Algorithmique parallèle pour les machines à mémoire distribuées (applications aux algorithmes matriciels)

Bernard Tourancheau

THESE
Présentée par:

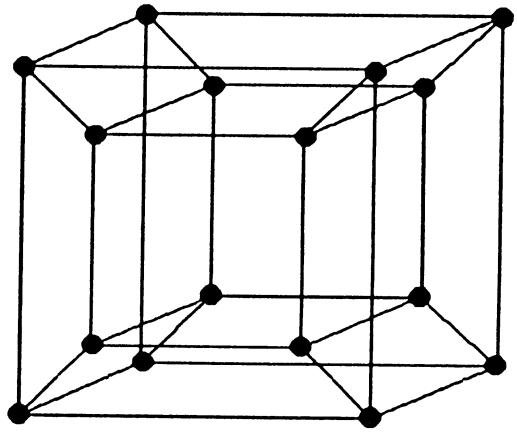
Bernard Tourancheau

pour obtenir le titre de **DOCTEUR**
de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**
(arrêté ministériel du 5 juillet 1984)

Spécialité: **Informatique**

Titre de la thèse:

Algorithmique Parallèle pour les machines à mémoire distribuée
(applications aux algorithmes matriciels)



Date de soutenance: 20 février 1989

Composition du jury:

Président:	Jean Pierre VERJUS
Rapporteurs:	Jean Claude BERMOND
	Yves ROBERT
Examinateurs:	Michel COSNARD
	Jacques LENFANT

Thèse préparée au sein du Laboratoire TIM3-IMAG de l'INPG.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD

Année 1988

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G
CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

E.F.P.
CHARUEL Robert

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Patience, patience,
Patience dans l'azur !
Chaque atome de silence
Est la chance d'un fruit mûr !

Paul Valéry

Personne ne sait comment sont exactement les choses quand on ne les regarde pas.

Hubert Reeves

A ma maman,
A béné,
et à tout ceux qui m'ont permis d'en arriver là .....!!

Je voudrais remercier et exprimer ma profonde reconnaissance à tous les membres du jury:

Jean Pierre Verjus, pour l'honneur qu'il me fait en présidant ce jury et aussi pour sa sympathie.

Jean Claude Bermond, pour avoir accepté d'être rapporteur de ce travail malgré les contraintes temporelles. Sa diligence, ses remarques et sa gentillesse m'ont beaucoup apporté.

Yves Robert, pour sa compétence, ses encouragements, son recul, bref tout et plus. Mais aussi pour ses amusements, ses fisherman's et parce qu'il m'a prété son transparent d'Edika.

Michel Cosnard, pour m'avoir supporté pendant presque 3 ans et surtout m'avoir donné goût à la recherche. Je crois que le Directeur de thèse, à la fois, père, tuteur, modèle, conseil, savoir, est tout entier en lui mais en plus il a su me donner de l'initiative, du bon temps et il m'attend lors de nos joggings !

Jacques Lenfant, pour avoir accepté de faire le déplacement depuis Rennes pour participer à ce jury.

Je veux également remercier:
Gilles Villard pour tous les travaux que nous avons effectués ensemble et toute l'énergie dépensée sur l'hypercube
El Mustafa Daoudi pour le travail que nous avons fait ensemble
Kuppuswami pour les travaux que nous avons collés ensemble
Denis Trystram car tout est de sa faute et puis c'est mon brico... heu... rando... heu... guitariste préféré
Afonso Ferreira, son amitié, la caipirinha, la samba de Bahia, etc.....
Jean Laurent Philippe pour sa sympathie et avoir corrigé mes fautes d'orthographe
Jean Michel Muller entre autre actionnaire dans toutes les patisseries du quartier
Bertrand Braschi pour n'avoir pas faibli à la 834635ième question sur Word ou Unix
Pierre Fraignaud pour savoir très bien lire l'anglais
tous les membres de l'équipe d'algorithmique et de calcul formel parallèle du laboratoire pour leur gentillesse et leur sympathie

Je tiens à remercier aussi Françoise Renzetti et toute l'équipe de la Médiathèque, sans qui nous ne pourrions travailler efficacement, ainsi que le services de reprographie de l'IMAG.

SOMMAIRE

INTRODUCTION

La complexité des problèmes à traiter et leur nombre évoluent de manière exponentielle , nous cherchons par quels moyens atteindre la puissance de calcul nécessaire pour les résoudre.

Aujourd'hui, la demande en puissance de calcul est sans cesse croissante dans le domaine des applications scientifiques et des simulations de toutes sortes.

Les puissances actuelles sont loin de satisfaire la demande. En effet, le produit a suscité le besoin [IBM]: lorsque sont apparus les premiers calculateurs performants, un grand nombre d'applications ont été développées à petite échelle, par exemple des modèles mathématiques de la physique et de la chimie. Réalisant le potentiel disponible, les utilisateurs ont compliqué leurs modèles pour mieux représenter la réalité et améliorer leurs résultats. La demande en temps de calcul s'est ainsi accrue.

A partir de cette simulation de la réalité, validant les concepts, le pas vers les extrapolations a été vite franchi. Une grande part du temps de calcul de tous les supercalculateurs mondiaux est maintenant utilisé pour la simulation de phènomènes divers. Je cite quelques domaines ayant recours aux simulations sur ordinateurs parmi la multitude existant: la cristallographie, la mécanique des fluides, de la tectonique des plaques terrestres, l'économie, la météorologie, la génétique, la conception de circuit, etc ...

La qualité d'un ordinateur se mesure par le nombre de millions d'opérations qu'il est capable d'effectuer en une seconde (lorsque les opérations sont effectuées sur des nombres flottants, l'unité est le Mflops (Mega floating point operations per second), lorsque ce sont des opérations de base en Mips (Mega instructions per second)).

Actuellement, les plus gros ordinateurs dont on dispose ont une puissance réelle de quelques dizaines de Mflops, dans le meilleur des cas [Her] (les constructeurs annoncent des puissances bien supérieures, mais il s'agit de résultats exprimés dans les cas les plus favorables avec une parfaite utilisation de toutes les ressources, ce qui est très rare dans la réalité).

De nombreux travaux font mention de l'état de l'art actuel sur les calculateurs les plus performants [Her].

Les besoins en puissance de calcul ont maintenant rejoint et dépassé la courbe représentant l'amélioration des performances des machines [Her].

Cela représente un "challenge" suffisant pour les recherches en informatique, mais je crois qu'une autre raison apparaît. L'informatique s'affirmant en temps que science, doit poursuivre des buts inaccessibles. Comme la physique raffinant de plus en plus ses connaissances sur la structure élémentaire de la matière, les informaticiens poursuivent la performance pure dans le domaine des machines, ou selon divers critères de qualité dans les autres domaines.

Devant ce défi, quelles sont les solutions développées par l'esprit humain?

Solution 1: Augmenter la vitesse du traitement de l'information: les limites physiques

C'est en tirant le meilleur parti des solutions existantes que l'on est arrivé aux performances actuelles. Le développement de la technologie des circuits intégrés à très haute intégration (VLSI pour Very Large Scale Integration) a permis d'accroître considérablement les performances des ordinateurs.

Mais la vitesse des composants atteint maintenant des limites. A la base de toute opération de l'ordinateur, il y a les portes logiques, matérialisées par des transistors. Dans les années 1970, on a assisté à un développement considérable de la technologie des circuits intégrés sur Silicium.

Pour obtenir une bonne rapidité il faut diminuer le temps de commutation des transistors, pour cela deux paramètres entrent en compte:

- La taille du transistor afin de réduire le temps de commutation en diminuant la longueur du trajet des charges.
- La célérité des électrons dans le matériau utilisé (elle représente la vitesse de déplacement des charges). C'est une caractéristique physique du matériau et de l'état (température surtout) dans lequel on l'utilise.

Le premier critère peut être satisfait en augmentant la densité des circuits intégrés. Aujourd'hui, les composants basés sur la technologie au Silicium atteignent des tailles de l'ordre du micron et l'on semble proche des limites à la fois technologiques et physiques de ce genre de portes logiques.

Le deuxième critère, en pleine évolution, introduit des changements radicaux de technologie et il existe des axes de recherches pour tenter de découvrir, de nouveaux semi-conducteurs, par exemple l'Arséniure de Gallium (AsGa) ou d'utiliser des propriétés supraconductrices (Effet Josephson).

La technologie AsGa est déjà une réalité, puisque certains composants existants sont déjà basés sur ce principe. Elle permet de créer des composants plus rapides. Pour un même champ électrique, les électrons sont accélérés davantage. Un problème cependant, le prix de revient est nettement plus élevé que dans le cadre de la technologie actuelle au Silicium.

Dans les travaux sur les phénomènes supraconducteurs utilisant l'effet Josephson, des progrès sont annoncés régulièrement, mais on est encore loin de pouvoir disposer d'une architecture basée sur ces principes et les recherches dans ce domaine sont importantes.

De toutes les façons, la vitesse de la lumière constitue une limite physique infranchissable, pour la célérité des déplacements de charges.

En une nanoseconde, une particule parcourt trente centimètres à la vitesse de la lumière, et un transistor commute. La technologie n'est pas loin des limites. On peut prédire, sans risques, que le coût pour obtenir un gain dans cette voie sera exponentiel car on se rapproche des limites physiques.

Solution 2: Améliorer les algorithmes

Le nombre d'opérations élémentaires nécessaires pour résoudre un problème dépend de l'algorithme utilisé. La vitesse d'un ordinateur pour un problème donné est donc fonction de la qualité algorithmique de la solution.

Il y a une trentaine d'années, Cooley et Tuckey ont proposé un algorithme très rapide pour calculer les transformées de Fourier discrètes de n échantillons d'un signal: la fameuse FFT (Fast Fourier Transform). L'exécution séquentielle de l'algorithme "simple" pour un problème de taille n, coutait $O(n^2)$ opérations arithmétiques. Avec beaucoup de clairvoyance et au prix d'un peu de stockage mémoire pour réutiliser des résultats précédemment calculés, l'algorithme de la FFT permet de gagner un ordre de grandeur sur la complexité en ramenant à $O(n\log_2(n))$ le nombre d'opérations arithmétiques nécessaires.

Ce simple exemple montre combien la recherche algorithmique peut être importante dans la course au gain en vitesse d'exécution pour des problèmes donnés.

La limite reste la complexité intrinsèque du problème. Si les gains algorithmiques sont importants pour les problèmes dont la complexité est polynomiale, il reste une large classe de problèmes où les gains algorithmiques sont non-significatifs car leur complexité est exponentielle en fonction de leur taille n. La classe des problèmes NP-complets (voyageur de commerce, sac à dos, ...) fait partie de ces problèmes intrinsèquement difficiles, dont l'ordinateur ne pourra jamais calculer la solution exacte lorsque n est grand.

Solution 3: Multiplier les unités qui traitent l'information

Nous venons de voir les limites des deux solutions classiquement (!)

utilisées. Depuis le début des années 70, une autre voie, prometteuse pour l'avenir, est apparue. Elle provient de la remise en question de l'architecture classique des ordinateurs avec l'introduction du concept de "parallélisme".

L'idée qui préside au parallélisme est simple :

Lorsque différentes parties d'une tâche sont indépendantes, il est possible d'envisager leur résolution simultanément pourvu que l'on dispose de plusieurs unités de traitement adéquates. On dit alors que l'on effectue la résolution de ces parties en parallèle et que la tâche est parallélisable.

Le meilleur exemple de la vie courante est la tâche représentant la construction d'une maison. Certaines parties (ou sous-tâches) sont dépendantes: on ne peut pas construire le toit avant d'avoir fini les murs par exemple. D'autres peuvent être exécutés en parallèle: par exemple électricité et pose des fenêtres, pourvu que l'on dispose d'un éléctricien et d'un maçon. Meilleur sera le chef de chantier (l'algorithmicien) meilleurs seront les délais de construction de la maison (le temps d'exécution du programme).

Dans un premier temps le parallèlisme fut utilisé au niveau du matériel, plusieurs unités fonctionnelles étant pilotées par un seul séquenceur, et une instruction contenait en fait plusieurs instructions.

Ensuite à un niveau encore plus bas: les opérations élémentaires ont été découpées en opérations encore plus fines à l'intérieur même des opérateurs. Cela a permis d'appliquer les techniques de pipeline sur les opérateurs ainsi découpés.

Aujourd'hui apparaissent sur le marché des calculateurs parallèles qui regroupent plusieurs processeurs indépendants reliés par un réseau d'interconnexions. Chacun des processeurs peut intègrer les solutions précédentes et toutes les améliorations technologiques. Dans la plupart des cas la brique de base est un microprocesseur puissant: c'est sans doute la solution la plus prometteuse pour obtenir des puissances élevées à faible coût.

Quelle en sont ses limites ?

Pour une taille totale de mémoire donnée, les limites sont sans aucun doute données par le degré d'indépendance des sous-tâches du problème à traiter, c'est à dire la part parallélisable du problème.

Dans le cas où la taille de la mémoire varie avec le nombre de processeurs, nous verrons que les limites sont moins nettes (§ 8).

Aujourd'hui, une seule limite semble prépondérante: les techniques algorithmiques (solution 2) pour utiliser ces ressources sont encore peu developpées, à tous les niveaux du logiciel.

En pratique, nous verrons aussi que la vitesse des accès mémoire (ou des communications) n'est pas encore du même ordre que la vitesse de calcul, qui a été obtenue après des années d'améliorations des processeurs séquentiels.

Nous allons étudier les machines correspondant à la solution 3 et tenter de concevoir de bonnes méthodes algorithmiques (solution 2). Pour dinstinguer les machines, le critère le plus important, à mes yeux, concerne le type d'accès aux données. Nous reprendrons par la suite la classification usuelle.

Répartition de la mémoire et classification des machines parallèles

Reprenons la troisième solution, un problème important est la manière dont seront connectés les processeurs entre eux. On peut choisir de les connecter par l'intermédiaire d'une mémoire. Les processeurs communiquent en partageant les données qu'ils utilisent. On parle de mémoire partagée. Chaque processeur peut communiquer directement avec tous ceux qui ont accès à la mémoire.

Si chaque processeur possède sa propre mémoire indépendante, la communication des informations peut avoir lieu grâce à un réseau d'interconnexions sur lequel transitent des messages. Chaque processeur ne peut alors communiquer directement qu'avec ses voisins directs sur le réseau. On parle alors de mémoire distribuée sur le réseau de processeurs.

Les communications vers des processeurs éloignés font appel au contrôle des processeurs intermédiaires ou "routage".

Sur les machines à mémoire partagée, les problèmes de blocage, d'écriture simultanée à un même emplacement, ... sont réglés par le matériel utilisé. Cela limite fortement le nombre de processeurs d'une telle machine mais facilite la programmation (transparence). Pour les machines à mémoire distribuée, certains processeurs n'étant pas reliés de manière directe, les capacités du réseau sont importantes. Il faut que le diamètre du graphe d'interconnexion soit faible pour ne pas pénaliser les performances de communication. De plus le degré maximum du graphe doit être limité car il est coûteux de gérer un réseau trop complexe. Sous ces deux conditions, le nombre de processeurs du réseau peut être important.

Ces deux approches différentes du parallélisme ont mené à des machines tout à fait opposées:

- supercalculateurs vectoriels multiprocesseurs à mémoire partagée:

  Cray, IBM, ... avec une dizaine de processeurs, ces machines sont "general purpose", i.e. destinées à tout type d'application.

- réseaux de processeurs à mémoire distribuée:

  Réseaux systoliques, Connection Machine, Ipsc, FPS T, Ncube,

Tnode, ... de la dizaine à la dizaine de milliers de processeurs suivant les configurations. Ces machines, le plus souvent expérimentales s'orientent plutôt vers la résolution de problèmes dédiés très parallèles.

Compte tenu de ces deux types, la taxinomie naturelle consiste à décomposer le parallèlisme en trois classes; suivant le mode de contrôle des séquences d'opérations élémentaires effectuées par les différents processeurs et distinguant flots de données et flots d'instructions [Fly].

Architecture pipeline:
C'est le parallélisme à un niveau très bas de l'architecture de la machine, sur les opérations élémentaires. Il nécessite un amorçage et n'est donc rentable que pour des séries d'opérations du même type, on parle de programmation vectorielle. Les opérations parallèles pipelinées possibles sont figées une fois pour toutes, mais le pipeline a l'avantage d'être peu coûteux, car il ne nécessite pas la duplication de toutes les ressources.
Mais surtout le pipeline est compatible (et complémentaire) avec les autres formes de parallélisme que nous détaillons ci-dessous.

Architecture de type SIMD (Single Instruction Multiple Data):.
Les processeurs peuvent intervenir complètement indépendamment. Ils sont chargés d'effectuer le même traitement sur des données différentes. L'ordinateur opère sur des vecteurs, et les manipulations sur des vecteurs deviennent alors des instructions élémentaires. En général, les instructions, opérateurs fonctionnels et données sont pipelinés pour permettre directement la manipulation de tableaux. Notons pour terminer que ce type d'ordinateur est un peu tombé en désuétude.
Mais il est facile de gérer beaucoup de cellules élémentaires avec cette approche: ce sont les "réseaux systoliques" [Kun]. Dans ce cas, l'initiative laissée aux processeurs est faible, et les machines restent dédiées à des types d'applications assez spécifiques, on parle alors de coprocesseurs systoliques.

Architecture de type MIMD (Multiple Instructions Multiple Data):.
Les processeurs peuvent être chargés d'exécuter des parties de code différentes. L'architecture se compose de plusieurs unités, d'une mémoire ou plusieurs mémoires et un réseau d'interconnexion et enfin d'un réseau d'interruptions pour pouvoir contrôler l'ensemble par exemple en cas de blocage ou de panne. Les algorithmes peuvent alors être asynchrones.
On dit qu'un ordinateur MIMD est fortement (faiblement) couplé si les interactions entre les processeurs sont (peu) importantes .

Cette classification est la plus ancienne et la plus connue. Elle ne permet pas cependant de distinguer les machines hybrides.

Notre propos dans la suite sera centré sur les machines MIMD à mémoire distribuée et les algorithmes qui permettent leur utilisation performante.

Pour justifier l'utilisation des algorithmes et des méthodes proposés nous userons, tour à tour, des expériences sur une machine hypercube et des analyses de complexité des algorithmes.

La ligne conductrice de l'organisation de ce travail est l'augmentation de la complexité des architectures et des modèles utilisés. Les premiers chapitres utilisent des modèles simples puis systoliques. Après la présentation de la machine MIMD à mémoire distribuée servant aux expérimentations, les autres chapitres portent sur des modèles se rapprochant le plus possible de la réalité expérimentale. Les résultats théoriques sont alors corroborés par les expériences.

RÉSUMÉ

Nous développons dans la suite, les résultats de nos recherches concernant l'algorithmique parallèle, pour des machines à mémoire distribuée du type MIMD.

Grâce aux expérimentations menées sur la machine hypercube, nous avons pu concevoir et affiner des méthodes de conception d'algorithmes parallèles.

Les modèles d'études analytiques qui en découlent permettent de comprendre le comportement des algorithmes sur des machines à mémoire distribuée et de le prévoir de manière théorique.

Des traits se dégagent et des qualités s'affirment être indispensables pour obtenir de bonnes performances algorithmiques avec les machines MIMD à mémoire distribuée.

Dans la première partie l'étude est centrée sur des architectures contenant des processeurs "simples". Cela permet de comprendre l'importance des communications dans les algorithmes utilisés en exemple.

Nous décrivons des résultats de complexité [§ 1, CDT] sur ces modèles pour la classe d'algorithmes matriciels de type élimination de Gauss. Ils donnent l'optimalité asymptotique de certains algorithmes et mettent en évidence l'importance de la manière dont l'hôte est relié au réseau de processeurs.

A partir de l'architecture linéaire de processeurs simples, nous montrons comment dériver un réseau systolique qui résout le même problème. Il est aussi performant que ceux de la littérature mais de surface inférieure [§ 2, CTT].

Dans la suite les processeurs sont du type de complexité de ceux des stations de travail performantes sur le marché aujourd'hui. Nous abordons des études plus réalistes, avec des applications en vraie grandeur sur une machine hypercube. Son architecture est décrite précisément. L'évaluation de ses performances tant sur le plan du calcul que des communications permet d'effectuer une comparaison avec diverses machines du commerce du type supercalculateurs [§ 3, STo2, TVi2].

Nous étudions ensuite divers algorithmes de communication des données entre processeurs sur cette machine [§ 4, Tou2], et nous en appliquons les résultats au problème des communications dans l'algorithme de Gauss.

Nous étudions alors l'influence de la répartition des données sur les performances de l'algorithme d'élimination de Gauss [§ 5, RTV, RT4]. Nous introduisons la répartition entrelacée par bloc des ligne (ou des colonnes) de la matrice initiale.

Nous appliquons les algorithmes de diffusion sur l'hypercube pour vérifier nos études analytiques et la stratégie de répartition.

Pour la topologie d'anneau et l'algorithme de Jordan, nous montrons que la répartition optimale est la répartition par blocs de taille maximale. Nous obtenons ainsi l'optimalité de l'algorithme sur cette architecture.

Avec l'algorithme de Gauss, nous donnons l'expression analytique de complexité qui permet de choisir le compromis de la taille des blocs donnant le meilleur temps d'exécution.

Pour la grille, nous donnons des résultats expérimentaux qui laissent penser que le comportement des algorithmes de Gauss et Jordan est de même type que celui de Gauss sur l'anneau.

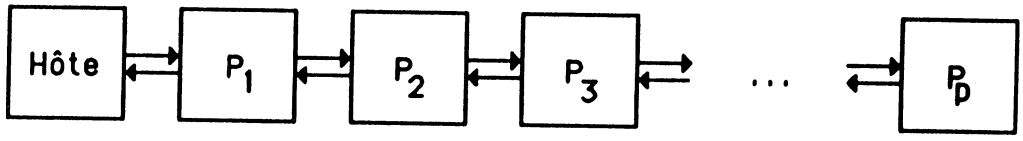
Avec les hypothèses classiques nous introduisons de nouveaux algorithmes sur l'anneau qui exploitent au mieux le parallèlisme de l'architecture sous-jacente pour le problème donné et diminuent la durée des communications interprocesseurs [§ 6, CTV].

Nous montrons que certaines techniques d'optimisations classiques pour les machines à mémoire partagée peuvent être adaptées pour améliorer la performance algorithmique sur les machines à mémoire répartie [§ 7, RT2, Tou1]. Elles accélèrent dans ce cas à la fois le calcul et les communications. Nous obtenons la performance de 42 Mflops pour l'élimination de Gauss avec un hypercube de 16 processeurs, ce qui place, au moins théoriquement, ce type de machine parmi les supercalculateurs

Pour la comparaison des performances des machines à mémoire distribuée, nous étudions l'accélération des algorithmes lorsque l'on augmente le nombre de processeurs suivant la méthode de Gustafson [§ 8, CRT], et nous dérivons une manière originale pour aborder ce problème analytiquement ou pratiquement.

Comme le montre les références, beaucoup de ces travaux ont été effectués en collaboration avec Michel Cosnard, El Mustafa Daoudi, Kuppuswami, Yves Robert, Maurice Tchuenté, G. Villard. Je voudrai leur témoigner ici ma reconnaissance et le grand plaisir que j'ai eu à travailler avec eux.

1. Algorithmique sur les réseaux linéaires 3



1. ALGORITHMIQUE SUR LES RÉSEAUX LINÉAIRES

Où nous montrons que le passage de mémoire partagée à mémoire distribuée dégrade peu les performances (facteur constant) et que des réseaux linéaires on déduit facilement des réseaux systoliques performants.

1.1. INTRODUCTION

Dans les premières machines parallèles commercialisées, l'architecture de base était constituée d'une large mémoire commune partagée par plusieurs processeurs. Les problèmes architecturaux résident essentiellement dans les difficultés d'accès à cette mémoire: le débit (largeur de bande) de la mémoire vers chaque processeur doit être important pour fournir les données aux opérateurs pipelines qui sont couramment utilisés.

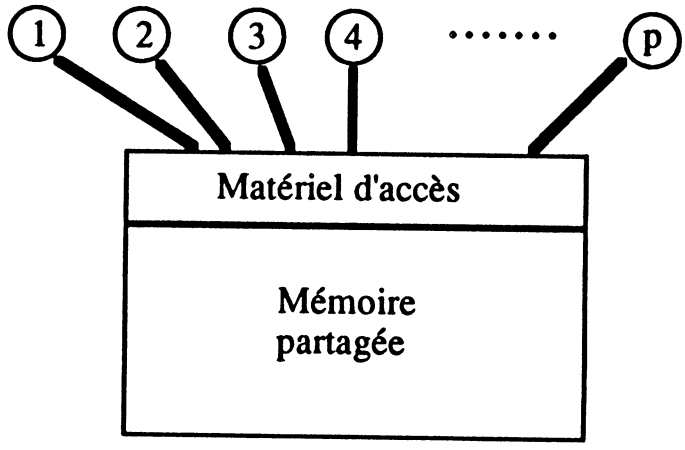


Figure 1.1: Machine à mémoire partagée (MP) avec p processeurs

Pour améliorer le débit de la mémoire vers les processeurs, de nombreuses solutions ont été proposées: partage en plusieurs bancs mémoire, hiérarchisation de la mémoire par l'introduction de caches, mémoires d'arrière plan, etc ...[Her][HBr]. Plusieurs de ces

supercalculateurs sont commercialement disponibles (IBM, Cray, Alliant). Ils utilisent tous de façon diverse les solutions citées.

Le logiciel système de ces machines tente d'obtenir, au travers de toutes ces facilités matérielles, une gestion optimale du parallélisme en garantissant l'intégrité des données stockées dans la mémoire commune. Les processeurs étant tous identiques, la méthodologie de conception d'algorithmes parallèles repose essentiellement sur des techniques d'ordonnancement de tâches [CRo2][RTr].

La contention mémoire limite le nombre de processeurs à la dizaine (IBM 6, 12 (expérimentale); Cray 4, 8; Alliant 8, 16 (en construction)), au delà, les performances des accès mémoire se dégradent.

Pour obtenir des machines à parallélisme massif (c'est à dire avec un nombre de processeurs supérieur à la centaine), on a donc recours à des architectures où la mémoire est décentralisée et où les informations sont échangées au travers d'un réseau d'interconnexion. Chaque processeur dispose d'une mémoire locale à accès rapide et n'est connecté qu'à un certain nombre de processeurs voisins.

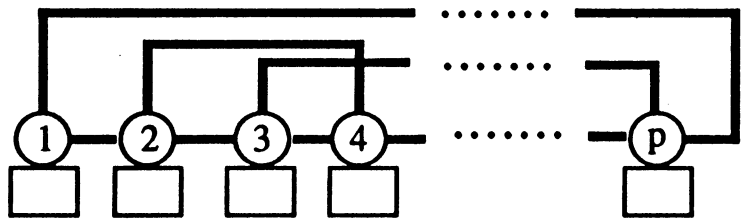


Figure 1.2: Machine à mémoire distribuée (MD) avec p processeurs (réseau d'interconnexions quelconque)

Les principes de base de ce type d'architecture ne sont pas nouveaux [HBr]. De nombreux prototypes ont vu le jour et les réseaux systoliques en sont directement inspirés [Kun]. Par contre ce n'est que récemment que sont apparus les premiers supercalculateurs construits suivant ce principe. Leur programmation est encore malaisée et, en dehors des algorithmes systoliques [QRo][Qui][CTc], il n'existe pas de méthodologie de conception d'algorithmes parallèles pour ces machines.

Dans le premier modèle, les processeurs travaillent en partageant des données de la mémoire commune: chaque processeur lit en mémoire les données dont il a besoin, effectue son traitement, puis écrit les résultats en mémoire. Le temps d'exécution ne dépend que du temps de calcul et des conflits d'accès mémoire (que l'on peut supposer limités avec un nombre faible de processeurs) [IBM][RRS].

Dans le second modèle, les processeurs travaillent en échangeant des messages: chaque processeur reçoit sur un ou plusieurs canaux de

communication les données extérieures dont il a besoin en provenance d'autres processeurs, effectue son traitement dans sa mémoire locale, puis transfère les résultats en direction des processeurs qui les utiliseront. Dans ce cas le temps d'exécution d'une suite de tâches ne dépend plus uniquement du temps d'exécution de chaque tâche, mais aussi du temps de communication des données nécessaires (opérandes, résultats). Ce temps dépend bien sûr de la topologie du réseau et de la répartition des données sur les processeurs (voir § 4 et 5).

Un algorithme séquentiel conduit à plusieurs versions parallèles sur une architecture à mémoire partagée [CRo2]. Un algorithme sur une architecture à mémoire partagée peut lui aussi conduire à plusieurs versions sur un réseau de processeurs.

Nous comparons dans la suite la complexité de trois algorithmes de base d'algèbre linéaire, Gauss, Givens et Jordan, sur différents modèles d'architectures multi-processeurs parallèles fonctionnant par partage des données dans une MP ou par échange de messages sur un réseau d'interconnexion.

L'approche que nous adoptons ici prend en compte le fait que dans le cas des mémoires partagées, les transferts de données entre processeurs ne sont pas négligeables devant le temps de calcul. L'unité de temps est prise égale au temps nécessaire à la réception des données, l'exécution d'une transformation et l'envoi des nouvelles valeurs. Notre but ici, n'est pas une analyse quantitative des résultats de plusieurs algorithmes sur une machine particulière, mais de montrer l'influence qualitative de l'architecture, en particulier des connexions entre processeurs, sur la complexité des algorithmes parallèles.

Les études sur la complexité des algorithmes d'algèbre linéaire sur une architecture à mémoire partagée sont nombreuses. Dans [Sam2], A. Sameh propose plusieurs algorithmes pour résoudre des problèmes d'algèbre linéaire sur un anneau de processeurs. Certains des algorithmes que nous présentons par la suite sont directement inspirés de ces résultats. Plusieurs méthodes de résolution de systèmes linéaires sont présentées dans [ISS] sur un anneau de processeurs, disposant d'un bus de communication. Dans ce modèle, le nombre de processeurs est petit devant la taille du problème. Dans [GRo], l'influence des communications sur la conception d'algorithmes parallèles est étudiée dans le cas d'architectures à mémoire partagée et de réseaux de processeurs à connexions locales. Y. Saad, [Saa1], examine les possibilités de communication de plusieurs architectures possédant un grand nombre de processeurs par rapport à la taille n du problème, en prenant comme algorithme test la méthode de Gauss. Son résultat principal montre que la prise en compte des communications ne permet pas d'obtenir des temps d'exécutions linéaires, car le temps de communication est de l'ordre de $O(n^2)$ quel que soit le nombre de

processeurs. Dans [Saa2], il étudie l'implémentation de l'algorithme de Gauss sur un hypercube de processeurs. Il montre que les sous réseaux d'anneau et de grille bidimensionnelle permettent de résoudre efficacement le problème en dupliquant des lignes de la matrice (grille) ou en les pipelinant (anneau). C'est de ce dernier principe que nous nous inspirons dans ce travail pour obtenir des algorithmes asymptotiquement optimaux.

Aprés avoir rappelé les résultats de complexité sur une architecture à mémoire partagée, nous étudions la complexité des trois méthodes sur un réseau linéaire et sur un anneau de processeurs. Le point commun de ces deux derniers réseaux réside dans le fait que les liens de communication inter-processeurs sont peu nombreux et que les échanges avec l'extérieur se font par l'intermédiaire d'un seul processeur (dans une certaine mesure privilègié). Nous montrons que pour ces algorithmes, le rôle joué par les communications entre les processeurs et l'hôte est important. La différence de complexité entre MP et MD sur ces réseaux n'est qu'un facteur constant, qui peut être compensé lorsque l'on utilise un plus grand nombre de processeurs avec une MD qu'avec une MP.

1.2. RAPPELS D'ORDONNANCEMENTS OPTIMAUX POUR LES MACHINES À MÉMOIRE PARTAGÉE

Supposons que l'architecture sous jacente soit une machine MIMD avec p processeurs. Ces processeurs communiquent par partage de données grâce à une mémoire commune à laquelle ils accèdent directement et indépendamment.

1.2.1. MÉTHODE DE GAUSS

Soit A une matrice de taille nxn. L'algorithme de Gauss permet de décomposer A en le produit de deux matrices triangulaires A=LU. Dans le cas de la résolution d'un système linéaire Ax=b, on borde la matrice A par le vecteur b et on effectue ainsi sur b les mêmes opérations que sur A. Le système se transforme en Ux=c où c= $L^{-1}b$ que l'on résoud directement car U est triangulaire supérieure. L'algorithme séquentiel est classique [GLo]. Il consiste à éliminer chacun des éléments sous diagonaux de A en utilisant deux lignes.

Nous appelons E(i,j,k), $i \neq j$, $1 \leq i,j \leq m$ et $1 \leq k \leq n$, la fonction d'élimination qui annule l'élément a_{ik} en utilisant les lignes i et j. Nous supposons que E s'exécute en un temps e. Par exemple, E(i,j,1) effectue une combinaison des lignes i et j de manière à annuler a_{i1}:

$$\begin{pmatrix} a_{j1} & a_{j2} & \dots & a_{jn} \\ 0 & \alpha_{i2} & \dots & \alpha_{in} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{a_{i1}}{a_{j1}} & 1 \end{pmatrix} \begin{pmatrix} a_{j1}\, a_{j2} \dots a_{jn} \\ a_{i1}\, a_{i2} \dots a_{in} \end{pmatrix}$$

Remarquons que seule la ligne i est modifiée. Par conséquent une même ligne j peut servir dans plusieurs éliminations sans duplication de données dans la mémoire. Dans l'algorithme séquentiel, à l'étape k on utilise la ligne k, appelée ligne pivot, pour éliminer successivement les éléments en position (k+1,k), ..., (n,k). Mesuré en nombre d'éliminations E le coût de l'algorithme est de n(n-1)/2 e unités (ce qui correspond au nombre d'éléments sous diagonaux de A).

Pour obtenir une plus grande stabilité numérique, on a généralement recours à une technique de pivotage qui consiste à rechercher par colonne l'élément maximal. Nous n'aborderons pas ici cet aspect essentiellement numérique. Notons cependant que les versions équivalentes par colonne de cet algorithme permettent de résoudre efficacement le problème.

Pour simplifier l'étude, l'unité de temps e correspond à l'exécution des opérations suivantes: calcul des coefficients de l'opérateur d'élimination et application de celui-ci (modification de la ligne i). En pratique, le temps d'exécution de l'application de la fonction dépend de la longueur des lignes en question donc de l'étape de l'élimination, mais nous ne prendrons pas ce paramètre en compte dans notre étude qualitative.

Dans la décomposition d'un problème en tâches en vue de la parallèlisation de son exécution, on définit la granularité comme la taille des tâches choisies. Ici la granularité est moyenne, la tâche E comprenant environ O(n) opérations élémentaires. La granularité la plus fine définit les tâches élémentaires égales aux opérations de base de la machine (+, -, *, ...). L'algorithme complet représente par contre la tache de base dans la granularité la plus grosse.

Un algorithme parallèle est alors un ensemble de triplets (E(i,j,k),prc,tps) où prc est le numéro du processeur exécutant E(i,j,k) et tps la date du début de son traitement. Sous ces hypothèses et en supposant que le traitement débute à la date un sur tous les processeurs, le temps d'exécution d'un algorithme correspond à la date de fin de traitement du dernier processeur actif.

Prenons p=n-1 et considérons l'algorithme GaussMP suivant : à la date t=k, pour q variant de k à n-1, le processeur q applique l'opérateur E(q+1,k,k) aux deux lignes k et q+1. Pour n=8, la trace temporelle de cet

algorithme est décrite dans la figure 1.3. L'entier t est placé en position (i,k) si l'élément correspondant a_{ik} a été annulé à la date t.

```
*
1  *
1  2  *
1  2  3  *
1  2  3  4  *
1  2  3  4  5  *
1  2  3  4  5  6  *
1  2  3  4  5  6  7  *
```

Figure 1.3: Date des éliminations de l'algorithme de GaussMP sur une architecture à mémoire partagée avec 7 processeurs et une matrice 8x8

Le temps d'exécution minimal d'un algorithme sera égal à la longueur de la plus longue suite d'opérations élémentaires intrinsèquement séquentielles nécessaire à la résolution du problème (c'est le chemin critique du graphe de dépendance des tâches [CMRT]).

Un algorithme est optimal lorsque son temps d'exécution est minimal. La taille des opérations élémentaires dépend de la définition de la granularité pour la décomposition du problème.

Théorème 1.1 [CMRT]: Sur une architecture à mémoire partagée, l'algorithme GaussMP est optimal. Son temps d'exécution est égal à (n-1)e. Le nombre minimum de processeurs pour exécuter cet algorithme en temps optimal est égal à n-1

1.2.2. MÉTHODE DE JORDAN

La méthode de Jordan permet de résoudre directement le système linéaire Ax=b, en éliminant tous les éléments non diagonaux [GLo]. La fonction d'élimination est la même que celle de la méthode de Gauss. Dans l'algorithme séquentiel, à l'étape k on utilise la ligne k, appelée ligne pivot, pour éliminer successivement les éléments en position (1,k), ..., (k-1,k), (k+1,k), ..., (n,k). Mesuré en nombre d'éliminations e le coût de l'algorithme est de $(n^2-n)e$ unités (ce qui correspond au nombre d'éléments non diagonaux de A).

La parallélisation de cet algorithme sur l'architecture à mémoire partagée décrite plus haut est évidente et reprend les idées du paragraphe précédent (voir [CRTr] pour une étude plus complète).

Théorème 1.2 [CMRT]: Sur une architecture à mémoire partagée, l'algorithme JordanMP est optimal. Son temps d'exécution est égal à n e. Le nombre minimum de processeurs pour exécuter un algorithme en temps optimal est égal à n.

```
* 2 3 4 5 6 7 8
1 * 3 4 5 6 7 8
1 2 * 4 5 6 7 8
1 2 3 * 5 6 7 8
1 2 3 4 * 6 7 8
1 2 3 4 5 * 7 8
1 2 3 4 5 6 * 8
1 2 3 4 5 6 7 *
```

Figure 1.4: Ordonnancement des éliminations de l'algorithme de JordanMP sur une architecture à mémoire partagée

1.2.3. MÉTHODE DE GIVENS

L'algorithme de Givens permet de construire la décomposition orthogonale de la matrice A en utilisant des rotations planes, c'est à dire de calculer une matrice triangulaire supérieure R telle que QA = R où Q est une matrice orthogonale. Dans le cas de la résolution d'un système linéaire Ax=b, on borde la matrice A par le vecteur b et on effectue aussi sur b les mêmes opérations que sur A. Le système se transforme en Rx=c où c=Qb que l'on résout directement. L'algorithme séquentiel est classique [GLo][Ste] et présente l'avantage d'une grande stabilité numérique (rotations). Nous appelons R(i,j,k), $i \neq j$, $1 \leq i,j \leq m$ et $1 \leq k \leq n$, la rotation dans le plan (i,j) qui annule l'élément a_{ik} en temps r.

Par exemple, R(i,j,1) effectue une combinaison des lignes i et j de manière à annuler a_{i1}:

$$\begin{pmatrix} \alpha_{j1} & \alpha_{j2} & \dots & \alpha_{jn} \\ 0 & \alpha_{i2} & \dots & \alpha_{in} \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a_{j1} & a_{j2} & \dots & a_{jn} \\ a_{i1} & a_{i2} & \dots & a_{in} \end{pmatrix}$$

Remarquons que, contrairement à l'algorithme de Gauss, les deux lignes servant à effectuer la rotation sont modifiées. Dans l'algorithme séquentiel, l'ordre des rotations est tel qu'un élément annulé précédemment n'est pas restauré par la suite: par exemple de gauche à droite et de bas en haut. Mesuré en nombre de rotations R le coût de l'algorithme est de n(n-1)/2 (ce qui correspond au nombre d'éléments sous diagonaux de A).

L'unité de temps r sera prise égale à l'exécution de la rotation sur les lignes i et j en mémoire.

L'idée de base pour paralléliser cet algorithme consiste à affecter les rotations indépendantes, c'est à dire portant sur des paires de lignes différentes, à des processeurs différents. Nous renvoyons à [LKK] [MCl] [SKu] [BBK] [CDMR] [CMR] [CRo1] [CTo1] pour des résultats sur l'implémentation de cet algorithme sur un réseau systolique et sur une architecture à mémoire partagée. Les principaux résultats concernant ce dernier type d'architecture sont rappelés dans la suite.

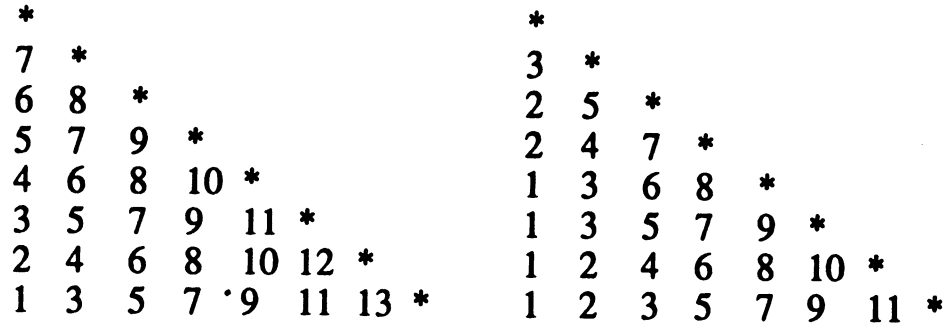
L'architecture considérée est la même que dans les deux cas précédents. Comme une rotation modifie les deux lignes utilisées, il est inutile de disposer de plus de $\lfloor n/2 \rfloor$ processeurs puisque l'on ne pourra faire plus de $\lfloor n/2 \rfloor$ rotations simultanément.

Nous allons présenter rapidemment deux algorithmes: l'algorithme SK de Sameh et Kuck et l'algorithme Glouton G. Une description détaillée de ces algorithmes se trouve dans [CRo1]. Pour n=8, l'ordre d'exécution des rotations dans les deux algorithmes est décrit dans la figure 1.5. L'entier t est placé en position (i,k) si l'élément correspondant a été annulé à la date t.

L'algorithme SK annule les éléments par colonnes de bas en haut: le traitement de la colonne k débute à la date 2k-1, c'est à dire dès que les deux premiers éléments de la colonne k-1 ont été annulés ce qui est nécessaire pour pouvoir effectuer une rotation portant sur la colonne k (les lignes correspondantes possèdent alors des zéros pour les positions 1 à k-1).

L'algorithme G consiste quant à lui à annuler à chaque instant le maximum d'éléments en effectuant toutes les rotations possibles.

Cette description n'est pas complète puisqu'elle ne précise pas quelle ligne j est utilisée pour générer la rotation R(i,j,k). On prend systématiquement j=i-1 dans le cas de l'algorithme SK. Dans le cas de l'algorithme G, si à la date t on exécute dans la colonne k les rotations $R(d,j_d,k),\ldots,R(s,j_s,k)$, on peut prendre $j_m=m+d-s-1$ $(d \leq m \leq s)$, ceci pour chaque colonne k où se trouve une série d'élimination.



