` est le type des buffers.

```
typedef struct {
   char sizedata;  /* taille des données élémentaires     */
   void *buf1;     /* pointeur sur le 1er buffer alloué   */
   void *buf2;     /* pointeur sur le 2eme buffer alloué  */
   void *buf3;     /* pointeur sur le 3eme buffer alloué  */
} sBufferOto;
```

Une routine d'initialisation renvoie 0 s'il n'y a pas eu de problème d'allocation mémoire, -1 sinon. Une routine de désallocation des buffers alloués existe également. Cette routine a le format suivant :

- `loccs_close_XXX(buffer, type)`

A.1.3 La taille des paquets

La taille des paquets est ν. Cette taille doit être calculée avant l'appel à la routine LOCCS pour la première version de la bibliothèque.

A.1.4 Les tâches

Ce sont les parties calcul des routines. Elles sont définies par l'utilisateur et sont divisées en deux parties distinctes : job_{before} et job_{after}.

job_{before} est la tâche qui doit être exécutée sur un processeur avant la communication.

job_{after} est la tâche qui doit être exécutée sur un processeur après la communication.

Une tâche est décrite par la structure `sJob` :

```
typedef struct {
   void (*job)();       /* pointeur sur une fonction              */
   int ** parameters;  /* pointeur sur un tableau de paramètres  */
} sJob;
```

A un instant donné, une tâche s'exécute avec comme donnée en provenance du macro-pipeline un paquet i de taille ν (cette taille peut varier si le paquet est le dernier). Ce paquet est rangé dans un buffer dont l'adresse a été donnée soit par l'utilisateur, soit par la routine LOCCS d'initialisation de buffers. La séquence d'appel d'une tâche est donnée par :

$job_{before|after}$`(context,parameters,buffer)`

```
sContexte *context;
int **parameters;
type **buffer;
```

Où la structure `context` est donnée par :

```
typedef struct {
   int Nu;        /* taille du paquet (sauf le dernier)  */
   int nu_c;      /* taille du paquet courant            */
   int num;       /* numéro du paquet                    */
   int sender;    /* numéro de l'envoyeur                */
} sContexte;
```

L'utilisateur n'est pas obligé de définir une tâche si elle n'est pas nécessaire. Il suffit de mettre la constante prédéfinie `NOJOB` à la place du pointeur sur la structure. Dans ce cas, il n'y aura pas d'appel.

A.2 Les routines LOCCS

L'ensemble des routines LOCCS est volontairement important car il correspond aux principaux schémas classiques de communication de l'algorithmique parallèle. Par contre, les routines de communications globales non-bloquantes ne sont pas disponibles sur toutes les machines, ce qui nous empêche de pouvoir implémenter toute la bibliothèque. Une implémentation était toutefois possible en utilisant les capacités de routage de ces machines et en utilisant un algorithme de communication naïf, mais les performances ne seraient valables que pour des nombres de processeurs très petits. La bibliothèque ne serait donc pas scalable.

L'ensemble des routines présenté se découpe donc en deux sous-ensembles : celui des routines implémentées et testées sur iPSC/860 et Paragon (`loccs_oto`, `loccs_exchange`, `loccs_ota` et `loccs_shift`) et celui des routines uniquement spécifiées (`loccs_ato`, `loccs_pota`, `loccs_ata` et `loccs_pata`). Nous ne présentons ici que les routines implémentées et testées. La description des autres routines peut être trouvée dans un rapport futur.

A.2.1 Routine d'échange : `loccs_exchange`

Cette routine est très proche de la routine `loccs_oto` puisqu'elle met en œuvre deux processeurs, mais cette fois, les deux processeurs ont le même comportement. La communication est donc un envoi et une réception.

Séquence d'appel

Un appel à la routine `loccs_exchange` est réalisé par :

– loccs_exchange(me, exchange, buffer, ν, size, job_b, job_a)

```
int         me;
sExchange   exchange;
sBufferOto  buffer;
int         ν;
int         size;
sJob        *job_b;
sJob        *job_a;
```

Paramètres

Nous avons :

`me`:	Numéro logique de l'appelant de `loccs_exchange`

`exchange`:	`typedef struct {`
	`int partner1; /* Num. log. d'un des deux nœuds */`
	`int partner2; /* Num. log. de l'autre nœud */`
	`} sExchange;`

`buffer`:	Buffers des messages
ν:	Taille d'un paquet
`size`:	Taille totale du message
job_b:	Tâche devant être faite sur un paquet avt l'éch.
job_a:	Tâche devant être faite sur un paquet après l'éch.

Remarques concernant l'implémentation

Attention, la taille des messages échangés doit être la même. De la même manière, nous pouvons optimiser la communication en utilisant des chemins disjoints.

A.2.2 Routine de diffusion : `loccs_ota`

Dans cette routine, nous cherchons à combiner la diffusion d'un message d'un processeur vers tous les autres et des calculs avant et après diffusion.

Séquence d'appel

Un appel à la routine `loccs_ota` est réalisé par :

- loccs_ota(me, ota, buffer, ν, size, job_b, job_a)

```
int         me;
sOta        ota;
sBufferOta  buffer;
int         ν;
int         size;
sJob        job_b;
sJob        job_a;
```

Paramètres

Nous avons :

`me` :	Numéro logique de l'appelant de `loccs_ota`
`ota` :	`typedef struct {`
	`int root;` `/* Numéro logique de la racine de la diffusion */`
	`} sOta;`
`buffer` :	Buffers messages
ν :	Taille d'un paquet
`size` :	Taille totale du message
job_b :	Tâche devant être faite sur un paquet avt la com.
job_a :	Tâche devant être faite sur un paquet après com.

Remarques concernant l'implémentation

Cette routine pourra être optimisée suivant la machine cible. Plus le coût de la diffusion est petit, plus le grain de calcul peut être réduit et de ce fait l'équilibrage plus important. Une variation de cette routine ne concernant qu'un sous-ensemble de tous les processeurs du réseau serait également très utile. L'opération de diffusion associée est alors appelée *multicast*.