Algorithme de Sameh et Kuck Algorithme Glouton

Figure 1.5: Date des rotations sur une architecture à mémoire partagée (matrice 8x8, 7 processeurs)

Il n'est pas difficile de voir que le temps d'exécution de l'algorithme SK est égal à 2n-3 (l'élément a_{nk} est annulé à t=2k-1, donc a_{nn-1} à t=2n-3). Par contre celui de l'algorithme G est plus difficile à évaluer Le résultat qui suit est tiré de [CRo1].

Théorème 1.3 [CRo1]: Sur une architecture à mémoire partagée, l'algorithme Glouton est optimal pour effectuer la décomposition QR. Son temps d'exécution est égal à (2n-o(n))r avec n≥2. Sur une architecture à mémoire partagée, l'algorithme de Sameh et Kuck est asymptotiquement optimal. Son temps d'exécution est égal à (2n-3)r avec n≥2. Le nombre minimum de processeurs pour exécuter ces algorithmes est égal à $\lfloor n/2 \rfloor$

Remarquons que [CDR] ont donné un algorithme asymptotiquement optimal pour la décomposition QR utilisant $n/(2+\sqrt{2})$ processeurs.

1.3. ALGORITHMES SUR UN RÉSEAU LINÉAIRE

Nous avons vu que l'idée de base pour paralléliser l'algorithme d'élimination de Gauss est d'affecter les éliminations indépendantes, c'est à dire portant sur des lignes différentes, à des processeurs différents. On peut aussi utiliser simultanément la ligne pivot dans plusieurs éliminations, si elle n'est pas modifiée par l'opérateur d'élimination.

L'architecture à mémoire partagée considérée dans la partie précédente peut être vue comme un réseau complet d'interconnexion entre des processeurs. Un tel réseau comporte $O(n^2)$ canaux de communication (n(n-1)/2) exactement). Ce réseau possède les possibilités maximales de communication et par conséquent, un algorithme pour un réseau particulier

de processeurs peut être exécuté sur une architecture à mémoire partagée. Bien entendu l'inverse n'est pas vrai !

Le réseau de processeurs possédant les possibilités minimales de communication est le réseau linéaire (le graphe connexe de degré minimal est la chaine [Ber]). Dans cette partie nous étudions l'implantation des méthodes précédentes sur un tel réseau.

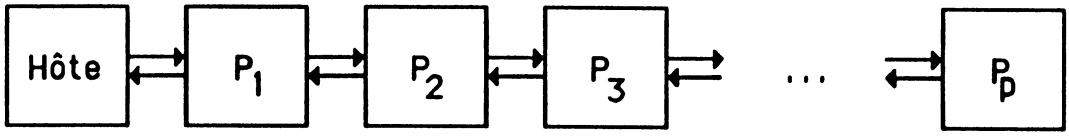


Figure 1.6: Réseau linéaire de processeurs

L'architecture considérée est représentée dans la figure ci dessus. L'ordinateur hôte communique avec le processeur P_1 par un canal d'émission et un canal de reception. Les processeurs restent des processeurs "simples" c'est à dire capables d'effectuer les opérations E et et R sur des données en mémoire locale. Chaque processeur P_i ($2 \leq i \leq p-1$) communique avec P_{i-1} et P_{i+1} aussi par l'intermédiaire d'un canal d'émission et d'un canal de réception. Chaque P_i ($1 \leq i \leq p$) possède une mémoire locale pouvant contenir 2 lignes de A.

Nous supposerons qu'en une unité de temps e, P_i est capable de recevoir une ligne sur un lien de communication, d'effectuer une élimination ou une rotation (quelle que soit la position qu'occupe l'élément à annuler) et de transmettre une ligne sur un lien de communication . En pratique, le temps d'exécution de la fonction dépend de la longueur des lignes en question donc de l'étape de l'élimination, mais nous ne prendrons pas ce paramètre en compte dans notre étude qualitative.

Nous supposerons aussi que deux communications sur des liens différents d'un même processeur ne pourront pas s'exécuter en parallèle, il n'y a pas de recouvrement (cette hypothèse est vérifiée si, par exemple, un seul DMA existe pour tous les canaux).

Au début de l'algorithme l'hôte contient la matrice A. A la fin, il doit contenir le résultat cherché.

1.3.1. MÉTHODE DE GIVENS

Nous allons tout d'abord étudier la méthode de Givens sur cette architecture. Il est clair qu'on ne peut pas implanter l'algorithme Glouton puisqu'il effectue plusieurs rotations à la date 1 et que la matrice n'est pas encore dans le réseau. L'algorithme de Sameh et Kuck s'implante par contre directement.

L'algorithme GivensRL1 consiste en deux étapes. Les lignes sont introduites dans le réseau les unes après les autres par le lien hôte-P_1, de la ligne n à la ligne 1. De t=1 à t=n-1, les lignes sont "pipelinées" dans le

réseau, chaque ligne restant deux unités de temps dans P_{i-1} avant d'être transférée dans P_i ($i \leq n-1$). A chaque unité de temps, P_i reçoit une ligne de P_{i-1}, effectue la rotation correspondante et envoie une ligne à P_{i+1}. A t=n, les processeurs P_i reçoivent une ligne de P_{i-1}, calculent puis envoient une ligne à P_{i-1}. Le processeur P_1 délivre la première ligne de la matrice R. De t=n+1 à t=2n-1, le sens de communication est renversé: P_i reçoit une ligne à partir de P_{i+1}, effectue la rotation correspondante et envoie le résultat. Pendant cette phase, le processeur P_1 délivre à chaque unité de temps une nouvelle ligne de la matrice R à l'ordinateur hôte. Remarquons que ce fonctionnement ne nécessite pas un contrôle global puisque le changement du sens de communication s'effectue lorsque le processeur P_i lit la ligne (2i-1). Il suffit donc de transmettre l'indice de la ligne avec celle-ci.

L'exécution de l'algorithme est décrite dans la figure 1.8. Les rotations sont effectuées dans le même ordre que dans l'algorithme de Sameh et Kuck.

	P_1		P_2		P_3		P_4	
t=1	8							
t=2	78	R(8,7,1)						
t=3	67	R(7,6,1)	8					
t=4	56	R(6,5,1)	78	R(8,7,2)				
t=5	45	R(5,4,1)	67	R(7,6,2)	8			
t=6	34	R(4,3,1)	56	R(6,5,2)	78	R(8,7,3)		
t=7	23	R(3,2,1)	45	R(5,4,2)	67	R(7,6,3)	8	
t=8	12	R(2,1,1)	34	R(4,3,2)	56	R(6,5,3)	78	R(8,7,4)
t=9	23	R(3,2,2)	45	R(5,4,3)	67	R(7,6,4)	8	
t=10	34	R(4,3,3)	56	R(6,5,4)	78	R(8,7,5)		
t=11	45	R(5,4,4)	67	R(7,6,5)	8			
t=12	56	R(6,5,5)	78	R(8,7,6)				
t=13	67	R(7,6,6)	8					
t=14	78	R(8,7,7)						
t=15	8							

Figure 1.7: Déroulement de l'algorithme GivensRL1 sur une matrice de taille (8x8), les ligne contenues dans les mémoires locales sont représentés pour chaque top.

Son temps d'exécution est égale à (2n-1)r si l'on suppose que la première et la dernière communication s'effectuent aussi en temps r. La différence entre les temps d'exécution des deux algorithmes (2r) provient de la lecture et de l'écriture sans élimination de la ligne 8 dans l'hôte aux instants 1 et 2n-1. On peut remarquer que ce temps d'exécution peut être plus "serré" car la première lecture et la dernière lecture ont une durée totale inférieure à r, ce qui donne $T_{SK} < (2n-2)r$ mais diminue la symétrie et la simplicité de la

présentation. Dans cette explication les dates correspondent à l'horloge de P_1 et ce n'est, bien sûr, pas une horloge globale, mais cela permet une projection plus simple de l'ordonnancement des processeurs.

L'ordre des éliminations de GivensRL1 n'est pas le seul possible, on peut aussi progresser de haut en bas et de gauche à droite par exemple. L'algorithme GivensRL2 consiste lui aussi en deux étapes. Les lignes sont introduites dans le réseau les unes après les autres, de la ligne 1 à la ligne n. Ce fonctionnement ne nécessite pas, lui non plus, un contrôle global puisque le changement de sens de communication s'effectue lorsque le processeur P_i lit la ligne (n-i+1). L'algorithme est décrit dans la figure 1.9. Le temps d'exécution de cet algorithme est aussi égal à (2n-1)r.

	P_1	P_2	P_3	P_4
t=1	1			
t=1	21	R(2,1,1)		
t=1	31	R(3,1,1)	2	
t=1	41	R(4,1,1)	32	R(3,2,2)
t=1	51	R(5,1,1)	42	R(4,2,2)	3	
t=1	61	R(6,1,1)	52	R(5,2,2)	43	R(4,3,3)	
t=1	71	R(7,1,1)	62	R(6,2,2)	53	R(5,3,3)	4	
t=1	81	R(8,1,1)	72	R(7,2,2)	63	R(6,3,3)	54	R(5,4,4)
t=1	82	R(8,2,2)	73	R(7,3,3)	64	R(6,4,4)	5	
t=1	83	R(8,3,3)	74	R(7,4,4)	65	R(6,5,5)		
t=1	84	R(8,4,4)	75	R(7,5,5)	6			
t=1	85	R(8,5,5)	76	R(7,6,6)				
t=1	86	R(8,6,6)	7					
t=1	87	R(8,7,7)						
t=1	8							

Figure 1.9: Déroulement de l'algorithme GivensRL2 sur une matrice de taille (8,8)

On remarque que P_4 n'est utilisé que pour une seule rotation dans les 2 algorithmes, ce qui bien sur est du à la faible connectivité du réseau.

La figure 1.10 donne la date d'annulation des éléments de la matrice A dans les deux algorithmes.

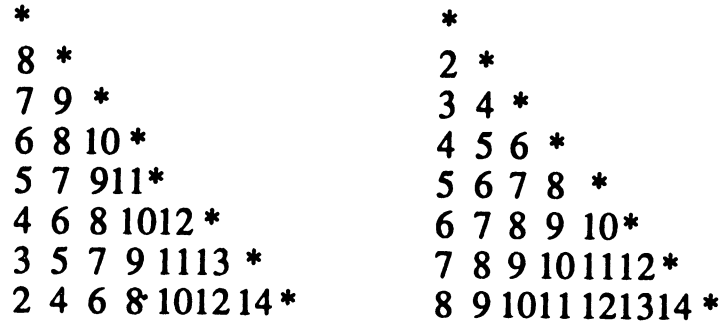


Figure 1.10: Date d'exécution des rotations dans les algorithmes GivensRL1 et GivensRL2 (matrice 8x8, 4 processeurs)

Théorème 1.4: Sur un réseau linéaire, les algorithmes GivensRL1 et GivensRL2 sont optimaux pour calculer la décomposition QR d'une matrice carrée. Leur temps d'exécution sont égaux à (2n-1)r. Le nombre optimal de processeurs pour exécuter ces algorithmes en temps optimal est égal à $\lceil n/2 \rceil$.

Démonstration :

Il faut n unités de temps pour transférer les n lignes de A de l'hôte vers P_1 (un seul canal) et il faut n unités de temps pour transférer les n lignes de R de P_1 vers l'hôte (idem). Si l'étape n de l'algorithme fait une émission puis une réception vers l'hôte il suffit, pour montrer l'optimalité des deux algorithmes, de prouver qu'il est impossible qu'une ligne ne soit sortie définitivement du réseau avant que toutes les lignes de A ne soient entrées.

Remarquons que si un élément sort non définitivement, il devra ré-entrer, ce qui, avec l'hypothèse de non recouvrement des communications, augmente le temps total.

Pour l'étape 1 de l'élimination de Givens, la ligne 1 doit effectuer la dernière élimination car elle est la seule qui restera de longueur n. Lorsque cette (dernière) élimination est effectuée, toutes les lignes sont entrées et ont été éliminées. Comme une seule ligne peut rentrer à chaque top on est donc au moins au top n.

A chaque étape k, la ligne k doit effectuer la dernière élimination car elle est la seule qui restera de longueur n-k+1. A chaque étape k, les lignes qui peuvent être définitivement hors du réseau sont donc les lignes d'indice < k.

Comme une seule ligne peut sortir à chaque top et que les sorties débutent au top n (après la dernière élimination de l'étape 1), la dernière sortie aura lieu au plus tôt lors du top 2n-1.

Appelons Rot(t) le nombre de rotations qu'exécute à la date t l'algorithme optimal considéré. Comme les lignes de A ne peuvent être introduites et

délivrées que séquentiellement, nous déduisons que $Rot(t) \leq \lfloor t/2 \rfloor$ pour $1 \leq t \leq n$ et $Rot(t) \leq \lfloor (2n-t)/2 \rfloor$ pour $n+1 \leq t \leq 2n$. Comme le nombre total de rotations exécutées par l'algorithme est égal à $n(n-1)/2$, les inégalités précédentes sont en fait des égalités, cela donne l'optimalité de l'algorithme. Pour $t=n$, $\lfloor n/2 \rfloor$ rotations doivent donc être exécutées simultanément, ce qui prouve que $\lfloor n/2 \rfloor$ processeurs sont nécessaires. Dans le cas où n est pair cela clôt la démonstration.

Supposons maintenant n impair. Nous allons montrer qu'il faut un processeur supplémentaire qui servira au stockage d'une ligne. Nous supposons le contraire c'est à dire qu'il n'y a que $\lfloor n/2 \rfloor$ processeurs. Pour ne pas contredire l'hypothèse selon laquelle une ligne ne peut être sortie définitivement avant que toutes soient entrées, il faut qu'une qu'une ligne sorte puis ré-entre.

Mais pour le nombre de rotations, par récurrence, on peut montrer qu'à l'issue des transformations à la date $t=n-1$, P_i contient une ligne avec $i-1$ zéros et une autre avec i zéros. La dernière ligne rentrant dans le réseau, avec $\lfloor n/2 \rfloor$ processeurs, une ligne du processeur P_1 doit sortir du réseau. Pour permettre une rotation, la ligne qui sort est celle qui posséde un zéro (celle qui reste est compléte). Il restera donc dans les processeurs P_2 ou P_3 une seule ligne possédant un seul zéro, ce qui implique qu'il existe au moins un processeur inactif à $t=n$. Ceci contredit le fait que $Rot(n)= \lfloor n/2 \rfloor$ pour l'exécution de l'algorithme en $(2n-1)r$.♠♥♦♣

Lorsque n est impair, le processeur $P_{\lceil n/2 \rceil}$ n'effectue aucune rotation. En fait, il ne sert qu'à stocker une ligne comme le montre la figure 1.11.

	P_1		P_2		P_3		P_4
t=1	1						
t=2	21	R(2,1,1)					
t=3	31	R(3,1,1)	2				
t=4	41	R(4,1,1)	32	R(3,2,2)			
t=5	51	R(5,1,1)	42	R(4,2,2)	3		
t=6	61	R(6,1,1)	52	R(5,2,2)	43	R(4,3,3)	
t=7	71	R(7,1,1)	62	R(6,2,2)	53	R(5,3,3)	4
t=8	72	R(7,2,2)	63	R(6,3,3)	54	R(5,4,4)	
t=9	73	R(7,3,3)	64	R(6,4,4)	5		
t=10	74	R(7,4,4)	65	R(6,5,5)			
t=11	75	R(7,5,5)	6				
t=12	76	R(7,6,6)					
t=13	7						

Figure 1.11: Déroulement de l'algorithme GivensRL2 sur une matrice de taille (7,7)

1.3.2. MÉTHODE DE GAUSS

Si, dans GivensRL1 et GivensRL2, nous remplaçons l'opérateur de rotation R(i,j,k) par un opérateur d'élimination E(i,j,k), nous obtenons deux algorithmes GaussRL1 et GaussRL2. L'étude du paragraphe 1.3.1 s'étend à ces algorithmes. Nous en déduisons le résultat suivant.

Théorème 1.5: Sur un réseau linéaire, les algorithmes GaussRL1 et GaussRL2 sont optimaux pour effectuer l'élimination de Gauss d'une matrice carrée. Leur temps d'exécution est égal à (2n-1) e. Le nombre optimal de processeurs pour exécuter un algorithme en temps optimal est égal à $\lceil n/2 \rceil$.

1.3.3. MÉTHODE DE JORDAN

L'algorithme de Jordan peut être vu comme un double algorithme de Gauss: élimination des éléments sous-diagonaux suivie de l'élimination des éléments sur-diagonaux. A partir des algorithmes GaussRL1 et GaussRL2, on peut donc construire 4 algorithmes JordanRL. La figure 1.12 représente l'algorithme JordanRL4 correspondant à l'application de deux algorithmes GaussRL2. Ces quatres algorithmes n'ont pas le même temps d'exécution puisque certains ordonnancements nécessitent que toutes les lignes sortent du réseau alors que d'autres ré-utilisent les dernières données. L'algorithme

JordanRL4 est le meilleur des 4. Son temps d'exécution est égal à (4n-4)e. Nous ne savons pas si cet algorithme est optimal. Le théorème 1.6 montre que la meilleure borne inférieure que nous connaissons est égale à (3n-2) e.

		P_1		P_2		P_3		P_4
t=1	1							
t=2	21	E(2,1,1)						
t=3	31	E(3,1,1)	2					
t=4	41	E(4,1,1)	32	E(3,2,2)				
t=5	51	E(5,1,1)	42	E(4,2,2)	3			
t=6	61	E(6,1,1)	52	E(5,2,2)	43	E(4,3,3)		
t=7	71	E(7,1,1)	62	E(6,2,2)	53	E(5,3,3)	4	
t=8	81	E(8,1,1)	72	E(7,2,2)	63	E(6,3,3)	54	E(5,4,4)
t=9	82	E(8,2,2)	73	E(7,3,3)	64	E(6,4,4)	5	
t=10	83	E(8,3,3)	74	E(7,4,4)	65	E(6,5,5)		
t=11	84	E(8,4,4)	75	E(7,5,5)	6			
t=12	85	E(8,5,5)	76	E(7,6,6)				
t=13	86	E(8,6,6)	7					
t=14	87	E(8,7,7)						
t=15	87	E(7,8,8)						
t=16	86	E(6,8,8)	7					
t=17	85	E(5,8,8)	76	E(6,7,7)				
t=18	84	E(4,8,8)	75	E(5,7,7)	6			
t=19	83	E(3,8,8)	74	E(4,7,7)	65	E(5,6,6)		
t=20	82	E(2,8,8)	73	E(3,7,7)	64	E(4,6,6)	5	
t=21	81	E(1,8,8)	72	E(2,7,7)	63	E(3,6,6)	54	E(4,5,5)
t=22	71	E(1,7,7)	62	E(2,6,6)	53	E(3,5,5)	4	
t=23	61	E(1,6,6)	52	E(2,5,5)	43	E(3,4,4)		
t=24	51	E(1,5,5)	42	E(2,4,4)	3			
t=25	41	E(1,4,4)	32	E(2,3,3)				
t=26	31	E(1,3,3)	2					
t=27	21	E(1,2,2)						
t=28	1							

Figure 1.12: Déroulement de l'algorithme JordanRL4 sur une matrice de taille (8,8)

Théorème 1.6: Sur un réseau linéaire avec $\lfloor n/2 \rfloor$ processeurs, le temps optimal t_{Jor} pour exécuter la méthode de Jordan vérifie:

$$(3n-2)\,e \leq t_{Jor} \leq (4n-4)\,e$$

Démonstration : La borne supérieure représente le temps d'exécution de l'algorithme JordanRL4 (temps de deux GaussRL une communication finale et une initiale). La borne inférieure est obtenue comme suit. Il faut 2n-1 unités de temps pour effectuer les entrées et les sorties comme précédemment. Pendant ce temps le nombre maximum d'éliminations possibles est égal à $(n^2/2 - n/2)$. Comme le nombre total d'éliminations est égal à n^2-n, nous en déduisons qu'il faut au moins n-1 unités de temps supplémentaires pour effectuer toutes les éliminations avec l'efficacité maximum du réseau (n/2 processeurs). ♠ ♥ ♦ ♣

Nous ne connaissons pas la complexité de la méthode de Jordan sur un réseau linéaire. Nous verrons au paragraphe suivant qu'une légère modification du réseau conduit à un algorithme en (3n-1) e.

Remarquons de plus que, dans les algorithmes que nous avons présentés, les deux canaux de communication reliant P_i et P_{i+1} ne sont jamais utilisés simultanément. Les théorèmes sont donc valables sur un réseau linéaire où deux processeurs communiquent entre eux par un unique canal bidirectionnel. Un tel réseau est représenté dans la figure 1.13.

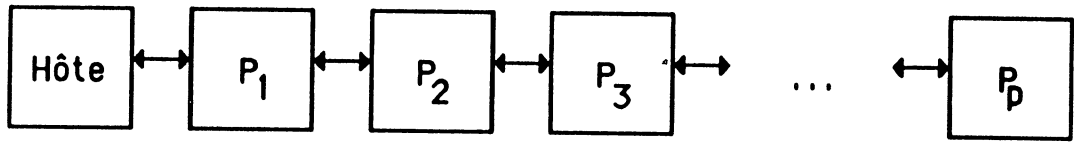


Figure 1.13: Réseau linéaire de processeurs communiquant par un canal bidirectionnel

1.4. ALGORITHMES SUR UN ANNEAU

Un anneau de processeurs est un réseau possédant des propriétés de communication très peu supérieures à celle du réseau linéaire. L'architecture considérée est représentée dans la figure 1.14. L'ordinateur hôte communique avec le processeur P_1 par un canal bidirectionnel. Chaque processeur P_i ($2 \leq i \leq p$) communique avec P_{i-1} et P_{i+1} (modulo p) aussi par l'intermédiaire d'un canal bidirectionnel. P_i ($1 \leq i \leq p$) possède une mémoire locale pouvant contenir 2 lignes de A. Nous supposerons qu'en une unité de temps P_i est capable de recevoir une ligne, d'effectuer une élimination (ou une rotation) et une émission quelle que soit la position qu'occupe l'élément à annuler. Au début de l'algorithme, l'hôte contient la matrice A. A la fin, il doit contenir le résultat cherché.

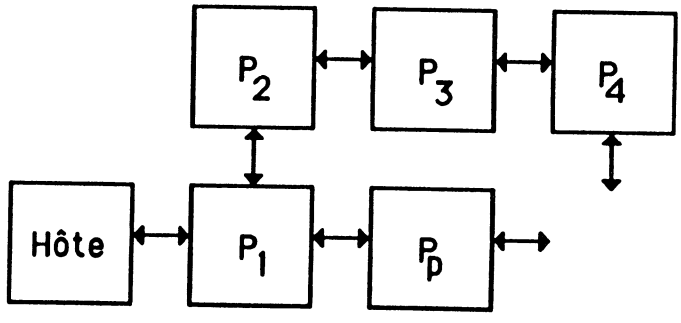


Figure 1.14: Anneau de processeurs

Le réseau linéaire est une sous topologie de l'anneau. Par conséquent, les algorithmes du paragraphe précédent s'appliquent à ce dernier type d'architecture. En reprenant la démonstration du théorème 1.4, il n'est pas difficile de montrer que les théorèmes 1.4 et 1.5 s'étendent à un anneau.

Théorème 1.7: Sur un anneau, les algorithmes GivensRL1, GivensRL2, GaussRL1 et GaussRL2 sont optimaux pour calculer la décomposition QR et l'élimination de Gauss d'une matrice carrée. Leur temps d'exécution est égal à (2n-1) r. Le nombre optimal de processeurs pour exécuter un algorithme en temps optimal est égal à $\lceil n/2 \rceil$.

Par contre, dans le cas de la méthode de Jordan, il est possible d'améliorer l'algorithme précédent pour obtenir un résultat de complexité avec l'algorithme JordanA1. Nous présentons l'ordonnancement des éliminations et les dates d'exécution de l'algorithme JordanA1 dans les figures 1.15 et 1.16.

		P_1		P_2		P_3		P_4
t=1	1							
t=2	21	E(2,1,1)						
t=3	31	E(3,1,1)	2					
t=4	41	E(4,1,1)	32	E(3,2,2)				
t=5	51	E(5,1,1)	42	E(4,2,2)	3			
t=6	61	E(6,1,1)	52	E(5,2,2)	43	E(4,3,3)		
t=7	71	E(7,1,1)	62	E(6,2,2)	53	E(5,3,3)	4	
t=8	81	E(8,1,1)	72	E(7,2,2)	63	E(6,3,3)	54	E(5,4,4)
t=9	51		82	E(8,2,2)	73	E(7,3,3)	64	E(6,4,4)
t=10	65	E(6,5,5)	12	E(1,2,2)	83	E(8,3,3)	74	E(7,4,4)
t=11	62		13	E(1,3,3)	84	E(8,4,4)	75	E(7,5,5)
t=12	76	E(7,6,6)	23	E(2,3,3)	14	E(1,4,4)	85	E(8,5,5)
t=13	73		24	E(2,4,4)	15	E(1,5,5)	86	E(8,6,6)
t=14	87	E(8,7,7)	34	E(3,4,4)	25	E(2,5,5)	16	E(1,6,6)
t=15	84		35	E(3,5,5)	26	E(2,6,6)	17	E(1,7,7)
t=16	18	E(1,8,8)	45	E(4,5,5)	36	E(3,6,6)	27	E(2,7,7)
t=17	28	E(2,8,8)	5		46	E(4,6,6)	37	E(3,7,7)
t=18	38	E(3,8,8)			56	E(5,6,6)	47	E(4,7,7)
t=19	48	E(4,8,8)			6		57	E(5,7,7)
t=20	58	E(5,8,8)					67	E(6,7,7)
t=21	68	E(6,8,8)					7	
t=22	78	E(7,8,8)						
t=23	8							

Figure 1.15: Déroulement de l'algorithme JordanA1 sur une matrice de taille (8,8)

*	10	11	12	13	14	15	16
2	*	12	13	14	15	16	17
3	4	*	14	15	16	17	18
4	5	6	*	16	17	18	19
5	6	7	8	*	18	19	20
6	7	8	9	10	*	20	21
7	8	9	10	11	12	*	22
8	9	10	11	12	13	14	*

Figure 1.16: Ordre d'exécution des éliminations de JordanA1

Théorème 1.8: Sur un anneau de $\lfloor n/2 \rfloor$ processeurs, l'algorithme JordanA1 est asymptotiquement optimal pour effectuer l'élimination de Jordan d'une matrice carrée.
Son temps d'exécution est égal à (3n-1)e.

Démonstration : Le temps d'exécution de l'algorithme JordanA1 est égal à la borne inférieure+1 du théorème 1.6.

La différence avec l'optimum s'explique par le fait que dans la phase centrale de l'algorithme le réseau ne calcule "à plein" qu'une fois sur deux, cela introduit n/2 éliminations en retard qui sont effectuées en une unité de temps supplémentaire avec le réseau plein.♠♥♦♣

1.5. INFLUENCE DE L'ARCHITECTURE

Nous avons étudié les méthodes de Givens, Gauss et Jordan sur 3 architectures différentes: le réseau complet (mémoire globale avec fonctionnement par partage de données), le réseau linéaire et l'anneau (mémoire locale avec fonctionnement par échange de messages). Dans le cas de la mémoire partagée, les processeurs ont un accès en parallèle à la mémoire de stockage (ici la mémoire partagée), alors que, dans les deux autres cas, l'accés est séquentiel.

Ceci a peu d'influence sur la méthode de Givens puisque les performances dans chaque cas sont du même ordre: (2n-1) r pour le réseau linéaire et l'anneau, et (2n-o(n)) r pour le réseau complet.

Par contre la différence n'est pas négligeable pour les méthodes de Gauss et Jordan. Cela s'explique par le fait que l'utilisation simultanée (jusqu'à p fois) d'une ligne non modifiée permet d'accélérer les algorithmes dans ces deux méthodes.

Algorithmes / Architectures	Gauss	Jordan	Givens
Mèmoire partagée	(n-1)e *	ne *	(2n-o(n))r *
Réseau linéaire	(2n-1)e *	(4n-4)e	(2n-1)r *
Anneau	(2n-1)e *	(3n-1)e *	(2n-1)r *

Figure 1.18: Tableau récapitulatif des temps d'exécution (les * représentent les temps asymptotiquement optimaux)

On peut généraliser ainsi les théorèmes précédents car ils sont basés sur les entrées/sorties:

Théorème 1.9: Sur un réseau connexe quelconque de processeurs communiquant avec l'hôte par l'intermédiaire d'un seul canal de lecture et d'un seul canal d'écriture, les algorithmes GivensRL1 et GivensRL2 (resp. GaussRL1 et GaussRL2) sont optimaux pour calculer la décomposition QR d'une matrice carrée. Leur temps d'exécution est égal à (2n-1) r (resp (2n-1)e). Le nombre optimal de processeurs pour exécuter un algorithme en temps optimal est égal à $\lceil n/2 \rceil$.

Théorème 1.10: Sur un réseau connexe, composé de $\lfloor n/2 \rfloor$ processeurs, contenant comme sous réseau un anneau communiquant avec l'hôte par l'intermédiaire d'un seul canal de lecture et d'un seul canal d'écriture, l'algorithme JordanA1 est asymptotiquement optimal pour effectuer l'élimination de Jordan d'une matrice carrée. Son temps d'exécution est égal à (3n-1) e.

1.6. CONCLUSIONS

Les résultats précédents montrent que pour les algorithmes considérés, les performances sur les divers types de réseaux sont proches. Dans les cas où ces différences sont non-négligeables (Jordan), il est clair que l'augmentation du temps d'exécution est liée aux moyens de communications entre l'hôte (mémoire de stockage) et les processeurs. Nous en déduisons que lorsque les processeurs sont du même type que ceux

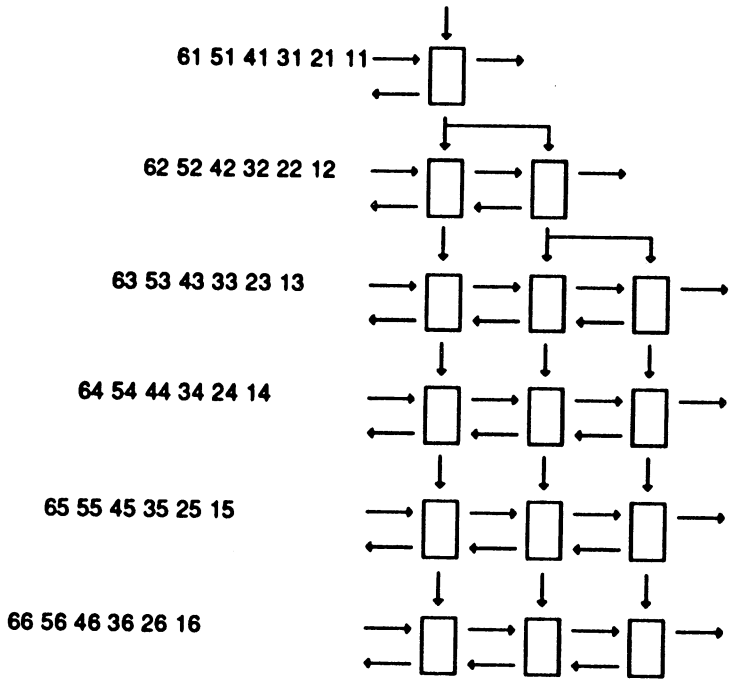
de notre modèle, le rôle joué par les communications entre les processeurs et la mémoire de stockage est plus important que les communications entre processeurs. Par exemple, un réseau hypercube dont un seul des processeurs est connecté à l'hôte n'est utilisé que comme un réseau linéaire pour les algorithmes de ce type.

On s'aperçoit ici de la nécessité d'un système matériel performant pour décharger les mémoires locales dans les machines à mémoires distribuées. Ce problème a été clairement pris en compte dans la Connection Machine (de Thinking Corp) où les mémoires peuvent être chargées et déchargées très rapidement vers les disques (hôte) au travers d'un système appelé "Vault".

Bien sûr, notre modèle d'architecture est très imparfait. En effet, nous supposons que les processeurs ne peuvent pas communiquer sur plusieurs liens en même temps ou pendant qu'ils calculent. L'unité de temps d'exécution est grossière (indépendante de la position de l'élément à annuler). De plus, pour les comparaisons, raisonner à nombre de processeurs égal entre MP et MD peut apparaitre comme une restriction puisque l'avantage des machines à mémoire distribuée est d'augmenter la largeur du parallélisme.

Un exemple de programmation des processeurs pour l'exécution des algorithmes sur réseau linéaire et l'anneau est développé dans [CDT].

2 Algorithmique systolique 27



2 ALGORITHMIQUE SYSTOLIQUE

Où, à partir d'un réseau linéaire, nous déduisons un réseau systolique pour la diagonalisation de Jordan.

2.1 INTRODUCTION

Pour augmenter les performances des calculateurs, l'approche traditionnelle consiste à rechercher une augmentation de la vitesse des composants matériels (on diminue la période d'horloge) et à améliorer le système d'exploitation pour minimiser le coût du controle. Aujourd'hui, cette approche est remise en cause. Tout d'abord, l'industrie des semi-conducteurs est, d'une manière générale, plus à même de développer des technologies à très haute densité d'intégration que de réaliser des circuits ultra rapides. Ensuite, il semble que l'on a atteint des limites physiques incontournables jusqu'à l'arrivée sur le marché de technologies plus rapides (à l'Arsenure de Gallium par exemple). De plus, le prix à payer pour gagner un ordre de grandeur sur la vitesse des composants en utilisant ces nouvelles technologies sera très vraisemblablement prohibitif pour la plupart des applications.

Plutôt que de compter seulement sur l'augmentation de la vitesse des composants, une solution consiste à dupliquer des processeurs "peu" chers pour obtenir des machines parallèles. Nous allons étudier dans ce chapitre une utilisation du parallélisme à un niveau microscopique.

De plus pour que ces systèmes parallèles puissent être implémentés efficacement sous forme de circuits intégrés VLSI (Very Large Scale Integration), la complexité de leur conception doit rester dans le cadre des meilleures possibilités industrielles.

Le modèle systolique, introduit en 1978 par Kung et Leiserson [KLe], s'est révélé être un outil puissant pour la conception de processeurs intégrés spécialisés. En un mot, une architecture systolique est agencée en forme de réseau (orthogonal par exemple). Ces réseaux se composent d'un grand nombre de cellules élémentaires identiques et localement interconnectées.

Chaque cellule reçoit des données en provenance des cellules voisines, effectue un calcul simple, puis transmet les résultats, toujours aux cellules voisines, un temps de cycle plus tard.

Pour fixer un ordre de grandeur, disons que chaque cellule a la complexité, au plus, d'un petit microprocesseur.

Les cellules évoluent en parallèle, sous le contrôle d'une horloge globale (synchronisme total): plusieurs calculs sont effectués simultanément sur le réseau, et on peut "pipeliner" la résolution de plusieurs instances du même problème sur le réseau en présentant les données les unes derrières les autres [QRo].

Enfin aucun contrôle global n'est permis sur l'ensemble des cellules, pour conserver l'indépendance vis à vis des données en entrée [Ull]

La dénomination "systolique" provient d'une analogie entre la circulation des flots de données dans le réseau et celle du sang humain, l'horloge qui assure la synchronisation globale constituant le "coeur" du système.

Les réseaux systoliques sont donc des processeurs intégrés spécialisés qui répondent à cette approche. Deux caractéristiques sont dominantes: un parallélisme massif et un mode opératoire synchrone [QRo].

Notre point de vue algorithmique du modèle systolique sera le suivant: conception d'algorithmes pour l'utilisation efficace en parallèle d'un très grand nombre de circuits simples répondant à la définition ci dessus.

De fait, la complexité des circuits intégrés disponibles à l'heure actuelle rend possible la réalisation à un faible coût de tels sytèmes parallèles. Deux restrictions cependant:

- il s'agit de processeurs dédiés que l'on adjoint à un processeur hôte de type conventionnel.
- la classe d'applications est bien délimitée: les problèmes où le volume de calculs à effectuer prime largement sur les transferts de données à réaliser.

Ces deux caractéristiques sont effectives dans les algorithmes et les architectures étudiés précédemment.

Nous dérivons, d'un réseau linéaire de processeurs simples comportant deux hôtes, un réseau systolique pour résoudre l'élimination de Jordan. Il présente de bonnes performances et une surface réduite par rapport à celles obtenues dans la littérature [Mel][RTc][GKu] par exemples.

2.2 RÉSEAU LINÉAIRE À 2 HOTES POUR LA MÉTHODE DE JORDAN

Ce réseau est issu des l'algorithmes JordanRL4 et JordanA1 des parties précédentes. Il tient compte du fait que sur un réseau systolique les données peuvent entrer à une extrémité (ou une face) et sortir à l'autre (sur une autre face), ce qui revient à construire un réseau linéaire avec deux hôtes (figure 2.1). Nous supposerons qu'il ne peut pas se produire de communications entre le réseau et les deux hôtes simultanément.

Le premier hôte contient les données initiales, le second contiendra les données finales.

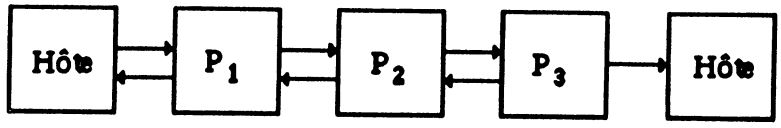


Figure 2.1: Réseau linéaire à deux hôtes

Lorsque le réseau est chargé, de manière identique à JordanRL4, le flot de données effectue des va-et-vient pour que les éléments et les coefficients soient utilisés à la manière de JordanA1. Ce mode de transfert permet d'effectuer la décomposition de Jordan, l'algorithme JordanRL2H est décrit dans la figure suivante.

	P1		P2		P3		Sens des transferts
t=1		1					----->
t=2		21	E(2,1,1)				----->
t=3		31	E(3,1,1)	2			----->
t=4		41	E(4,1,1)	32	E(3,2,2)		----->
t=5		51	E(5,1,1)	42	E(4,2,2)	3	----->
t=6		61	E(6,1,1)	52	E(5,2,2)	43	E(4,3,3) ----->
t=7	1	62	E(6,2,2)	53	E(5,3,3)	4	<-----
t=8		12	E(1,2,2)	63	E(6,3,3)	54	E(5,4,4) ----->
t=9	2	13	E(1,3,3)	64	E(6,4,4)	5	<-----
t=10		23	E(2,3,3)	14	E(1,4,4)	65	E(6,5,5) ----->
t=11	3	24	E(2,4,4)	15	E(1,5,5)	6	<-----
t=12		34	E(3,4,4)	25	E(2,5,5)	16	E(1,6,6) ----->
t=13		4		35	E(3,5,5)	26	E(2,6,6) ----->
t=14				45	E(4,5,5)	36	E(3,6,6) ----->
t=15				5		46	E(4,6,6) ----->
t=16						56	E(5,6,6) ----->
t=17						6	----->

Figure 2.2: Ordonnancement des éliminations pour l'algorithme de Jordan sur le réseau à 2 hôtes avec 3 processeurs et une matrice 6x6

Lorsque le système présente r seconds membres, les processeurs de ce réseau linéaire effectuent des opérations sur des lignes de taille n+r (E(i,j,k) est l'élimination du $k^{ième}$ élément de la ligne i par la ligne j). Un registre tampon est nécessaire pour permettre la migration des données, il peut être dans l'hôte ou dans le premier processeur. Les unités e et le modèle sont les mêmes que § 1.

*	8	9	10	11	12
2	*	10	11	12	13
3	4	*	12	13	14
4	5	6	*	14	15
5	6	7	8	*	16
6	7	8	9	10	*

Figure 2.3: Date d'exécution des éliminations JordanRL2H pour une matrice 6x6

Théorème 2.1: Sur un réseau linéaire de ce type, avec $\lfloor n/2 \rfloor$ processeurs, l'algorithme de JordanRL2H s'exécute de manière asymptotiquement-optimale en (3n-1)e unités de temps.

Démonstration: Il faut n unités pour charger le réseau, pendant lesquelles on effectue $(2\sum_{1\leq k<n/2}k + n/2)$ E. Viennent ensuite les étapes avec changement du sens du flot de données où, en n pas, ((n-1)*n/2) E sont exécutés et le déchargement du réseau en n-1 pas qui donne $2\sum_{1\leq k<n/2}k$ E. Ce qui au total résout bien les n^2-n éliminations de l'algorithme de Jordan en (3n-1)e (entrées par un hôte, sorties par le deuxième comprises).

Comme dans le § 1 on peut dire que ce réseau est asymptotiquement optimal avec l'hypothèse qui précise que les deux hôtes ne peuvent communiquer simultanément. Le réseau effectue les 2n-1 unités de temps nécessaires aux entrées/sorties (car les données sont et vont dans un seul hôte), n/2 unités de temps de calcul avec le réseau plein et n/2 unités de temps de calcul avec le réseau "presque plein" (c'est à dire lorsque n/2-1 opérations d'élimination sont réalisées à chaque unité de temps). Le théorème 1.8 est étendu avec 2 hôtes.♠♥♦♣

2.3 TRANSFORMATION DU RÉSEAU LINÉAIRE EN RÉSEAU SYSTOLIQUE

2.3.1 SYSTOLISATION DES TRANSFORMATION VECTORIELLES

Les transformations vectorielles E du chapitre 2.2 et du § 1 peuvent être généralisées sous la forme:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} \leftarrow \begin{bmatrix} F(X,Y) \\ G(X,Y) \end{bmatrix}$$

avec X,Y des vecteurs de $\mathbb{R}^n$ et F et G des fonctions de $\mathbb{R}^{2n}$ dans $\mathbb{R}^n$.

L'analyse descendante introduite par [CTc] repose sur un principe simple. Si les fonctions F et G peuvent se décomposer sous la forme:

$$F=(f_1, ..., f_n) \text{ et } G=(g_1, ..., g_n)$$

Et si les fonctions f_i et g_i sont indépendantes des données, elles peuvent être mémorisée dans des cellules systoliques, en conservant la propriété de base des réseaux sytolique: l'indépendance par rapport au données [Ull]. Chaque cellule opèrera sur les éléments des vecteurs au fur et à mesure de leur propagation dans le réseau. En général les fonctions f_i et g_i ne varient pas avec i et sont égales à f et g.

$$\begin{bmatrix} x'_1, ..., x'_n \\ y'_1, ..., y'_n \end{bmatrix} \leftarrow \begin{bmatrix} f(x_1, y_1), ..., f(x_n, y_n) \\ g(x_1, y_1), ..., g(x_n, y_n) \end{bmatrix}$$

Pour la construction du réseau systolique dans notre cas, le choix d'une solution statique [CTc] ,où les fonctions f et g restent dans la cellule en propageant le résultat de la transformation et les données, est le plus approprié et moins complexe que la solution dynamique où les fonctions sont transmisent.

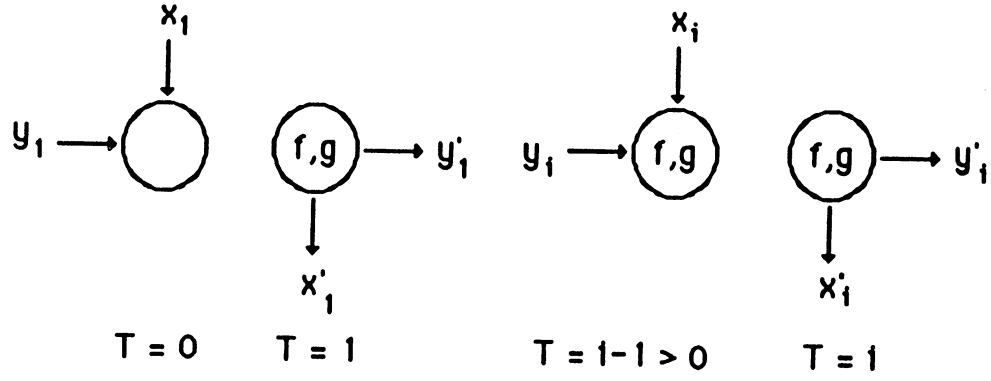


Figure 2.4: Cellule systolique correspondant à la solution statique

La systolisation statique consiste alors en la décomposition des opérations vectorielles en opérations équivalentes sur les éléments des vecteurs pour permettre au flot de données du problème de traverser le réseau en recevant la transformation adéquate. La figure suivante présente une cellule systolique établie de cette manière et le flot des données pour plusieurs cycle de l'horloge du réseau.

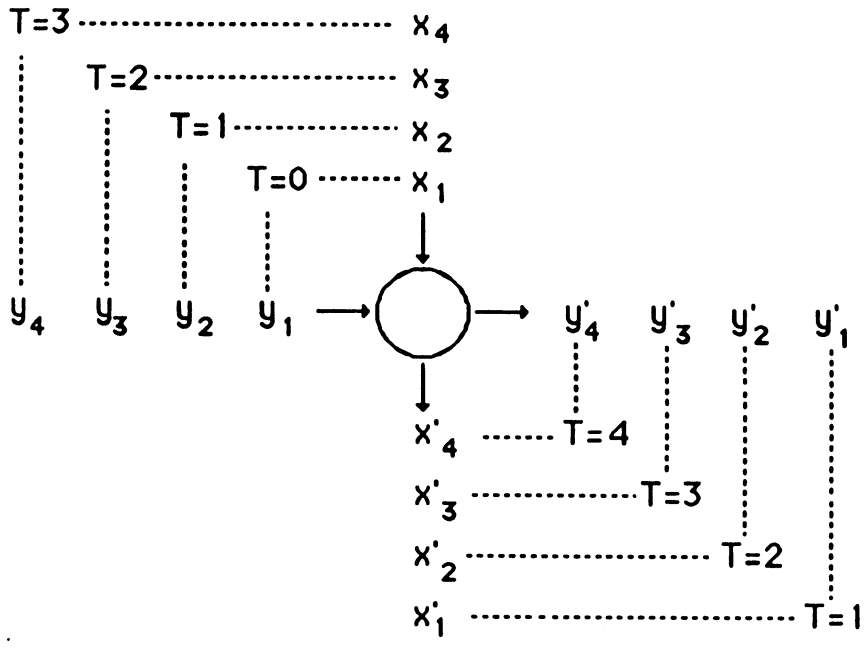


Figure 2.5: Flot des données sur la cellule systolique

En poursuivant cette analyse descendante, le réseau linéaire RL2H du chapitre 2.2 entre dans la catégorie des réseau systoliques "réversibles" [CTc].

L'explication provient du graphe de dépendance des taches de l'algorithme. Les croisements de données sont bilatéraux car chaque élément doit rencontrer tous les éléments de sa colonne dans l'algorithme de diagonalisation de Jordan. Si l'on définit l'algorithme de la manière suivante:

pour k=1 **à** n
  pour j≠k
    tâche T_{kj}
  finpour
finpour

et la relation de précédence par ">>", on obtient les deux relation qui définissent complètement le graphe de dépendance des tâches [CRTr]:

$T_{k,k+1} >> T_{k+1,j}$ $(j \geq k+2)$
$T_{k,j} >> T_{k+1,j}$ $(j > k)$

Avec une stratégie "gloutonne" pour l'exécution des tâches, on obtient le graphe suivant [CTc].

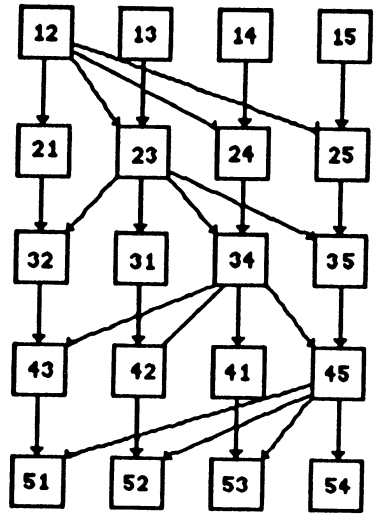


Figure 2.6: Graphe de dépendance des taches dans l'algorithme de diagonalisation de Jordan pour une matrice 5x5

Ce graphe peut être "étiré" pour faciliter le pipeline des données sans perte sur le temps total [CMRT], puis ré-arrangé pour diminuer la surface totale du rectangle dans lequel il s'inscrit. Le flot des données devra alors être inversé à chaque étape pendant les phases centrales du calcul pour que les tâches de chaque colonne soient affectées aux processeurs correspondants.

Cette méthode donne presque tous les éléments pour la construction du réseau systolique définitif.

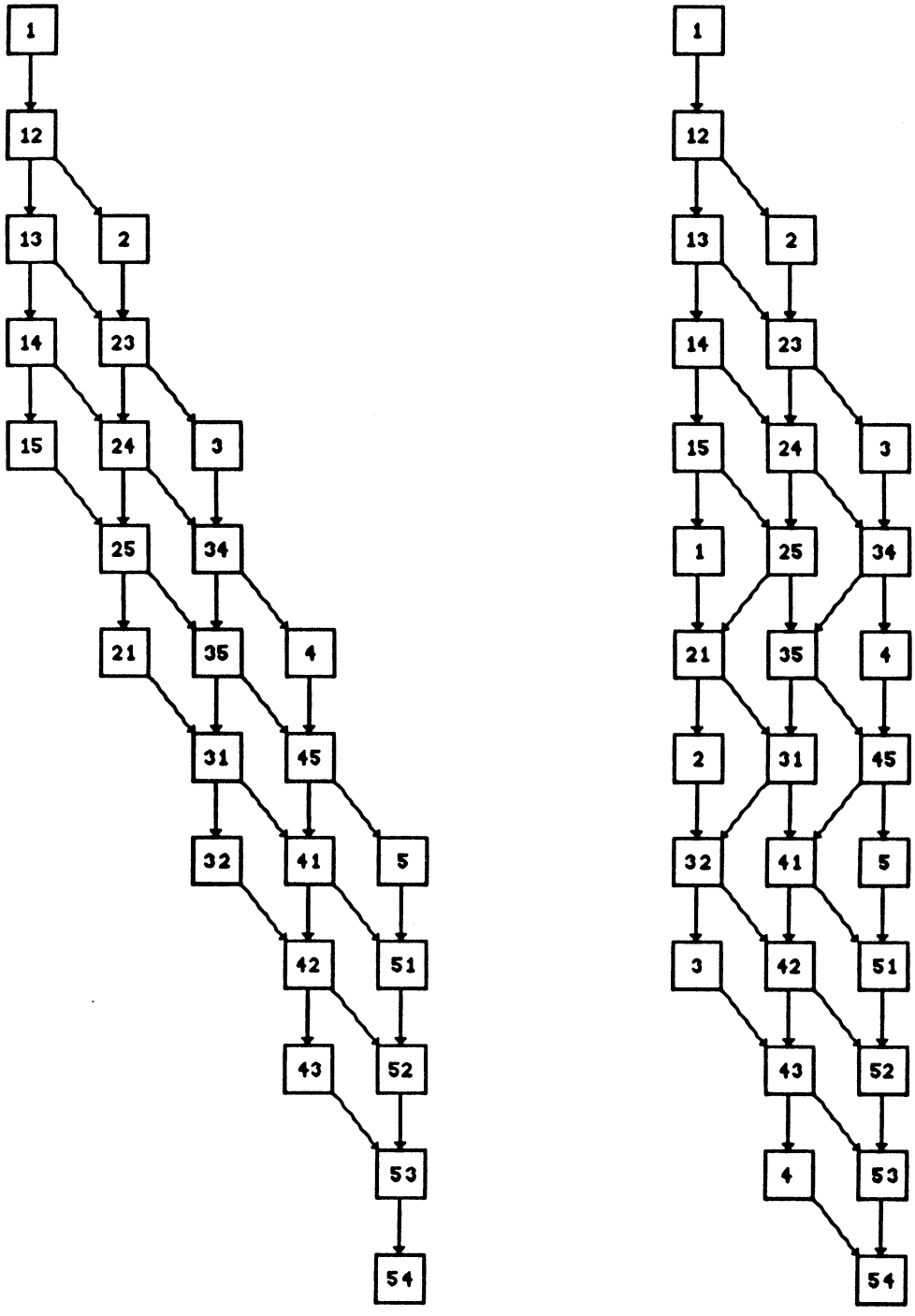


Figure 2.7: Graphes de précédence pour obtenir une plus petite surface

2.3.2 RÉSEAU SYSTOLIQUE POUR LA MÉTHODE DE JORDAN

Avec l'étude suivant la technique d'analyse descendante, généralisée dans [CTc], on peut dire grossièrement que le réseau systolique est obtenu par

"étirement" du réseau linéaire dans le sens vertical (représentant ici les lignes). Le croisement des données satisfait l'algorithme de Jordan et dans la suite nous allons décrire un des réseaux systoliques qui peut être obtenu ainsi. Les données pénètrent le réseau par lignes (équivalent au premier hôte du réseau linéaire RL2H) et quittent le réseau à l'opposé (deuxième hôte).

Pour des lignes de taille n+r (résolution avec r seconds membres) chaque processeur P_i est divisé en n+r-i+1 cellules systoliques élémentaires. La largeur du réseau reste n/2 car il ne faut pas plus pour exécuter le graphe des tâches (figure 2.6), cela donne un nombre total de cellules de $3n^2/8 + rn/2$.

La topologie d'interconnexion est décrite figure 2.8. On peut ajouter à la base du réseau autant de rangées de cellules que de seconds membres à résoudre.

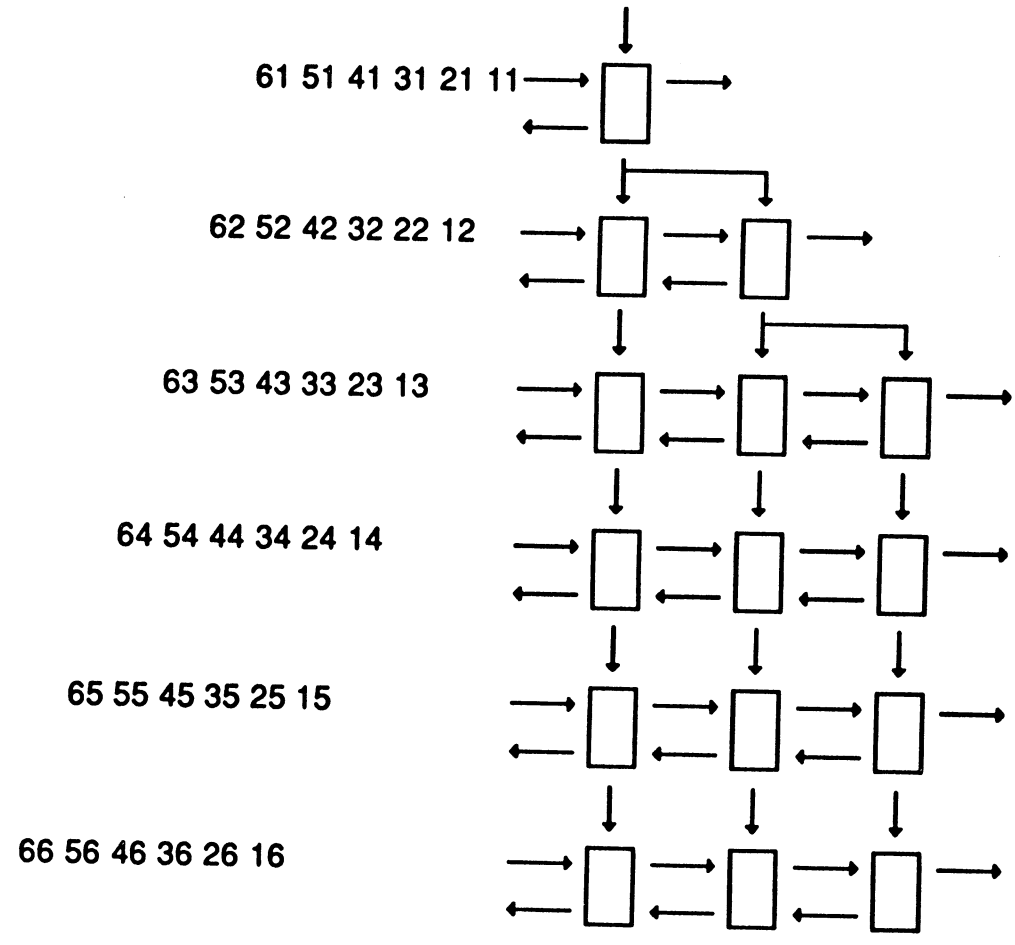


Figure 2.8: Le réseau d'interconnexion des cellules systoliques pour une matrice 6x6 sans second membre (les éléments ij pénètrent dans le réseau)

Chaque cellule possède 2 registres pouvant contenir un réel et est capable d'effectuer les opérations arithmétiques D_{ij} et U_{ikj} (qui correspondent aux fonctions f et g précédentes).

L'opération D_{ij} consiste en le calcul du coefficient de l'élimination ($coef=a_{ij}/a_{ii}$) et l'opération U_{ikj} est la mise à jour des éléments de la ligne i à l'étape k par le coefficient D_{ij} ($a_{kj} = a_{kj} - a_{ki}*coef$).

Les trois phases de l'algorithme décrit pour le réseau linéaire sont conservées:

1/ Chargement du réseau et début des Eliminations

2/ Eliminations et flot de donnée alterné

3/ Déchargement du réseau et fin des Eliminations

La difficulté de l'étape 2/ provient du changement de sens du flot de données, à chaque pas. Le contrôle du sens des transferts est donc assuré par un booléen introduit de haut en bas dans le réseau.

Pour effectuer le choix entre les opérations D_{ij} et U_{ikj} il faut que les éléments diagonaux soient reconnaissables. On introduit donc avec chaque coefficient un booléen qui sert de marque pour l'algorithme interne.

Comme pour le réseau linéaire on peut supposer que les cellules de la première colonne de processeurs contiennent une case mémoire de plus pour effectuer le stockage de l'élément qui sort du réseau puis rentre tous les 2 tops lors de la phase 2/.

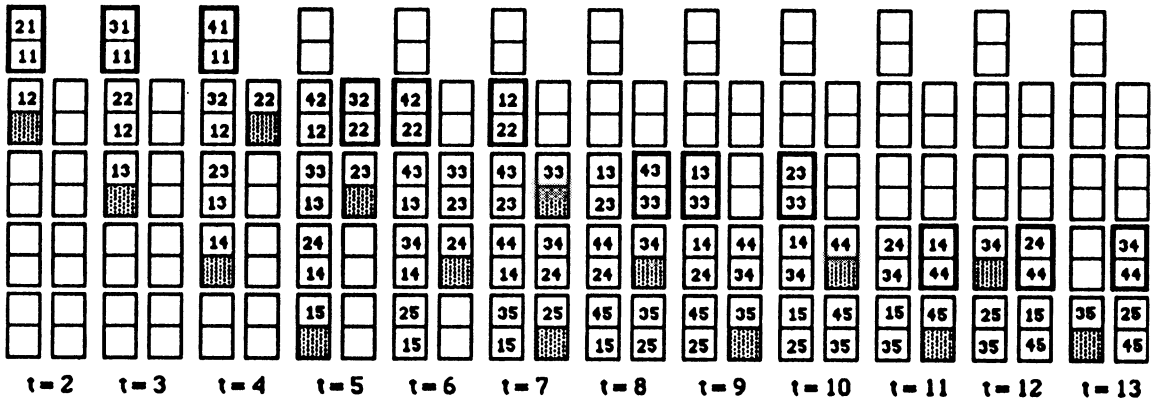


Figure 2.9: Exécution de l'élimination de Jordan d'une matrice 4x4 avec un second membre, sur un réseau systolique de 9 cellules (cases entourées de gras D_{ij}, normales U_{ikj}, grisés stockage d'un élément)

Théorème 2.2: Ce réseau de $3n^2/8+rn/2$ cellules exécute l'algorithme d'élimination de Jordan avec r seconds membres en 4n-2+r unités de temps.

Preuve: le réseau effectue l'élimination de Jordan de la même manière que l'algorithme JordanRL2H dont il est issu mais élément par élément (la granularité est plus fine).

L'élément A_{ii} pénètre dans le réseau à t=2i-1 et il est mis à jour par (i-1) opérations U_{ikj} puis après un temps mort où il passe de la position haute à la position basse dans la cellule, il génère les (n-1) coefficients correspondants à la colonne i. Le dernier coefficient de a_{nn} sera généré à t=4n-3 et appliqué à la $r^{ième}$ colonne second membre à t=4n-3+r permettant au dernier élément de quitter le réseau (t=4n-2+r). ♠♥♦♣

Decription du fonctionnent interne de chaque cellule. A chaque étape les transferts de données sont les suivants:

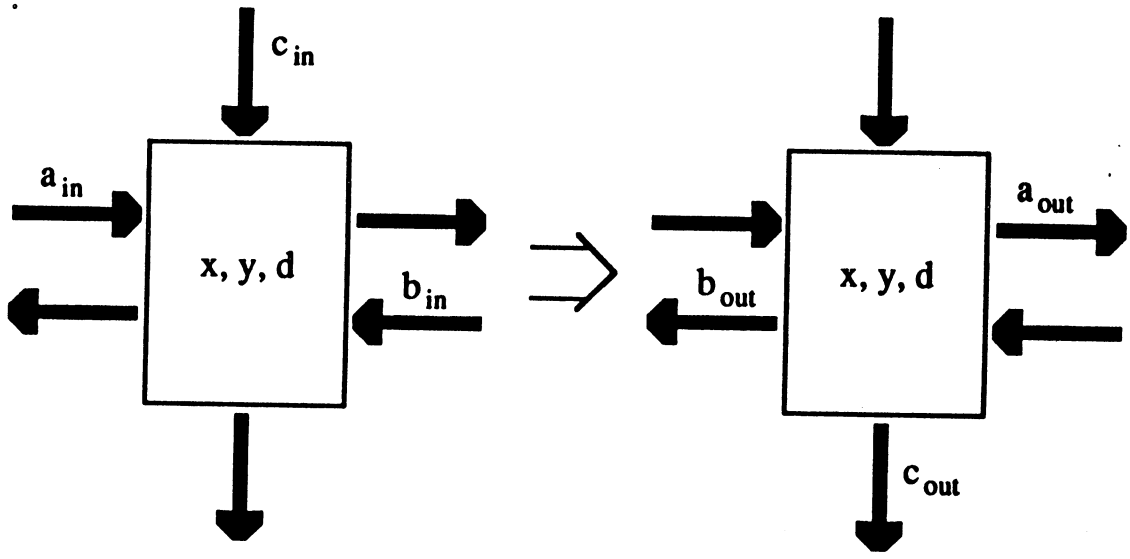


Figure 2.10: Une étape systolique sur la cellule

Les variables a_{in}, b_{in}, a_{out}, b_{out} pour les communications horizontales comprennent 2 éléments: le coefficient de la matrice (x ou y) et le booléen (x_{bool} ou y_{bool}) qui lui est associé. c_{in} et c_{out} sont les variables des communications verticales, du même type, elles contiennent le coefficient de l'élimination courante (d) et le booléen qui contrôle le sens du flot des données (sens).

Le programme interne de chaque cellule comprend deux procédures de calcul pour D_{ij} et U_{ikj} et deux fonctions de communications correspondant aux deux sens des données et aux communications verticales. Le programme interne de chaque cellule s'écrit alors:

```
/* programme d'une cellule du réseau */
{     /* descente dans la cellule pour remplir le réseau */
      si y=nil et x ≠nil alors y:=x finsi;
      /* remonté dans la cellule pour vider le réseau */
      si x=nil et y ≠nil alors x:=y finsi;
      /* choix de la fonction de calcul */
      si ybool=true      alors calcul_Dij
                         sinon calcul_Uikj finsi;
      /* choix du sens de communication */
      si sens=true       alors com_droite
                         sinon com_gauche finsi;
      /* communication verticale du résultat */
      com_vert;
}
```

Les procédures sont les suivantes, elles n'ont pas été incluses dans le corps du programme pour faciliter sa compréhension:

calcul_D_{ij}	{d:=x/y}
calcul_U_{ikj}	{x:=x-d*y}
com_droite	{a_{out}:=(x, x_{bool}); (x, x_{bool}):=a_{in}}
com_gauche	{b_{out}:=(y, y_{bool}); (y, y_{bool}):=b_{in}}
com_vert	{c_{out}:=(d, sens); (d, sens):=c_{in}}

On peut remarquer que la diminution du nombre de cellules de $n^2/2$ à $3n^2/8$ est préjudiciable à la période du réseau: 3n au lieu de n dans [RTc][Mel].

Les registres additionnels ne sont pas pris en compte dans la surface car ils représentent des éléments de stockage qui pourraient être inclus dans les cellules de la première colonne, sans perte réelle de place mais avec une perte de généralité sur l'indépendance par rapport aux données. La solution consiste à effectuer, au niveau matériel, un rebouclage de la sortie à l'extrême gauche du réseau sur son entrée, en intercalant une cellule retard (figure 2.12).

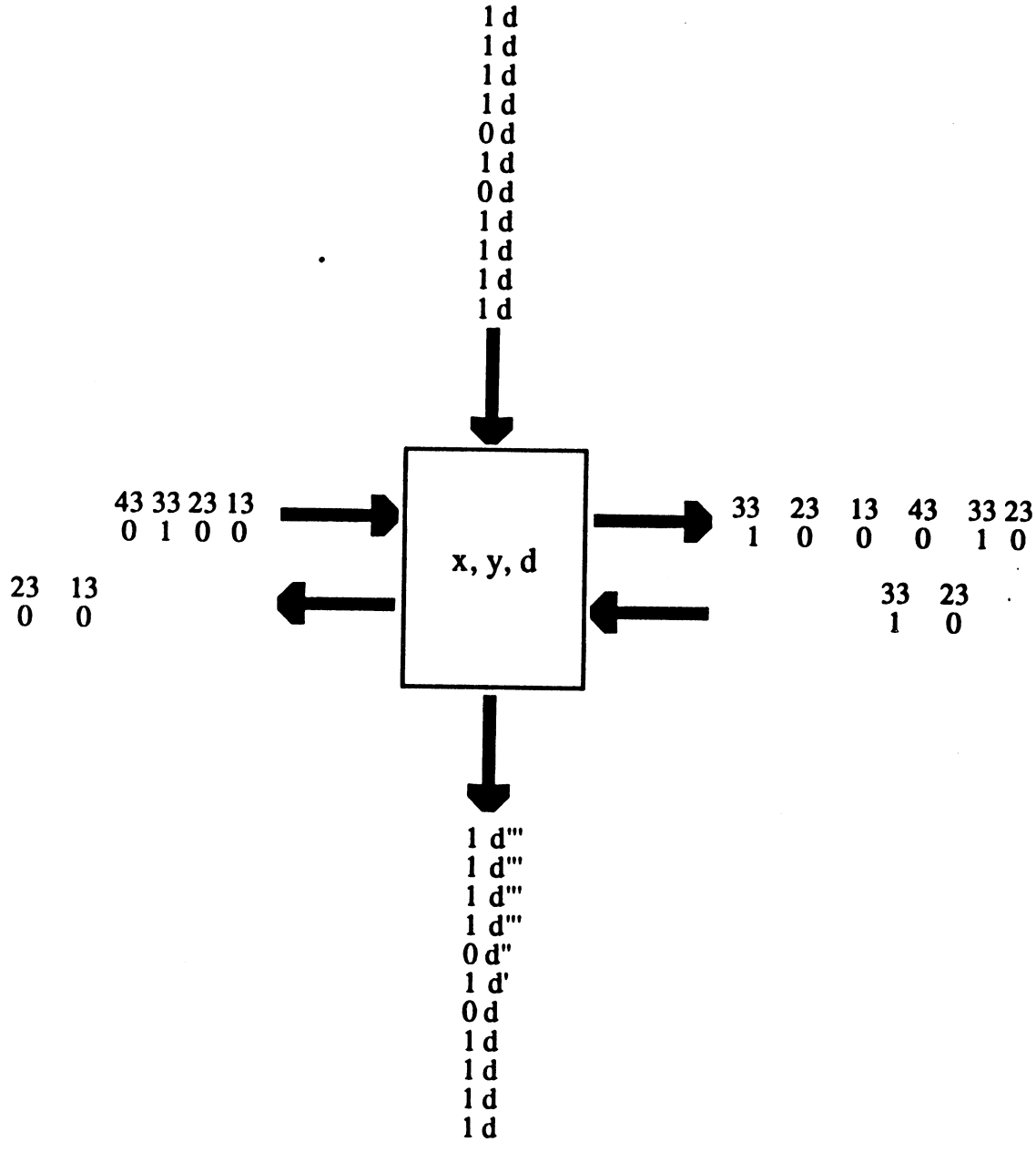


Figure 2.11: La cellule (3,1) et ses flots de données pour une matrice 4x4

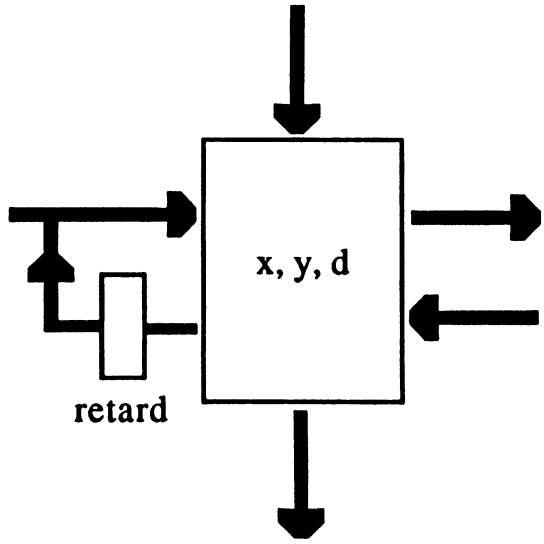


Figure 2.12: Solution matérielle pour la partie extrême gauche du réseau

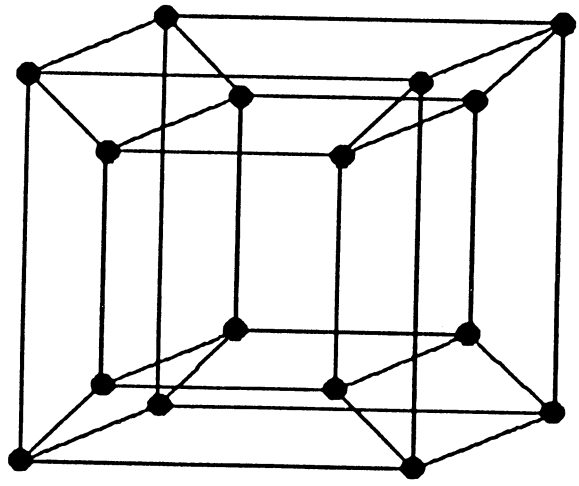
2.4 CONCLUSIONS

Nous avons développé, à partir des résultats précédents, un réseau linéaire plus spécialisé pour la diagonalisation de Jordan. Par une sorte de dépliage vertical [CTc] nous avons obtenu un réseau systolique équivalent. Il présente les même performances que ceux de la littérature et surtout une surface plus faible ($3n^2/8$ cellules).

Le fonctionnenment de ce réseau a été testé à l'aide du simulateur de réseaux systoliques SISYC écrit par A. Benaini pendant sa thèse de Doctorat [Ben] réalisée au laboratoire.

L'utilisation de cette méthode pour d'autres algorithmes sur un réseau linéaire est possible et donne de bons résultats pour la surface des réseaux. Nous travaillons actuellement pour obtenir un réseau systolique résolvant le problème des plus courts chemins (APP) qui présenterait aussi une surface inférieure à ceux de la littérature.

3. L'hypercube FPS T20 43



3. L'HYPERCUBE FPS T20

Où nous présentons l'architecture et les performances d'une machine multiprocesseurs à mémoire distribuée de topologie hypercube.

3.1. INTRODUCTION

Le marché des supercalculateurs parallèles à mémoire distribuée regroupe maintenant plusieurs compagnies: Thinking Machine (Connection Machine), Intel (Ipsc), Ncube, FPS, ... [Dun]. Floating Point Systems a lancé ses ordinateurs parallèles très performants (processeurs pipelines) en 1987, la gamme est maintenant arrêtée mais les idées phares se sont introduites chez les autres constructeurs. On voit maintenant apparaître des processeurs vectoriels en option dans la plupart des machines citées précédemment. Le laboratoire TIM3, abrite une machine FPS depuis Juin 1987. Nous avons essuyé les premiers plâtres d'une machine prototype, mais par la suite, l'utilisation s'est trouvée facilitée, jusqu'à la version actuelle du système d'exploitation tout à fait acceptable.

Cette machine a été utilisée pour les expérimentations des prochains §. C'est un hypercube multiprocesseurs vectoriels. FPS proposait plusieurs machines dans sa série T, de 16 à 128 processeurs. Le modèle le plus utilisé dans la suite est le T20 du laboratoire TIM3 (16 processeurs) et quelques expériences ont été exécutés sur le modèle T100 de FPS à Portland (64 processeurs).

Nous décrivons sommairement son architecture et l'architecture interne de chaque noeud. Nous présentons ensuite plus précisément les opérateurs vectoriels et nous décrivons les fonctions de communication qui sont fournies avec le système. Le but de ce chapitre est de fournir des éléments pour faciliter la compréhension des § 4, 5, 6, 7 et 8 et de montrer comment une machine à mémoire distribuée peut être évaluée de manière objective.

3.2. ARCHITECTURE

Le FPS T20 est une machine comportant 16 processeurs, chacun capable d'effectuer des opérations vectorielles pipelinées. Il ne possède pas de

mémoire partagée par tous les processeurs. Chacun d'entre eux a sa propre mémoire. La mémoire est distribuée sur les processeurs (par opposition au terme mémoire partagée).

Pour l'exécution de la plupart des algorithmes, les processeurs doivent communiquer entre eux. Il existe donc des canaux de communication reliant les processeurs. L'idée de base de la série T est la topologie du réseau de connexions entre les processeurs: du type hypercube. Les intérêts de ce réseau sont les suivants: peu de liens physiques entre processeurs et un plus court chemin entre processeurs en $\log_2$(nombre de processeurs). Ceci permet de garantir la faisabilité du système (nombre de canaux=O(nombre de processeurs) et la rapidité dans les communications (diamètre du graphe d'interconnexion).

La figure suivante montre la manière dont sont interconnectés les processeurs (noeuds) sur une machine hypercube de dimension 4.

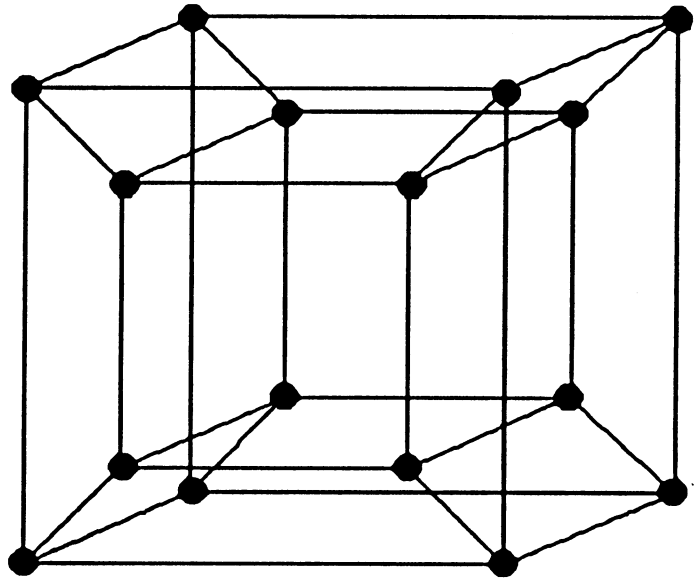


Figure 3.1: Hypercube de dimension 4, 16 noeuds

Il n'existe pas de synchronisation globale entre les processeurs. Le T20 est donc une machine MIMD asynchrone [Fly] où les processeurs communiquent par échange de messages.

Plus précisément, l'hypercube T20 est formé de 2 modules. Chaque module comprend un cube de dimension 3, c'est à dire 8 processeurs vectoriels. Il est relié à un processeur système par un bus (link-bus) et aux autres modules par des canaux suivant l'architecture hypercube. Le noeud système de chaque module contrôle un disque dur (Winchester) pour le stockage des données.

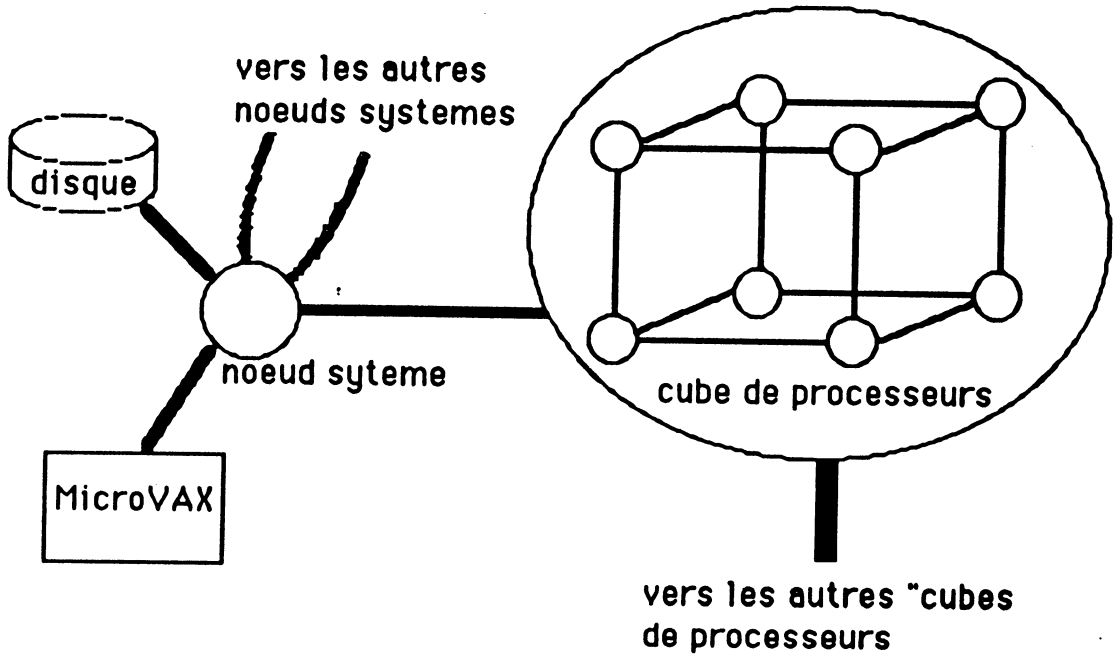


Figure 3.2: Un module de la série T de FPS

3.3. ENVIRONNEMENT DE PROGRAMMATION

La machine hôte est un microVax de DEC. Il sert d'interface avec les utilisateurs et son système d'exploitation est Ultrix. Il est relié au réseau local grenoblois Grenet par Ethernet et aux réseaux internes du laboratoire.

Les noeuds systèmes sont reliés entre eux suivant une structure d'anneau. La liaison entre cet anneau et le microVax s'effectue grâce à un bus (Q bus) connecté au noeud système "0" (noeud système privilégié pour les communications vers l'extérieur).

La programmation de la machine s'effectue en FORTRAN ou en C. Les programmes sont développés et compilés sur l'hôte, puis chargés sur l'hypercube via les noeuds systèmes. Toutes les facilités d'Unix sont disponibles à ce niveau.

La modularité des ordinateurs de la série T de FPS permet de construire des systèmes plus importants (T200 à Los-Alamos (USA) avec 128 processeurs) car on peut facilement augmenter la dimension de l'hypercube en augmentant le nombre de modules. Nous verrons par la suite que la topologie hypercube n'est utilisable complètement à un instant donné qu'avec 16 processeurs.

3.4. ARCHITECTURE D'UN NOEUD DE L'HYPERCUBE

Tous les noeuds sont identiques, nous allons décrire leur architecture. Ils sont formés de 3 parties principales :

- Un microprocesseur 32 bits spécialisé (Inmos Transputer® T414) qui assure la gestion des communications entre les processeurs ou avec son noeud système (pour le disque et les échanges clavier/écran), la fonction Unité Arithmétique et Logique scalaire et le contrôle du programme.

- Une unité de calcul vectorielle (Weitek) qui assure des calculs très rapides en virgule flottante (additionneur et multiplieur pipelinés) pour des vecteurs de réels codés sur 64 bits (format double précision IEEE). Quatre registres vectoriels à décalage sont présents pour alimenter les opérateurs.

- Une mémoire vidéo très performante (Hitachi Video RAM). Elle présente 2 accès (aléatoire vers le Transputer et série vers les registres du VPU) et sa capacité est de 1 Moctet.

Ces unités sont regroupées sur une carte (40x40 cm) enfichée dans l'un des 16 emplacements du T20. Huit processeurs sont regroupés par étages pour former un module avec un processeur système et un disque.

Un processeur système est formé de 2 parties principales:
- Une mémoire de masse (disque dur de 85 MOctets)
- Un transputer et sa mémoire locale

Le processeur système sert d'interface avec le microVAX hôte pour les entrées/sorties et il assure le controle du stockage des données sur le disque qui lui est associé. Cette unité est aussi installée sur une carte, plus petite, placée à coté du disque.

Lors de l'exécution, un des processeurs système reçoit le code de l'hôte. Il le diffuse aux autres processeurs systèmes par l'intermédiaire de l'anneau système, puis chacun d'entre eux charge le code reçu sur le cube de 8 processeurs de son module. Ceci assure le chargement simultané de tous les processeurs de l'hypercube.

On remarque ici que tous les processeurs reçoivent le même code, mais ils connaissent leur numéro absolu, suivant un code de Gray dans l'architecture hypercube, par le système. Ils peuvent donc n'exécuter que la partie du code les concernant grace à d'une instruction conditionnelle explicite du programmeur. La machine fonctionne ainsi suivant le mode SPMD (Single Program Multiple Data).

3.4.1. MÉMOIRE ET UNITÉ VECTORIELLE

Les registres vectoriels ont une taille de 1024 octets, soit 128 réels. La mémoire est divisée en tranches de la taille des registres pour faciliter leurs accès. Le chargement/déchargement série est donc très rapide (trois cycles du Transputer), ce qui correspond à l'accès aléatoire d'un mots de 32 bits sur le Transputer.

Pour alimenter les deux opérandes des unités vectorielles indépendamment, la mémoire est partitionnée en 2 bancs: l'un de 256 tranches, l'autre de 3x256 tranches. Les échanges entre mémoire et registres se font par tranche complète, quelle que soit la longueur des données à transferer. Il faut donc faire attention à ne pas écraser des informations.

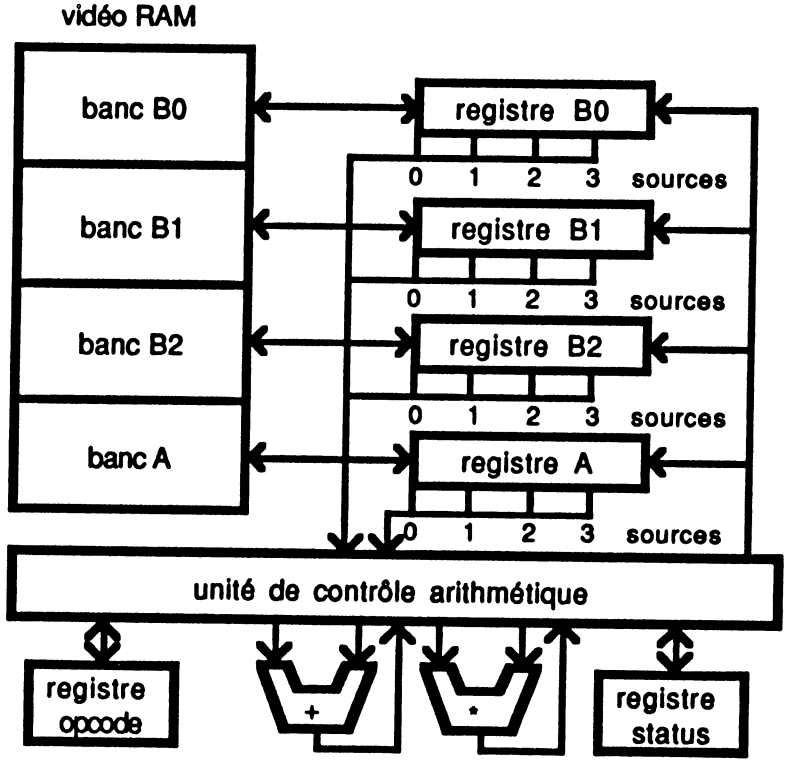


Figure 3.3: L'unité vectorielle, les registres et la mémoire de chaque noeud

Le déchargement d'un registre vers les unités de calcul se fait au travers de ses sources, par décalage. Il existe 4 sources par registre. Chaque registre étant une copie comforme de la tranche de mémoire corrrespondante, il faut faire attention à l'emplacement des données pour que la première soit en face d'une source et que le vecteur puisse "couler" par décalages successifs dans les pipelines. Pour ne pas trop

compliquer l'utilisation des registres, nous utilisons uniquement la source 0 à l'extrémité des registres.

L'unité vectorielle est composée de deux opérateurs pipeline et leurs connexions internes sont décrites dans la figure 3.4.