A.2.3 Routine de shift : `loccs_shift`

Comme nous l'avons montré précédemment, cette routine propose une généralisation des schémas de communication de type shift (Chap. 5, Par. 5.5.1).

Séquence d'appel

Un appel à la routine `loccs_shift` est réalisé par :

- `loccs_shift(me, shift, buffer,` ν`, size,` job_b`,` job_a`)`

int	me;
sShift	*shift;
sBufferShift	*buffer;
int	ν;
int	size;
sJob	$*job_b$;
sJob	$*job_a$;

Paramètres

Nous avons :

`me` : Numéro logique de l'appelant de `loccs_shift`

`shift` :

```
typedef struct {
   int node;            /* Numéro du processeur                          */
   int *senders;        /* table des nœuds qui envoient un message
                           au processeur                                 */
   int *receivers;      /* table des nœuds qui reçoivent un message
                           du processeur                                 */
   int taille;          /* taille des tables                             */
   void (*change)();    /* fonction devant être appliquée à la table
                        des nœuds de qui le processeur reçoit            */
} sShift;
```

`buffer` : Buffers des messages

ν : Taille d'un paquet

`size` : Taille totale des messages

job_b : Tâches devant être faites sur les paquets avt la com.

- Avant la réception si l'appelant est le nœud
- Avant l'envoi si l'appelant est un `sender`
- Avant la réception si l'appelant est un `receiver`

job_a : Tâches devant être faites sur les paquets avt la com.

- Après la réception et avant l'envoi si l'appelant est le nœud
- Après l'envoi si l'appelant est un `sender`
- Après la réception si l'appelant est un `receiver`

Remarques concernant l'implémentation

Une routine de shift simplifié mettant en jeu uniquement trois processeurs (une racine, un pour les réceptions et un pour les émissions) est à l'étude. Cela permettrait de réduire la partie remplissage des structures pour ce type de problèmes. Nous verrons dans l'Annexe B que pour la somme globale, la plus grosse partie du code concerne le remplissage des structures LOCCS.

Annexe B

Programme de la réduction utilisant les LOCCS

Dans cette annexe, nous donnons le code LOCCS permettant d'effectuer une somme globale de vecteurs à l'aide de la routine LOCCS `loccs_shift`. Remarquons que la majeure partie du code concerne l'initialisation des structures et pourrait être commune à différents appels. L'utilisation d'une routine aussi générale que `loccs_shift` nécessite la mise en place de nombreux paramètres. Une routine de shift à trois processeurs est également à l'étude.

```
/* ---------------------------------------- */
/* Somme globale anneau sur iPSC860 d'Intel */
/* ---------------------------------------- */

#include "definitions.h"   /* Definitions des structures LOCCS   */
#include <stdio.h>
#include <cube.h>
#define NU 3000

/*--- positionne le buffer sur la partie du vecteur a envoyer ---*/
void position(contexte,parametres,buffer)
sContexte   *contexte;
int      **parametres;
double      **buffer;
{
 (*buffer) = (double*)(&(((double*)(parametres[0]))[contexte->Nu*contexte->num]));
}

/*--- additionne le vecteur local avec le vecteur recu
       et met le resultat dans le buffer ---*/
void calc(contexte,parametres,buffer)
sContexte   *contexte;
int      **parametres;
double      **buffer;
{
 int   i;
 double   *add;

 add = (double*)(&(((double*)(parametres[0]))[contexte->Nu*contexte->num]));
 for(i=0; i<contexte->nu_c; i++) (*buffer)[i] += add[i];
}

/*--- idem que calc mais mets le resultat dans le vecteur local ---*/
void calcRoot(contexte,parametres,buffer)
sContexte   *contexte;
int      **parametres;
double      **buffer;
{
 int   i;
 double   *add;
```

```
 add = (double*)(&(((double*)(parametres[0]))[contexte->Nu*contexte->num]));
 for(i=0; i<contexte->nu_c; i++) add[i] += (*buffer)[i];
}


/*--- Global combine sur le vecteur de taille donne --*/
void gc(vecteur,taille)
double   *vecteur;
int   taille;
{
 int me,       /* my id                    */
    pos,       /* position dans l'anneau   */
    n;         /* nombre total de noeud    */
 char type;    /* type du buffer utilise   */
 sShift   shift;
 sJob *jobb, *joba;
 sBufferShift buffer;
 int **paramBefore, **paramAfter;
 int err;

 n=numnodes(); me=mynode(); pos=ginv(me);
 if (pos==0) return; /* le noeud 0 n'execute pas le GC */

 jobb = (sJob*)malloc(sizeof(sJob));
 joba = (sJob*)malloc(sizeof(sJob));

 paramBefore = (int**)malloc(sizeof(int*));
 paramAfter = (int**)malloc(sizeof(int*));
 paramBefore[0] = paramAfter[0] = (int*)vecteur;
 shift.change = NULL;

 if (pos==n/2) {      /* centre   */
  type = BUF;
  shift.node = me;
  shift.senders = (int*)malloc(2*sizeof(int));
  shift.receivers = (int*)malloc(2*sizeof(int));
  shift.senders[0] = gray(n/2+1);
  shift.senders[1] = gray(n/2-1);
  shift.receivers[0] = shift.receivers[1] = -1;
  shift.taille = 2;
  jobb = NOJOB;
  joba->job = calcRoot;
  joba->parameters = paramAfter;
 }
 else {
  shift.senders = (int*)malloc(sizeof(int));
```

```
  shift.receivers = (int*)malloc(sizeof(int));
  shift.taille = 1;
  if (pos == 1 || pos == n-1) {   /* extremite   */
    type = NOBUF;
    jobb->job = position;
    jobb->parameters = paramBefore;
    joba = NOJOB;
    shift.senders[0] = me;
    if (pos == n-1) {
     shift.node = gray(pos-1);
     shift.receivers[0] = gray(pos-2);
    }
    else {
     shift.node = gray(pos+1);
     shift.receivers[0] = gray(pos+2);
    }
  }
  else { /* ni centre, ni extremite   */
    type = BUF;
    jobb = NOJOB;
    joba->job = calc;
    joba->parameters = paramAfter;
    shift.node = me;
    shift.senders[0] = gray(pos+1);
    shift.receivers[0] = gray(pos-1);
    if (pos < (n/2)) {   /* de gauche a droite   */
      shift.senders[0] = gray(pos-1);
      shift.receivers[0] = gray(pos+1);
    }
  }
 }

 err = initShift(&buffer, NU, sizeof(double), type, me, &shift);
 if (err < 0) { fprintf(stderr,"Probleme d'allocation memoire\n"); exit(-1);}

 /* ///////
   APPEL DE LA ROUTINEE LOCCS
   ////////             */
 loccs_shift(me, &shift, &buffer, NU, taille, jobb, joba);

 closeShift(&buffer,type);
}
```