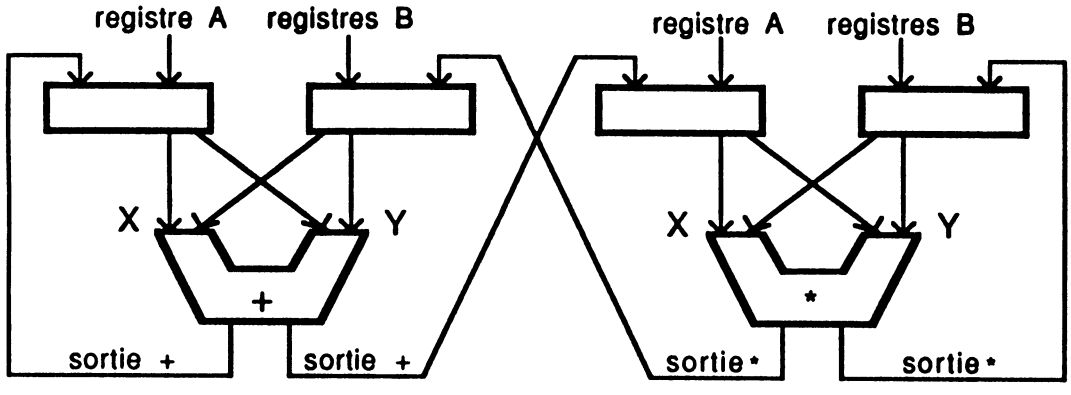


Figure 3.4: Chemin des données dans l'unité vectorielle

3.4.2. COMMUNICATIONS

Plusieurs sous-topologies de l'hypercube sont utilisables; anneaux, grilles, etc... avec des dimensions inférieures à la taille de la machine (!) et qui doivent être une puissance de 2 (le système utilise les facilités d'un code de Gray).

Différentes fonctions permettent de connaître la position relative des processeurs dans les topologies choisies.

Pour communiquer, il faut ouvrir un descripteur de communication où l'on précise le nombre r de communications non-bloquantes possibles. Ensuite il suffit d'utiliser les fonctions de communication entre voisins classiques (envoi/reception) ou celles plus générales fournies par le système (diffusion, ragots, ...).

L'instruction "wait" est bloquante sur une communication en cours et cela permet ainsi d'assurer la cohérence des données.

3.4.3. PROGRAMMATION

Le code est exécuté sur le Transputer. L'accès au VPU se fait par appel à des fonctions de la bibliothèque vectorielle (il n'y a pas de compilateur vectoriseur). Il y a 3 différents niveaux de programmation ("singlenode", "generic", "parameter blocks") dont les performances sont inversement proportionnelles à la facilité de programmation [TVi2].

Les niveaux "singlenode" et "generic" offrent une gestion automatique de l'alignement des données en mémoire. Elles assurent l'intégrité des

données au prix de plusieurs recopies des vecteurs [FPS]. Le coût de cette gestion mémoire peut être énorme (figure 3.6).

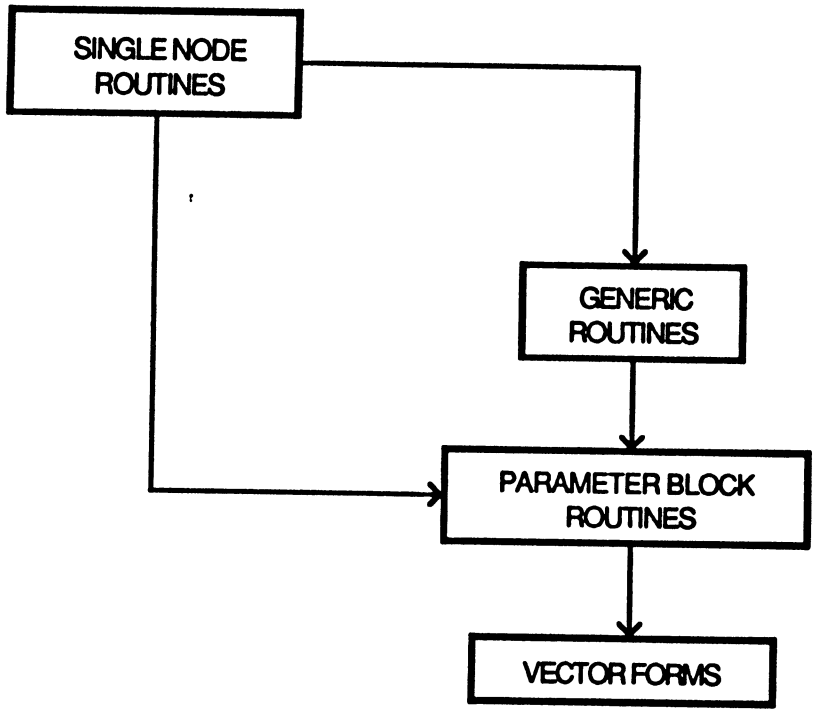


Figure 3.5: Les niveaux de programmation et leurs dépendances

Pour effectuer l'alignement des données et contrôler l'espace mémoire, l'utilisateur dispose du "locator", un outil intervenant après la compilation et avant l'édition de liens en fixant les adresses des variables globales à des valeurs passées en paramètres.

Lorsque les emplacements particuliers des variables utilisées par l'unité vectorielle sont bien choisis, la vitesse d'exécution du programme est améliorée (figure 3.6) [TVi2].

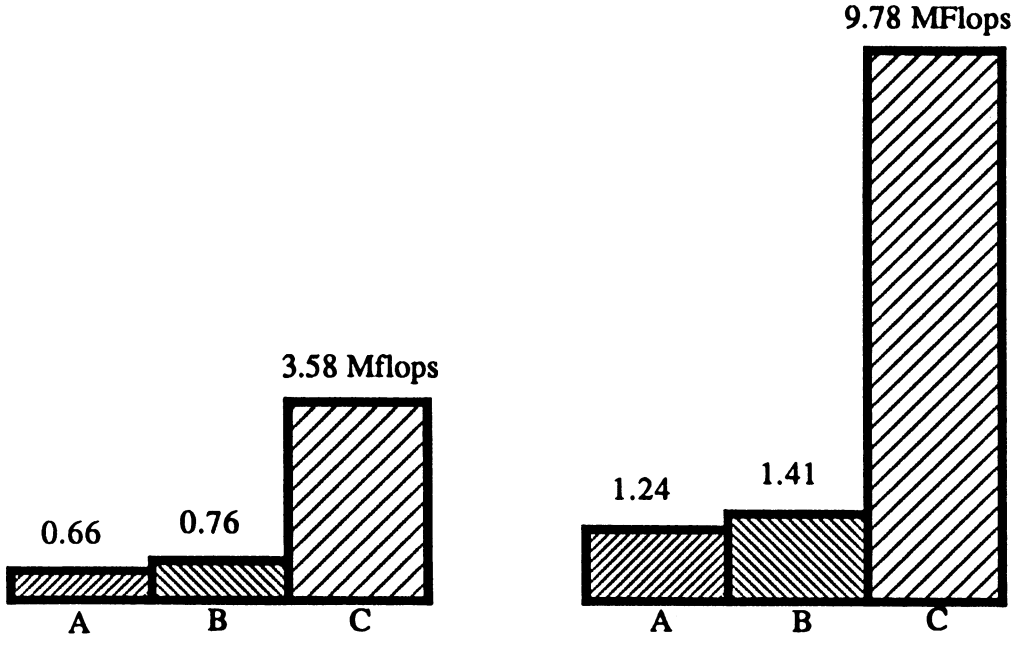


Figure 3.6: Vitesses d'une fonction vectorielle en fonction de l'alignement des données en mémoire (A: 2vecteurs non alignés, B: 1 vecteur non aligné, C: 2 vecteurs alignés)

3.5. PERFORMANCES

3.5.1. SITUATION DE CETTE MACHINE AVEC 16 PROCESSEURS PARMI LES SUPERCALCULATEURS EXISTANTS

Les performances théoriques du système sont les suivantes :

- pour le calcul, chaque processeur peut effectuer 7.5 MIps (Transputer T414) et 12 MFlops soit 120 MIps et 192 MFlops théoriques pour la machine complète.
- pour le transfert des données, chaque processeur peut envoyer ou recevoir 692 KOctets par canal et par seconde et environ 2,5 MOctets pour les quatre canaux d'un processeur en communication parallèle (un seul sens). Cela représente 40 MOctets par seconde (deux sens) pour la machine complète.

A titre de comparaison voiçi une table comprenant la plupart des super-ordinateurs actuels [Her].

Cray1	250 MFlops
Cray X-MP-48 (4 processeurs)	840 MFlops
IBM 3090-400 (4 processeurs)	432 MFlops
CDC Cyber 205	400 MFlops
Réseau hypercube Intel IpscII (16 processeurs)	4,8 MFlops 5,6 MOct/s (2 sens)

Table 3.1: Performance de quelques supercalculateurs actuels

3.5.2. PERFORMANCE VECTORIELLE

Nous avons vu que sur chaque noeud se trouvait une unité vectorielle avec 2 opérateurs. Les algorithmes que nous présentons utilisent tous l'opération SAXPY qui enchaine le multiplieur et l'additionneur. La courbe suivante présente ses performances à différents niveaux de programmation suivant la longueur des vecteurs opérandes.

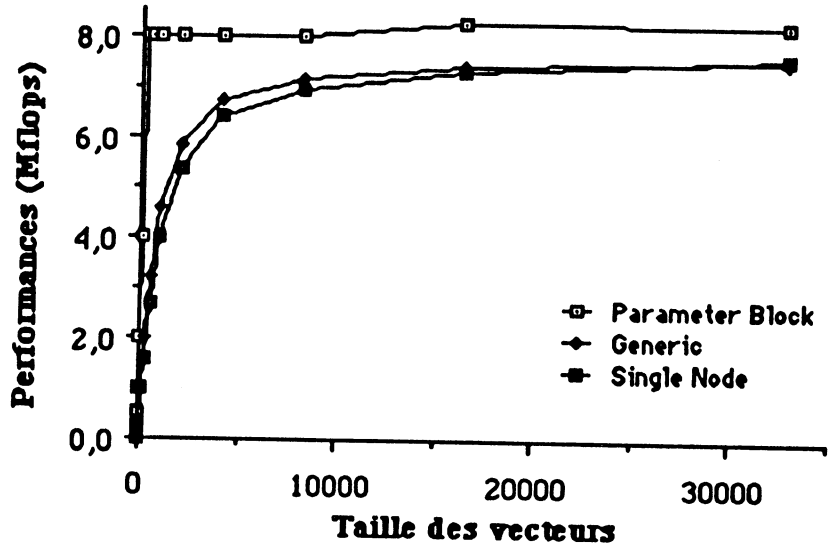


Figure 3.7: Performances de l'opération SAXPY aux trois niveaux de programmation (influence de la longueur des opérandes).

Deux remarques s'imposent, les performances sont, bien sûr, une fonction de la longueur des vecteurs (courbes 3.7) et aussi de la longueur des registres (courbes 3.8).

La durée d'une opération vectorielle est composée d'une partie initialisation (appel de la procédure et contrôle, chargement du pipeline) et d'une partie où une donnée résultat est fournie à chaque cycle pendant la longueur du registre. On ajoute un chargement/déchargement des registres

par tranche de 128 données. Asymptotiquement la partie d'initialisation étant constante, la performance tend vers une limite.

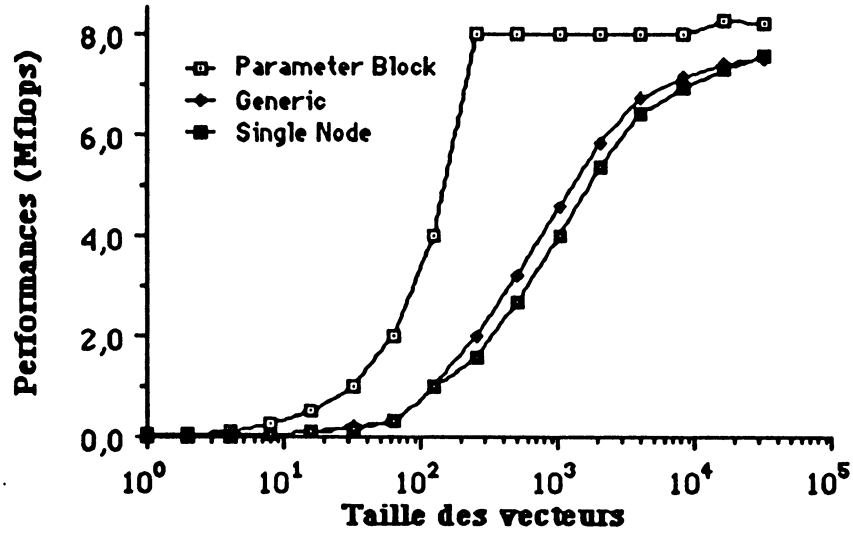


Figure 3.8: Performances de l'opération SAXPY aux trois niveaux de programmation (influence de la longueur 128 des registres).

Le chargement ou déchargement des registres coûte un temps constant quelque soit le nombre d'éléments se trouvant à l'intérieur. Il est donc intéressant d'utiliser ce temps constant pour le plus grand nombre de données possibles (ici 128).

Ce n'est pas l'opération la plus rapide sur cette machine, le produit scalaire est plus performant (11 Mflops en pointe) car le résultat intermédiaire (somme partielle) est stocké dans un registre scalaire interne pendant la série d'opération. Cela diminue les transferts de données car on ne décharge pas de résultats en mémoire avant la fin des vecteurs.

Les opérations non enchainées présentent des performances égales à la moitiée de celle de l'opération SAXPY, avec les mêmes caractéristiques [STo2].

3.5.3. PERFORMANCE DES CANAUX DE COMMUNICATION

3.5.3.1. Largeur de bande

Les valeurs sont présentées dans cette partie pour la première (langage Occam [STo1]) et la dernière version du système (langage C [STo2]), afin de fixer les potentialités et l'influence du système qui les exploite.

Il est clair que la dernière version du système ne permet plus d'utiliser toutes les possibilités des liens de communication: le parallélisme sur les deux sens de communication d'un même canal n'est plus possible (figure 3.9).

De plus, on voit un décalage des courbes sur l'axe des abscisses. Le microprocesseur chargé des communications possède 4 canaux, pour permettre la modularité de la machine jusqu'aux hypercubes de dimension 15 (plus un lien vers l'hôte), les canaux ont été multiplexés de un vers quatre. Dans les premières versions du système les valeurs de ce multiplexage était fixées par le programmeur avec l'appel d'une procédure à chaque changement de topologie. L'évolution du système vers plus d'automatisme dans l'utilisation des procédures et plus de simplicité (?) de programmation a amené ses auteurs à inclure cette procédure coûteuse dans chaque procédure de communication. Il résulte une diminution de l'efficacité pour les messages inférieurs à 10000 mots alors que le Transputer permet un temps d'initialisation remarquablement court (l'équivalent de la transmission de 1,5 mot). Ce qui explique le décalage.

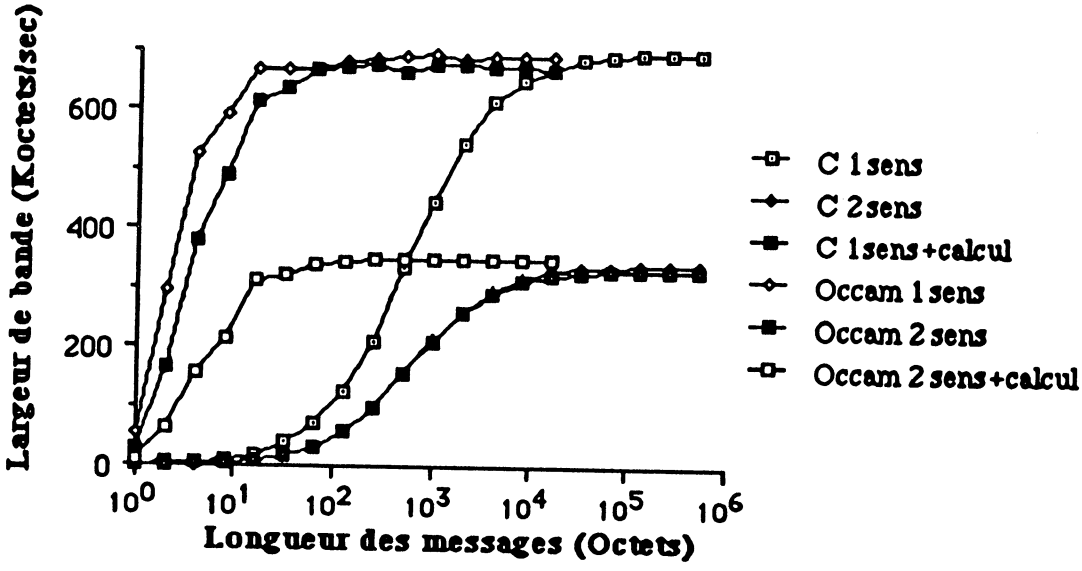


Figure 3.9: Vitesse de transmission des données sur un canal de communication pour les languages C et Occam

La largeur de bande maximale est atteinte avec le langage C avec une valeur de 0,70 MOctets/sec sur un canal dans un seul sens. Cela représente 56% des 1,25 MOctets/sec de la performance maximale des liens.

3.5.3.2. Calcul du temps d'initialisation et du temps de transfert

Suivant le modèle classique, le temps de communication, $T_c=\beta+L\tau_c$, pour la communication d'un message de longueur L (cf § 4, 5).

Le calcul de τ et β s'effectue avec les valeurs extrèmes de cette courbe. Lorsque l'on utilise pas de routage automatique, sur un seul lien de communication, on trouve les valeurs suivantes:

nombre de liens en //	τ (µsec)	β (µsec)	β_{moy} (µsec)
1	1,43	768≤≤970	830
4	1,555	2835≤≤3123	2992

Table 3.2: Valeurs expérimentales des temps d'initialisation d'une communication (β) et de transmission d'un octet (τ).

τ est calculé à partir des plus grands messages possibles. Le calcul est plus délicat pour β aux petites valeurs car l'unité accéssible de l'horloge interne est 68 micro-secondes (de l'ordre des mesures).

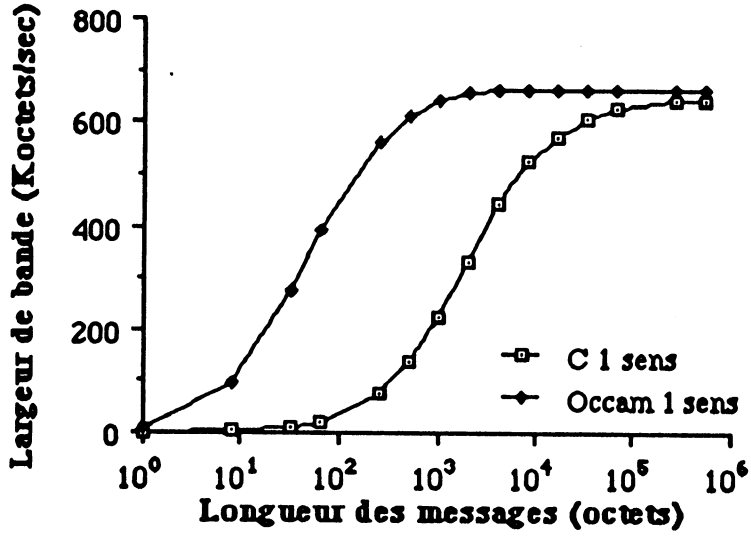


Figure 3.10: Vitesse de transmission des données sur 4 canaux de communication en parallèle

Avec 4 canaux en parallèle, la largeur de bande maximale est 0,64 Moctets/sec par lien et les résulats sont aussi moins bons en se qui concerne β. On peut penser que cela est dû à la mise à jour du multiplexeur des liens séquentiellement pour chacun d'eux. En effet pour la mise en place des canaux physiques, une vérification est effectuée, il faut donc une synchronisation implicite des processeurs qui sera exécuté plusieurs fois (donc séquentiellement).

Sur le FPS avec la version C01 du système la transmission de messages courts est donc très couteuse car le temps d'initialisation est 579 fois plus grand, dans le meilleur cas, que le temps de transmission d'un octet (β est équivalent à la transmission d'un message de 72 réels).

3.6. EVALUATION

Sur une machine classique, le temps des accès à la mémoire est du même ordre que le temps des opérations élémentaires c'est à dire quelques cycles. Cet équilibre permet de ne pas avoir de "goulot d'étranglement" dans une partie de la machine. Pour les machines vectorielles, il faut utiliser des registres à chargement et déchargement rapides pour améliorer l'accès aux données et atteindre.l'équilibre avec les unités de calcul flottant.

Sur une machine multi-processeurs vectoriels à mémoire distribuée, il est sûr que certains opérandes ou résultats devront être transférés de, ou vers, les voisins. La vitesse de communication sur des liens entre processeurs est physiquement limitée (électricité). Les techniques optiques, opto-électroniques et supra conductrices n'étant pas encore à des stades industriels il y a donc une différence entre performances de communications et de calcul. Cette différence peut-être compensée par la décomposition du problème en granularité plus forte, mais ce n'est pas toujours possible ou suffisant. Nous allons étudier, dans le meilleur des cas, le rapport entre le temps de calcul d'une donnée et son temps de transfert sur le réseau d'interconnections.

Pour cela nous comparons à taille égale à 16, les machines du commerce [Dun].

	FPS T C	FPS T Occam	FPS T C rout	Ametek	Ncube	Ipsc I
β (μsec)	830	15	1024	564	447	650
τ (μsec/octets)	1,43	1,44	1,61	9,5	2,4	1,8
V (Kmots/sec)	87,4	86,8	77,6	13,2	52,1	69,4
Perf. (Kflops)	11000	11000	11000	40	120	40
rapport comm./calcul	0,008	0,008	0,007	0,33	0,43	1,73

Table 3.3: Comparaison de quelques machines existantes

Les chiffres montrent bien le déséquilibre entre vitesse de communication et de calcul sur les machines FPS T. Ceci oblige pour une programmation performante à utiliser une grosse granularité de décomposition pour minimiser autant que possible les communications.

D'un point de vue plus local, pour chaque noeud, l'équilibre est bien conservé entre opérateurs vectoriels et chargement des registres et la performance maximum des composants est presque atteinte (11Mflops pour 12 Mflops de crête).

L'utilisation d'une machine pour le calcul numérique en virgule flottante pose un autre problème d'équilibre celui des performances scalaires et

vectorielles. Quel que soit le problème, la part non vectorisable du calcul est toujours existante, et pour ne pas avoir un moteur de Ferrari dans une 2 CV, il faut que l'unité scalaire soit capable de performances honorables. Nous présentons les valeurs de performances des principales opérations réelles scalaire.

Opération	Unité vectorielle		Transputer	
	Generic	Vector F	T414	T800
Multiplication	0,008	0,038	0,006	1,1
Addition	0,008	0,038	0,009	1,1

Table 3. 4: Opérations scalaires (Mflops)

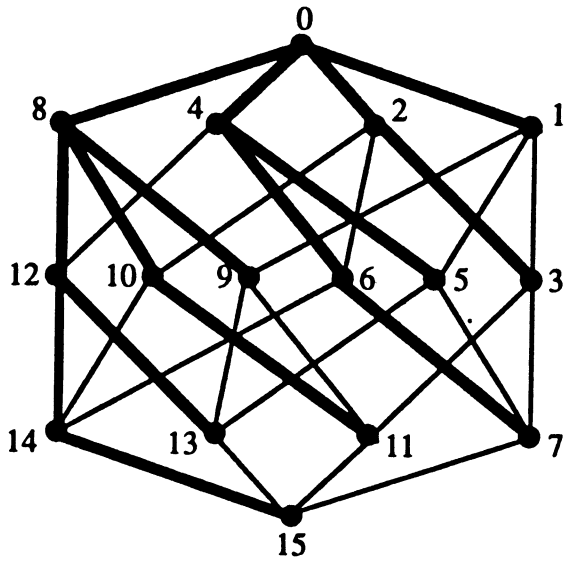
L'unité scalaire T414 [Inm1] est beaucoup trop lente: 0,009 Mflops par rapport aux 11 Mflops du calcul vectoriel. Même en utilisant l'unité vectorielle pour le calcul scalaire (avec des vecteurs de longueur 1), le rapport est du même ordre, et la programmation devient alors infernale (un appel de procédure par opération arithmétique élémentaire !). Ce problème serait résolu avec la nouvelle version de l'unité scalaire (Transputer T800, 1,1 Mflops [Inm2]) et nous le mettons en évidence § 7.

3.7. CONCLUSION

Quelques points sont à dégager. Pour les communications, le matériel permet des performances honorables (par rapport aux autres machines) grâce au Transputer. Cela n'est pas suffisant, la vitesse importante des unités vectorielles introduit un double déséquilibre. D'une part vis à vis des communications, ce qui oriente l'utilisateur vers une décomposition en grosses tâches des problèmes et vers des algorithmes où le calcul domine et d'autre part et surtout vis à vis des performances scalaires du Transputer modèle T414, qui ne permet pas d'utiliser pleinement les capacités des unités vectorielles (§ 7).

En utilisation courante la performance de l'ensemble se rapproche de celle des "super-minis" avec pour certaines applications les capacités d'un "super-calculateur" pour un prix de revient très inférieur.

4. Algorithmes de diffusion sur hypercube 59



4. ALGORITHMES DE DIFFUSION SUR HYPERCUBE

Où nous étudions différents algorithmes pour effectuer la diffusion sur un hypercube issus d'un article de [HJo].

4.1. PROPRIÉTÉS TOPOLOGIQUES DES HYPERCUBES

Nous avons besoin de quelques propriétés des hypercubes pour l'étude de la diffusion sur cette topologie.

Les hypercubes se construisent récursivement par duplication. A partir de 2 hypercubes de dimension m, on construit un hypercube de dimension m+1 en reliant les sommets de même emplacement dans chaque hypercube (figure 4.1). L'hypercube de dimension 0 est un sommet isolé.

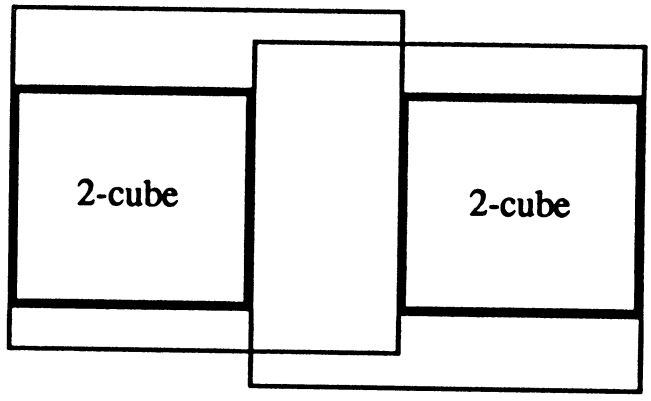


Figure 4.1: Construction d'un 3-cube à partir de deux 2-cubes

Le nombre de sommets d'un m-cube est 2^m, le nombre d'arêtes $m2^{m-1}$, le nombre de voisins de chaque sommet m, le nombre de voisins à la distance i $\binom{m}{i}$ et la profondeur du graphe (plus long chemin) m. Deux sommets quelconques sont reliés par m chemins disjoints.

Les noeud d'un hypercube peuvent être numérotés de 0 à 2^m-1 de telle manière que leur écriture en binaire ne diffère que d'un bit entre voisins [SSc][Saa2]. Ce sont les codes de Gray, qui permettent aussi de définir des

anneaux et des grilles contenues dans un hypercube, ainsi que des arbres de recouvrement (figure 4.3)

Par exemple le code de Gray standard. On débute avec la séquence de dimension 1: $G_1 = \{0, 1\}$. Ensuite, si l'on appelle R la fonction de Réfléction et ^ l'opération de concaténation, le code de numérotation de la dimension m+1 est obtenu à partir de ceux de la dimension m:

$G_{m+1} = \{0\text{^}G_m, 1\text{^}R(G_m)\}$.

Par exemple:

$G_2=\{0\text{^}G_1, 1\text{^}R(G_1)\}=\{0\text{^}\{0, 1\}, 1\text{^}\{1, 0\}\}=\{00, 01, 11, 10\}$

Ainsi lorsque l'on fixe un bit à une valeur donnée, en faisant varier les autres on parcourt un hypercube de dimension m-1. Chaque bit de cette numérotation binaire correspond à une dimension de la topologie.

Une propriété intéressante est le fait que de nombreuses sous-topologies soient incluses dans les hypercubes (figure 4.2 anneau en gras), en plus bien sûr des "bonnes" propriétés qui sont: le diamètre et le nombre de liens en $\log_2$ du nombre total de processeurs [SSc].

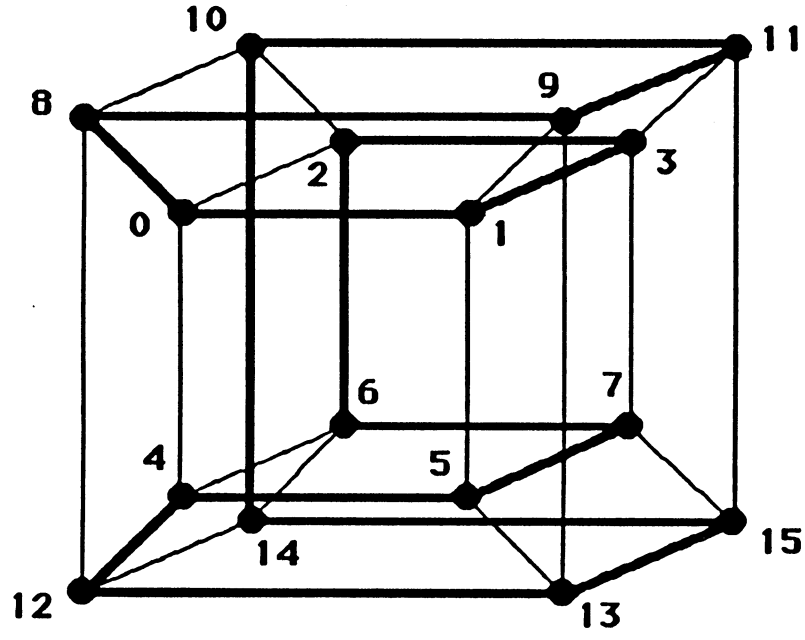


Figure 4.2: 4-cube dont les sommets sont numérotés par un code de Gray (en gras une sous topologie anneau)

4.2. DIFFUSION STANDARD

La diffusion d'une donnée à partir d'un des noeuds de l'hypercube peut être faite simplement en suivant les arêtes d'un arbre de recouvrement minimal inclus dans le m-cube [SSc][HJo]. Le chemin critique est alors de longeur m (chapitre 4.1) et le nombre d'arêtes utilisées égal à 2^{m-1}, le nombre d'arêtes utilisées pour 2^m sommets atteints est de ce fait minimum.

D'autres méthodes sont proposées dans [HJo] avec des arbres plus équilibrés, mais la profondeur reste la même (diamètre du graphe l'hypercube) et le temps total de la procédure aussi.

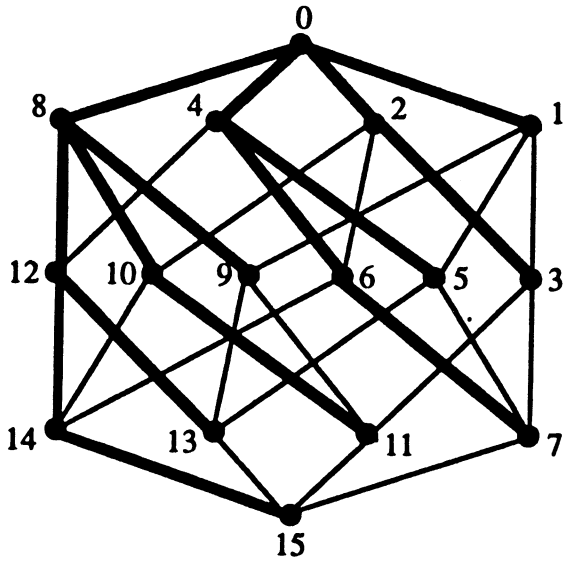


Figure 4.3: Un arbre de recouvrement (racine 0) dans un 4-cube

On suppose que les processeurs connaissent leur numéro suivant un code de Gray abs et que les communications entre voisins sont réalisées avec les procédures suivantes (de base sur les machines à mémoire distribuée):

```
recep(lien_hyp,buf_émis,long)    /* reception */
emis(lien_hyp,buf_reçu,long)     /* émission */
```

Avec lien_hyp la dimension sur laquelle on communique (i.e. le numéro du bit qui diffère dans le code de Gray entre les 2 processeurs communiquants), buf_reçu et buf_émis les adresses des messages en mémoire et long la taille des envois. Notons que l'hypothèse de parallélisme sur les liens de communications est réaliste. Des messages peuvent être envoyés parallèlement sur des liens différents de manière satisfaisante (la dégradation des performances est faible) pour le Transputer et donc le FPS T20. Notons que sur le nouveau Transputer (T800) cette dégradation sera encore plus faible, si l'on reste avec des topologies de degré inférieur ou égales à quatre.

Si l'on suppose que ces procédures sont non-bloquantes pour le processus père, il faut introduire une fonction fin_com(lien_hyp) qui est bloquante tant que les processus initialisés sur lien_hyp ne se sont pas exécutés. Cette procédure sera utilisée dans les processus pères lorsque ceux-ci ont besoin d'assurer la cohérence des données reçues, et aussi dans la procédure de diffusion elle-même pour ne transmettre les données que, lorsqu'elles se trouvent valide en mémoire. La procédure de diffusion peut être écrite (avec des opérateurs pseudo C) comme suit:

```
/* Procédure de diffusion depuis le processeur racine */
{définir bit(A,b) ((A>>b)&1)     /* bit(A,b) est le bième bit de A */
/* on se ramène au cas où la racine = 0000 */
pos = abs xor racine;
/* indice du premier bit égal à 1 */
     prem_1=0;
     tantque ((bit(pos,prem_1)=0) et (prem_1<m)) {prem_1 ++}
          fintantque
     /* diffusion du message */
pour (i=m-1; i≥0; i--)
     {    si (i = first_1)
               {recep(i, buf_reçu, long);
               fin_com(i);}
          sinon
               {si (i < first_1) {émis(i, buf_émis, long)} finsi}
          finsi
     }finpour
}
```

Les processeurs reçoivent sur le lien correspondant à leur premier bit égal à un et envoient sur les bits égaux à zéro (s'il en existe !) à droite de celui (de valeur un) qui a permit la réception.

Avec le modèle de communication choisi on obtient le temps total de la procédure pour un message de longueur L en suivant le chemin critique de longueur m: $T_d = m*(\beta + L\,\tau_c)$.

On remarque (figure 4.3) qu'un arbre de recouvrement minimal n'utilise pas toutes les arêtes de l'hypercube. Avec ce type d'arbre, sur chaque processeur les communications peuvent s'effectuer séquentiellement sur des liens différents. Le temps d'exécution n'est pas perturbé si l'on ordonne les communications suivant les distances à la feuille la plus éloignée de l'arbre. A chaque étape, les communications réalisées concernent les plus longs chemins restant à parcourir et il n'y a pas d'augmentation du temps total de la procédure.

4.3. DIFFUSION PIPELINE

La première optimisation, décrite par [Saa2], est obtenue en pipelinant les données. On découpe le message en ν paquets égaux (de longueur L/ν) que l'on envoient successivement le long des chemins de communication. Bien sûr $1 \leq \nu \leq L$. C'est une méthode générale qui permet d'éviter que des processeurs restent inactifs pendant la majeure partie du temps.

Le processeur à distance m recevra le premier paquet après $m*(\beta + L\tau_c/\nu)$, puis les ν-1 autres paquets tous les $(\beta + L\tau_c/\nu)$. Le temps total est donc $T_{dp} = (m+\nu-1)*(\beta + (L/\nu)\ \tau_c)$.

On calcule le nombre optimal de paquets avec la dérivée $T_{dp}/\theta\nu$:

$\nu_{opt} = (L\tau_c(m-1)/\beta)1/2$,

Le temps optimal pour cette méthode en remplaçant ν par sa valeur optimale est alors:

$T_{dpopt} = ((L\tau_c)^{1/2} + ((m-1)\beta)^{1/2})^2$.

La procédure de diffusion est peu modifiée:

```
/* Procédure de diffusion pipelinée depuis le processeur raçine */
{définir bit(A,b) ((A>>b)&1) /* bit(A,b) est le bième bit de A */
/* on se ramène au cas où la racine = 0000 */
pos = abs xor racine;
/* indice du premier bit égal à 1 */
prem_1=0;
tantque ((bit(pos,prem_1) = 0) et (prem_1<m)) {first_1++}
    fintantque
/* calcul de νopt */
νopt = (Lτc(m-1)/b)1/2
/* diffusion du message */
pour (i=m-1; i≥0; i--)
    {pour (j=0; j<νopt; j++)
        {si (i == first_1)
            {recep(i, buf_reçu+(j*νopt), long/νopt);
            fin_com(i);}
        sinon
            {si (i<first_1) {émis(i, send_buf+(j*νopt), long/νopt)}
                finsi
        }finsi
    }finpour
finpour
}
```

Un processeur doit attendre d'avoir terminé la réception d'un message pour commencer à le ré-émettre et ceci est obtenu avec fin_com. Ici par contre l'hypothèse de communication parallèle sur des liens différents est

nécessaire puisque lorsque le processus est initialisé, plusieurs étapes se déroulent en même temps.

Théorème 1: Si l'on ne peut recevoir qu'un seul message à la fois sur chacun des processeurs, dans un réseau dont la topologie est un m-cube, le temps optimal pour effectuer la diffusion d'un message est (selon le modèle défini précédemment):

$$m\beta + L\tau_c .$$

Preuve: Le diamètre du graphe étant m, le temps minimal pour diffuser une information sera constitué d'au moins m envois le long du chemin critique donc $m\beta$ et le message arrivera au dernier processeur en au mois $L\tau_c$ unités de temps (longueur du message). ♠♥♦♣

Nous remarquons que $((L\tau_c)^{1/2} + ((m-1)\beta)^{1/2})^2 \leq 2(m\beta + L\tau_c)$ car $(\sqrt{A}+\sqrt{B})^2 \leq 2(A+B)$. Cette borne est donc obtenue à un facteur 2 près dans cette majoration de la diffusion pipeline des données:

$m\beta + L\tau_c \leq T_{dp} \leq 2(m\beta + L\tau_c)$

La différence exacte est le double produit $2(L\tau_c(m-1)\beta)^{1/2}$ qui est négligeable devant L dans une étude asymptotique.

Une manière pour augmenter l'efficacité du procédé pipeline, consiste à faire débuter le travail des processeurs éloignés le plus tôt possible. Il faudrai donc essayer d'envoyer de tout petits messages au départ pour initialiser plus rapidement le procédé et ensuite, augmenter progressivement la taille des messages pour diminuer leur nombre total et ainsi la somme des temps d'initialisation. Mais du fait du caractère discret de la taille des messages (!), le choix du découpage par paquets de tailles égales pour l'envoi pipeliné est la meilleure stratégie possible:

Théorème 4.2: pour la diffusion pipeline précédente, la stratégie d'envois de messages de tailles égales est optimale

Preuve:

Nous montrons que quel que soit la stratégie déterminant la taille et le nombre des paquets, le meilleur compromis est atteint lorsque les paquets sont tous de taille égale à L/ν_{opt}.

Reprenons la diffusion standard pipelinée dans un graphe de diamètre m avec ν messages de taille L/ν,

$T_{dp}=(m+\nu-1)*(\beta + (L/\nu)\,\tau_c)$

que l'on décompose en:

$m\beta$: Les initialisations pour arriver au dernier processeur du chemin critique

$L\tau$: Le temps de transmission du message

$(\nu-1)\beta$: Les initialisations ajoutées par le découpage en ν paquets (au lieu de 1)

$(m-1)\tau L/\nu$: Le temps d'attente introduit par la transmission du premier paquets au dernier processeur

Si l'on prend k paquets, chacun de taille L_i , on peut exprimer le temps total sous la forme:

$T = m\beta + L\tau + (k-1)\beta + (m-1)\tau \Sigma_{1\le i\le k}(L_i-L_{i-1})^+$

avec $L_0=0$ et $x^+ = \max(0, x)$

Et on a la majoration suivante du dernier terme quel que soit k:

$(m-1)\tau \Sigma_{1\le i\le k}(L_i-L_{i-1})^+ \ge ((m-1)\tau) \max(L_i) \ge (m-1)\tau L/k$

Or $(k-1)\beta + (m-1)\tau L/k$ est minimum lorsque $k=\nu_{opt}$

d'où $T \ge T_{dp}$ quel que soit k. ♠♥♦♣

4.4. DIFFUSION TOURNANTE

En reprenant la remarque du chapitre 4.2 sur la non-utilisation de tous les liens de l'hypercube lors de la diffusion, on peut chercher à améliorer l'efficacité de la procédure. L'arbre de recouvrement minimal utilise 2^{m-1} arêtes et l'hypercube en contient m fois plus ($m2^{m-1}$).

Si nous utilisons tous les liens de l'hypercube, nous pouvont générer m arbres de recouvrement différents sur l'hypercube [HJo].

Avec l'hypothèse où m communications en parallèle à partir d'un même processeur sont possibles, il est possible d'ordonnnancer les diffusions pour qu'il n'y ai pas de collisions sur les liens [HJo].

Les arêtes sont alors toutes utilisées à l'étape m. Cette procédure est décrite figure 4.4 pour m=4. La construction des arbres de recouvrement est faite, de manière automatique, en effectuant une rotation des codes de Gray de chaque noeud. Cette construction est valide quelque soit m [HJo].

Les données sont divisées en m parties égales de taille L/m et diffusées en temps $m(\beta + L\tau_c/m)$ sur chacun des arbres de recouvrement. Le parallélisme des communications étant parfait (modèle) et l'ordonnancement sans collision sur un même lien, le temps de la procédure complète est le même: $T_{dt} = m\beta + L\tau_c$.