Annexe C

Publications personnelles

Chapitre de livre

[CD94] M. Cosnard and F. Desprez. Quelques Architectures de Nouvelles Machines. In *Ecole d'Automne CAPA - Port d'Albret*. 1994.

Conférences Internationales avec comité de lecture et publication des actes

[BD92] C. Bonello and F. Desprez. Implementation of Linear Algebra and Communication Libraries on the TNode Reconfigurable Machine. In JJ. Dongarra and B. Tourancheau, editors, *Environments and Tools For Parallel Scientific Computing - St Hilaire du Touvet*, pages 41–52. Elsevier, September 1992.

[BDT93] C. Bonello, F. Desprez, and B. Tourancheau. Parallel Linear Algebra and Communication Subroutines on a Transputer Based Reconfigurable Machine. In S. Atkins and A.S. Wagner, editors, *NATUG6 - Transputer Research and Applications 6*, pages 21–38. IOS Press, 1993.

[CD93] C. Calvin and F. Desprez. Minimizing Communication Overhead Using Pipelining for Multi-Dimensional FFT on Distributed Memory Machines. In *Parallel Computing'93*, 1993.

[DG93] F. Desprez and M. Garbey. Parallel Computing of a Combustion Front. In *Parallel CFD'93 - Implementations and Results Using Parallel Computers*. North-Holland/Elsevier, 1993.

[DT91] F. Desprez and B. Tourancheau. Reconfiguration versus Fixed Topology. In *EPCA Bohn Workshop*, pages 1–10. European Community, 1991.

[DT92] F. Desprez and B. Tourancheau. A Theoretical Study of Reconfigurability for Basic Communication Algorithms. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *CONPAR 92 - VAPP V*, number 634 in Lecture Notes in Computer Science. Springer Verlag, 1992.

[DT93] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. In *High Performance Computing and Networking Conference - Amsterdam*, May 1993.

Conférences Internationales avec comité de lecture sans publication des actes

[DT91a] F. Desprez and B. Tourancheau. Matrix Multiplication and Matrix Transpose: Fixed Topology vs. Switching Network (poster). In *EDMCC2 Munich*, 1991.

[DT91b] F. Desprez and B. Tourancheau. Reconfiguration versus Fixed Topology. In *European Parallel Computing Action - Bonn Workshop*, pages 1–10. European Community, 1991.

Revues nationales avec comité de lecture

[DGJP93] F. Desprez, C. Gavoille, B. Jargot, and M. Pourzandi. Tests des Performances des Communications de la Machine VOLVOX IS-860. Technical Report 93-02, LIP-IMAG, March 1993. to appear in La lettre du Transputer et des calculateurs distribués.

[DT90] F. Desprez and B. Tourancheau. Modélisation des Performances des Communications sur le Tnode avec le Logical System Transputer Toolset. *La lettre du Transputer et des Calculateurs Distribués*, 7:65–72, September 1990.

Rapports de recherches et techniques

[DFL93] F. Desprez, E. Fleury, and M. Loi. T9000 et C104, La Nouvelle Génération de Transputers. Technical Report 93-01, LIP-IMAG, 1993.

[DFT93a] F. Desprez, A. Ferreira, and B. Tourancheau. Efficient Communication Operations in Reconfigurable Parallel Computers. Technical Report CS-93-209, The University of Tennessee - Knoxville USA, 1993.

[DFT93b] F. Desprez, P. Fraigniaud, and B. Tourancheau. Successive Broadcast on Hypercube. Technical Report CS-93-210, The University of Tennessee - Knoxville USA, 1993.

[DT93] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. Technical Report 92-44, LIP - ENS Lyon, December 1993.

Publications en cours de soumission

[BDT93] C. Bonello, F. Desprez, and B. Tourancheau. Basic Routines for Linear Algebra on a Reconfigurable Network. Submitted to Parallel Computing, 1993.

[CCD+93] C. Calvin, L. Colombet, F. Desprez, B. Jargot, P. Michallon, B. Tourancheau, and D. Trystram. Towards mixed computation - communication in scientific libraries. Submitted to the Scalable High Performance Computing Conference'94, 1993.

[DDT93] F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Complexity of LU Factorization with Efficient Pipelining and Overlap on a Multiprocessor. Submitted to Parallel Processing Letters, 1993.

[DFT93] F. Desprez, A. Ferreira, and B. Tourancheau. Efficient Communication Operations on Passive Optical Star Networks, 1993.

[DP93] F. Desprez and M. Pourzandi. A Comparison of Three Matrix Product Algorithms on the Intel Paragon and Archipel Volvox Machines. Submitted to the Scalable High Performance Computing Conference'94, 1993.

Références

[ABB+91] J.M. Adamo, J. Bonneville, C. Bonello, P. Moukeli, N. Alhafez, and L. Trejo. Developing a High Level Programming Environment for Supernode. In *Proceedings of Transputer Application - Glasgow, (UK)*, pages 632–637. IOS Press, 1991.

[ABD+91] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. Van de geijn. Blacs: Basic linear algebra communication subroutines. In *Sixth Distributed Memory Computing Conference*, Portland, , U.S.A., 1991.

[ADDM92] P.R. Amestoy, M.J. Daydé, I.S. Duff, and P. Morère. Linear Algebra Calculations on a Virtual Shared Memory Computer. Technical Report RT/APO/93/2, ENSEEIHT - Département Informatique N7 -I.R.I.T. Gpe Algorithmes Parallèles, 1992.

[AGGM90] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan. A Parallel FFT on an MIMD Machine. *Parallel Computing*, 15:61–74, 1990.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley Publishers, 1974.

[AL88] M. Ashworth and A.G. Lyne. A Segmented FFT Algorithm for Vector Computers. *Parallel Computing*, 6:217–224, 1988.

[Amd67] G.M. Amdhal. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conf. Proc.*, 30:483–485, 1967.

[Bai88] D.H. Bailey. Extra High Speed Matrix Multiplication on the CRAY-2. *SIAM Journal on Science and Statistical Computing*, 9(3), May 1988.

[BBHM92] A. Bayliss, T. Belytdchko, D. Hansen, and M. Minkoff. Adaptive Multi-Domain Spectral Methods. In Meurant Scroggs Keyes, Chan and Voigt, editors, *5th SIAM Conference on Domain Decomposition Methods for Partial Differential Equations*. SIAM Philadelphia, 1992.

[BBM91] A. Bauch, R. Braam, and E. Maehle. DAMP - A Dynamic Reconfigurable Multiprocessor System with a Distributed Switching Network. In *Proceedings of the 2nd European Distributed Memory Computing Conference - Munich*, Lecture Notes in Computer Science, pages 495–504. Springer Verlag, April 1991.

[BCH93] J. Bruck, R. Cypher, and C.T. Ho. Efficient Algorithms for the Index Operation in Message-Passing Systems. Technical Report RJ 9300 (80030), IBM Research Division - Almaden Research Center - San Jose, April 1993.

[BDD+89] C.H. Bischof, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. LAPACK Working Note: Provisional Contents. Technical Report 5, Mathematics and Computer Science Division - Argonne National Laboratory, 1989.

[BDL91] R.H. Bisseling, L. Daniel, and J.C. Loyens. Towards Peak Parallel LINPACK Performance on 400 Transputers. *Supercomputer*, VIII-5(45):20–27, September 1991.

[Ber89] J. Berntsen. Communication Efficient Matrix Multiplication on Hypercubes. *Parallel Computing*, 12:335–342, 1989.

[Ber92] J.C. Berthillier. Machines RISC: Des Prototypes Aux Superscalaires. *Publications Scientifiques et Techniques d'IBM France*, 3:9–37, June 1992.

[BGM93] A. Bayliss, M. Garbey, and B.J. Matkowsky. Pseudo-Spectral Domain Decomposition and the Approximation of Multiple Layers. Northwestern University Preprint, Evanston, IL, 1993.

[BGMM89] A. Bayliss, D. Gottlieb, B.J. Matkowsky, and M. Minkoff. An Adaptive Pseudo-Spectral Method For Reaction Diffusion Problems. *J. Comput. Phys.*, 81:421–443, 1989.

[BH93a] J. Bruck and C.T. Ho. Concatenating Data Optimally in Message-Passing Systems. Technical Report RJ 9191 (81499), IBM Research Division - Almaden Research Center - San Jose, January 1993.

[BH93b] J. Bruck and C.T. Ho. Efficient Global Combine Operations in Multi-Port Message-Passing Systems. Technical Report RJ 9333 (82457), IBM Research Division - Almaden Research Center - San Jose, May 1993.

[BLPG93] M. Barnett, R. Littlefield, D.G. Payne, and R. Van De Geijn. On the Efficiency of global Combine for 2-D Mesh With Wormhole Routing, 1993.

[BM90] P. Berger and P. Morere. Evaluation d'un Noyau BLAS 3 sur une Configuration Multi-Transputer. *La Lettre du Transputer et des Calculateurs Distribués*, March 1990.

[BMM87] A. Bayliss, B.J. Matkowsky, and M. Minkoff. Adaptive Pseudospectral Computation of Cellular Flame Stabilized by a Point Source. *Appl. Math. Letters*, 1:19–24, 1987.

[BMSV92] P. Bjorstad, F. Manne, T. Sorevik, and M. Vajtersic. Efficient Matrix Multiplication on SIMD Computers. *SIAM Journal on Matrix Analysis and Applications*, 13(1):386–401, January 1992.

[BNK93] A. Bar-Noy and S. Kipnis. Multiple Message Broadcasting in the Postal Model. In *International Parallel Processing Symposium*, pages 463–470. IEEE Computer Society Press, April 1993.

[BP86] A. Brass and G.S. Pawley. Two and three dimensional FFTs on highly parallel computers. *Parallel Computing*, 3:167–184, 1986.

[BPG91] M. Barnett, D.G. Payne, and R. Van De Geijn. Optimal Broadcasting in Mesh Connected Architectures, December 1991.

[Bré88] C. Brésillon. Transputer, Composant du Parallélisme. *Micro-Systèmes*, Février 1988.

[Bre92] R.P. Brent. The LINPACK Benchmark on the Fujitsu AP 1000. In H.J. Siegel, editor, *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 128–135. IEEE Computer Society Press, 1992.

[BS92] J.C. Bermond and M. Syska. Routage Wormhole et Canaux Virtuels. In M. Cosnard, M. Nivat, and Y. Robert, editors, *Algorithmique Parallèle*, Etude et Recherche en Informatique, chapter 10, pages 150–158. Masson, 1992.

[BS93] P.E. Bjorstad and T. Sorevik. Data-Parallel BLAS as a basis for LAPACK on Massively Parallel Computers. Technical Report 93-xx, Para//ab, Institutt for Informatikk, Bergen, Norway, 1993.

[Bur89] G.D. Burns. A Local Area Multicomputer. In *Fourth Conference on Hypercube, Concurrent Computers and Applications.* ACM, 1989.

[Can69] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.

[Cap87] P.R. Capello. Gaussian Elimination on a Hypercube Automaton. *Journal of Parallel and Distributed Computing*, 4:288–308, 1987.