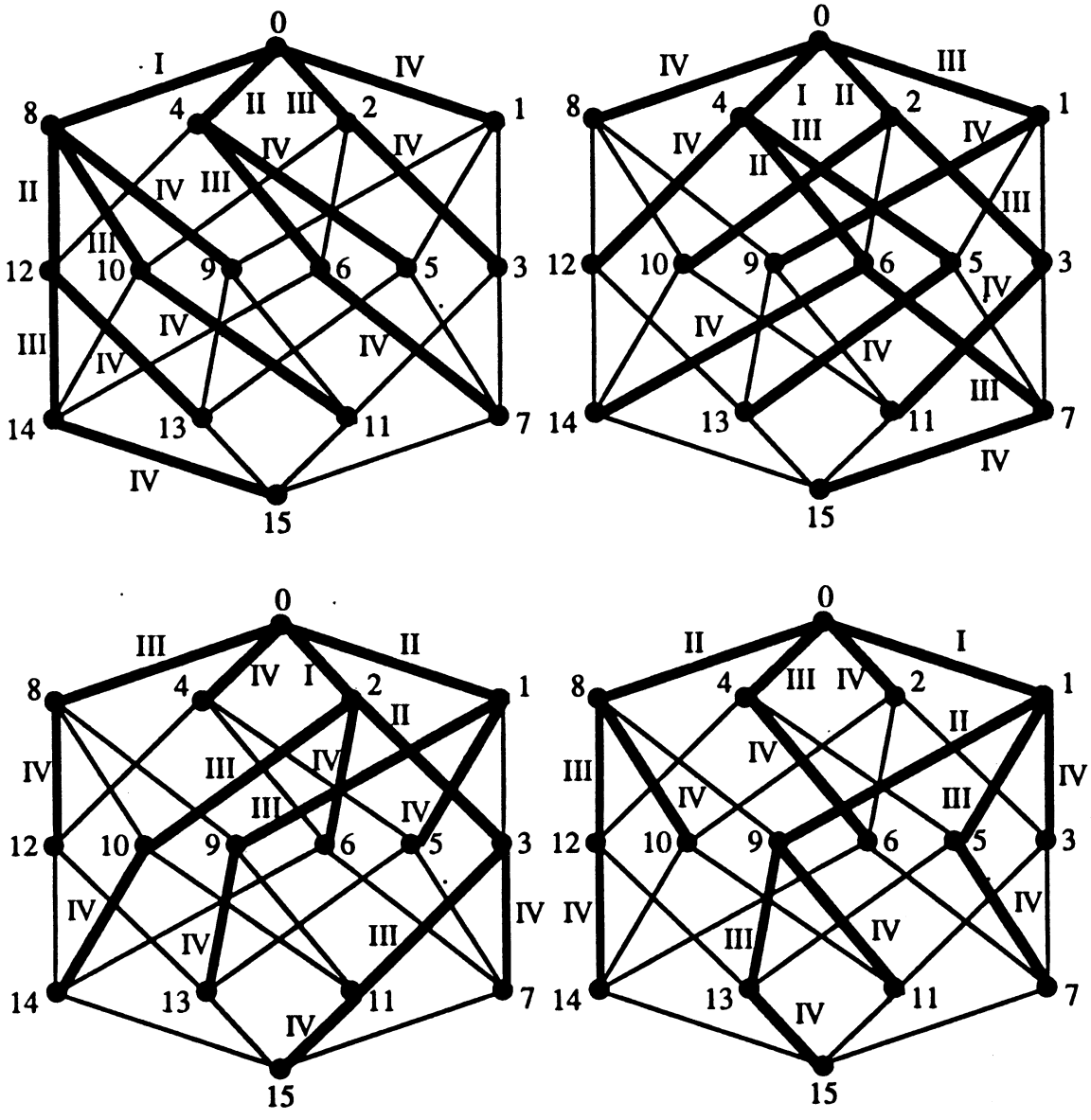


Figure 4.4: 4 arbres de recouvrement distinct dans un 4-cube (les chiffres Romains représentent les macro-unités de temps d'un ordonnancement sans collisions)

L'algorithme se déduit facilement des procédures précédentes en générant m arbres de recouvrement avec des rotations des numéros binaires:

```
/* Procédure de diffusion tournante depuis le processeur raçine */
définir bit(A,b) ((A>>b)&1) /* bit(A,b) est le bième bit de A */
définir rot(A,b) (((A>>(m-b))|(A<<b))&(2m-1)) /* rotation de A de b
bits vers la gauche */
/* on se ramène au cas où la racine = 0000 */
pos = abs xor racine;
```

```
/* pré-calcul des adresses et indice du premier bit égal à 1*/
long_m = long / m;
buf_émis[0]=buf_émis; buf_reçu[0]=buf_reçu;
pour (i=1; i≤m; i++)
     {prem_1[i-1]=0;
     tantque ((bit(rot(pos,i-1)),prem_1[i-1]) = 0) et (prem_1[i-1]<m))
          {prem_1[i-1]++;} fintantque
     /* buf[m] n'est pas une valeur significative */
     buf_émis[i]=buf_émis[i-1]+long_m;
     buf_reçu[i]=buf_reçu[i-1]+long_m;}
/* diffusion du message */
pour (i=m-1; i≥0; i--)  /* étapes de la diffusion */
     {pour (j=0;j<m;j++)     /* arbres de recouvrement */
          {lien=(i-j+m) mod m;
     si (i == first_1[j])
               {recep(lien, buf_reçu[j], long_m);
               fin_com(lien);}
          sinon
               {si (i<first_1[j]) {émis(lien, send_buf[j], long_m)}}finsi
          finsi
     }finpour
finpour
}
```

Théorème 4.3: Si chacun des processeurs peut communiquer en parallèle sur des liens différents, dans un réseau dont la topologie est un graphe de diamètre m et de degré k, le temps minimal pour effectuer la diffusion d'un message est (selon le modèle défini précédemment): $m\beta + (L/k)\tau_c$.

Preuve: Le diamètre du graphe étant m, le temps minimal pour diffuser une information sera constitué d'au moins m envois donc $m\beta$ et le message arrivera au dernier processeur depuis (ou partira du premier vers) au plus k voisins. Cela prendra donc au moins $(L/k)\tau_c$ unités de temps (longueur minimale de la transmission totale telle que la somme des longeurs des messages envoyés par les k voisins soit L). ♠♥♦♣

Le théorème est vrai pour l'hypercube où k=m, la borne est alors $m\beta+(L/m)\tau_c$. [HJo] ont donné une borne minimale légèrement supérieure, $m\beta+(L/m+m-1)\tau_c$ et montré que le pipeline ne peut être appliqué à la diffusion tournante. En effet, les communications se chevauchent sur un même lien dès l'envoi du deuxième paquet. Par contre une méthode plus compliquée pour générérer des arbres de diffusion à arêtes disjointes

permet d'utiliser m diffusions pipelinées à la fois en augmentant de un la profondeur des arbres.

4.5. EXPÉRIMENTATIONS

Figure 4.5, nous reportons les performances des 3 diffusions pour la machine FPS T20 (§ 3), avec 16 processeurs. Chaque noeud possède un Transputer T414 dont les m liens peuvent être activés en parallèle.

La diffusion "tournante" devient 4(=m) fois plus rapide que la diffusion standard pour les grandes valeurs de la taille des messages (les effets des différences de temps d'initialisation diminuent) comme le prédisait l'étude analytique.

La diffusion pipelinée reste elle entre les deux courbe extrèmes et se rapproche de la courbe de la diffusion tournante pour de grandes valeurs de la taille des messages. Ceci corrobore parfaitement l'analyse théorique car, d'une part le poids des différences de temps d'initialisation devient négligeable et d'autre part le poids du double produit $(2(L\tau_c(m-1)\beta)^{1/2})$ diminue par rapport aux termes de plus haut degré.

Les courbes se croisent plus ou moins vite au départ suivant le poids du contrôle dans les procédures et le temps nécéssaire à l'établissement d'un régime "pseudo-asymptotique". La diffusion pipeline est la plus interressante jusqu'à la taille 512, puis pour de grandes valeurs de L les résultats sont meilleurs avec la diffusion tournante. Mais en accord avec l'étude théorique l'écart diffusion tournante - diffusion pipeline ne cesse de diminuer.

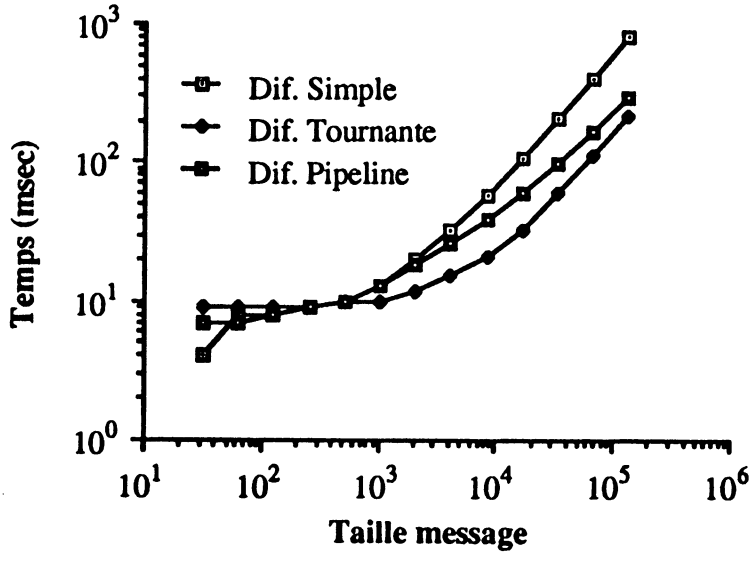


Figure 4.5: Performances des procédures de diffusion

Pour la version pipeline, on calcule le nombre optimal des paquets avec les valeurs de τ_c et β lorsque plusieurs communications sont exécutées en parallèles, ce qui correspond à la réalité dans la majeure partie du temps, entre les courtes périodes d'initialisation et de fin du procédé pipeline.

En pratique le nombre optimal est peu différent de celui calculé. Pour les "petits" messages, on trouve la puissances de deux la plus proche car la taille des messages du test est une puissance de deux. Il n'y a donc pas, avec cette valeur, de reste à la division de la taille du message par le nombre de paquets. Ainsi il ne faut pas d'envoi suplémentaire de longueur le reste de la division.

Pour les grands messages où l'importance de la communication du reste de la division disparait devant le temps total, la meilleure valeur pratique est égale à la valeur optimale calculée.

Taille message (oct)	2048	4096	8192	16364	65536	262144
Diffusion pipeline						
ν_{opt}	1,78	2,52	3,56	5,04	10,08	20,16
ν_{exp}	2	2	4	4	8	20

Figure 4.6: Nombre de paquets optimal et expérimental

4.6. APPLICATION À L'ALGORITHME DE GAUSS

Nous allons présenter les complexités des diffusions pour l'algorithme de gauss sur les différentes topologies d'anneau, d'hypercube et de réseau complet.

On diffuse pendant l'algorithme de Gauss des messages de longueur n-k pour k variant de 1 à n-1. Le temps total d'une stratégie de diffusion synchrone avec l'algorithme de Gauss est le suivant:

$T_c=\Sigma_{1\le k\le n-1}$(diffusion de longueur k)

Avec la procédure standard $T_c=\Sigma_{1\le k\le n-1}\phi(\beta+k\tau)$, où ϕ est le diamètre de la topologie et τ et β sont indicés par r, h et c pour l'anneau, l'hypercube et le réseau complet:

- anneau $\phi=\lceil p/2\rceil$
$T_c=\lceil p/2\rceil/2(n(n-1))\tau_r+\lceil p/2\rceil(n-1)\beta_r$

- hypercube $\phi=\log(p)$
$T_c=(\log(p)/2)(n(n-1))\tau_h+(\log(p))(n-1)\beta_h$

-hypercube diffusion tournante
idem mais remplaçer τ par $\tau/\log(p)$
$T_c=(n(n-1))\tau_h/2+(\log(p))(n-1)\beta_h$

- réseau complet $\phi=1$
$T_c=(n(n-1)/2)\tau_c+(n-1)\beta_c$

Pour la diffusion d'un message de longueur L en ν paquets de tailles égales, il en coûte $(\phi+\nu-1)(\beta+L\tau/\nu)$, on peut calculer le nombre optimal de paquets $\nu_{opt}=((\phi-1)k\tau/\beta)^{1/2}$, d'où
$T_c=\Sigma_{1\le k\le n-1}(\phi+\nu_{opt}-1)(\beta+k\tau/\nu_{opt})$
$T_c=\Sigma_{1\le k\le n-1}(((\phi-1)\beta)^{1/2}+(k\tau)^{1/2})^2$

et les approximations suivantes [CTV] des temps de diffusion par paquets de taille optimale:

-anneau
$$T^r_{comm}=\gamma^r_{n,p}\left[\frac{n(n-1)}{2}\tau_r+\left(n-\frac{p}{4}\right)\left(\frac{p}{2}-1\right)\beta_r\right] \qquad 1\le\gamma^r_{n,p}\le 2$$

-hypercube
$$T^h_{comm}=\gamma^h_{n,p}\left[\frac{n(n-1)}{2}\tau_h+\left(\left(n-\frac{p}{2}\right)(\log_2 p-1)+\log_2\left(\left(\frac{p}{2}-1\right)!\right)+\alpha'_{n,p}\left(\frac{p}{2}-1\right)\right)\beta_h\right]$$
$0\le\alpha'_{n,p}\le 1/2,\ 1\le\gamma^h_{n,p}\le 2$

-le réseau complet présente bien sur le même temps
$$T^c_{comm}=\frac{n(n-1)}{2}\tau_c+(n-1)\beta_c$$

De manière évidente les temps de communication décroissent avec l'augmentation de complexité du réseau. Mais les coefficients de τ diffèrent d'un facteur 2 seulement. Si la bande passante d'un réseau faiblement interconnecté est meilleure T_{com} sera plus faible sur celui ci que sur les réseaux plus complexes.

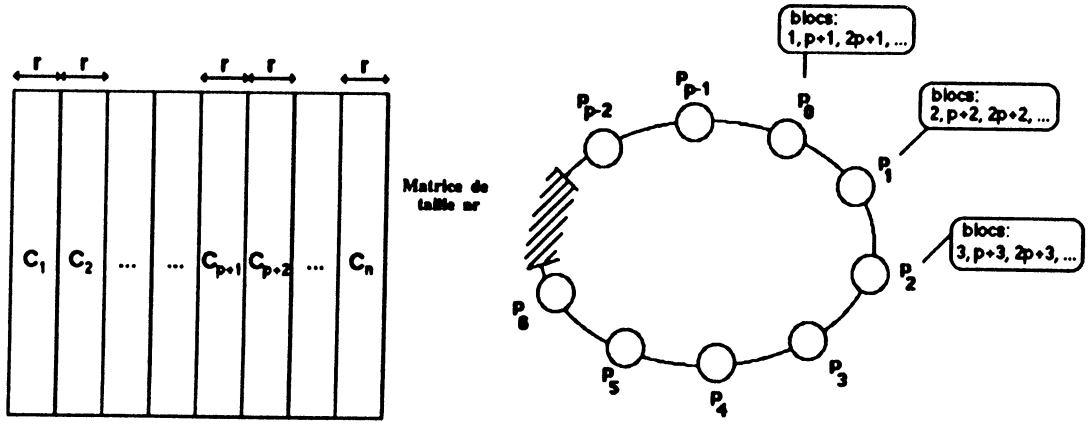
Enfin les coefficients de β diffèrent plus mais peuvent être négligés dans les études sur le comportement des algorithmes asymptotiques en n (avec p fixé).

4.7. CONCLUSION

Nous venons de voir l'importance du choix d'algorithmes performants et bien adaptés pour obtenir de bonnes performances dans les échanges de messages.

Ainsi lorsque l'on utilise des méthodes pipelinées, le réseau complet ne présente pas des capacités bien supérieures à celles de l'anneau pour l'algorithme de Gauss.

5. Répartition des données 75



5. RÉPARTITION DES DONNÉES

Où nous présentons, à partir d'une fonction d'allocation des données, des résultats de complexité pour les algorithmes d'éliminations avec diffusion sur hypercube et avec pipeline sur un anneau et une grille.

5.1. INTRODUCTION

De nombreux auteurs ont étudié la complexité des algorithmes élémentaires usuels de l'algèbre linéaire sur des machines à mémoire partagée (voir par exemple [CMRT][CRTr][LKK][MRo]).

Pour les machines à mémoire locale, l'analyse est plus complexe, car le temps de communication entre deux processeurs qui veulent échanger des données dépend de leur éloignement dans le réseau. Plusieurs auteurs ont proposé des algorithmes pour des machines disposées en anneau, en grille ou en hypercube [CTo2][CTV][GHe][GNe][Hip][ORo][Saa2], mais peu de résultats de complexité sont disponibles, si ce n'est des bornes inférieures [Gei][Gen][ISS][Saa1].

Nous proposons dans cette partie plusieurs implémentations des algorithmes de Gauss et de Jordan sur l'anneau, la grille et l'hypercube de processeurs. Nous améliorons et étendons certains résultats de [CTo2,GNe].

Sur l'hypercube nous nous servons des résultats du § 4 sur la diffusion, étape prépondérante pour la rapidité de l'algorithme de Gauss avec diffusion.

Sur l'anneau nous étudions le problème de la répartition des données et nous donnons des résultats de complexité pour l'algorithme pipeline. En particulier, nous donnons une version asymptotiquement optimale de l'algorithme de Jordan sur l'anneau.

Sur la grille nous donnons des résultats expérimentaux qui laissent penser que le comportement est le même que sur l'anneau.

5.2. MODELES UTILISÉS

5.2.1. ALGORITHMES

Nous rappelons tout d'abord ci-dessous les versions séquentielles des deux algorithmes pour une matrice de taille n x n:

pour k = 1 **à** n
  Jordan: **pour** i = 1 **à** n, **et** i≠k
  Gauss: **pour** i = k+1 **à** n
    tâche T_{ki}: {$a_{ik} \leftarrow a_{ik} / a_{kk}$
      pour j = k+1 **à** n
        $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$}

5.2.2. COMPLEXITÉ

5.2.2.1. Complexité arithmétique

Soit τ_a le temps nécessaire pour réaliser une opération arithmétique. Le temps séquentiel est $T_{seq} = 2\,n^3\,\tau_a / 3 + O(n^2)$ pour l'algorithme de Gauss et $T_{seq} = n^3\,\tau_a + O(n^2)$ pour l'algorithme de Gauss-Jordan.

5.2.2.2. Complexité communication

Le temps nécessaire pour communiquer n données entre deux processeurs voisins est modélisé comme dans la littérature [Saa1] par l'expression $\beta + n\,\tau_c$, où β est un temps d'initialisation et τ_c le temps de transfert élémentaire d'une donnée. Pour nos analyses asymptotiques en n, nous prendrons $\beta = 0$ sans perte de généralité [Saa1][ISS].

Il est clair que β et τ_c dépendent de la machine, par exemple pour deux hypercubes commercialisés installées dans des laboratoires Français:

Hypercubes	β (µs)	τ (µs/mot)
FPS T20	830	11,42
Ipsc II	650(IpscI)	11,42

Table 5.1: β et τ pour deux hypercubes (mot de 64 bits)

Il n'y a presque pas de coût de contrôle pour les communications car ce problème est réglé par le protocole d'échange de messages: P_i peut envoyer un message à P_j si et seulement s'ils sont voisins et P_j est prêt à recevoir le message.

On suppose que les voisins sont reliés par un seul canal de communication qui peut être utilisé pour envoyer ou pour recevoir des messages. Mais

émission et réception sur un même canal ne peuvent pas être exécutées en même temps.

Les communications suivent un protocole de rendez-vous du type CSP et les procédures qui les utilisent peuvent être non bloquantes. Un processeur peut par contre effectuer plusieurs communications en parallèle sur des liens différents.

Tous les canaux peuvent être utilisés en parallèle pour l'émission sans aucune perte de temps du type contension de la mémoire locale, et de même en réception si les emplacements mémoire des messages ne se chevauchent pas (pas de conflit en écriture).

5.2.2.3. Temps parallèle

Le temps d'exécution d'un algorithme parallèle sera modélisé comme la somme $T_p = T_a + T_c$ où T_c contient les temps d'attente dûs aux synchronisations.

5.2.3. HYPOTHESES GÉNÉRALES SUR L'ARCHITECTURE

L'architecture cible est un réseau de processeurs interconnectés. Le fonctionnement global de la machine est asynchrone, de type MIMD [Fly]. Chaque processeur a sa propre mémoire, le système ne comporte pas de mémoire globale partagée, les communications se font donc seulement par échange de messages [HBr].

Notre première hypothèse interdit le recouvrement entre communications et calcul. Cette hypothèse n'est pas trop restrictive si il n'y a pas de temps mort du aux synchronisations des échanges de messages. Le recouvrement permettrait alors un gain d'un facteur 2 seulement [Saal]. Nous avons montré que ce facteur peut être beaucoup plus grand lorsque les attentes ne sont plus négligeables [CTV]. Aussi, la prise en compte des temps morts de synchronisation est elle très importante. Ceci correspond à la réalité de la machine expérimentale FPS T20 (l'introduction de ce parallélisme de manière globale peut être réalisé si l'on connait le coefficient r qui représente le taux de recouvrement calcul / communication: le temps total devient $T_p = r\,Ta + (1-r)\,Tc$).

Dans les algorithmes qui suivent,

- sur l'hypercube chaque processeur doit envoyer et recevoir de tous ses autres voisins
- sur l'anneau (figure 5.1) P_i reçoit en provenance de P_{i-1}, et envoie en direction de P_{i+1} (les indices sont pris modulo p).

- sur la grille, qui est plus exactement un tore de dimension 2 (figure 5.2), P_{ij} reçois en provenance de $P_{i,j-1}$ et $P_{i-1,j}$ et envois en direction de $P_{i+1,j}$ et $P_{i,j+1}$ (les indices sont pris modulo $\sqrt{p}$).

Les deux dernières topologies peuvent donc être orientées pour les communications.

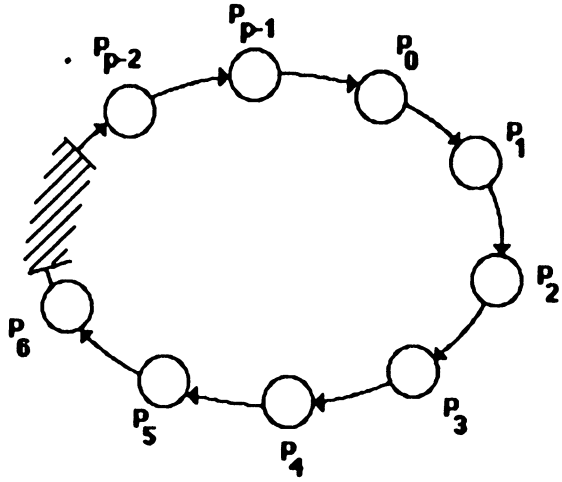


Figure 5.1: Anneau orienté de p processeurs

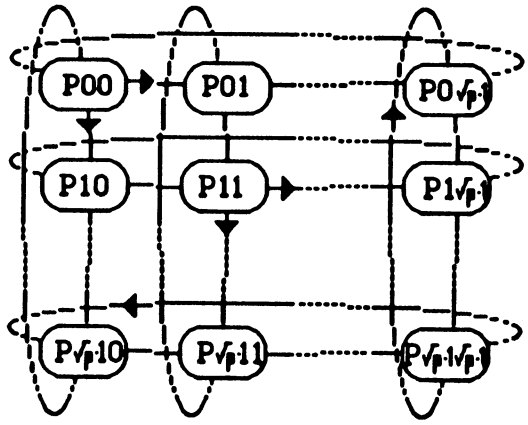


Figure 5.2: Grille orientée de p processeurs

Chaque processeur connait le nombre p de processeurs utilisés et son identité (abs ou mon_numéro). Il peut ainsi calculer l'identité de ses voisins (par simple calcul modulo).

L'élément a_{ij} de la matrice est allouée au processeur alloc(i) pour l'anneau et au processeur (alloc_l(i,j), alloc_c(i,j)) pour la grille.

5.2.4. HYPOTHESES SUR LA RÉPARTITION DES DONNÉES ET LE NOMBRE DE PROCESSEURS

Nous supposons que le nombre de processeurs est un diviseur de la taille n la matrice. Ainsi chaque processeur peut contenir exactement n/p lignes de la matrice initiale dans sa mémoire locale.

Pour étudier l'implémentation parallèle des algorithmes de Gauss et de Gauss-Jordan, nous faisons les hypothèses suivantes utilisées dans [GNe][Saa1]:

(1) **Equidistribution des données**: le nombre de processeurs p est un diviseur de la taille du problème n. Chaque processeur possède exactement n/p lignes (de taille n) de la matrice dans sa mémoire locale pour l'anneau et l'hypercube, $n/\sqrt{p}$ morceaux de lignes (contenant $n/\sqrt{p}$ éléments consécutifs) pour la grille

(2) **Principe de localité**: un processeur ne peut modifier que les données qui résident dans sa mémoire locale

(3) **Principe de précédence**: un processeur doit terminer tous les calculs liés au dernier message reçu avant de commencer les calculs liés au message suivant.

L'hypothèse 1 permet d'équilibrer au mieux la charge de travail entre les processeurs, 2 assure la cohérence des données résultantes et 3 garantit que la sémantique de l'algorithme est vérifiée.

Nous voulons répartir n/p lignes de la matrice nxn de départ (numérotées 0 à n-1) dans les mémoires locales, en équilibrant au mieux la charge dans les processeurs. Le temps de calcul est fonction de la répartition car il dépend du nombre de mises à jours effectuées sur un processeur donné.

Les trois fonctions d'allocations décrites dans [Saa2] et [Gei] sont les suivantes ($0 \le i < p$):

- la répartition entrelacée, chaque processeur contient les lignes dont les indices modulo p sont égaux à son numéro (P_i contient i, p+i, 2p+i, 3p+i, ..., (n/p-1)p+i)
- la répartion par blocs de lignes consécutives de taille n/p (P_i contient i(n/p), i(n/p)+1, i(n/p)+2, ..., i(n/p)+p-1)
- la répartition miroir, si Pi-1 et Pi+1 sont voisins de Pi (il existe un anneau sous-jacent), on alloue les lignes 2kp+i (k=0, 1, ..., $\lceil n/2p \rceil$) et 2kp-i-1 (k=1, 2, ..., $\lfloor n/2p \rfloor$) au processeur i (P_i contient i, 2p-i-1, 2p+i, 4p-i-1, 4p+i, ...).

Dans la littérature, la méthode miroir donne une meilleure répartition de la charge (nombre d'opérations) sur les processeurs mais le même temps d'exécution que l'entrelacée. La raison est la suivante: le processeur qui termine doit attendre la fin des tâches dont dépendent les siennes pour travailler et cela rajoute des délais de synchronisation.

Nous introduisons **la répartition par blocs de taille r entrelacés**: r lignes consécutives sont allouées au processeur P_0 puis les r suivantes au processeur P_1 etc. Nous supposons que le nombre de processeurs fois la taille des blocs est un diviseur de la taille n la matrice. Ainsi chaque processeur peut contenir exactement n/pr blocs de lignes de la matrice initiale dans sa mémoire locale.

Par exemple pour r=2 et p=5 avec une matrice 30x30, le processeur P_0 contiendra les lignes 0, 1, 10, 11, 20, 21. La fonction d'allocation est la suivante alloc(i) = $(\lceil i / r \rceil - 1) \bmod p$. Ainsi le processeur P_i contient les lignes:

krp+ri, krp+ri+1, krp+ri+2, ..., krp+ri+(r-1)

pour k variant de 0 à n/pr-1.

Dans la répartition entrelacée courante les processeurs sont en général numérotés suivant une numérotation usuelle où deux processeurs d'indice consécutifs sont voisins dans la topologie considérée. On prendra par exemple pour l'hypercube un code de Gray, pour l'anneau la numérotation simple, etc... La figure 5.3 représente l'allocation par blocs de colonnes entrelacées pour les versions par colonne des algorithmes.

Dans les analyses asymptotiques qui suivent, nous supposerons que le nombre de processeurs est proportionnel à la taille n du problème: $p = \alpha n$ avec $0<\alpha\leq 1$ une constante.

La raison de ce choix est simple: en mode parallèle parfait $T_a=O(n^3/p)$ avec p processeurs et $T_c=O(n^2)$ (volume des communications) lorsque $p\leq n$, par contre, nous savons par [Saa1] que les communications dominent le calcul si $p=O(n^2)$, il faut donc étudier le rapport temps de calcul, temps de communication quand T_a et T_c sont proportionnels c'est à dire pour $p=\alpha n$ ($\alpha\leq 1$).

Ainsi lorsque n tend vers l'infini le coût des communications et du calcul deviennent de même ordre: $T_a=O(n^3/p)= O(n^2/\alpha)=O(n^2)=T_c$.

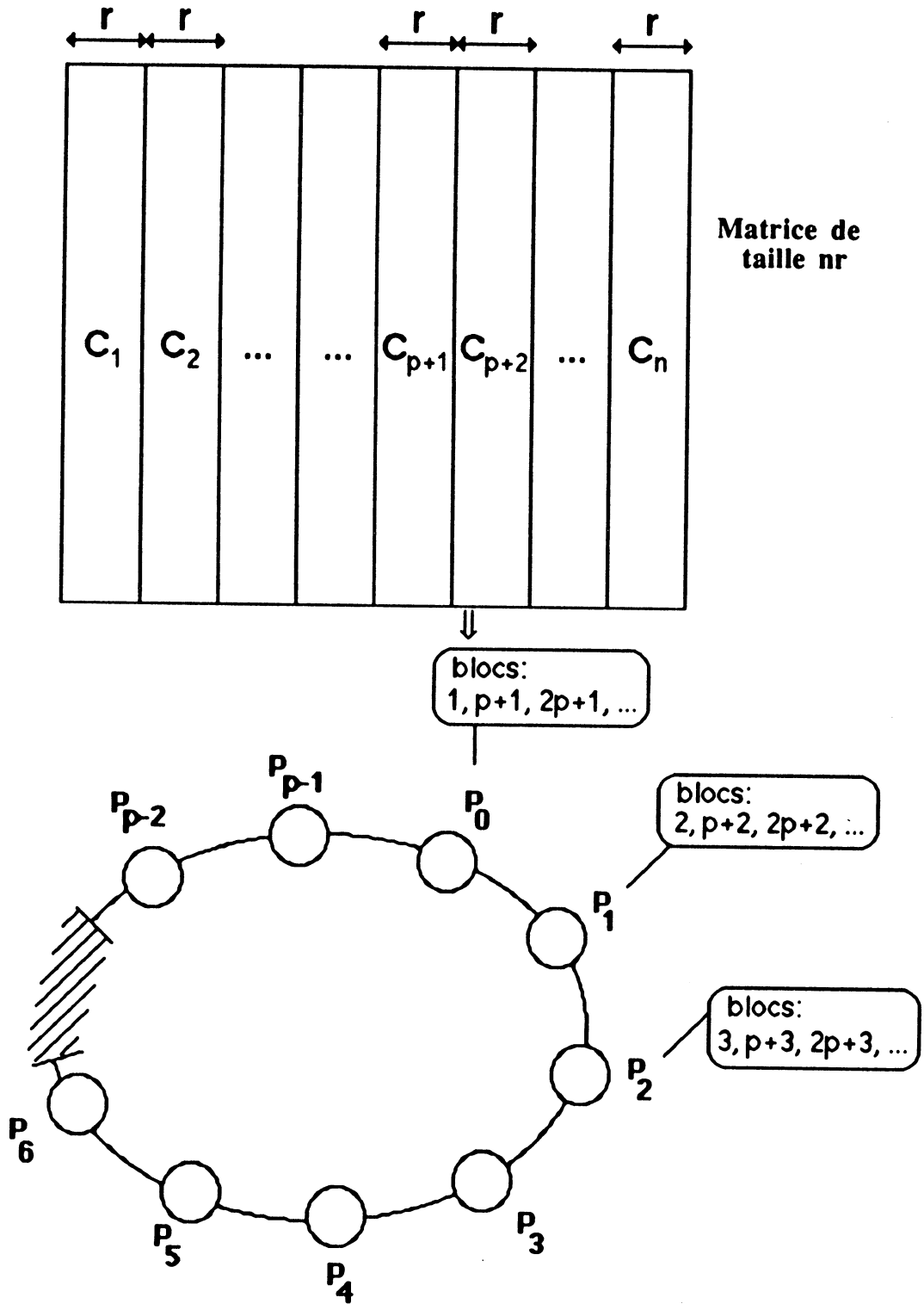


Figure 5.3: Répartition entrelacée par blocs de colonnes sur l'anneau pour une matrice de taille N=nr (les blocs sont numérotés de 1 à n)

5.3. ALGORITHMES CENTRALISÉS (DIFFUSION)

Suivant [Saa2], un moyen simple de paralléliser l'élimination de Gauss lorsque la matrice est divisée en p parties de n/p lignes pour chacun des processeurs, est l'envoi de la ligne pivot à chacun des processeurs. L'algorithme est alors du même type qu'en séquentiel, le processeur qui contient la ligne pivot diffuse simplement l'information et chacun des P_i ne met à jour que les lignes de sa mémoire locale (!). Le problème se décompose naturellement en 2 phases : diffusion et éliminations.

Algorithme de diffusion :
/* programme du processeur P_t */
{**pour** k=1 **à** n-1
  {**si** ligne k **dans** P_t
    {exécuter T_{kk} ;
    diffuser la ligne pivot k aux processeurs P_1,..., P_{t-1}, P_{t+1}, ..., P_p}
  sinon
    {recevoir la ligne pivot k}
  finsi
  /* mettre à jour sa partie de la matrice */
  exécuter T_{ki} pour les lignes $i \geq k+1$}
finpour
}

Dans ces algorithmes "centralisés", à chaque étape le processeur qui contient l'information courante (la ligne pivot) dans sa mémoire locale, effectue un certain travail T_{kk} (ou rien) ensuite il diffuse cette ligne à tous les autres processeurs pour qu'ils puissent mettre à jour leurs propres données.

Le pivotage partiel peut être introduit facilement au niveau de T_{kk} dans la version par colonne de l'algorithme [ISS][CGe].

La topologie choisie pour cette étude est l'hypercube. C'est la topologie de connectivité la plus riche sur les machines actuelles commercialisées. Elle présente les "bonnes" propriétés que nous avons présentés dans le § 4.

5.3.1. ALGORITHME SYNCHRONE

5.3.1.1. Description

A chaque étape k de l'algorithme on trouve 2 phases distinctes (communication, élimination) et dans ce paragraphe nous supposerons que

les processeurs travaillent de manière synchrone, c'est à dire qu'ils attendent la fin de chaque phase sur tous les processeurs pour commencer la suivante.

• phase de communication: diffusion de la ligne pivot k depuis le processeur P_i qui la contient (alloc(k) = i) .

• phase de calcul: chaque processeur met à jour de manière indépendante ses propres lignes.

Si l'on suppose que l'opération de diffusion est synchrone, cela implique que lorsque tous les processeurs commencent leur exécution en même temps, il n'y a aucun recouvrement entre calcul et communication. En fait comme il n'y a que ces 2 phases, chacune des itérations est synchrone.

Nous pouvons donc étudier séparément les 2 noyaux de l'algorithme:

- le choix d'une fonction de diffusion avec les résultats du § 4.
- le choix d'une fonction d'allocation des données qui minimise le temps de calcul c'est à dire qui répartit le mieux possible les calculs sur les différents processeurs.

5.3.1.2. Temps de calcul et temps d'exécution

Soit A une matrice de taille n, nous avons supposé que le nombre de processeurs p fois la taille des blocs r divise n. Equilibrer la charge de calcul des processeurs revient à allouer les lignes de manière à ce que chaque processeur est le même nombre d'éléments à mettre à jour à chaque étape au cours de l'algorithme.

Nous calculons le coût arithmétique d'une ligne pendant l'algorithme d'élimination Gauss:

Soit τ_a le coût d'une opération arithmétique de notre modèle.

- Le coût d'une tache T_{ki}, $1 \leq k < i$, est:

$$W(T_{ki}) = [2(n-k)+1]\tau_a.$$

- La ligne i est mise à jour pendant les étapes 1 à i-1 dans l'algorithme de Gauss, le processeur la contenant doit donc effectuer le nombre d'opération suivant:

$$W(i) = \Sigma_{1 \leq k < i} W(T_{ki}) = (i-1)(2n-i+1)\tau_a$$

(W(i) est une fonction croissante de i.)

- La charge du processeur t est alors:

$$WL(t) = \Sigma_{alloc(i)=t} W(i).$$

- Le temps parallèle total:

$$T_a = \max_{0 \leq t < p} WL(t).$$

Avec la répartition entrelacée par blocs de taille r, le processeur P_{p-1} (numérotation de 0 à p-1) effectue le plus grand nombre d'élimination. En effet il contient des lignes d'indices supérieures à celles des autres processeurs pour chaque cycle (de longueur p) de la distribution.

Le calcul de $WL(P_{p-1})$ donne alors T_a:

$$T_a = n/(6p)\,[4n^2 + 3rpn - r^2p^2]\,\tau_a + o(pn)$$

T_a est une fonction croissante de r: plus les blocs de colonnes consécutives sont grands plus le calcul est mal réparti.

Le problème du calcul du temps parallèle sur une machine MIMD n'est pas notre propos et nous supposerons que, les décalages d'horloge sont plus faibles que la précision de nos mesures et que les processeurs commencent au même instant. Nous mesurons ensuite le temps d'exécution du processeur qui termine pour avoir sous ces hypothèses le temps d'exécution total.

Les transmissions restent les mêmes quelque soit la taille des blocs Le coût des communications ne varie donc pas avec r pour la diffusion synchrone. Le temps total de la procédure de diffusion correspondra au temps calculés § 4 pour l'exécution complète de la procédure car il n'y a pas de recouvrement dans l'algorithme synchrone

Le temps total d'exécution $T_p=T_a+T_c$ augmente donc avec la taille des blocs comme T_a. Les résultats expérimentaux suivent dans la table 5.2 pour une matrice de taille 1024 répartie suivant des blocs de taille 1 à 64 sur 16 processeurs.

taille des blocs	1	2	4	8	16	32	64=n/p
diffusion tournante	**67.3**	67.7	68.4	70.0	73.1	79.2	90.6
diffusion standard	**77.4**	77.8	78.5	80.0	83.2	89.2	100.7
diff standard pipeline	**75,7**	76,0	76,7	77,9	80,5	85,6	95,0

Table 5.2: Temps d'exécution (secondes) pour différentes tailles de blocs (matrice de taille 1024 et 16 processeurs)

Le temps de synchronisation est d'environ 20 secondes lorsque r=1 et augmente avec la taille des blocs.

La meilleure répartition est celle qui minimise le temps de calcul car les communications ne varient pas avec la taille des blocs. Comme le prédisait l'analyse précédente ce résultats est atteint avec r=1.

Les résultats du § 4 sur la diffusion tournante sont ici mis en évidence par les écart du temps d'exécution (8 et 10 secondes) pour le même algorithme avec les deux autres fonctions de diffusion.

5.3.2. ALGORITHME ASYNCHRONE

5.3.2.1. Pipeline des diffusions

Si on laisse l'algorithme s'exécuter de manière asynchrone, les décalages dûs aux différents programmes exécutés par les différents processeurs peuvent être bénéfiques en remplissant certains temps morts où le processeur était en attente de synchronisation pour une communication.

Dans la procédure de diffusion standard l'arbre des communications est très déséquilibré. Ainsi si l'on suppose que les temps de calculs sont peu différents, l'ordre dans lequel s'effectuent les différentes diffusions influe sur la longueur totale des communications (empilement des arbres). Par exemple la figure 5.4 montre deux diffusions enchainées (raçine 0 puis 1). Elles s'exécutent avec une profondeur 4 alors que la somme de deux diffusions est 6.

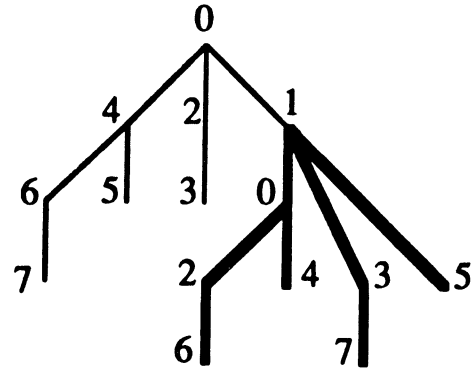


Figure 5.4: 2 arbres de diffusion emboités pour un 3-cube

Cet empilement n'est pas possible dans la procédure de diffusion tournante dont l'arbre est équilibré, mais à chaque niveau du graphe correspond une date de début de communication. Cette différence peut se conjuguer avec celle des autres étapes pour diminuer le temps total.

Le cas de la diffusion pipeline, est le cas le moins favorable pour recouvrir décalage et communication, en effet le découpage en petits paquets restreint l'écart entre les dates de début et de fin du haut et du bas de l'arbre.

Si l'on étudie de plus près l'algorithme de type Gauss, un décalage du même type apparait dans le graphe des tâches, lorsque un processeur exécute une tache T_{kk} (non vide), les autres ne font rien et les graphes de taches peuvent aussi s'emboiter entre deux itérations.

Toutes ces remarque (il en manque sûrement !) montrent ici la complexité de l'analyse du problème lorsque la procédure de

communication est complexe (diffusion) et le déroulement de l'algorithme totalement asynchrone

5.3.2.2. Meilleur chemin dans l'hypercube

Pour effectuer la répartition entrelacée des lignes de la matrice de départ sur l'hypercube, on choisi implicitement un chemin cyclique de taille p dans le graphe hypercube, que l'on parcourt n/pr fois.

Il n'y a aucune raison de penser que cette stratégie est la meilleure, on peut voir le problème plus globalement comme la recherche d'une suite d'indices de processeurs (non-nécessairement voisins) dans l'hypercube. Chacun est cité exactement n/pr fois et les colonnes pivots ainsi affectées respectent une "bonne" allocation pour ne pas diminuer l'efficacité globale de l'algorithme.

Nous nous interressons aux chemins cycliques, leur choix est effectué classiquement en utilisant les propriétés des codes de Gray de la numérotation binaire. Aucune notion de voisinage entre deux successeurs n'est nécessaire dans la topologie.

Pour Gauss différents essais permettent de mettre en évidence de meilleurs chemins dans l'hypercube pour la succession des pivots. Sans pouvoir dégager de régle générale (pour l'instant !)

Pour l'enchainement des procédures de diffusion, on peut discerner une stratégie pour emboiter les diffusions ou les diffusions rotatives en comptant le nombre de bits d'écart entre chaque processeurs dans le chemin décrit. En effet, chaque niveau de l'arbre de diffusion n'a des liens que vers le précédent ou le suivant. Les processeurs d'un même niveau auront fini juste une étape après ceux du niveau supérieur. Si il n'y a qu'un bit d'écart entre deux processeurs, ils appartiennent à des niveaux voisins. Ils terminerons donc une diffusion avec au plus une macro unité de temps d'écart. Ils seront donc prêts à enchainer les diffusions avec peu de perte de temps.

Cela donne une "mesure" de la qualité de l'enchainement des diffusions, en faisant la somme du nombre des bits d'écart entre tous les successeurs du chemin cyclique emprunté dans l'hypercube (Σnb bits$\neq$).

Nous avons choisi trois chemins cycliques dans l'hypercube pour de l'exécution de l'algorithme de Gauss. Les cycles suivis sont la numérotation binaire classique, l'anneau suivant un code de Gray et un chemin qui maximise la distance entre deux successeurs. Ils sont représentés figure 5.5.