[CDW92] J. Choi, J.J. Dongarra, and D.W. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools For Parallel Scientific Computing.* Elsevier, 1992.

[CDW93a] J. Choi, J.J. Dongarra, and D.W. Walker. Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers. Technical Report ORNL/TM-12309, Oak Ridge National Laboratory, October 1993. Submitted to Parallel Computing.

[CDW93b] J. Choi, J.J. Dongarra, and D.W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. Technical Report ORNL/TM-12252, Oak Ridge National Laboratory, April 1993.

[CG87] E. Chu and A. George. Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[CG90] E. Chu and A. George. QR Factorization of a Dense Matrix on a Hypercube Multiprocessor. *SIAM Journal on Science and Statistical Computing*, 11(5):990–1028, September 1990.

[Cha88] R.M. Chamberlain. Gray codes, Fast Fourier Transforms and hypercubes. *Parallel Computing*, 6:225–233, 1988.

[Cha90] H.P. Charles. Le Processeur i860. Technical Report 90-02, LIP-IMAG, 1990.

[Cha93] H.P. Charles. *De la Micro-Optimisation à l'Algorithmique Parallèle.* PhD thesis, Institut Polytechnique de Grenoble, February 1993.

[CHQZ87] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral Methods in Fluid Dynamics.* Springer Verlag - New York, 1987.

[Chu88] C. Y. Chu. Comparison of Two-Dimensional FFT Methods on the Hypercube. In Geoffrey Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, 1988.

[CLT92] M. Cosnard, M. Loi, and B. Tourancheau. A framework for data migrations on the hypercube. In *NATO Advanced Research Workshop - Software for Parallel Computation (Cetraro)*, June 1992.

[CMT93] L. Colombet, P. Michallon, and D. Trystram. Parallel Matrix-Vector Product on a Ring with (Almost) no Communication, 1993.

[Com92] R. Comerford. How DEC Developped Alpha. *IEEE Spectrum*, pages 26–31, July 1992.

[CRA92] CRAY Research Inc. *MPP Technology Preview*, 1992.

[CRT89] M. Cosnard, Y. Robert, and B. Tourancheau. Evaluating speedups on distributed memory architectures. *Parallel Computing*, 10:247–253, 1989.

[CT65] C.W. Cooley and J.W. Tuckey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.

[CT93a] C. Calvin and D. Trystram. Matrix Transpose for Block Allocation on Processor Networks. Technical Report APACHE-2, LMC-IMAG, INPG Grenoble, France, 1993.

[CT93b] M. Cosnard and D. Trystram. *Algorithmes et Architectures Parallèles.* Interéditions, 1993.

[CTV87] M. Cosnard, B. Tourancheau, and G. Villard. Gaussian elimination on message passing architecture. In *International Conference on Supercomputing*, Patras, Grece, June 1987. Springer-Verlag.

[CW76] H.J. Curnow and B.A. Wichmann. A Synthetic Benchmark. *Comput. Journal*, 19(1):43–49, 1976.

[DCDH90] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transaction on Mathematical Software*, 16(1):1–17, 1990.

[DCHH88] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Transaction on Mathematical Software*, 14(1):1–17, March 1988.

[DDC+87] J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. LAPACK Working Note: Prospectus for the Development of a Linear Algebra Library for High-Performances Computers. Technical Report 1, Mathematics and Computer Science Division - Argonne National Laboratory, 1987.

[DDGW93] J. Demmel, J.J. Dongarra, R. Van De Geijn, and D. Walker. LAPACK for Distributed Memory Architectures: The Next Generation. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 323–329. SIAM, 1993.

[DDK91] J. Demmel, J.J. Dongarra, and W. Kahan. LAPACK Working Note: LAPACK: On Designing Portable High-Performance Numerical Libraries. Technical Report 39, Department of Computer Science - University of Tennessee, 1991.

[DDP92] M.J. Dayde, I.S. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. Technical Report RT/APO/93/1, ENSEEIHT - Département Informatique N7 -I.R.I.T. Gpe Algorithmes Parallèles, 1992.

[DDSV91] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. Van Der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, 1991.

[DGMJ93] J.J. Dongarra, A. Geist, R. Manchek, and W. Jiang. Using PVM 3.0 to Run Grand Challenge Applications on a Heterogenous Network of Parallel Computers. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 873–877. SIAM, 1993.

[DGW92] J.J. Dongarra, R. Van De Geijn, and D.W. Walker. A Look at Dense Linear Algebra Libraries. Technical Report ORNL/TM-12126, Oak Ridge National Laboratory, July 1992.

[DHHW93] J.J. Dongarra, R. Hempel, A.J.G. Hey, and D.W. Walker. A Proposal for a User-Level, Message Passing Interface in a Distributed Memory Environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[DMS93] J.J. Dongarra, H.W. Meuer, and E. Strohmaier. TOP500 Supercomputers. Technical Report RUM 33/93, Computing Center - university of Mannheim - Germany, July 1993.

[DNS81] E. Dekel, D. Nassimi, and S. Sahni. Parallel Matrix and Graph Algorithms. *SIAM Journal on Computing*, 10(4), November 1981.

[DO90] J.J. Dongarra and S. Ostrouchov. LAPACK Working Note: LAPACK Block Factorization Algorithms on the Intel iPSC/860. Technical Report 24, Department of Computer Science - University of Tennessee, 1990.

[Don90] J.J. Dongarra. Performance of Various Computers Using Standard Equations Software in a FORTRAN Environnement. *ACM Comput. Architecture News*, 18(1):17–31, March 1990.

[DPW93] J.J. Dongarra, R. Pozo, and D.W. Walker. An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures, 1993.

[DRT92] B. Dumitrescu, J.L. Roch, and D. Trystram. Fast Matrix Multiplication Algorithms on MIMD Architectures. Technical Report RT-87, Laboratoire LMC-IMAG, July 1992.

[DS86] W.J. Dally and C.L. Seitz. The Torus Routing Chip. *Journal of Parallel and Distributed Computing*, 1(3):187–196, 1986.