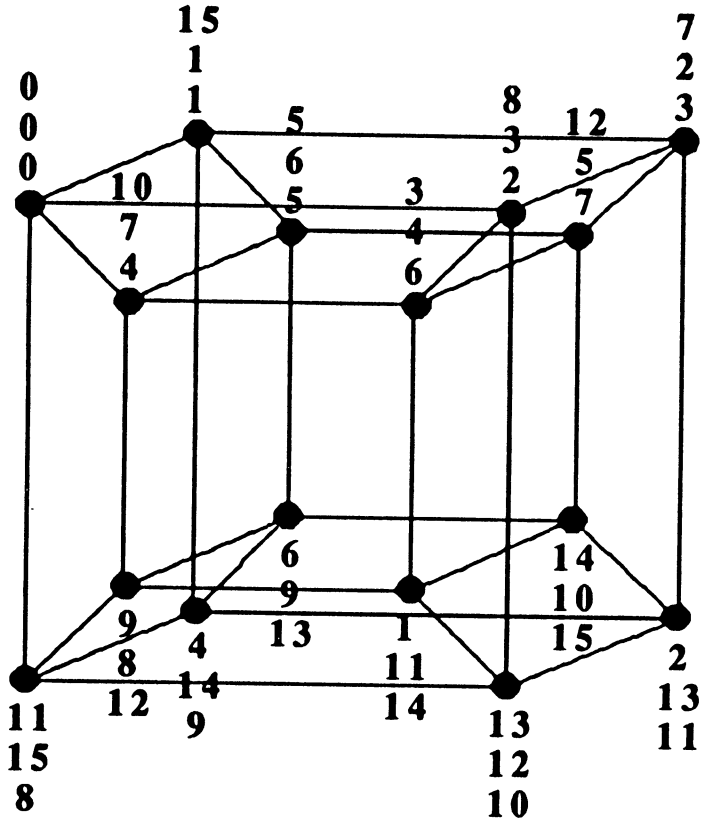


Figure 5.5: Numérotation des sommets suivant les trois chemins dans un 4-cube (de haut en bas: le plus dur, l'anneau, la numérotation standard)

Nous donnons les résultats avec une répartition entrelacée des lignes le long des trois chemins dans la table suivante:

Chemin	Σnb bits≠	meilleur tps (s)	taille bloc	tps bloc=1 (s)
code Gray	30	42,75	4	46,50
anneau	16	42,34	4	44,61
le plus dur	54	44,00	8	55,23

Table 5.3: Meilleur temps d'exécution pour différents cheminements du pivot dans l'algorithme de Gauss avec diffusion classique.

Les écarts sont en accords avec la "mesure" de qualité choisie. La taille des blocs donnant le meilleur temps d'exécution est aussi influencée par la stratégie d'enchainement des pivots.

En prenant comme base de comparaison la solution qui aurait été la meilleure en mode synchrone, le gain par un bon choix de la taille des blocs est de 8, 7 et 20 % suivant les chemins suivis. L'écart entre les stratégies pour le meilleur temps est de 4%.

En mode asynchrone, pour la diffusion, le temps des communications est fonction de la taille r des blocs de la répartition.

La recherche de la meilleure solution permet de gagner presque un quart du temps d'exécution !

5.4. ALGORITHMES À CONTROLE LOCAL (PIPELINE)

Dans la suite nous utilisons deux sous topologies de l'hypercube, anneau et grille, sur lesquelles nous étudions les algorithmes pipelines en fonction des stratégies d'allocation des données.

5.4.1. INTRODUCTION

Nous décrivons des algorithmes qui, basés sur la topologie d'anneau ne demandent pas de communications globales de diffusion mais des communications plus locales pipelinées.

L'anneau est une topologie simple (graphe de degré 2) incluse dans la grille et l'hypercube. Dans un anneau P_i a pour voisins $P_{i-1(\text{mod } p)}$ et $P_{i+1(\text{mod } p)}$ (figure 5.1). Dans la grille, nous utiliserons en fait, des communications le long des anneaux verticaux et horizontaux (figure 5.2).

La diffusion est une opération chère car elle effectue un routage de l'information dans tout le graphe en recopiant les valeurs dans les processeurs traversés. Chaque appel à cette procédure demande donc le concours de tous les processeurs. L'algorithme développé dans la suite n'utilise que les communications entre voisins (envoi/reception) qui sont les primitives de base des machines multi-processeurs interconnectés. Ce type d'algorithme pipeline a été présenté pour la première fois dans la littérature en 1985 par Y. Saad [Saa2].

L'algorithme pipeline est asynchrone. A l'étape k, le processeur qui contient la ligne pivot, P_i, l'envoie à son voisin de droite P_{i+1}, qui à son tour la transmet à P_{i+2}, et ainsi de suite jusqu'au processeur P_{i-1}.
L'avantage de cette stratégie asynchrone provient du fait que le calcul sur un processeur peut recouvrir la communication sur un autre. Ainsi un processeur peut commencer à mettre à jour ses lignes dès qu'il a reçu et retransmis la ligne pivot. Il n'y a pas d'attente globale de la fin de l'étape de communication ou de routage global avant de commencer le calcul, l'algorithme est construit pour utiliser au maximum l'asynchronisme des processeurs suivant un procédé pipeline.

Algorithme pipeline sur l'anneau:
/* programme du processeur P_t */
{**pour** k=1 **à** n-1
{**si** t=k mod p
{executer T_{kk} ;
envoyer la ligne k à P_{t+1}}
sinon
{recevoir la ligne pivot k de P_{t-1}
si t≠k-1 mod p {envoyer la ligne k à P_{t+1}} **finsi**}
finsi
/* mettre à jour sa propre partie de la matrice */
executer T_{ki} pour les lignes i≥k+1}
finpour
}

Pour l'algorithme de Gauss, il suffit à l'étape k de modifier les lignes d'indice supérieur à k, et la boucle d'indice i doit être modifée pour Jordan comme dans le cas séquentiel, pour modifier toutes les ligne différentes de k. Par contre, les conditions de réception et d'émission, qui dépendent de la fonction d'allocation, sont complexes à expliciter dans le cas général.

Le problème qui nous intéresse maintenant est la recherche de la fonction d'allocation par blocs qui minimise le temps total d'exécution T_p, défini comme la somme du temps arithmétique T_a, et du temps de communication T_c (dans lequel on inclut le temps d'attente dû au protocôle de rendez-vous).

Nous avons vu chapitre 5.3.1.2 que T_a était croissant avec la taille r des blocs entrelacés dans l'algorithme de Gauss.

Pour l'algorithme de Jordan, toutes les lignes représentent la même charge de calcul (elles doivent être misent à jour n-1 fois), T_a est donc indépendant de r.

Nous allons étudier T_c pour obtenir le $T_p = T_a + T_c$ minimum.

5.4.2. DESCRIPTION DES COMMUNICATIONS AVEC LES DEUX STRATÉGIES EXTREMES D'ALLOCATION

Pour comprendre l'influence de la fonction d'allocation des données sur les temps de communication, nous étudions tout d'abord sur un exemple la stratégie d'entrelacement de blocs de taille r décrite chapitre 5.2.4.

L'algorithme de Gauss pour une répartition entrelacée des blocs de taille r est présenté dans les figures:

5.6 pour r= n/p (taille maximale).

5.7 pour r=1 (taille minimale)

Seules les communications sont représentées (on suppose qu'il y a peu de différence entre les processeurs pour le temps de calcul) et sont supposées

égales à une unité de temps pour simplifier la présentation (en fait la longueur des transmissions diminue au cours de l'élimination).

	P_0	P_1	P_2	P_3	P_4
lignes	1,2,3,4	5,6,7,8	9,10,11,12	13,14,15,16	17,18,19,20
t = 1	1	1			
t = 2		1	1		
t = 3	2	2	1	1	
t = 4		2	2	1	1
t = 5	3	3	2	2	
t = 6		3	3	2	2
t = 7	4	4	3	3	
t = 8		4	4	3	3
t = 9			4	4	
t = 10		5	5	4	4
t = 11			5	5	
t = 12		6	6	5	5
t = 13			6	6	
t = 14		7	7	6	6
t = 15			7	7	
t = 16		8	8	7	7
t = 17			8	8	
t = 18				8	8
t = 19			9	9	
t = 21				9	9
t = 22			10	10	9
t = 23				10	10
t = 24			11	11	
t = 25				11	11
t = 26			12	12	
t = 27				12	12
t = 28				13	13
t = 29				14	14
t = 30				15	15
t = 31				16	16

Figure 5.6: Date des communication pour Gauss, répartition par blocs de taille maximale (n=20, p=5, r=4)

T_c est plus faible avec la répartition par blocs de taille n/p qu'avec celle par blocs de taille 1. En effet, dans la stratégie pipeline, à chaque changement du processeur émetteur du message on perd une unité de temps à attendre que son voisin soit pret à recevoir. Par contre lorsque le même processeur émet plusieurs fois de suite on retrouve seulement l'attente due au fait qu'il ne reçoit pas sa propre ligne.

Ce résultat est général pour T_c dans les algorithmes de Gauss et Jordan, lorsque les données sont entrelacées par bloc de taille r. Plus les blocs sont grands plus T_c diminue.

	P_0	P_1	P_2	P_3	P_4
lignes	1,6,11,16	2,7,12,17	3,8,13,18	4,9,14,19	5,10,15,20
t = 1	1	1			
t = 2		1	1		
t = 3			1	1	
t = 4		2	2	1	1
t = 5			2	2	
t = 6				2	2
t = 7	2		3	3	2
t = 8				3	3
t = 9	3				3
t = 10	3	3		4	4
t = 11	4				4
t = 12	4	4			
t = 13	5	4	4		5
t = 14	5	5			
t = 15		5	5		
t = 16	6	6	5	5	
t = 17		6	6		
t = 18			6	6	
t = 19		7	7	6	6
t = 20			7	7	
t = 21				7	7
t = 22	7		8	8	7
t = 23				8	8
t = 24	8				8
t = 25	8	8		9	9
t = 26	9				9
t = 27	9	9			
t = 28	10	9	9		10
t = 29	10	10			
t = 30		10	10		
t = 31	11	11	10	10	
t = 32		11	11		
t = 33			11	11	
t = 34		12	12	11	11
t = 35			12	12	
t = 36				12	12
t = 37	12		13	13	12
t = 38				13	13
t = 39	13				13
t = 40	13	13		14	14
t = 41	14				14
t = 42	14	14			
t = 43	15	14	14		15
t = 44	15	15			
t = 45		15	15		
t = 46	16	16	15	15	
t = 47	16	16			
t = 48			16	16	
t = 49		17	17	16	16
t = 50			17	17	
t = 51				17	17
t = 52				18	18
t = 53				18	18
t = 54				19	19

Figure 5.7: Date des communication pour Gauss, repartition par blocs de taille minimale (n=20, p=5, r=1)

5.4.3. ALGORITHME DE GAUSS

Pour l'élimination de Gauss, T_a est plus faible avec la répartition par blocs de taille 1 qu'avec celle par blocs de taille n/p. Plus les blocs sont grands plus les données sont mal réparties (cf 5.3.1.2).

Les temps d'exécution des parties communication et calcul de l'algorithme suivent deux lois de variations opposées. Il faut donc rechercher le meilleur compromis possible (en fonction de la taille r des blocs) qui assure le temps total T_p minimum. Nous allons calculer les complexités T_a et T_c, pour trouver la taille r des blocs qui minimise le temps total d'exécution.

Les temps T_a et T_c dépendent de la fonction d'allocation. Nous ne savons pas calculer T_a et T_c pour toute fonction générale d'allocation. Mais la proposition 5.1 donne la complexité sur l'anneau pour une distribution des lignes par blocs consécutifs de taille r variable.

Proposition 1: Sur un anneau de $p = \alpha\, n$ processeurs, la fonction d'allocation par blocs consécutifs de taille r, $1 \le r \le n/p$ est définie par $alloc(i) = (\lceil i / r \rceil - 1) \bmod p$. Le temps d'exécution de l'algorithme de Gauss est alors:

$$T_p=[4/\alpha+3r-\alpha r^2]n^2\tau_a/6+[1+1/(2r)+\alpha-\alpha^2 r/2]n^2\tau_c+O(n)$$

Preuve :

- Nous avons calculé T_a pour l'algorithme de Gauss avec diffusion pour le processeur qui avait la plus grande charge de travail (§ 5.3.1.2). C'est ici le dernier processeur de l'anneau P_{p-1} qui est le plus chargé:

$$T_a = n/(6p)\,[4n^2 + 3rpn - r^2p^2]\,\tau_a + o(pn)$$

Avec $p=\alpha n$, nous obtenons:

$$T_a = [\,4/\alpha + 3r - \alpha r^2\,]\, n^2\, \tau_a / 6 + O(n)$$

- Pour déterminer T_c, nous utilisons la figure 5.10 où seules les communications apparaissent et sont toutes supposées avoir la valeur unité. Dans chaque encadrement rectangulaire se trouve 2r unités de temps pour envoyer et recevoir r lignes plus une unité due aux contraintes de synchronisation. D'ou l'expression:

$T_c = (p-2)\,(n+1)\,\tau_c$ {initialisation: envoi de la ligne 1 de taille n+1}

$+2\,\Sigma_{1\le k\le n-r}\,(n-k+2)\,\tau_c$ {réception et envoi des lignes k de taille n-k+2}

$+\Sigma_{0\le k\le n/r-p+1}(n-kr+1)\tau_c$ {synchronisation dans chaque encadrement}

et après calcul, $T_c = (\,pn + n^2 + (n^2/(2r)-p^2r/2)\,)\,\tau_c + O(n)$. ♠♥♦♣

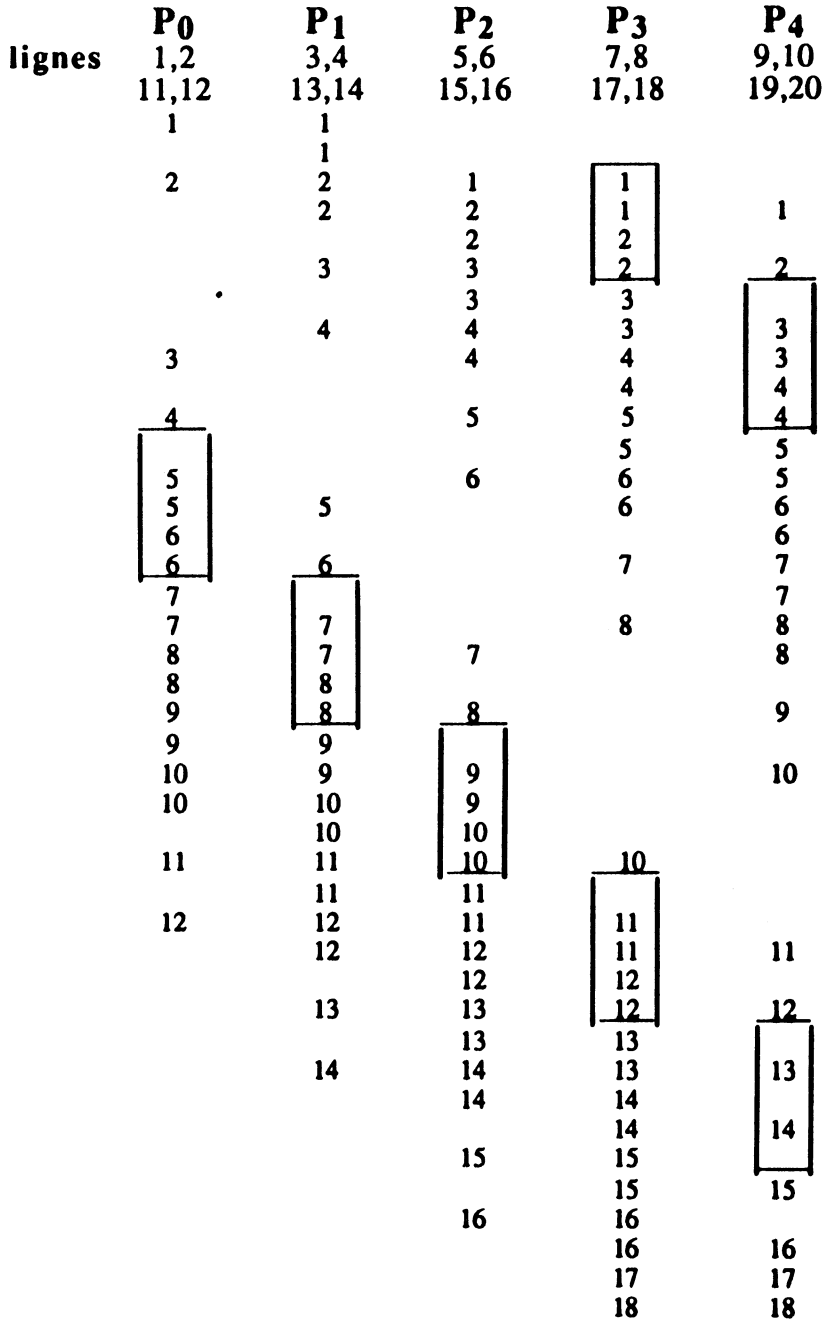


Figure 5.10: Date des communication pour Gauss, répartition par blocs de taille r (n=20, p=5, r=2)

Nous vérifions que T_c est décroissante avec r, comme nous l'avions remarqué dans l'exemple.

En reportant les résultats obtenus dans la formule de l'efficacité classique:

$$e_\alpha = T_{séq}/(pT_p)$$

Nous obtenons pour l'algorithme de Gauss:

$$e_\alpha = 1 / [\ 1 + \lambda_1 \alpha + \lambda_2 \alpha^2 + \lambda_3 \alpha^3\ ]$$

avec $\lambda_1 = 3r/4 + 3\rho/2 + 3\rho/(4r)$, $\lambda_2 = 3\rho/2 - r^2/4$, $\lambda_3 = -3\rho\ r/4$ et $\rho = \tau_c/\tau_a$

ρ est le rapport des coûts élémentaires communication/arithmétique, c'est le paramètre caractéristique de la machine (cf § 3).

ρ et α étant donnés, il est facile de calculer la taille r des blocs qui maximise l'efficacité (ie minimise le temps total) en factorisant ce polynome du second degré.

Pour les petites valeurs de α, e_α peut-être approchée (sans tenir compte des termes en α^2 et α^3) par :

$$e_\alpha = 1 / [\ 1 + (3r/4 + 3\rho/2 + 3\rho/(4r))\alpha\ ]$$

L'efficacité est alors maximum pour $r = \sqrt{\rho}$

La comparaison des deux stratégies les plus couramment utilisées (r=1 et r=n/p) dans [CTV][GNe][Saa2] par exemple est présentée dans le tableau suivant:

	Répartition par blocs	**Répartition entrelacée**
Arithmétique	$\tau_a [\ n^2 / \alpha + O(n)\ ]$	$\tau_a [n^2 (4/\alpha+3-\alpha)/6 + O(n)]$
Communication	$\tau_c [\ n^2 (1+\alpha) + O(n)\ ]$	$\tau_c [n^2 (3+2\alpha-\alpha^2)/2 + O(n)]$

Table 5.4: Complexité de l'algorithme de Gauss avec les répartition entrelacées par blocs de taille 1 et n/p

Nous donnons finalement une borne inférieure:

Lemme 5.1 : Pour l'algorithme de Gauss sur l'anneau avec $p = \alpha n$ processeurs, $0<\alpha\leq 1$, $2\tau_a n^2/(3\alpha)+O(n) \leq T_a$
$\tau_c n^2(1+\alpha)+O(n) \leq T_c$

Preuve : La borne sur T_a est simple: $T_a > T_{seq}/p$. La borne pour T_c est obtenue comme dans le lemme 5.2 démontré dans le chapitre 5.4.4 pour l'algorithme de Jordan. ♠♥♦♣

La borne sur T_c est atteinte avec la distribution par blocs de taille n/p mais la borne sur T_a n'est pas atteinte avec la répartition entrelacée par bloc de taille un. En fait pour la distribution entrelacée:

$$T_a = \tau_a [\ T_{borne} + n^2 (3 - \alpha)/6\ ] + O(n), \text{ avec } T_{borne} = 2 n^2 / (3\alpha)$$

La borne n'est donc pas serrée pour les grandes valeurs de α et T_a ne tend pas vers T_{borne} même pour α petit. Pour atteindre cette borne sur T_a il faut utiliser d'autres répartitions, qui équilibrent encore mieux la charge sur les processeurs: par exemple

- la répartition miroir [GHe][GNe] (cf § 5.2.4) qui donne

$$T_a = \tau_a [\ T_{borne} + \alpha n^2 /3\ ] + O(n)$$

- la double répartition miroir: si $\alpha < 1/4$ le processeur i ($0 \le i < p$), a les lignes $4mp+i+1$, $4mp+2p-i$, $4mp+3p-i$ et $4mp+3p+i+1$, $0 \le m \le n/(4p)-1$.

$$T_a = \tau_a [T_{bound} + \alpha n^2 /12] + O(n).$$

Pour ces deux répartitions T_a tend bien vers T_{borne} lorsque α est petit

5.4.4. ALGORITHME DE JORDAN

Dans ce cas, pour toute fonction d'allocation sur l'anneau, $T_a = T_{seq} / p$ en vertu du principe d'équidistribution des données (car chaque ligne représente une quantité de travail équivalente).

Pour les communications, le coût des synchronisations est du même type que pour l'élimination de Gauss. La fonction T_c est donc décroissante et nous cherchons sa valeur minimale car elle donnera le temps d'exécution total minimum.

De manière générale, nous établissons tout d'abord une borne inférieure:

Lemme 5.2: Sur un anneau de $p = \alpha n$ processeurs, $0 < \alpha \le 1$,

$$(1+\alpha) n^2 \tau_c + O(n) \le T_c$$

Preuve: Considérons le voisin de gauche du processeur qui contient la première ligne, appelons le m. Il faut $(p-1)\, n\, \tau_c = \alpha\, n^2\, \tau_c + O(n)$ unités de temps pour que la ligne 1 arrive à P_m. De plus P_m doit recevoir au moins $n-n/p-1$ lignes de son propre voisin de gauche car il ne contient que n/p lignes dans sa mémoire locale. P_m aura besoin de $n^2/2\, \tau_c + O(n)$ unité de temps (n/p est négligeable devant n) pour recevoir ces lignes de longueur n à 1. Or, le voisin de droite de P_m a aussi besoin de recevoir $n-n/p-1$ lignes de P_m (ce ne sont , bien sur , pas nécessairement celles recues par P_m), et P_m passera $n^2/2\tau_c + O(n)$ unité de temps à les lui envoyer. ♠♥♦♣

La trace temporelle de l'exécution pour une répartition par blocs de taille n/p est décrite dans la figure 5.8 pour $p=4$ et $n=8$. Elle montre le bon équilibre de la charge des processeurs et la faible perte de temps due aux synchronisations.

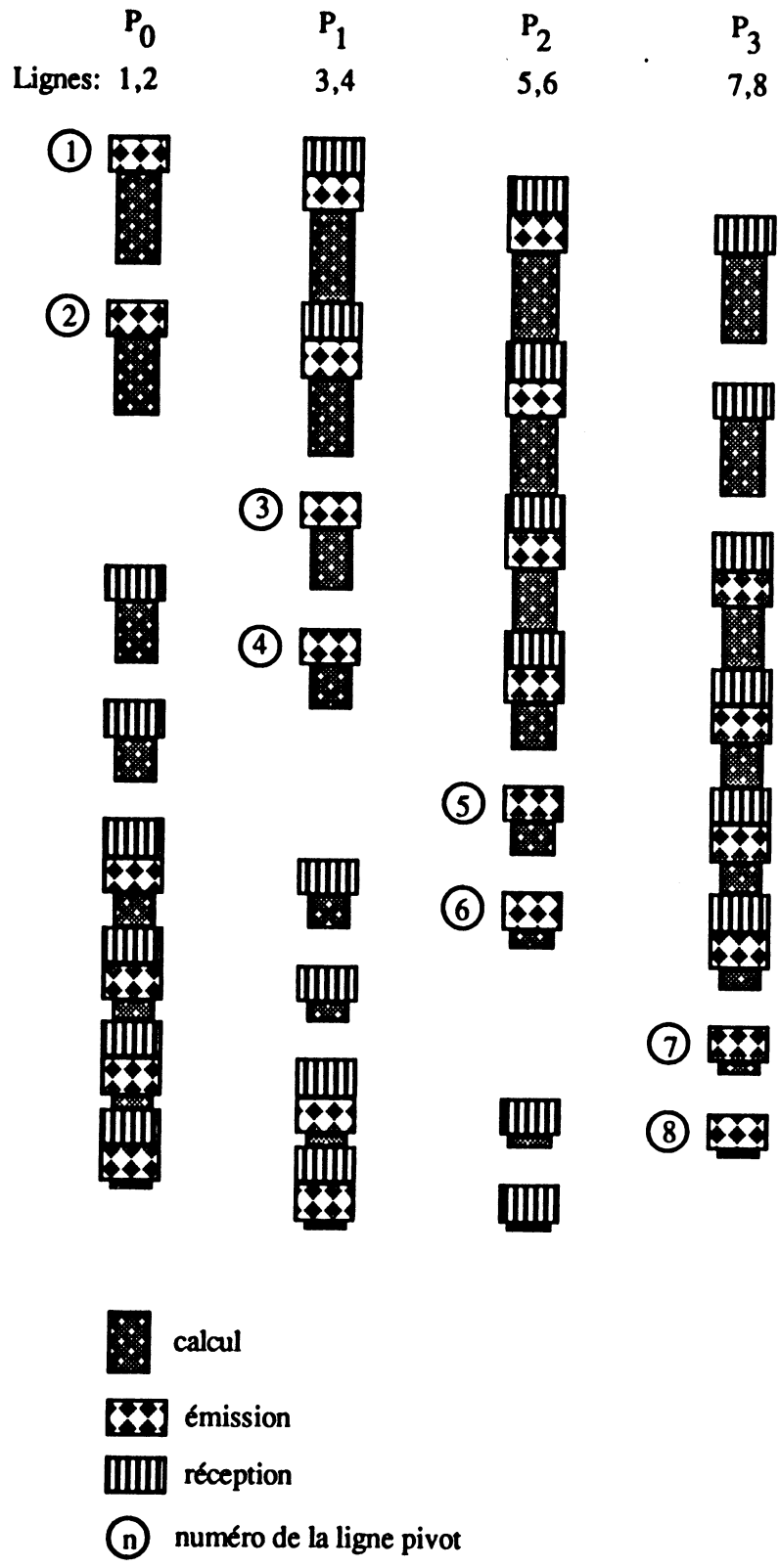


Figure 5.8: Algorithme de Jordan avec une allocation par blocs consécutifs de taille n/p (n=8, p=4)

L'optimalité de la fonction d'allocation par blocs de taille n/p est établie en montrant que la borne du lemme 5.2 est atteinte.

Théorème 5.1: Sur un anneau de $p = \alpha\, n$ processeurs, la fonction d'allocation $alloc(i) = \lceil i\, p / n \rceil - 1$ (allocation par blocs de lignes consécutives) est optimale. La complexité de l'algorithme de Jordan est donc: $T_p = [\, n^2 / \alpha\,]\, \tau_a + [\, (1+\alpha)\, n^2\,]\, \tau_c + O(n)$.

Preuve : Nous devons seulement montrer que $T_c = \tau_c\, [n^2\, (1+\alpha)+O(n)]$ pour l'allocation par blocs. Pour faciliter le calcul de T_c, considérons le schéma simplifié de la figure 5.9 où les calculs sont supprimés et les communications sont prisent égales à l'unité. Pour le dernier processeur nous calculons T_c comme:

$T_c = (p-1)\,(n+1)\,\tau_c$ {initialisation: envoi de la ligne 1 de taille n+1}

$+2\, \Sigma_{1 \le k \le n}\, (n-k+2)\, \tau_c$ {réception puis envoi de la ligne k (longueur n-k+2)}

$+ \quad (p-1)\, 2\, \tau_c$ {envoi de la dernière ligne sur l'anneau}

d'ou $T_c = (pn + n^2)\, \tau_c + O(n)$. ♠♥♦♣

Pour un problème de taille n et un nombre de processeurs $p=\alpha n$ $(0<\alpha\le 1)$, l'efficacité correspondant à l'algorithme de Jordan avec la fonction d'allocation optimale est:

$$e_\alpha = 1 / [\, 1 + \alpha\, (1+\alpha)\, \rho\,]$$

	P_0	P_1	P_2	P_3	P_4	
lignes	1,2	3,4	5,6	7,8	9,10	
	1	1				
		1	1			première ligne
	2	2	1	1		
		2	2	1	1	
			2	2		
		3	3	2	2	
			3	3		
		4	4	3	3	
	3		4	4	3	
				4	4	
	4		5	5	4	
				5	5	receptions/envois
	5		6	6	5	
	5	5		6	6	
	6				6	
	6	6		7	7	
	7				7	
	7	7		8	8	
	8	7	7		8	
	8	8				
	9	8	8		9	
	9	9				
	10	9	9		10	
	10	10	9	9		
		10	10			dernière ligne
			10	10		

Figure 5.9: Trace des communications avec la répartition par blocs de taille n/p pour l'algorithme de Jordan (n=10, p=5)

5.4.5. RÉSULTATS EXPÉRIMENTAUX

Les expériences ont été menées sur un hypercube FPS série T ([GHS], § 3), utilisant jusqu'à 16 processeurs. Nous avons programmé les répartitions par bloc de taille r pour les algorithmes de Gauss et Jordan. Remarquons ici que sur l'anneau T_c est le même pour Gauss et Jordan avec la répartitions par bloc de taille r.

Pour commencer nous vérifions que le temps d'exécution est une fonction décroissante avec la taille des blocs pour l'algorithme de Jordan sur l'anneau (figure 5.11).

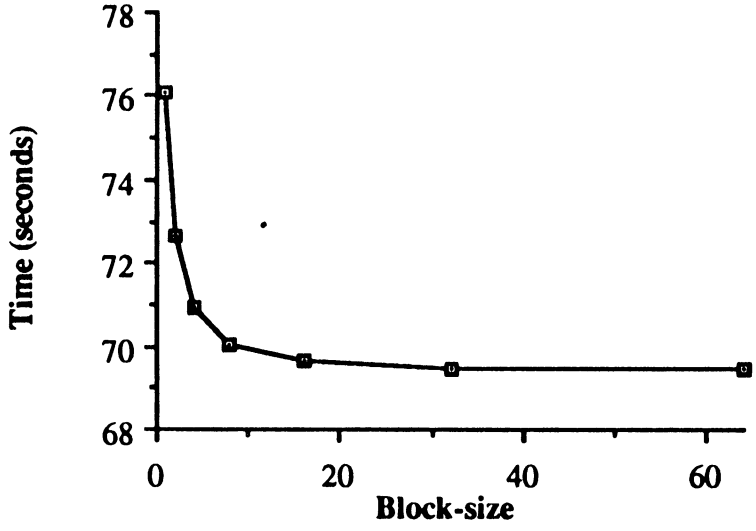


Figure 5.11: Temps d'exécution pour l'algorithme de Jordan sur l'anneau (n=1024, p=16)

Pour Gauss l'expérience montre que, pour obtenir la meilleure performance, le compromis entre T_a et T_c est atteint lorsque r=4.

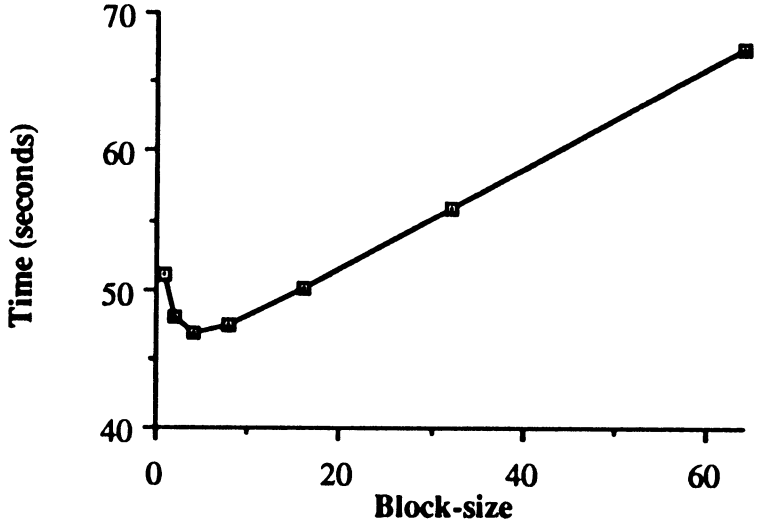


Figure 5.12: Temps d'exécution pour Gauss sur l'anneau (n=1024, p=16)

Pour l'algorithme de Gauss, avec n=480 et en utilisant 4, 8 et 16 processeurs (α varie de 1/120 à 1/30).

Nous donnons dans la table les temps d'exécution pour r=3, 4 et 5 avec des matrices 480x480, avec une programmation de haut niveau des unité vectorielles. La meilleure valeur expérimentale est toujours r=4

Sur la machine FPS, nous estimons, à partir des performances du § 3, la valeur de ρ:

$$\rho=\tau_c/\tau_a=11,4/0,8\approx14,13$$

(messages de taille moyenne 240 et 1,23 Mflops sur des vecteurs de longueur 240).

Nous reportons ensuite cette valeur dans l'expression de l'efficacité e_α pour calculer la valeur de r qui la maximise; cela donne la "valeur optimale" du tableau.

Taille des blocs r	p = 4	p = 8	p = 16
3	35.58	19.97	12.37
4	35.53	19.96	12.26
5	35.71	20.14	12.56
valeur optimale r	3.80	3.85	3.97

Table 5.5: Temps en secondes pour des blocs de taille r et n=480

Pour les petites valeurs de α, e_α est maximale quand $r = \sqrt{\rho} \approx 3,76$, ce qui correspond bien aux résultats obtenus.

5.4.6. ALGORITHMES SUR LA GRILLE

Dans la littérature, [Saa2][ISS] et [Hip] étudient l'implentation de l'algorithme sur la grille avec différentes méthodes d'allocation des données. [Hip] étudie la répartition entrelacée des colonnes par blocs de taille 1 et des lignes par blocs de taille n/p. Il donne uniquement des raisons expérimentales à son choix.

La répartition que nous utilisons sur la grille (ou plus exactement un tore de dimension deux) est un double entrelacement par blocs de taille r des lignes et des colonnes. La fonction d'allocation est la suivante, elle donne pour l'élément a_{ij} de A, le processeurs $P_{alloc\_l(i,j)alloc\_c(i,j)}$ correspondant de la grille:

$$alloc\_l(i,j) = (\lceil i / r \rceil - 1) \bmod \sqrt{p}$$
$$alloc\_c(i,j) = (\lceil j / r \rceil - 1) \bmod \sqrt{p}$$

Avec cette répartition, les pivots sont localisés dans les processeurs d'indices i=j et une même ligne (resp. colonne) de la matrice se trouve dans une même ligne (resp. colonne) de la grille de processeurs.

A l'étape k, chaque processeur doit connaitre l'élément pivot et la partie de la ligne et de la colonne pivot lui correspondant pour mettre à jour ses

propres morceaux de lignes. Ce type d'algorithme parallèle nécessite deux phases de communication distinctes et dépendantes. Il peut être écrit:

```
  /* Algorithme du processeur P_d_horiz, d_vert */
{pour k = 1 à n
      {/* envoi des morceaux de la ligne pivot verticalement */
      si (alloc_l(k,j)=d_horiz)
            {envoi(ligne k,successeur_vert(d_horiz))}
      sinon
            {réception(ligne k,prédécesseur_vert(d_horiz))
            si successeur_vert(d_horiz)≠alloc_l(k,j)
                  {envoi(ligne k,successeur_vert(d_horiz))} finsi
      }finsi
      /* envoi des morceaux de la colonne pivot horizontalement (y
compris l'élément a_kk) */
      si (alloc_c(i,k)=d_vert)
            {envoi(colonne k,successeur_horiz(d_vert))}
      sinon
            {réception(colonne k,prédécesseur_horiz(d_vert))
            si (successeur_horiz(d_vert)≠alloc_c(i,k))
                  {envoi(colonne k,successeur_horiz(d_vert))} finsi
      }finsi
      pour (i = 1 à n, et i≠k) {tâche Tki} finpour
finpour
}
```

Dans le cas de l'algorithme de Jordan sur l'anneau, pour toute fonction d'allocation, $T_a = T_{seq}/p$ en vertu du principe d'équidistribution des données (chaque ligne représente une quantité de travail équivalente). Ceci n'est pas vérifié sur la grille où une ligne n'est pas entièrement contenue dans un processeur: le travail correspondant à la boucle sur les colonnes n'est donc pas équitablement réparti.

Nous vérifions sur la figure 5.13 que pour la diagonalisation de Jordan, la courbe est du même type que celle de l'élimination de Gauss, mais de pente moins forte (la charge est quand même moins déséquilibrée).

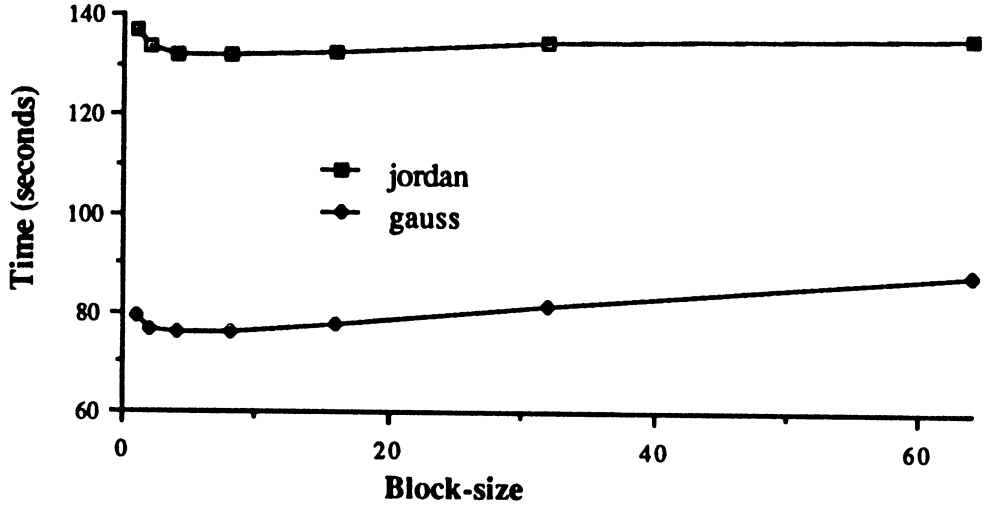


Figure 5.13: Temps des algorithmes de Gauss et Gauss-jordan sur la grille pour différentes tailles de blocs

Sur une grille, avec une version de l'algorithme de Jordan progressant par colonne, il faut utiliser une répartition qui ne conserve plus la géométrie verticale de la matrice de départ pour obtenir une répartition plus équilibrée de la charge arithmétique des processeurs (ie la colonne j n'est pas contenue dans les processeurs P_{kr} avec $0 \leq k \leq \sqrt{p}$).

La meilleure taille des blocs est r = 8 pour l'algorithme de Jordan et r=4 pour l'algorithme de Gauss avec des matrices de taille 128 à 1024 et p=16 processeurs.

Nous présentons une borne inférieure pour les communications des algorithmes de Gauss ou Jordan sur une grille de processeurs.

Lemme 5.3: Sur une grille de $p = \alpha n$ processeurs, $0<\alpha\leq 1$, $(4/\alpha)n\sqrt{n}\,\tau_c + O(n) \leq T_c$ pour les algorithmes de Gauss ou Jordan.

Preuve: Considérons le prédécesseur vertical du processeur qui contient a_{11}, appelons le mj. Il faut $(\sqrt{p}-1)\,(n/\sqrt{p})\,\tau_c = n\,\tau_c + O(1)$ unité de temps pour que la partie correspondante de la ligne 1 arrive à P_{mj} puis $(\sqrt{p}-1)\,(n/\sqrt{p})\,\tau_c = n\,\tau_c + O(1)$ unités de temps pour que les éléments des colonnes nécessaires arrivent à P_{mj} depuis son prédecesseur horizontal.

De plus P_{mj} doit recevoir au moins $n-n/\sqrt{p}-1$ parties de lignes de son propre prédécesseur vertical (chacune de taille $n/\sqrt{p}$ dans le pire des cas où les éléments sont $(i,n-n/\sqrt{p}-1)$ à (i,n)). Ceci car il ne contient que $n/\sqrt{p}$ parties de lignes dans sa mémoire locale. P_{mj} aura besoin de $n^2/\sqrt{p}\tau_c + O(n)$

unités de temps pour recevoir ces parties de lignes (n/p est négligeable devant n, pas $n/\sqrt{p}$). Il lui faudra $n^2/\sqrt{p}\tau_c + O(n)$ unités de temps comme précédemment pour les élements des colonnes correspondantes (la taille totale est toujours $n/\sqrt{p}$)

Or, le successeur vertical de P_{mj} a aussi besoin de recevoir $n-n/\sqrt{p}-1$ lignes de P_{mj} (ce ne sont , bien sur , pas nécessairement celles recues par P_{mj}), et P_{mj} passera $n2/\sqrt{p}\tau_c + O(n)$ unité de temps (dans le pire cas) à les lui envoyer. De la même manière le successeur horizontal de P_{mj} a besoin de recevoir $n-n/\sqrt{p}-1$ colonnes de P_{mj} et P_{mj} passera $n2/\sqrt{p}\tau_c + O(n)$ unité de temps à les lui envoyer ♠♥♦♣

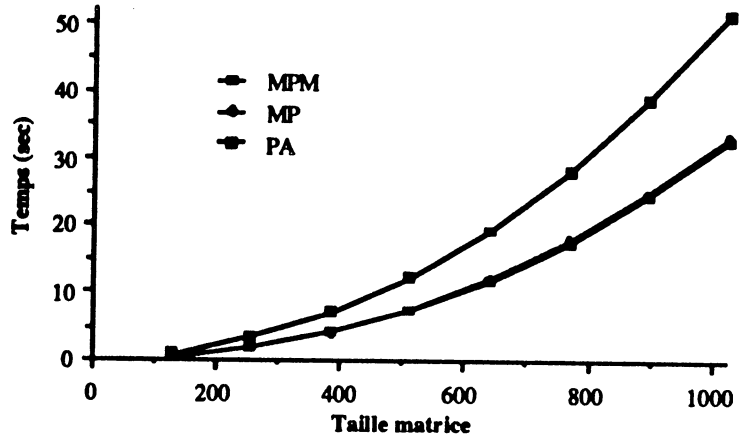
5.5. CONCLUSION

Sur l'hypercube avec l'algorithme de Gauss suivant des diffusion synchrones, la meilleure stratégie est la répartition par blocs de taille 1. Avec les diffusions asynchrones, la taille optimale des blocs dépend du chemin cyclique suivi par la répartition pour obtenir le meilleur compromis entre T_a et T_c.

Sur un anneau de $p = \alpha\ n$ processeurs, pour l'algorithme de Jordan, l'allocation par blocs consécutifs de taille n/p est optimale parmi toutes les fonctions d'allocation possibles. Pour l'algorithme de Gauss, nous connaissons le temps d'exécution pour les fonctions d'allocation par blocs consécutifs de taille r. Il est facile de calculer la taille des blocs la plus performante pour des valeurs données de α et du paramètre ρ qui caractérise la machine.

Sur une grille de $p = \alpha\ n$ processeurs, pour l'algorithme de Jordan, les fonctions d'allocation par blocs consécutifs de taille r ne produisent pas un bon équilibrage des calculs. La meilleure taille des blocs est donc comme pour l'algorithme de Gauss un compromis entre T_a et T_c.

6. Algorithmes multipivots 107



6. ALGORITHMES MULTIPIVOTS

Où nous introduisons, sur la base de l'algorithme pipeline sur un anneau, des algorithmes nouveaux utilisant plusieurs pivots à une même étape.

6.1. INTRODUCTION

Nous étudions une nouvelle approche parallèle des algorihmes de Gauss et Jordan, avec l'utilisation de plusieurs lignes pivots à une même étape de l'élimination. Les algorithmes multi-pivots et multi-pivots modifiés donnent de bons résultats de complexité sur un anneau de processeurs.

La perte de stabilité dans ces algorithmes peut être compensée par un pivotage par paires [Sor] ou un pivotage partiel local aux processeurs. L'algorithme MP a été utilisé par [Vil] pour dans les corps finis. Dans ce cas le processeur qui contient un pivot nul attend celui qu'il doit recevoir de son voisin et l'envoie. Si il n'y en a pas la matrice est de rang inférieur .

Notre étude est essentiellement algorithmique et nous ne developperons pas ce point numérique. Notre objectif est d'analyser le déroulement de l'exécution de nouveaux algorithmes plus performants sur une machine à mémoire distribuée.

Les modèles utilisés sont ceux du § 5, non asymptotiques (les temps d'initialisations apparaissent en communication et arithmétique) et la répartition est entrelacée avec des blocs de taille un.

6.2. ALGORITHME MULTIPIVOTS

6.2.1. INTRODUCTION

Lorsque l'on tente de réduire T_p, on remarque que T_a est figé dès que la répartition des données est fixée, il faut donc réduire T_c.

L'opération d'élimination consiste en la combinaison de 2 lignes de même "longueur" (nombre d'éléments non nuls à gauche) pour obtenir un zéro de plus sur l'une d'elles sans modifier l'autre. La ligne qui n'est pas modifiée

peut être n'importe quelle ligne de la matrice ayant la bonne longueur (cf § 1).

$$\begin{pmatrix} 1 & 0 \\ -\frac{a_{j1}}{a_{i1}} & 1 \end{pmatrix} \begin{pmatrix} a_{i1}\ a_{i2} \cdots a_{in} \\ a_{j1}\ a_{j2} \cdots a_{jn} \end{pmatrix} = \begin{pmatrix} a_{i1}\ a_{i2} \cdots a_{in} \\ 0\ \alpha_{j2} \cdots \alpha_{jn} \end{pmatrix}$$

Figure 6.1: Elimination de l'élément a_{j1}

A chaque étape k de l'élimination les processeurs transmettent la ligne pivot qu'ils ont précédemment reçue. La ligne qu'ils transmettent n'étant pas modifiée, ils peuvent envoyer n'importe laquelle de leur propres lignes correspondant à la "longueur" n-k de l'étape k.

Pour cette communication les processeurs n'ont pas besoin d'attendre d'avoir reçu la ligne de leur prédécesseur. Comme le modèle autorise le parallélisme interne des communications sur des canaux différents, on peut gagner du temps en effectuant l'émission et la réception simultanément.

Ligne	Proc	
1	1	*
2	2	1 *
3	3	2 2 *
4	4	3 3 3 *
5	1	4 4 4 4 *
6	2	1 5 5 5 5 *
7	3	2 2 6 6 6 6 *
8	4	3 3 3 7 7 7 7 *
9	1	4 4 4 4 8 8 8 8 *
10	2	1 5 5 5 5 9 9 9 9 *
11	3	2 2 6 6 6 6 10 10 10 10 *
12	4	3 3 3 7 7 7 7 11 11 11 11 11 *

Figure 6.2: Lignes pivots: l'entier r est en position (i,k) si a_{ik} est éliminé en utilisant les lignes i et r à l'étape k de l'algorithme multi-pivots (p=4,n=12).

Le processeur contenant la ligne k n'aura quant a lui pas besoin de recevoir de ligne pour effectuer ses éliminations car il contient la "vraie" ligne pivot de l'étape courante que l'on ne doit pas modifier. On peut toutefois considérer à l'étape k, que la ligne k est terminée et faire les éliminations avec la ligne envoyée par le voisin (pour rendre l'algorithme plus symétrique).

On remarque que cette stratégie est proche des idées du § 1 sur le réseau linéaire avec l'algorithme Glouton. Séquentiellement l'algorithme est le suivant :

```
/* Algorithme multipivots (MP) en séquentiel */
{pour  k=1 à n-1
    pour  i=n à k+1
          {éliminer a_ik avec les lignes i et ((i-k-1)mod p)+k}
    finpour
finpour
}
```

Les lignes ayant servi à l'élimination sont regroupées figure 6.2:

6.2.2. ALGORITHME PARALLELE

La parallélisation est directe, chaque processeur travaillant avec les lignes de sa mémoire locale: (PAR est l'opérateur équivalent au constructeur 'par' de CSP/OCCAM [Hoa] qui décrit le comportement PARallel des processus englobés).

Algorithme multi-pivots (MP) parallèle:

```
/* Programme du processeur P_i */
{pour k=1 à n-1
    {par
        {envoyer la ligne pivot locale (de longueur n-k) à Pi+1 mod p
        recevoir le ligne pivot locale (de longueur n-k) de Pi-1 mod p}
    finpar
    effectuer les éliminations pour toutes les lignes m > k}
finpour
}
```

Dans un certain sens l'algorithme multi-pivots est synchrone au travers du protocole de communication, son exécution est décrite figure 6.3.

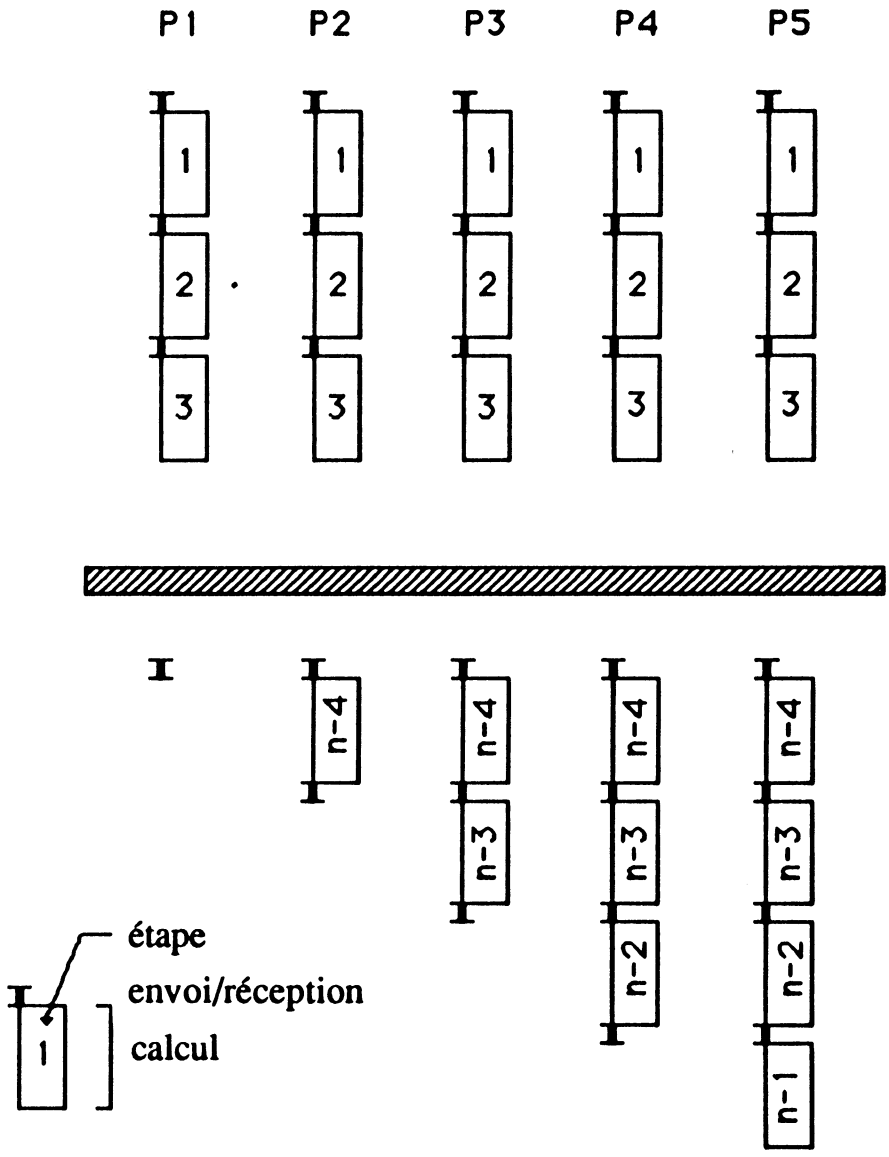


Figure 6.3: Décomposition temporelle de l'exécution de l'algorithme MP

6.2.3. TEMPS D'EXÉCUTION

Le calcul du temps d'exécution est simple: le dernier processeur P_p qui termine l'exécution effectue n-1 étapes de communication de longueur n-étape+1 et le temps de calcul est le même que celui de l'algorithme de diffusion du § 5.

$$T_{comm}^{lpr} = \sum_{j=1}^{n-1} ((n-j+1)\,\tau + \beta) = \left(\frac{n^2}{2}+\frac{n}{2}-1\right)\tau + (n-1)\,\beta$$

6.3. ALGORITHME MULTI-PIVOTS MODIFIÉ

6.3.1. INTRODUCTION

Si l'on poursuit la philosophie gloutonne précédente, on peut encore diminuer le nombre de communications en commencant le procédé d'élimination de manière interne. En effet, avant toute communication, rien n'empêche avec les n/p lignes de chaque processeur d'effectuer (n/p)(n/p-1)/2 éliminations (partie triangulaire inférieure de la sous matrice n/p x n contenue dans chaque processeur).

L'algorithme se déroule ensuite à la manière de MP mais les processeurs ne sont pas tous à la même étape du processus d'élimination au même temps. Cela implique un certain calcul pour connaitre la "longueur" de la ligne à transmettre pour que le receveur puisse l'utiliser de manière optimale (c'est à dire mettre à jour sa ligne la plus longue non terminée). Les éliminations suivantes sont alors effectuées de proche en proche avec la ligne qui vient juste d'être mise à jour car les lignes sont toutes de longueur différentes dès la fin de la première partie sans communication de l'algorithme.

```
/* Algorithme multipivot modifié (MPM) en séquentiel */
{pour  k=1 to n-1
    pour  i=n to k+1
          {éliminer a_ik avec les lignes i et fligne[i,k]}
    finpour
finpour
}
```

La table fligne(i,k) peut être pré-calculée avec l'algorithme suivant:

```
{k = 1;       /* étape de l'elimination pour avoir la long des lignes*/
fini = 0;     /*nombre de lignes termines*/
tantque ((k < n) et (fini < (n div p)))
{/*boucle sur k que l'on arrête dès la fin des mises à jours internes*/
      j = 0;{décalage du triangle initial}
      i = k;
      tantque ((i <= n) et (fini < (n div p)))
      {     si ((i mod p) = (t mod p)) /*si la ligne appartient a t*/
                  {si (i - k = 0) /*ligne pivot*/
                        {fini ++; j ++}
                  sinon
                        {si (fini < (n div p))
                              {si (i - k <= p)
                              /*la ligne est traitee par le pivot du voisin*/
                                    {fligne[i, k + j] := i - 1;
```

```
                                    j := j + 1}
                                sinon
                                /*la ligne est traitée par la précédente mise à
jour dans le processeur*/
                                    {fligne[i, k + j] := i - p;
                                    j := j + 1}
                                finsi
                            }finsi
                        }finsi
                    i := i + 1;
                    }finsi
            si (k mod p) = (t mod p) {k +=2;} /*la ligne pivot appartient a t*/
            sinon {k ++;} finsi
            } fintantque
}
```

Ligne	Proc	
1	1	*
2	2	1 *
3	3	2 2 *
4	4	3 3 3 *
5	1	**1** 4 4 4 *
6	2	**2** 2 5 5 5 *
7	3	**3** 3 3 6 6 6 *
8	4	**4** 4 4 4 7 7 7 *
9	1	**1** **5** 5 5 5 8 8 8 *
10	2	**2** **6** 6 6 6 6 9 9 9 *
11	3	**3** **7** 7 7 7 7 7 101010 *
12	4	**4** **8** 8 8 8 8 8 8 111111 *

Figure 6.4: Lignes pivots: l'entier r est placé en position (i,j) si a_{ij} a été éliminé en utilisant la ligne i et la ligne r pour p=4 et n=12 (les entiers en gras représentent la phase sans communication du début de l'algorithme).

La figure 6.4 permet de visualiser les lignes qui sont pivots pendant la progression de l'algorithme.

6.3.2. ALGORITHME PARALLELE

Dans la version parallèle de MPM, on suppose que p divise n et que le processeur P_i ($0 \leq i \leq p-1$) contient les lignes i, i+p, i+2p, ..., i+p(n/p-1) de la matrice numérotées 0, 1, 2, ..., n/p-1 L'algorithme s'écrit:

```
/* programme du processeur Pabs */
{
```

```
/* triangularisation avec les pivots locaux internes */
pour (k=0; k< n/p-1; k++)
      pour (j=k+1; j≤n/p; j++)
      {eliminer a_{i+(j-1)*p,k} de A avec les lignes j et k de P_abs} finpour
finpour
/*élimination du reste de la matrice */
k=0;
ligne1=0;   /* numéro de la première ligne non-terminée */
l_envoi=0;  /* numéro de la ligne à envoyer */
t_envoi=n-1;        /* taille de la ligne à envoyer */
t_reçois=n-1;       /* taille de la ligne à recevoir */
si abs=p-1
      {t_envoi--; l_envoi++}  finsi /* car P0 est déja à l'étape k=1 */
tantque (k<n-1)
      {r=kmodp; /* on determine quel processeur a la "vrai" ligne pivot */
      si (abs=r)
            {t_reçois--; ligne1++; k++;
            si (abs=p-1)
                  {t_envoi--; l_envoi++}finsi /* rattrappe le décalage */
            }finsi
      par
            {envoyer_à_droite(l_envoi, t_envoi);
            recevoir_de_gauche(buf, t_reçois)} finpar
      eliminer a_{ligne1,k} avec les lignes ligne1 et buf;
      pour (j=ligne1+1, decal=1; j<n/p; j++, decal++)
            {eliminer a_{jk+decal} avec les lignes j et j-1} finpour
      si ((abs =r) et (abs≠p-1))
            {t_envoi-=2; l_envoi+=2}
      sinon
            {t_envoi--; l_envoi++}
      finsi
      k++;
      t_reçois--;
      }
fintantque
}
```

L'ordonnancement des éliminations lorsque l'on utilise MPM parallèle est le suivant:

Ligne	Proc											
1	1	*										
2	2	3	*									
3	3	3	4	*								
4	4	3	4	5	*							
5	1	**1**	3	4	5	*						
6	2	**1**	3	4	5	6	*					
7	3	**1**	3	4	5	6	7	*				
8	4	**1**	3	4	5	6	7	8	*			
9	1	**1**	**2**	3	4	5	6	7	8	*		
10	2	**1**	**2**	3	4	5	6	7	8	9	*	
11	3	**1**	**2**	3	4	5	6	7	8	9	10	*
12	4	**1**	**2**	3	4	5	6	7	8	9	10	11 *

Figure 6.5: Etapes: l'entier r est en position (i,j) si l'élément a_{ij} est éliminé à la "l'étape relative" r (locale à chaque processeur) en utilisant l'algorithme multi-pivots modifié pour p=4 et n=12 (en gras la triangularisation interne initiale).

Les "étapes relatives" (1 à n/p-1) sont sans communication donc moins longues que les suivantes et leur durée décroit (triangularisation). Il reste donc n-1-(n/p-1) étapes avec communication de lignes pivots de longueurs différentes suivant les processeurs.

La longueur des lignes transmisent diminue au cours de l'algorithme (par pas de 1 ou de 2 suivant l'étape).

Le schéma d'exécution est simple:

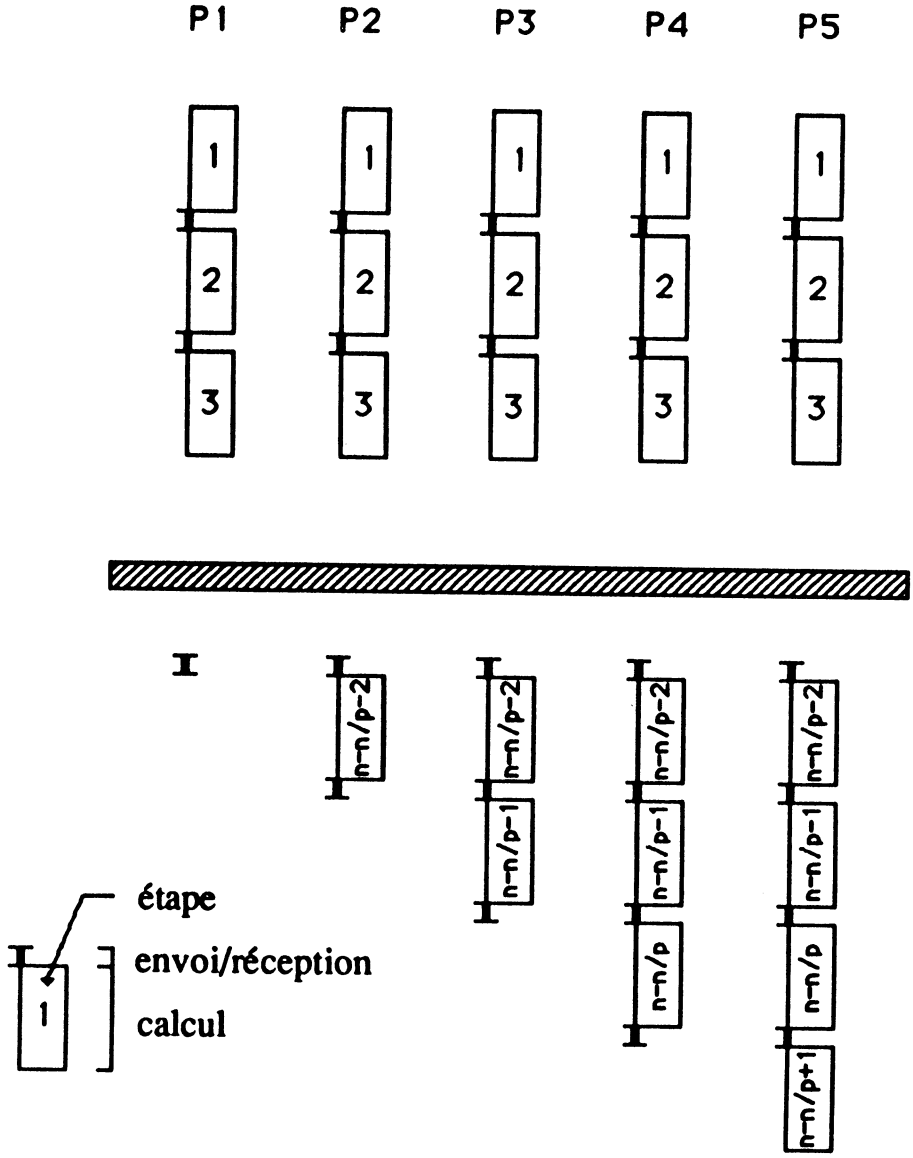


Figure 6.6: Schéma d'exécution de l'algorithme MPM

6.3.3. TEMPS D'EXÉCUTION

Le temps de calcul reste le même que pour MP mais le temps de communication est réduit à n-n/p-1 étapes. Les n/p étapes retranchées correspondent aux étapes de numéros égaux aux "longueurs" finales des lignes contenues dans P_p. Le temps de communication dépend donc de la répartition des lignes dans les processeurs. Pour minimiser les communications on songe à une distribution optimale où P_p contient les lignes 1, 2, 3, , n/p-1, et n ce qui maximise la somme retranchée. P_p termine quand même l'algorithme car il contient la ligne n. Malheureusement cela introduit des temps morts où le processeur attend les données de ses voisins, donc pas de gain. De plus le calcul est alors très

déséquilibré car P_p ne met plus à jour qu'une seule ligne à partir de l'étape n/p. Nous obtenons, avec la répartition entrelacée classique:

$$T_{comm}^{lpm} = \sum_{j=1}^{n-1} ((n-j+1)\,\tau + \beta) - \sum_{q=1}^{\frac{n}{p}-1} ((n-qp+1)\,\tau + \beta)$$

$$T_{comm}^{lpm} = \left(\left(\frac{n^2}{2}+n\right)\left(1-\frac{1}{p}\right)\right)\tau + \left(n\left(1-\frac{1}{p}\right)\right)\beta$$

Une dernière réduction des communications peut être obtenue en envoyant à chaque étape plusieurs ou toutes les lignes de longueur inférieure ou égale à la longueur courante.

En transmettant le maximum de lignes à chaque fois (n/p au début), on effectue autant d'étapes que de lignes transmisent avec une seule communication. Cela réduit à p le nombre d'initialisations pour le même volume de données.

6.4. COMPARISON DES ALGORITHMES

6.4.1. EN COMPLEXITÉ

Prenons un anneau de taille p fixée, p<<n et la répartition entrelacée par bloc de taille un, nous avons:

$T_a = 2n^3\tau_a/(3p) + O(n^2)$ pour tous les algorithmes

et les valeurs suivantes pour les communications:

Algorithme	$T_c \approx$
pipeline (§ 5)	$3n^2\tau_c/2 + 3n\beta$
MP	$n^2\tau_c/2 + n\beta$
MPM	$(1-1/p)n^2\tau_c/2 + (1-1/p)n\beta$

Table 6.1: termes de plus haut degré du temps de communication

Les algorithmes MP et MPM présentent des caractéristiques intéressantes pour les communications (proche de celles du réseau complet), qui ne demandent que la connectivité d'un anneau.

6.4.2. COMPARAISON EXPÉRIMENTALE

Pour l'algorithme MP, envois et réceptions peuvent être exécutés en parallèle comme supposé dans le modèle car le microprocesseur qui gère les communications le permet, si les liens utilisés sont différents. De plus les

lignes sont différentes et indépendantes (il n'y a pas de relation de précédence entre les 2 actions de communications).

Pour éviter une situation de blockage si l'opérateur "par" n'existe pas, il suffit par exemple d'effectuer envoi puis reception sur les processeurs d'indices pairs pour la numérotation de l'anneau et reception puis envoi sur les impairs.

La méthode est la même pour l'algorithme MPM, où la programmation est plus délicate car le processeur à l'extrémité de l'anneau doit initier le pipeline tous les p étapes avec une ligne de longueur de 2 unités inférieure à la précédente. Les résultats sont les suivants, en prenant pour base de comparaison l'agorithme pipeline (§ 5):

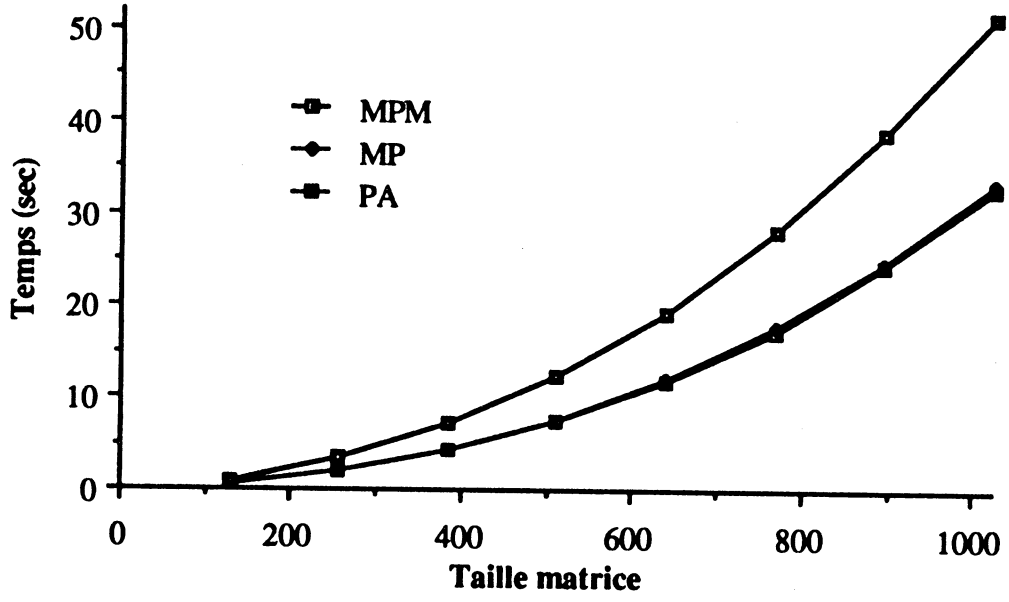


Figure 6.7: Elimination de Gauss, algorithmes MP, MPM et PA

Les deux algorithmes multipivots ont des temps d'exécution très inférieur à celui de l'algorithme pipeline classique sur l'anneau. La pratique confirme l'analyse de complexité, l'algorithme MPM est légèrement plus performant que MP et l'écart augmente avec la taille de la matrice lorsque le temps de contrôle additionnel devient négligeable.

6.5. EXTENSION DES MÉTHODES À LA DIAGONALISATION DE JORDAN

L'algorithme pipeline est facilement adaptable pour résoudre la diagonalisation de Jordan, les mises à jour sont simplement effectuées sur toutes les lignes. Le temps de calcul devient:

$$T_a = (n^3/p)\tau_a + O(n^2).$$

La différence dans le temps de communication est la suivante:
-à chaque pas la ligne pivot est envoyée à tous les processeurs
-la dernière ligne est aussi transmise

Les algorithmes deviennent:

DA Jordan:
```
/* processeur Pi */
pour k=1 to n
    si ligne k dans Pi
        diffuser la ligne k à
        P1,...,Pi-1,Pi+1,...,Pp
    sinon recevoir la ligne pivot k
    finsi
    mettre à jour les lignes de A en
    utilisant la ligne k
finpour
```

PA Jordan:
```
/* processeur Pi */
    pour k=1 to n
    si i=k mod p
        envoyer la ligne k à Pi+1
    sinon recevoir la ligne k de Pi-1
            si i≠(k mod p)-1
                envoyer ligne k à Pi+1
            finsi
finsi
    mettre à jour les lignes de A en
    utilisant la ligne k
finpour
```

Temps de communication sur l'anneau, méthode pipeline:

$$T_c \approx 3n^2\tau_c/2 + 3n\beta$$

L'algorithme MPM peut aussi être adapté à la diagonalisation de Jordan, les éliminations sont effectuées sur toutes les lignes. Les communications sont différentes dans les dernières p étapes où l'algorithme devient du type PA avec la répartition entrelacée des données.

Dans la figure suivante, nous présentons pour n=9 et p=3 la date d'élimination de chacun des éléments. Si l'élément (i,j) est annulé à la date r le nombre r est placé en position j sur la ligne i. Remarque: P_p est le premier à terminer pour pouvoir pipeliner sa dernière ligne.

Processeur 1	Processeur 2	Processeur 3
X 1 1 2 3 4 5 7 8	1 X 1 2 3 4 5 6 9	1 1 X 2 3 4 5 6 7
1 2 1 X 3 4 5 7 8	1 1 3 2 X 4 5 6 9	2 1 1 4 3 X 5 6 7
1 1 3 2 4 5 X 7 8	2 1 1 4 3 5 6 X 9	1 3 1 2 5 4 6 7 X

Le temps de communication est alors obtenu par la formule suivante:

$$T_c \approx (1-1/p)n^2\tau_c/2 + (1-1/p)n\beta$$

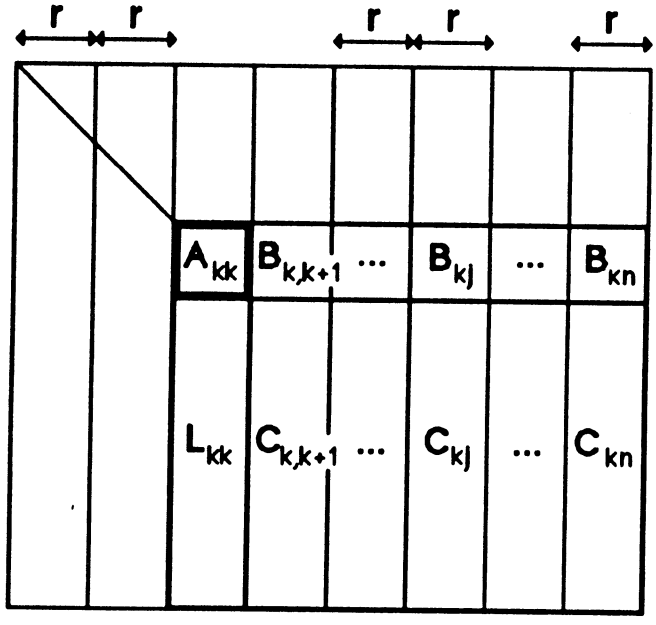
Ces résultats sont détaillés dans [CT].

6.6. CONCLUSION

Avec les formules précédentes nous pouvons dire que la diffusion n'est pas beaucoup plus rapide que le pipeline multipivot, même sur le réseau

complet. Pour les problèmes "bien conditionnés" il peut être intéressant d'utiliser cette architecture simple avec les algorithmes performants présentés.

7. Performances 123



7. PERFORMANCES

Où nous expliquons l'implémentation des méthodes précédantes. Dans le but d'obtenir des performances élevées nous introduisons la version par blocs de l'algorithme pour utiliser au maximum les unités vectorielles.

7.1. INTRODUCTION

Dans ce chapitre nous présentons l'élimination de Gauss sur un multiprocesseur vectoriel de topologie hypercube: le FPS T20 (§ 3). Nous introduisons ensuite les méthodes par blocs et nous les comparons au algorithmes précédents pour expliquer la supériorité de cette approche.

Nous utilisons l'hypercube avec seulement la topologie d'anneau car le système fournit une fonction de diffusion pipeline rapide (assembleur) sur cette topologie.

Les versions par blocs des algorithmes matriciels présentent deux avantages. Elles permettent de tirer le meilleur parti du calcul vectoriel, avec la ré-utilisation des registres vectoriels sans les décharger en mémoire. Elles permettent la communication de plus grands paquets de données et ainsi diminuent la part du temps due aux initialisations dans les communications (paramètre critique pour la plupart des hypercubes).

Si l'on utilise le plus haut niveau de programmation des unités vectorielles, la programmation est facile, similaire aux benchmarks de LINPACK pour les hypercubes [Mol] [Rat] où le SAXPY est remplacé par un appel à la fonction VSMA (Vector Scalar Multiply and Add).

Comme les auteurs [Cap] [Cha] [Gei] [GHe], nous étudions les versions par colonnes des algorithmes pour se réserver la possibilité d'inclusion du pivotage partiel. Comme nous ne nous intéressons pas ici à la résolution du sytème linéaire les versions par lignes ou par colonnes sont équivalentes en complexité.

La stratégie couramment utilisée [Gei][Saa2][LCo] dans la littérature est la stratégie d'entrelacement (par blocs de taille 1), nous l'utiliserons aussi pour pouvoir situer nos résultats.

7.2. ALGORITHME GAUSS PAR POINTS

7.2.1. IMPLÉMENTATION

Nous allons étudier l'augmentation des performances lorsque l'on utilise les différents niveaux de programmation pour les unités vectorielles (§ 3).

Les routines génériques nécessitent l'alignement des vecteurs sur les sources des registres pour donner de bonnes performances. Qui plus est cette condition est nécessaire aux "parameter blocs" pour fonctionner. Si les données sont bien rangées en mémoire, cela évite les manipulations logicielles de recopie des vecteurs dans des zones alignées de la mémoire, réservées au système. Le temps de chargement et déchargement des opérandes dans les appels aux procédures vectorielles est ainsi beaucoup diminué.

Lors de la réservation de la place en mémoire pour les vecteurs de la matrice, les vecteurs sont rangés de manière consécutive. Ensuite durant le processus d'élimination la longueur des vecteurs varie (de n à 1). Cette valeur n-k à l'étape k n'étant pas souvent un multiple de la longueur des registres (condition nécessaire pour l'obtention de bonnes performances sur les machines vectorielles (cf § 3) [CRRS], il faut essayer de diminuer les opérations sur des registres non pleins. Lorsque les vecteurs commencent et finissent au milieu des tranches de la mémoire on a 2 parties de tailles inférieures à la taille des registres (une au début l'autre à la fin figure 7.1).

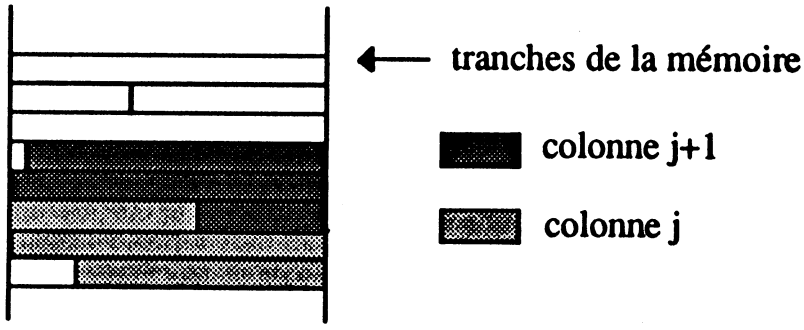


Figure 7.1: Mauvais rangement des données en mémoire, les colonnes sont non-alignées

L'emplacement du premier élément du vecteur varie pendant l'élimination ce qui est préjudiciable à l'alignement.

Pour améliorer le rangement en mémoire des opérandes, nous avons choisi de stocker les vecteurs de la matrice à l'envers, ainsi le dernier élément du vecteur (qui est le premier en mémoire) reste toujours aligné sur la source zéro dans les registres (figure 7.3). Aucun décalage n'est nécessaire pour aligner les vecteurs pendant l'élimination et une seule

partie, la dernière, n'est pas de la taille des registres mais est quand même alignée sur le début de la tranche.

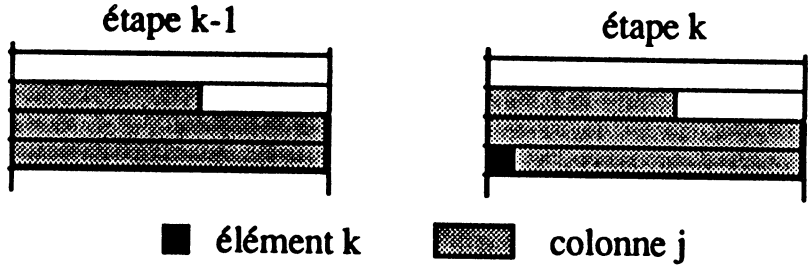


Figure 7.2: Mauvais rangement des données en mémoire, l'élément éliminé modifie l'alignement

Pour assurer que tous les vecteurs composant la matrice commencent bien sur la source 0, nous avons pris comme dimension horizontale de la matrice le multiple de 128 supérieur ou égal à la longueur des lignes (stride), quitte à perdre un peu de place à la fin de chaque vecteur (en blanc sur la figure).

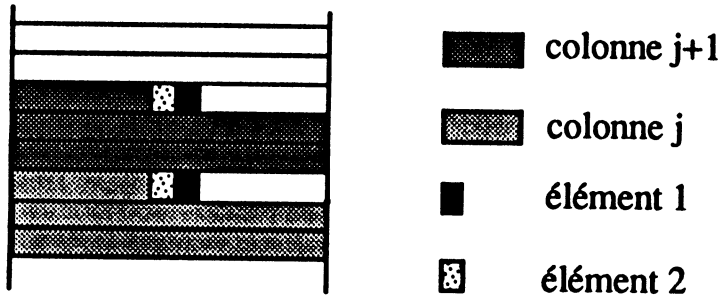


Figure 7.3: Bon rangement des données en mémoire, les éléments sont en ordre inverse, la colonne reste alignée.

La seule difficulté introduite par ce stockage inversé provient du fait que l'élément a_{ij} est en position (n-i+1,j).

La dernière optimisation de la gestion mémoire provient de l'utilisation de 2 bancs mémoire différents en entrée des opérateurs vectoriels. La colonne pivot est stockée dans le banc opposé à celui où se trouve la matrice pour augmenter la performance de l'accès mémoire des registres.

7.2.2. RÉSULTATS

Nous vérifions tout d'abord les résultats des analyses du § 5. La figure suivante donne les performances des versions pipeline et diffusion par paquets de taille égale sur le FPS T20 configuré en anneau de 16 processeurs. Comme prévu par l'étude théorique ci dessus, l'algorithme de diffusion par paquets sur l'anneau est légèrement meilleur que ceui du

pipeline. Pour une matrice 1024x1024 en utilisant la programmation vectorielle de haut niveau (singlenode), nous obtenons 12.5 Mflops ce qui est un bon résultat compte tenu de la facilité l'implémentation.

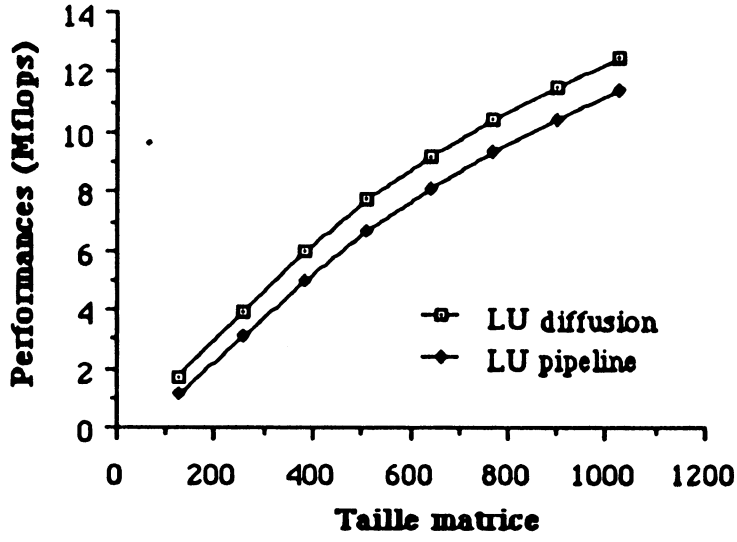


Figure 7.4: Elimination de Gauss (procédures singlenodes,16 processeurs)

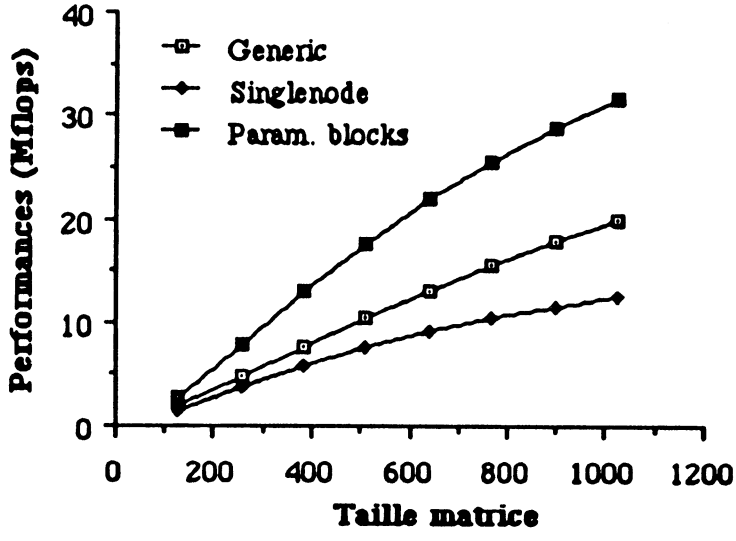


Figure 7.5: Elimination de Gauss (algorithme de diffusion, 16 processeurs)

En prenant comme base la programmation avec les routines de plus haut niveau (singlenode) l'augmentation des performances est de 36 % pour une matrice de taille 512 et de 59 % pour une matrice de taille 1024 (figure 7.5) avec les routines génériques. En utilisant les parameter bloc nous obtenons

des augmentations de 128 % et de 154 % dans les mêmes cas. Ce qui comfirme le coût logiciel des routines de haut niveau.

Le meilleur résultat obtenu (avec le niveau paramater block) est 32 Mflops pour l'élimination de Gauss d'une matrice 1000x1000. A titre comparatif, citons les résultats de [CRRS] sur IBM 3090 avec 1 processeur vectoriel: 22 Mflops pour un exemple de même taille et l'algorithme par points.

Pour étudier l'influence de l'augmentation du nombre de processeur sur les performances, nous avons envoyé nos programmes aux USA et obtenu de nos collègues les résultats pour 32 et 64 processeurs (hypercube de dimension 5 et 6). Le meilleur résultat atteint les 40 Mflops avec une matrice 1000x1000, que l'on peut comparer [CRRS] avec les 70 Mflops de la version assembleur sur IBM 3090 monoprocesseur.

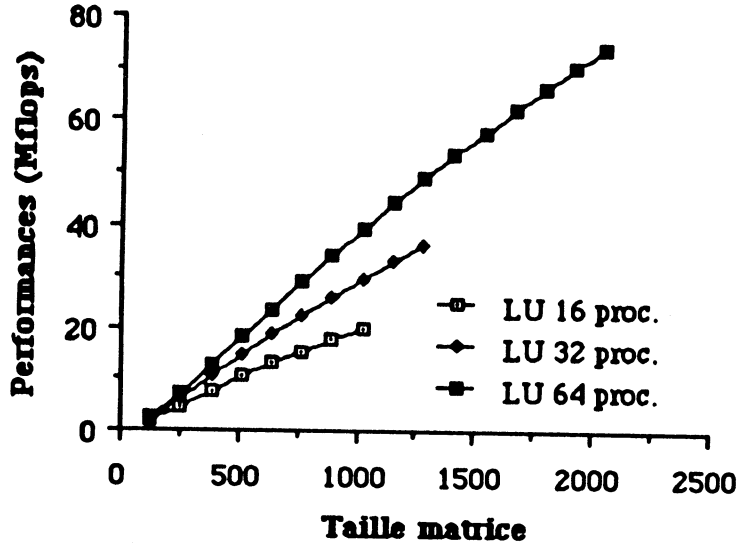


Figure 7.6: Elimination de Gauss (fonctions generics, algorithme de diffusion)

Pour Householder nous avons fait les mêmes expériences et les résultats sont reportés dans la figure 7.7

Les performances sont meilleures pour QR car la part du calcul sur le temps total est deux fois plus importante que pour LU alors que les communications qui représentent le goulot d'étranglement sur cette machine sont du même ordre.