[DT89] L. Desbat and D. Trystram. Analysis of the Fast Fourier Transform on a Hypercube Vector-Parallel Computer. In J.L. Delaye and E. Gelenbe, editors, *High Performance Computing*, pages 143–152. Elsevier Science (North Holland), 1989.

[DW93] J.J. Dongarra and D. Walker. LAPACK Working Note: The Design of Linear Algebra Libraries for High Performance Computer. Technical Report 58, Department of Computer Science - University of Tennessee, 1993.

[ECGS92a] T. Von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Message: a Mecanism for Integrated Communication and Computation. Technical Report UCB/CSD 92/675, Computer Science Division - EECS - University of California, Berkeley, March 1992.

[ECGS92b] T. Von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Message: a Mecanism for Integrated Communication and Computation. In ACM IEEE Computer Society, editor, *The 19th Annual Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.

[Eck79] W. Eckhaus. Asymptotic Analysis of Singular Perturbations. In *Studies in Mathematics and its Applications*, volume 9. North-Holland Publ. Co., 1979.

[EGP89] U. Ehrenstein, H. Guillard, and R. Peyret. Flame Computations by a Chebyshev Multi-Domain Method. volume 9, pages 499–515, 1989.

[EK92] T. Von Eicken and D.E. Kuller. Building Communication Paradigms with the CM-5 Active Message Layer, July 1992.

[EK93] R. Esser and R. Knetcht. Intel Paragon XP/S - Architecture and Software Environment. Technical Report KFA-ZAM-IB-9305, Zentralinstitut fur Angewandte Mathematik - Forschungszentrum Julich, April 1993.

[Els90] A.C. Elster. Basic Matrix Subprograms for Distributed Memory Systems. In *Proceedings of DMCC5 - Charleston*, pages 311–316. IEEE Computer Society Press, 1990.

[FJL+88] G. Fox, M. Johnson, G. Lycenza, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors - General Techniques and Regular Problems*, volume 1. Prentice Hall, 1988.

[FL91] P. Fraigniaud and E. Lazard. Methods and Problems of Communication in Usual Networks. Technical Report 91-33, Laboratoire LIP, 1991.

[FN92] E.W. Felten and D. Mc Namee. Improving and the Performance of Message Passing Applications by Multithreading. In IEEE Computer Society, editor, *Scalable High Performance Computing Conference*, pages 84–89. IEEE Computer Society Press, April 1992.

[FO87] G.C. Fox and S.W. Otto. Matrix Algorithms on a Hypercube I: Matrix Multiplication. *Parallel Computing*, 4:17–31, 1987.

[For93] HPF Forum. High Performance Fortran Language Specification, May 1993. Version 1.0.

[Fra92] P. Fraigniaud. Communications dans un Réseau de Processeurs. In M. Cosnard, M. Nivat, and Y. Robert, editors, *Algorithmique Parallèle*, Etude et Recherche en Informatique, chapter 9, pages 134–147. Masson, 1992.

[FSSS92] R.D. Falgout, A. Skjellum, S.G. Smith, and C.H. Still. The *multicomputer toolbox* Approach to Concurrent BLAS and LACS. In *Scalable High Performance Computing Conference*, pages 121–128. IEEE Computer Society and The Institute of Electrical and Electronics Engineers, Inc., 1992.

[FW93] I. Foster and P.H. Worley. Parallelizing the Spectral Transform Method: A Comparison of Alternative Parallel Algorithms. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 100–107. SIAM, 1993.

[Gar94] M. Garbey. Domain Decomposition to Solve Layers and Asymptotic. *SIAM Journal on Scientific Computing*, 15(4), July 1994. To appear.

[GH85] G.A. Geist and M.T. Heath. Parallel Cholesky Factorization on a Hypercube Multiprocessor. Technical Report ORNL-6190, Oak Ridge National Laboratory, 1985.

[GH86] D. Gottlieb and R.S. Hirsh. Parallel pseudospectral domain decomposition techniques. ICASE Report, 1986.

[GHPW90] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. PICL: A Portable Instrumented Communication Library. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[GK90] A. Gupta and V. Kumar. The Scalability of FFT on Parallel Computers. Technical Report TR 90-20, Department of Computer Science - University of Minnesota - Minneapolis, October 1990. Revised October 1992.

[GK91a] A. Gupta and V. Kumar. Analysing Scalability of Parallel Algorithms and Architectures. Technical Report TR 91-18, Department of Computer Science - University of Minnesota - Minneapolis, November 1991. Revised November 1992.

[GK91b] A. Gupta and V. Kumar. Scalability of Parallel Algorithms for Matrix Multiplication. Technical Report TR 91-54, Department of Computer Science - University of Minnesota - Minneapolis, September 1991. Revised September 1992.

[GK92] A. Gupta and V. Kumar. Analysing Performance of Large Scale Parallel Systems. Technical Report TR 92-32, Department of Computer Science - University of Minnesota - Minneapolis, November 1992.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computation.* The John Hopkins University Press, 1989. Second edition.

[Gla91] B. Glass. SPARC Revealed. *Byte*, April 1991.

[GO77] D. Gottlieb and S.A. Orszag. Numerical analysis of spectral methods: Theory and applications. In *C.B.M.S.-N.S.F. Conference Series in Applied Mathematics.* SIAM, Philadelphia, 1977.

[GP88] H. Guillard and R. Peyret. On the Use of Spectral Methods for the Numerical Solution of Stiff Problems. *Comput. Methods Appl. Mech. Engrg.*, 66:17–43, 1988.

[GREC91] J. Gustafson, D. Rover, S. Elbert, and M. Carter. The Design of a Scalable, Fixed-Time Computer Benchmark, August 1991.

[Gus88a] J.L. Gustafson. Reevaluating Amdahl's Law. *Communication of the ACM*, 31(5), May 1988.

[Gus88b] J.L. Gustafson. The Scaled Sized Model: A Revision of Amdahl's Law. In L.P Kartashev and S.I. Kartashev, editors, *Supercomputing'88*, volume II, pages 130–133. International Computing Institute, 1988.

[Haw93] S. Hawkinson. Personal communication, 1993. Intel SSD Corp.

[HD89] D.P. Helmbold and C.E. Mc Dowell. Modeling Speedup(n) Greater than n. In F. Ris and P.M. Kogge, editors, *International Conference on Parallel Computing*, pages III–219–225. The Pennsylvania State University Press, 1989.

[HE91] M.T. Heath and J.A. Etheridge. Visualizing Performances of Parallel Programs. Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, July 1991.

[HHL86] S.M. Hedetniemi, S.T. Hedetniemi, and A.L. Liestman. A Survey of Gossiping and Broadcasting in Communication Network. *Networks*, 18:319–349, 1986.

[Hig89] N.J. Higham. Exploiting Fast Matrix Multiplication Within the Level 3 BLAS. Technical Report TR89-984, Dept of Computer Science - Center of Applied Mathematics - Cornell University, April 1989.

[Ho93] C.T. Ho. Matrix-Transpose on Meshes with Wormhole and XY Routing. Technical Report RJ 9385 (82637), IBM Research Division - Almaden Research Center - San Jose, June 1993.

[HP91] Y. Huang and Y. Paker. A Parallel FFT Algorithm for Transputer Networks. *Parallel Computing*, 17:895–906, 1991.

[HR91] C.T. Ho and M.T. Raghunath. Efficient Communication Primitives on Hypercubes. Technical Report RJ 7932, IBM Research Division, February 1991.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism Scalability Programmability*. Mc Graw Hill, Inc., 1993.

[Int91] Intel Corporation - SSD. *Toward TeraFLOP Performance: An Update on the Intel/DARPA Touchstone Program*, March 1991.

[JH86] S.L. Johnsson and C.T. Ho. Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes. Technical Report YALEU/DCS/TR-500, Department of Computer Science - Yale University, November 1986.

[JH87] S.L. Johnsson and C.T. Ho. Algorithms for Matrix Transposition on Boolean n-cube Configured Ensemble Architectures. Technical Report YALEU/DCS/TR-572, Department of Computer Science - Yale University, September 1987.

[JH89] S.L. Johnsson and C.T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transaction on Computers*, 9(38):1249–1268, 1989.

[JK92] S. L. Johnsson and R. L. Krawitz. Cooley-Tuckey FFT on the Connection Machine. *Parallel Computing*, 18:1201–1221, 1992.

[Joh87] S.L. Johnsson. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing*, 4:133–172, 1987.

[KCN88] C.T. King, W.H. Chu, and L.M. Ni. Pipelined Data Parallel Algorithms - Concept and Modeling. In *International Conference on Supercomputing*, pages 385–395, July 1988.

[KHJS93] B. Kumar, C.H. Huang, R.W. Johnson, and P. Sadayappan. A Tensor Product Formulation of Strassen's Matrix Multiplication Algorithm with Memory Reduction. In *International Parallel Processing Symposium*, pages 582–588. IEEE Computer Society Press, 1993.

[Kin88] C.T. King. *Pipelined Data Parallel Algorithm: Concept, Design and Modeling.* PhD thesis, Michigan State University - Department of Computer Science, 1988.

[KLL93] B. Kaagstrom, P. Ling, and Charles Van Loan. Portable High Performance GEMM Based Level 3 BLAS. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 339–346. SIAM, 1993.

[KN88] CT. King and L.M. Ni. Large Grain Pipelining on Hypercube Multiprocessor. In G. Fox California Institute of Technology, editor, *Third Conference on Hypercube Concurrent Computers and Applications*, volume II - Applications, 1988.

[LA91] D.L. Lee and M.A. Aboelaze. Linear Speedup of Winograd's Matrix Multiplication Algorithm Using an Array Processor. In *Sixth Distributed Memory Computing Conference*, pages 427–430. IEEE Computer Society Press, 1991.

[LC89] G. Li and T.F. Coleman. A New Method for Solving Triangular Systems on Distributed Memory Message Passing Multiprocessors. *SIAM Journal on Science and Statistical Computing*, 10:382–396, 1989.

[Lei85] C.E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transaction on Computers*, 34(10):892–901, October 1985.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transaction on Mathematical Software*, 5:308–323, 1979.

[LKK83] R.E. Lord, J.S. Kowalik, and S.P. Kumar. Solving Linear Algebraic Equations on a MIMD Computer. *Journal of the ACM*, 30(1):103–117, January 1983.

[LP92] Z. Lahjomri and T. Priol. KOAN: A Shared Virtual Memory for the iPSC/2 Hypercube. In *CONPAR 92 - VAPP V*, Lecture Notes in Computer Science, pages 441–452. Springer Verlag, 1992.

[LS92] C. Lin and L. Snyder. A Matrix Product Algorithm and its Comparative Performance on Hypercubes. In IEEE Computer Society, editor, *Scalable High Performance Computing Conference*, pages 190–193. IEEE Computer Society Press, April 1992.

[LT91] M. Loi and B. Tourancheau. Trollius: an easy-to-use programming environment for distributed multicomputers. In *Workshop ESPRIT Parallel Computing Action*, volume 4, pages 66–71, Bohn, RFA, may 1991.

[Mah86] F.H. Mc Mahon. The Livermore Fortran Kernels: A Computer Test for the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National laboratory, 1986.

[MJ91] K.K. Mathur and S.L. Johnsson. Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer. Technical Report 216, Thinking Machine Corporation, December 1991.

[Moy91] S.A. Moyer. Performance of the iPSC/860 Node Architecture. Technical Report IPC-TR-91-007, Institute for Parallel Computation - School of Engineering and Applied Science - University of Virginia, May 1991.

[PBKP92] B.V. Purushotham, A. Basu, P.S. Kumar, and L.M. Patnaik. Performance Estimation of LU Factorisation on Message Passing Multiprocessors. *Parallel Processing Letters*, 2(1):51–60, 1992.

[PBV92] S. Poinson, Tourancheau B., and X. Vigouroux. Distributed monitoring for scalable massively parallel machines. In J. Dongarra and B. Tourancheau, editors, *Environment and Tools for Parallel Scientific Computing*, Saint Hilaire du Touvet - France, September 1992. Elsevier Sciences Publisher.

[Pel87] J. Pelaez. Stability of Premixed Flames with Two Thin Reactions Layers. *SIAM J. Appl. Math.*, 47:781–799, 1987.

[Pie88] P. Pierce. The NX/2 Operating System. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[Rob90] Y. Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm.* Manchester University Press - Manchester - UK, 1990.

[RSRM93] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability Study of the KSR-1. Technical Report GIT-CC 03/03, College of Computing - Georgia Institute of Technology - Atlanta, 1993.

[RT88] Y. Robert and B. Tourancheau. Block gaussian elimination on a hypercube vector multiprocessor. Technical Report 37, TIM3-IMAG, June 1988.

[RTV89] Y. Robert, B. Tourancheau, and G. Villard. Data allocation strategies for the gauss and jordan algorithms on a ring of processors. *Information Processing Letters*, 31:21–29, 1989.

[Rum93] J.D. Rumeur. *Communication Dans Les Réseaux de Processeurs.* Masson, 1993.

[Saa86a] Y. Saad. Communication Complexity of the Gaussian Elimination Algorithm on Multiprocessors. *Linear Algebra and Applications*, 77:315–340, 1986.

[Saa86b] Y. Saad. Gaussian Elimination on Hypercubes. In M. Cosnard, Y. Robert, P. Quinton, and M. Tchuente, editors, *Parallel Algorithms and Architectures.* North-Holland, 1986.

[SB91] P.E. Strazdins and P. Brent. The Implementation of BLAS level 3 on the AP1000: Preliminary Report. Technical report, Department of Computer Science and Computer Science Laboratory - Australian National University, November 1991.

[SBOP91] R. A. Sweet, W. L. Briggs, S. Oliveira, and J. Porsche. FFTs and Three-Dimensional Poisson Solvers for Hypercubes. *Parallel Computing*, 17:121–131, 1991.

[SG91] X.H. Sun and J.L. Gustafson. Toward a Better Performance Metric. *Parallel Computing*, 17:1093–1109, 1991.

[Sit92] R.L. Sites. RISC Enters a New Generation. *Byte*, August 1992.

[SLF91] S.R. Seidel, M.H. Lee, and S. Fotebar. Concurrent Bidirectional Communications on the Intel iPSC/860 and iPSC/2. In *Sixth Distributed Memory Computing Conference*, pages 283–286, 1991.

[SS85] Y. Saad and M.H. Schultz. Data Communication in Hypercubes. Technical Report YALEU/DCS/RR-428, Department of Computer Science - Yale University, October 1985.

[SS88] Y. Saad and M.H. Schultz. Topological Properties of Hypercubes. *IEEE Transaction on Computers*, 37(7):867–871, July 1988.

[SS89] Y. Saad and M.H. Schultz. Data Communication in Parallel Architectures. *Journal of Parallel and Distributed Computing*, 6:115–135, 1989.

[Str69] V. Strassen. Gaussian Elimination is not Optimal. *Numer. Math.*, 13:354–356, 1969.

[SW87] Q.F. Stout and B. Wagar. Intensive Hypercube Communications. Technical report, University of Michigan - Computing Research Laboratory, 1987.

[Swa87] P. N. Swarztrauber. Multiprocessors FFTs. *Parallel Computing*, 5:197–210, 1987.

[Sys92] M. Syska. *Communications dans les Architectures à Mémoire Distribuée*. PhD thesis, Université de Nice - Sophia Antipolis, December 1992.

[Tou89] B. Tourancheau. *Algorithmes Parallèles pour les Machines à Mémoire Distribuée (Applications aux Algorithmes Matriciels)*. PhD thesis, INP-Grenoble, France, February 1989.

[Tse93] C.W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[TV89] D. Trystram and F. Vincent. Programmation Avancée du TRANSPUTER: Architecture et Mécanisme. *La Lettre du Transputer et des Calculateurs Distribués*, April 1989.

[Ube93] S. Ubeda. *Algorithmes d'Amincissement d'Images sur Machines Parallèles*. PhD thesis, Ecole Normale Supérieure de LYON - Université Claude Bernard LYON I, February 1993.

[Vel90] E.F. Van De Velde. Data Redistribution and Concurrency. *Parallel Computing*, 16:125–138, 1990.

[Wal88] D. W. Walker. Portable Programming within a Message-Passing Model: the FFT as an Example. In Geoffrey Fox California Institute of Technology, editor, *The Third Conference On Hypercube Concurrent Computers and Applications*, volume II - Applications, 1988.

[WD81] C.K. Westbrook and F.L. Dryer. Simplified Reaction Mechanism for the Oxidation of Hydrocarbon Fuels in Flames. *Comb. Sci. Tech.*, 27:31–43, 1981.

[WD84] R.P. Weicker and R. Dhrystone. A Synthetic Systems Programming Benchmark. *Communication of the ACM*, 27(10):1013–1030, 1984.

[Wei91] R.P. Weiker. A Detailed Look at Some Popular Benchmarks. *Parallel Computing*, 17:1153–1172, 1991.

[Win73] S. Winograd. Some Remarks on Fast Multiplication of Polynomials. In J.F. Traub, editor, *Complexity of Sequential and Parallel Numerical Algorithms*, pages 181–196. Academic Press, 1973.

[Wor91] J. Worlton. Towards a Taxonomy of Performance Metrics. *Parallel Computing*, 17:1073–1092, 1991.

[WWD91] D.W. Walker, P.H. Worley, and J.B. Drake. Parallelizing the Spectral Transform Method-Part II. Technical Report ORNL/TM-11855, Oak Ridge National Laboratory, July 1991.

[ZRB+90] E.L. Zapata, F.F. Riviera, I. Benavides, J.M. Carazo, and R. Peskin. Multidimensional Fast Fourier Transform into SIMD Hypercube. In *IEEE Proceedings*, volume 4-137, pages 253–260, July 1990.