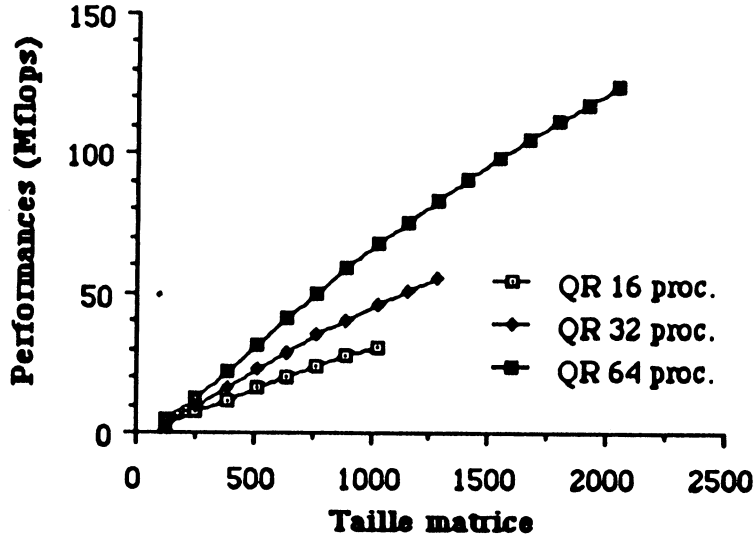


Figure 7.7: Décomposition QR (fonctions generics, algorithme de diffusion)

7.3. ALGORITHME DE GAUSS PAR BLOCS

7.3.1. INTRODUCTION

Nous allons présenter dans la suite la technique d'élimination par blocs pour améliorer les performances de la classe d'algorithmes que nous étudions (Gauss) sur les processeurs vectoriels.

Sur les machines multiprocesseurs vectoriels à mémoire partagée, l'amélioration des résultats est bien connue [RSg] par exemple. Pour les machines à mémoires distribuées, nous montrons que ces techniques améliorent aussi les performances des communications et nous donnons la taille optimum des blocs pour tirer le meilleur partie de l'hypercube FPS T20.

Sur les machines vectorielles à mémoire partagée, avec un système de mémoire hiérarchisée, il est bien connu qu'il faut réécrire les algorithmes en termes d'opérations matrice-matrice (noyaux BLAS 3) pour avoir de bonnes performances. Ceci est obligatoire pour l'utilisation maximum des mémoires caches et des registres vectoriels [BGWJ][Cal][DDDH][DSo] [GJMS][RRS][RSg]. Nous montrons que l'utilisation de ces noyaux produit de bonnes performances sur une machine multiprocesseur vectoriel à mémoire distribuée.

Le passage aux algorithmes par blocs réduit à la fois le temps de calcul arithmétique (par une meilleure utilisation des unités vectorielles) et le temps de communication (plus gros messages). En algébre linéaire la

programmation par blocs permet de manipuler des noyaux de niveau plus élevé, par exemple des opérations matrices-matrices et de simplifier la programmation.

7.3.2. DESCRIPTION DES ALGORITHMES

La répartition des données est du type entrelacé et le rangement des données en mémoire est optimisé comme pour l'algorithme par points. Pour simplifier, nous prendrons comme dans [GJMS] l'élimination de Gauss avec inversion des blocs diagonaux.

Cet algorithme est stable pour les matrices à diagonale dominante [DSo]. Nous implémentons la version par colonnes de l'algorithme pour faciliter une éventuelle implentation du pivotage partiel dans les blocs de colonnes [DSo]. Cette procédure serait exécutée de manière interne par chaque processeur, sans ajouter de communications. La raison est la suivante: si un processeur contient la colonne entière il peut effectuer dans la tâche T_{kk} la recherche du pivot et la multiplication de la colonne pour LU ou le calcul de la norme pour QR, sans faire aucune communication. Dans le cas de l'allocation par ligne, la recherche du pivot demande des communications entre tous les processeurs.

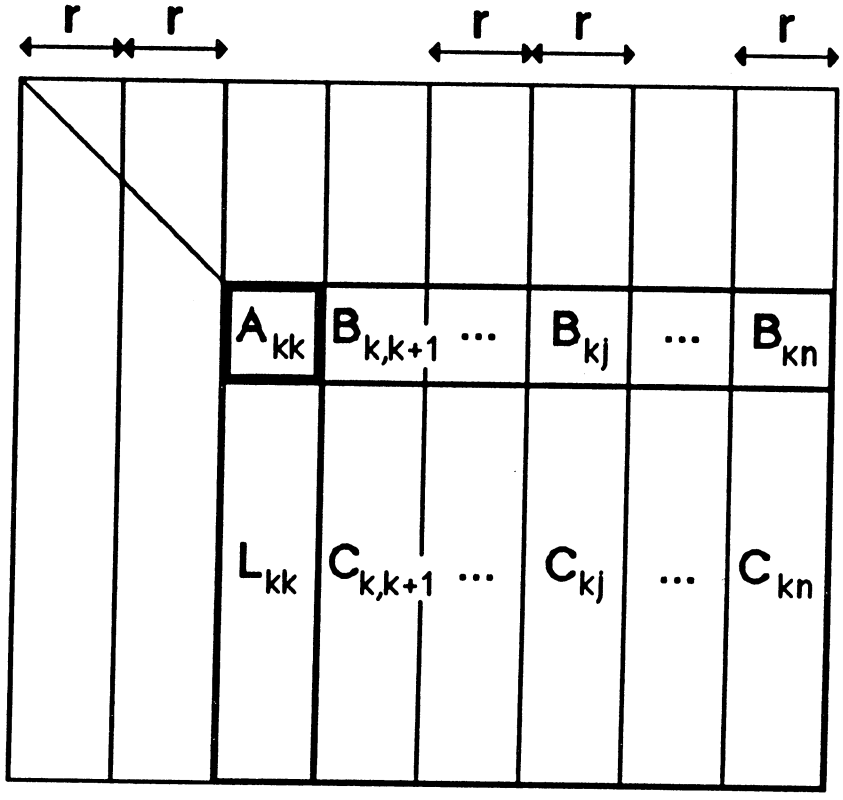


Figure 7.8: Elimination de Gauss par blocs de taille r (matrice de taille nr)

La matrice de départ est divisée en blocs de colonnes consécutives de taille r. Ensuite l'algorithme s'exécute comme dans sa version par points, mais les opérations ont lieu sur des matrices rxn et l'élément de base est la matrice rxr. Cela revient à faire les opérations sur r colonnes à la suite.

Algorithme de diffusion, élimination par blocs de taille r:

```
/* Programme du processeur Pi */
pour k=1 à n
    {si (bloc Ck dans Pi )
        /* calcul de l'inverse de la matrice rxr "pivot" */
        {Akk ← Akk-1 ;                          (1)
        /* modification correspondante du bloc colonne pivot */
        Lkk ← - Ckk Akk-1   ;                   (2)
        /* communication globale */
        diffuser le bloc de r colonnes Lkk à P1, .., Pi-1, Pi+1, .., Pp}(3)
    sinon
        {recevoir le bloc de r colonnes Lkk}
    finsi
/* effectuer les éliminations sur les blocs contenus dans le processeur */
    pour j := k+1 à n
        {Ckj ← Ckj + Lkk Bkj }                  (4)
    }finpour
finpour
```

La partie (1), inversion d'une matrice de taille r petit, est un calcul avec des vecteurs de longueur réduite qui sera effectué en mode scalaire sur le Transputer. Par contre (2) et (4) sont des opérations sur des vecteurs longs (de taille n), ils seront donc effectués à l'aide des unités vectorielles.

L'opération de base est un enchainement de r opérations SAXPY qui mettent à jour la variable de départ, par exemple pour (4):

$$c_j \leftarrow c_j + b_1 l_1 + b_2 l_2 + ... + b_r l_r$$

Ce type d'opération permet la ré-utilisation des données chargées dans les registres sans déchargements, donc un gain de temps appréciable pour l'opération complète.

7.3.3. ENCHAINEMENT DE R OPÉRATIONS SAXPY

L'opération $Z \leftarrow Y + aX$ avec X,Y et Z des vecteurs et a un scalaire est une instruction qui enchaine les 2 opérateurs vectoriels * puis +, qui sont habituellement disponibles sur les machines de ce type. C'est en général avec le produit scalaire l'opération la plus performante (2 opérations par cycle lorsque le pipeline est chargé).

Au plus bas niveau de programmation de cette machine (parameter bloc) on trouve une contrainte matérielle: les connexions des deux unités de calcul

pipelineet des registres n'autorisent qu'un seul chemin de données, le vecteur X doit être stocké dans le banc B et le vecteur Y dans le banc A (§ 3).

L'enchainement de r opérations SAXPY est utilisé à deux niveaux dans l'algorithme (la matrice initiale est stockée dans col[n]):

-à l'étape (2) pour la multiplication par la matrice inverse rangée dans les vecteurs a_i ($1 \le i \le r$). Le résultat (matrice L_{kk}) est stocké dans une variable buf[r] qui sera utilisée pour la diffusion et la mise à jour des autres blocs.

buf $\leftarrow a_1$ col[1] + a_2 col[2] + ... + a_r col[r] sur chaque ligne d'indice supérieur à k

-à l'étape (4) pour la mise à jour des blocs contenus dans le processeur. Les coefficients de B_{kj} sont calculés dans les vecteurs b_i ($1 \le i \le r$) et la matrice L_{kk} a été reçu dans buf.

col $\leftarrow$ col + b_1 buf[1] + b_2 buf[2] + ... + b_r buf[r] sur chaque ligne d'indice supérieur à k

Le banc B de chaque processeur est trois fois plus grand que le banc A (cf § 3), les blocs de colonnes (col[]) de la matrice initiale doivent être rangés dans le banc B. L'instruction enchainée ne pourra être utilisée que si la variable buf est aussi dans le banc B.

Pour passer outre la restriction du chemin de donnée dans l'unité vectorielle, nous ré-écrivons l'enchainement des r opérations SAXPY de la manière suivante:

col $\leftarrow$ col + b_1 buf[1] + b_2 buf[2] + ... + b_r buf[r]

col $\leftarrow$ col + (...((0 + b_1 buf[1]) + b_2 buf[2]) + ... + b_r buf[r])

où la première opération est initialisée avec un vecteur nul dans le banc A. A la fin des opérations SAXPY, une simple addition du résultat obtenu (dans le banc A) et de la colonne considérée (du banc B) fourni le résultat cherché à sa juste place (banc B).

L'enchainement des r SAXPY permet la ré-utilisation des valeurs contenues dans le registre du banc A, sans aucune écriture, ni lecture en mémoire entre la première addition (avec 0) et la dernière (avec col). La somme partielle reste valide dans le registre du banc A car la première addition s'effectue avec 0. La programmation de cette méthode est décrite dans [RTo2].

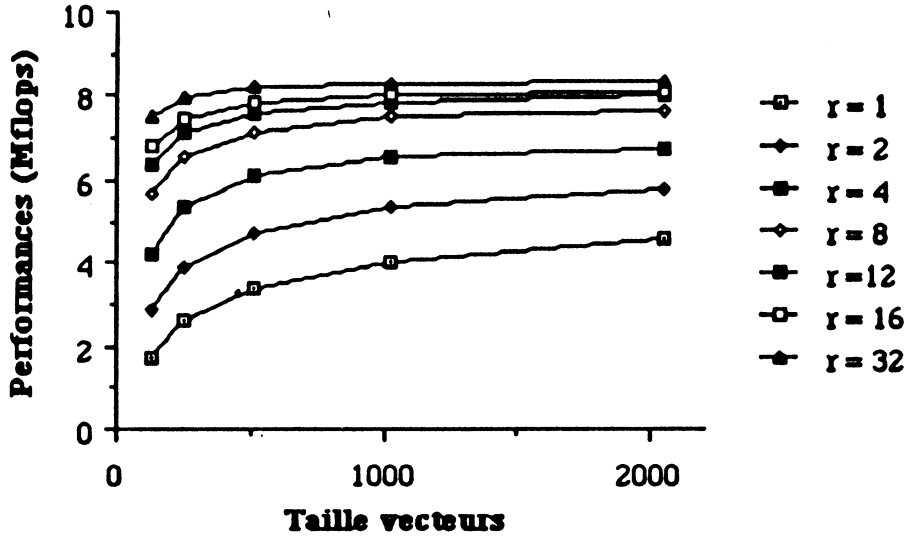


Figure 7.9: Performances de l'enchainement des r opérations SAXPY sur un noeud de l'hypercube FPS T20

Les performances de l'enchainement de r d'opérations SAXPY sont décrites figure 7.9. Pour $r \geq 8$, r opérations SAXPY vont deux fois plus vite que l'opération simple, cela représente 90% de l'optimum des opérateurs de la machine pour des vecteurs de taille 512 (la longueur moyenne dans la cas d'une matrice 1024x1024). Cela démontre l'efficacité de la programmation par blocs des unités vectorielles.

7.3.4. DIMINUTION DU TEMPS DE COMMUNICATION

Le volume total des communications est exactement le même que pour les méthodes par point. Deux solutions s'offrent pour faire décroitre le temps de diffusion de r vecteurs de taille m.

Les vecteurs sont stockés dans les variables buf[i] ($1 \leq i \leq r$), comme ces vecteurs ne sont pas consécutifs, on peut tenter de minimiser la longueur des messages, en effectuant r diffusions de taille exacte m

Par contre, si l'on veut minimiser le nombre d'initialisations des communications, on enverra plutot un grand message de taille (m+s)r, avec m+s le plus proche multiple de la taille des registres supérieur à L. On conserve ainsi l'alignement des vecteurs en mémoire (m+s) est la "stride").

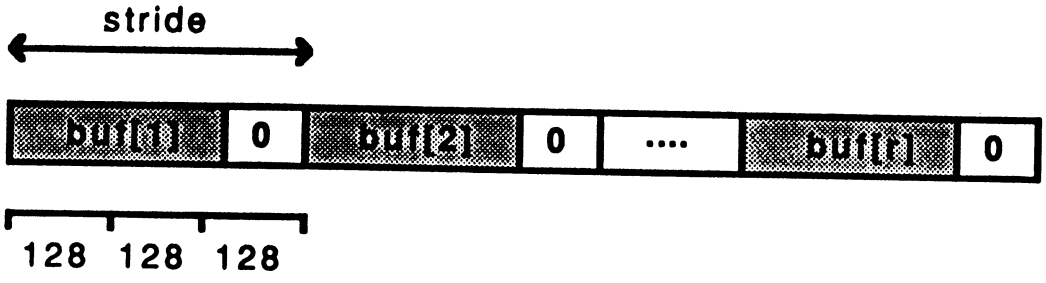


Figure 7.10: Diffusion d'un seul paquet, 128 est la taille des registres

7.3.5. RÉSULTATS

La comparaison des stratégies de diffusion est effectuée pour les valeurs de r = 1, 2 et 3 (figure 7.11 et 7.12). La stratégie de communication d'un seul grand message est la meilleure car comme nous l'avons vu en § 3 le temps d'initialisation est très important avec cette version du système. L'augmentation de la longueur du message ne compense pas le gain sur la durée d'initialisation des canaux.

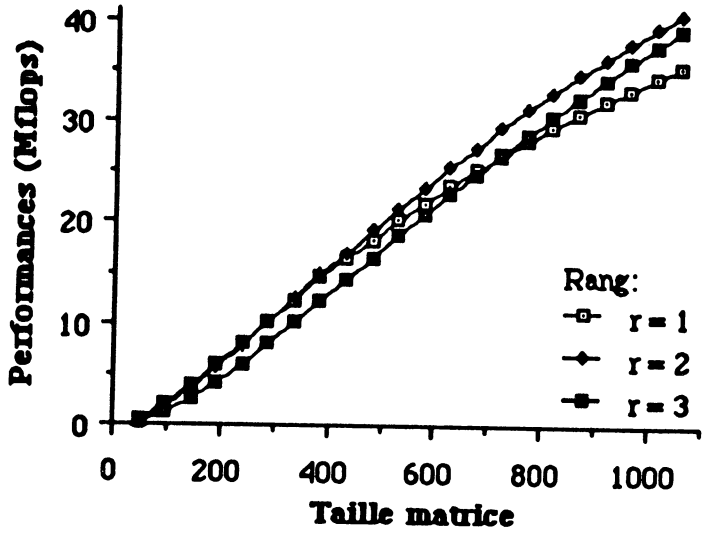


Figure 7.11: Performance de l'élimination de Gauss, stratégie de la longueur minimum, blocs de taille r (16 processeurs)

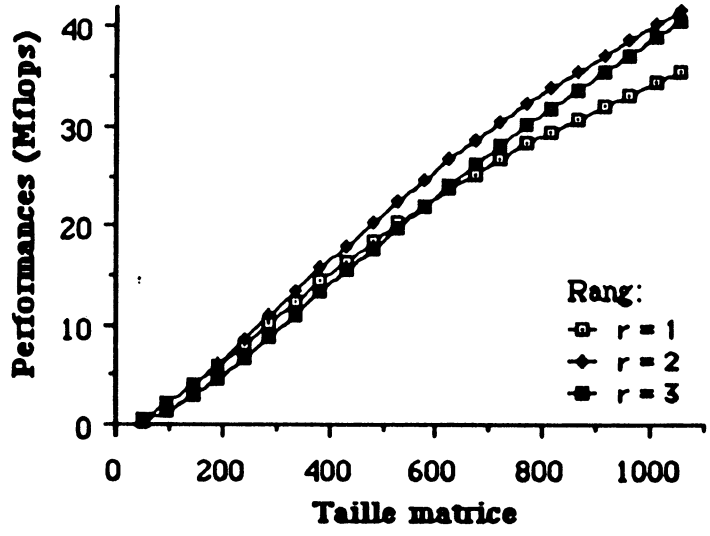


Figure 7.12: Performance de l'élimination de Gauss, stratégie du temps d'initialisation minimum, blocs de taille r (16 processeurs)

On remarque sur les courbes des figures 7.11 et 7.12 que les versions avec des blocs de taille 2 obtiennent de meilleurs résultats que celles avec des blocs de taille 3. Ceci est très surprenant, compte tenu du fait que l'enchainement des r SAXPY augmente en performance avec r (figure 7.9) et que le temps des communication diminue avec r lors de la stratégie de minimisation des temps d'initialisation. En fait nous ne reportons pas les résultats avec r>3 car ils sont désastreux !! (inférieurs aux valeurs pour r=1).

La raison est simple, la seule partie de calcul scalaire (1) consiste en l'inversion d'une matrice de taille r. La quantité de travail sur le Transputer T414 augmente donc en r^3 et ce microprocesseur est très lent pour le calcul réel car il n'a pas d'unité flottante.

Ainsi dès que r≥5 l'inversion de la matrice 5x5 prend plus de temps que la mise à jour de la matrice 1024x64 de chaque processeur!! L'unité vectorielle effectuant elle aussi assez lentement les multiplications (cf § 3) pour des vecteurs de taille un (!), nous avons simulé une machine possédant une unité scalaire flottante raisonnablement performante.

En effet la dernière génération du Transputer (T800) effectue des calculs scalaires flottants à la vitesse de 1,1 Mflops, on est loin des 0,009 Mflops du modèle T414 (cf § 3).

Nous avons remplacé l'inversion du bloc A_{kk} par r^3 assignements, la meilleure valeur est alors obtenue pour des blocs de taille 4 (figure 7.14). Cela donne le meilleur compromis entre l'efficacité du calcul (qui augmente avec r), le temps des diffusions (qui diminue avec r) et la répartition du travail sur les processeurs (qui diminue avec r car tous les processeurs attendent celui qui effectue l'inversion de la matrice (1) et la multiplication

avec le bloc pivot (2) et aussi car plus les blocs sont grands plus grand est le décalage entre les processeurs qui finissent l'algorithme cf § 5).

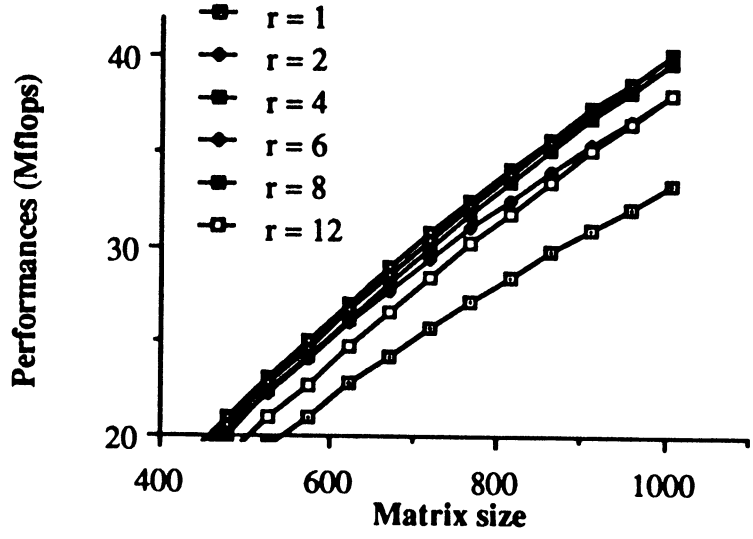


Figure 7.14: Performances de l'algorithme de Gauss par blocs de taille r, avec simulation d'une unité scalaire flottante (16 processeurs)

Le meilleur résultat avec des blocs de taille r=4 est 42 Mflops pour une matrice 1000x1000 avec 16 processeurs.

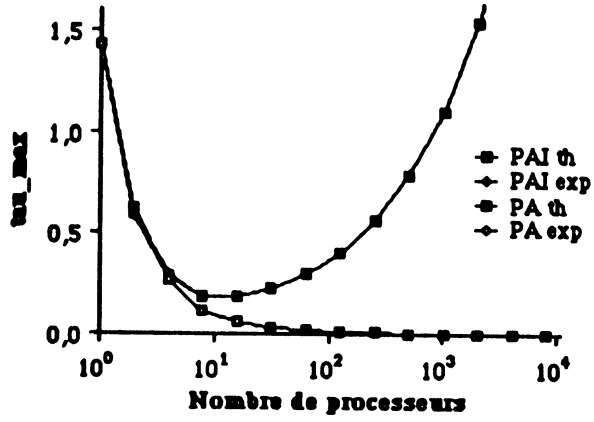
7.4. CONCLUSION

Nous avons montré que les performances d'une machine à mémoire distribuée (42 Mflops pour l'élimination de Gauss) peuvent être importantes (2 fois supérieures à celles de [LCo] sur Ipsc) et même rivaliser avec les supercalculateurs monoprocesseurs.

Les techniques pour obtenir ces exécutions performantes demandent, comme pour les calculateurs à mémoire partagée, une bonne gestion de la mémoire mais en plus une optimisation des communications. Par contre par rapport aux multiprocesseurs à mémoire partagée, l'efficacité des versions par blocs est limitée par le compromis entre répartition de la charge totale, communications et vitesse de calcul.

La comparaison avec les supercalculateurs multiprocesseurs à mémoire partagée demanderai beaucoup de processeurs: l'IBM 3090 atteint 270 Mflops sur une matrice 1000x1000 avec 6 processeurs vectoriels et des blocs de taille r=24 !

8. Accélération sur les machines à mémoire distribuée 139



8. ACCÉLÉRATION SUR LES MACHINES À MÉMOIRE DISTRIBUÉE

Où nous montrons que l'accélération pour les machines à mémoire distribuée doit tenir compte de tout l'apport matériel de l'augmentation du nombre de processeurs. Nous donnons une méthode efficace pour calculer cette accélération.

8.1. INTRODUCTION

La manière habituelle pour déterminer l'accélération consiste à prendre un problème de taille fixée et de calculer le rapport du temps d'exécution sur un seul processeur et sur plusieurs processeurs. La loi d'Amdahl [Amd] prédit que ce rapport est borné par l'inverse du pourcentage de calcul séquentiel du programme.

Supposons $T_{seq}=T_s+T_p$ (T_{seq} est la durée d'exécution séquentielle du programme, T_s la partie intrinsèquement séquentielle (dépendante dans le graphe de précédence des tâches), T_p la partie parallélisable). On a alors la durée d'exécution du programme en parallèle $T_{par} \geq T_s+T_p/p$ et l'accélération définie, à taille fixe, comme T_{seq}/T_{par}. Si l'on normalise $T_{seq}=T_s+T_p=1$, l'accélération $1/(T_s+T_p/p)$ tend vers $1/T_s$ lorsque p tend vers l'infini. Ce pourcentage n'étant jamais nul, cela démontre l'inutilité du parallélisme massif (!!).

Gustafson [Gus1][Gus2] a proposé récemment une nouvelle manière de calculer l'accélération. Pour chaque nombre de processeurs on considère la plus grande instance du problème qui peut être résolue sur la machine et on détermine le temps d'exécution nécessaire pour le même problème sur un seul processeur. Le rapport des temps obtenus est l'accélération.

Nous étudions ce point de vue et donnons une manière simple de calculer cette accélération, à partir des performances ou à partir de l'analyse de complexité de l'algorithme. Ces méthodes sont appliquées aux exemples précédents du § 7.

8.2. MODELE ET ALGORITHMES ÉTUDIÉS

Dans sa description, Gustafson a supposé que la partie séquentielle du programme est indépendante de la taille du problème et a négligé les communications.

Dans notre exemple, l'algorithme de Gauss, l'hypothèse sur la partie séquentielle n'est pas importatnte car le temps séquentiel est négligeable ($n^2 << n^3$). Mais nous ne pouvons pas négliger les communications et les temps morts dûs aux synchronisations qui sont contenus dans Tc comme dans les études des § 5.

Nous allons étudier l'accélération pour 2 algorithmes avec la répartition entrelassée par blocs de taille un (§ 5), l'élimination de Gauss avec des communication pipelines sur l'anneau (§ 5), et une version synchrone de cette méthode où les communications et le calcul ne peuvent pas se recouvrir. Cela permettra de mettre en exergue l'importance du coût des communications.

L'algorithme PAI (PA Inversé), utilise la même structure mais les communications se font dans le sens inverse de la répartition entrelacé, pour annuler l'effet pipeline.

Algorithme PAI :

```
/* programme du processeur Pi */
pour k=1 à n-1
    {si i=k mod p
        {Tkk
        envoi de la colonne k à Pi-1}
    sinon
        {recevoir la colonne k
        si i≠k+1  {envoi de la colonne k à Pi-1} finsi}
    finsi
    mettre à jour les colonnes internes j≥k+1;}
finpour
```

Le modèle utilisé est classique, $\beta_c + n\tau_c$ (§ 4 et 5) pour la transmission de n données et $\beta_a + n\tau_a$ pour les opérations arithmétiques sur des vecteurs de taille n, β_a est le temps de chargement du pipeline et τ_a le temps de cycle des opérateurs vectoriels.

8.3. ACCÉLÉRATION CLASSIQUE

Si nous choisissons le critère d'Amdahl (taille fixe), le problème de taille maximale qui peut être résolu sur un processeur de l'hypercube T20 est de

taille n=256. Les accélérations correspondantes sont dans les tables 8.1 et 8.2 pour les deux algorithmes.

Processeur	Taille du pb	Temps (sec)	Accélération	Efficacité
1	256	15.98	1	1
2	256	9.35	1.91	0.95
4	256	5.69	3.35	0.84
8	256	3.75	**4.53**	0.55
16	256	2.80	3.94	0.25

Table 8.1: Accélérations pour un problème de taille fixé (n=256), algorithme PA

Processeur	Taille du pb	Temps (sec)	Accélération	Efficacité
1	256	15.98	1	1
2	256	9.35	1.71	0.85
4	256	6.55	**2.44**	0.61
8	256	6.89	2.32	0.29
16	256	10.53	1.51	0.09

Table 8.2: Accélérations pour un problème de taille fixé (n=256), algorithme PAI

Les résultats sont très faibles pour p=16 et l'accélération décroit de 2,4 à 1,5 avec l'algorithme sans recouvrements communications/calcul. La raison principale est le coût prohibitif des communications pour un si petit problème sur un si grand réseau. La partie calcul est négligeable car les communications dominent. Lorsque la taille du réseau augmente, le coût de communication augmente et dépasse le gain en temps de calcul dû à la multiplication des processeurs. La meilleure accélération est atteinte pour 4 processeurs avec PAI et pour 8 avec PA.

Prendre un problème de taille fixée n'est pas très intéressant lorsque l'on dispose de beaucoup de mémoire au total mais peu sur chaque processeur, avec 16 processeurs nous aimerions résoudre des problèmes de taille 1024x1024. Et aussi évaluer les performances sur cette taille de problème.

8.4. ACCÉLÉRATION DE GUSTAFSON

A partir d'un nombre de processeurs donné, nous considérons le plus grand problème que nous pouvons résoudre en utilisant toute la mémoire disponible.

Nous calculons alors $\tau_{max}(p)$, le temps moyen que coûte une opération arithmétique sur la machine donnée pour le problème donné à sa taille

maximale. Ce temps inclut les communications, le controle etc... et représente, en fait, l'inverse de la performance en nombre d'opérations flottantes par seconde (flop). Pour des raisons d'échelle l'unité de $\tau_{max}(p)$ sera le temps d'un million d'opérations correspondant à l'inverse de la performance en Mflops.

Cela correspond à une re-normalisation du problème. L'accélération au sens de Gustafson est alors le rapport de $\tau_{max}(p)$ pour différentes valeurs de p.

Pour une machine distribuée donnée, avec p processeurs identiques, on calcule la taille totale de la mémoire égale à pM où M est la taille de la mémoire locale d'un processeur. Dans notre exemple la taille du problème est la dimension n de la matrice et le stockage nécessaire est de l'ordre de n^2. La taille maximale $n_{max}(p)$ de la matrice stockée sera donc proportionnelle à $\sqrt{p}$:

$$n2 \leq p\,M$$
$$\text{et } n_{max}(p) = (p\,M)^{1/2}.$$

Dans les tables 8.3 et 8.4 nous reportons les accélérations type Gustafson pour différents p. Sur le FPS T20 la valeur de M utilisable est égale à 65 536 mots (de 64 bits). On a alors $n_{max}(1)=256$ et $n_{max}(16)=1024$, les valeurs de n sont tronquées au plus proche multiple de la taille des registres vectoriels à décalage (128) pour ne pas pénaliser la performance du calcul.

Nb proc	Taille pb	Temps (sec)	$\tau_{max}(p)$	Accél.	Efficacité
1	256	15.98	1.43	1	1
2	384	21.75	0.58	2.47	1.23
4	512	23.78	0.27	5.26	1.31
8	768	35.13	0.12	12.31	**1.53**
16	1024	44.01	0.06	**23.26**	1.45

Table 8.3: Accélération pour le problème de taille maximum à p fixé, algorithme PA

Nb proc.	Taille pb	Temps (sec)	$\tau_{max}(p)$	Accél.	Efficacité
1	256	15.98	1.43	1	1
2	384	21.75	0.58	2.47	**1.23**
4	512	26.27	0.29	4.87	1.22
8	768	55.87	0.18	**7.72**	0.96
16	1024	134.05	0.19	7.63	0.48

Table 8.4: Accélération pour le problème de taille maximum à p fixé, algorithme PAI

L'accélération calculée de cette manière est beaucoup plus grande que la version usuelle car elle prend en compte toutes les possibilités offertes par la multiplication du matériel (opérateurs vectoriels, mémoire, registres, etc...).

Les valeurs super-linéaires montrent bien que sur les machines à mémoire distribuée, la performance résulte de l'ensemble du matériel, mémoire et unités de communications comprises.

Nous rappelons et calculons les formules de complexité des algorithmes PA et PAI.

$T_{pa}(n, p)$ a été évaluée § 5 (r=1):

$$T_{pa}(n, p)=(1/p)(2\tau_a n^3/3+\beta_a n^2) + (\tau_a n^2/2+\beta_a n) + \min(3,p-1)(\tau_c n^2/2+\beta_c n)$$

$T_{pai}(n, p)$ se décompose en deux parties distinctes T_a que nous avons déjà calculées précédemment § 5 et T_c qui est simple à calculer lorsque il n'y a pas de recouvrement (cf § 4 pour un graphe de diamètre p-1). Il suffit ensuite de faire la somme:

$$T_{pai}(n, p)=(1/p)(2\tau_a n^3/3+\beta_a n^2) + (\tau_a n^2/2+\beta_a n) + (p-1)(\tau_c n^2/2+\beta_c n)$$

Pour évaluer les performances des deux algorithmes nous calculons l'expression analytique de $\tau_{max}(p)$:

$$\tau_{max}(p) = T(n_{max}(p),p) / [\ 2n_{max}(p)^3/3\ ], \text{ avec } n_{max}(p) = (p\ M)^{1/2}$$

En reportant T_{pai} et T_{pa} dans l'expression de τ_{max} nous avons:

Algorithme PAI:

$\tau_{max}(p) = \alpha\ p^{1/2} + \beta + (\gamma + \gamma')\ p^{-1/2} + (\delta + \delta')\ p^{-1} + \varepsilon\ p^{-3/2}$ pour p>1

$\tau_{max}(1) = \gamma + \delta + \varepsilon$

Algorithme PA:

$\tau_{max}(p) = (g + g")\ p\text{-}1/2 + (d + d")\ p\text{-}1 + e\ p\text{-}3/2$ pour p>3

et les mêmes formules que PAI pour p≤3 car min(3,p-1)=p-1.

Les valeurs des constantes sont les suivantes:

$\alpha = 0.75\ \tau_c\ M^{-1/2} = 0.0337$
$\beta = 1.5\ \beta_c\ M^{-1} = 0.0183$
$\gamma = 0.75\ \tau_a\ M^{-1/2} = 0.0004$
$\gamma' = -0.75\ \tau_c\ M^{-1/2} = -0.0337$
$\gamma'' = 2.25\ \tau_c\ M^{-1/2} = 0.1011$
$\delta = \tau_a + 1.5\ \beta_a\ M^{-1} = 0.1420$
$\delta' = -1.5\ \beta_c\ M^{-1} = -0.0183$
$\delta'' = 4.5\ \beta_c\ M^{-1} = 0.0549$
$\varepsilon = 1.5\ \beta_a\ M^{-1/2} = 1.2832$

Elles sont calculées avec les valeurs expérimentales mesurées sur l'hypercube: $\tau_a = 1{,}37\ 10^{-7}$ secondes
$\beta_a = 2{,}19\ 10^{-4}$ secondes
$\tau_c = 1{,}15\ 10^{-5}$ secondes
$\beta_c = 8{,}30\ 10^{-4}$ secondes

La figure 8.1 comprend 4 courbes de $\tau_{max}(p)$ qui résument ces résultats, les 2 courbes expérimentales s'arrêtent pour 16 processeurs et les courbes théoriques vont jusqu'à 16 384 processeurs. On peut remarquer pour les deux algorithmes la bonne adéquation des valeurs expérimentales et théoriques.

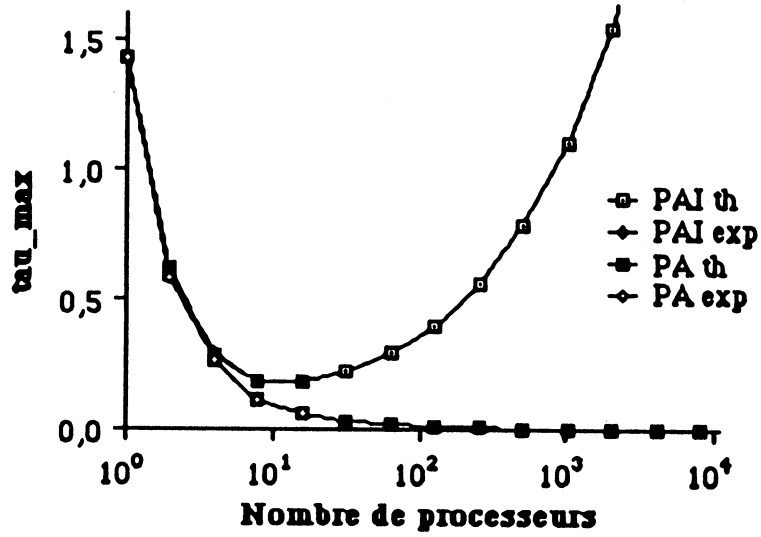


Figure 8.1: τ_{max} en fonction de p, valeurs experimentales et théoriques

La table ci-dessous donne les valeurs numériques de 1 à 16 processeurs.

nb processeurs	PA exp	PA th	PAI exp	PAI th
1	1.428	1.426	1.428	1.426
2	0.576	0.558	0.576	0.558
4	0.266	0.260	0.294	0.260
8	0.116	0.117	0.185	0.174
16	0.061	0.058	0.187	0.173

Table 8.5: τ_{max} expérimental et théorique de 1 à 16 processeurs

Il est interressant de noter le comportement de τ_{max} pour l'algorithme PAI. Il présente un minimum pour 11 processeurs, taille de la machine a meilleur rendement avec les processeurs utilisés.

8.5. UTILISATION DES RÉSULTATS SUR L'ACCÉLÉRATION POUR DIVERS ALGORITHMES

A partir des résultats obtenus § 7 pour 16, 32 et 64 processeurs, nous pouvons comparer l'accélération sous ses deux formes pour la décomposition LU suivant l'algorithme avec diffusion des données, la répartition entrelacée et une programmation des unités vectorielles au niveau "generic".

Nous comparons les algorithmes à taille de problème fixé en prenant p=16 et n=1024 comme base de comparaison (accélération=1) pour ne pas trop pénaliser l'accélération usuelle.

Nous mesurons donc le gain sur une architecture parallèle avec doublement ou quadruplement du nombre de processeurs, et pas le gain par rapport à la version séquentielle du problème. Cela peut servir par exemple pour choisir une configuration de machine à mémoire distribuée.

Nb processeurs	Temps (sec)	Accélération	Efficacité
16	36.11	1	1
32	24.12	1.49	0.75
64	18.25	1.98	0.49

Table 8.6: Accélération classique

Si nous utilisons l'accélération de Gustafson les résultats sont meilleures et correspondent beaucoup plus à la réalité:

Nombre processeurs	Taille problème	Temps (sec)	Temps pour 10M opér	Speed-up	Efficiency (1/Mflops)
16	1024	36.11	0.050	1	1
32	1280	38.73	0.028	1.82	0.91
64	2048	77.40	0.013	**3.73**	**0.93**

Table 8.7: Accélération de type Gustafson

Bien sûr les deux approches peuvent servir si l'on cherche à résoudre un problème sur une machine. On peut dire en fait que la première donne une évaluation figée de la machine pour le problème donné alors que la deuxième décrit le potentiel objectif de la machine.

Avec l'algorithme le plus performant que nous ayons implémenté sur cette machine, l'élimination de Gauss par blocs de taille 2, les résultats sont de même type et peuvent être estimés directement sur la figure 8.2 avec les courbes. Le rapport des ordonnées des points d'abscisses les plus grands pour Gustafson et rapport des ordonnées pour la même d'abscisses pour l'accélération usuelle.

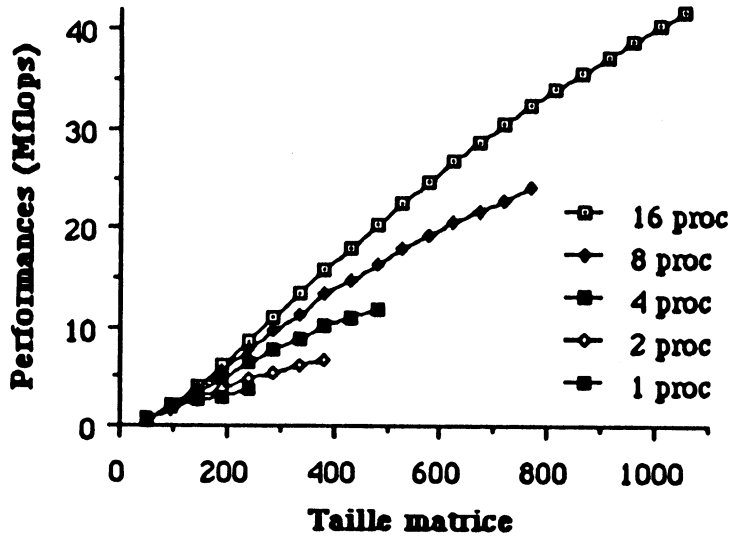


Figure 8.2: Performances de l'algorithme de Gauss par blocs de taille 2 pour différentes tailles d'anneaux (diffusion, répartition entrelacée)

Valeurs correspondantes de l'accélérations de type Gustafson sont reportées dans la table 8.8.

Nombre processeurs	Taille problème	Temps (sec)	Temps pour 1M opé	Accélération	Efficacité
1	240	2.54	0.275	1	1
2	384	5.56	0.147	1.87	0.93
4	480	6.23	0.084	3.26	0.81
8	768	12.60	0.042	6.60	0.82
16	1056	18.84	0.024	11.48	0.72

Table 8.8: Accélération de type Gustafson

Avec cette version optimisée de programmation, on ne trouve pas de valeurs super linéaires, mais l'accélération est proche du nombre de processeur.

8.6. CONCLUSION

Nous pensons que $\tau_{max}(p)$ est la variable qui donne le meilleur aperçu des performances d'un algorithme sur une machine à mémoire distribuée de taille p. Etant donné que tout le matériel est pris en compte, on peut penser que cette méthode est valable aussi pour les multiprocesseurs à mémoire partagée (elle tient compte de la multiplication des caches etc...).

Les formules montrent que le comportement asymptotique de $\tau_{max}(p)$ dépend essentiellement de la complexité des communications de l'algorithme étudié. Par exemple pour l'algorithme PA $\tau_{max}(p)= O(1/\sqrt{p})$.
$\tau_{max}(p)$ diminue lorsque p augmente, l'efficacité augmente sans limitation avec le nombre de processeurs. Cela contredit la loi d'Amdahl et montre bien le comportement tout à fait différent des deux accélérations.

L'algorithme PAI présente au contraire un minimum pour $\tau_{max}(p)$ lorsque p=11, le parallélisme massif n'apporterait rien en vitesse pure à cet algorithme pour un problème de taille fixée ($\tau_{max}(p)$ est plus grand pour 1 processeur que pour 2048, la courbe se comporte comme $\sqrt{p}$ à l'infini). Mais on pourrait quand même résoudre un problème de plus grande taille (lentement !) avec plus de processeurs. Une analyse de ce type est intéressante pour connaitre le nombre de processeurs qui présente le meilleur rendement à partir d'un algorithme donné.

Grâce aux études de complexité, pour les deux algorithmes nous pouvons calculer les performances sur une machine à p processeurs donnés. Par exemple avec 1024 processeur l'algortihme PAI atteindrai 300 Mflops pour un problème de taille 8192. L'accélération serait alors égale à 420.

Pour les algorithmes dont l'analyse de complexité est difficile, i.e.dont on ne connait pas de formule analytique précise, le calcul de $\tau_{max}(p)$ comme l'inverse des Mflops obtenus pour la taille maximale, est facile et permet de montrer l'accélération due à l'augmentation du nombre de tous les matériels (CPU, mémoire, ...).

L'orientation future comprend la prise en compte des performances de tout le matériel, mais aussi du logiciel système sous-jacent. Ainsi, un problème remplissant presque totalement la mémoire d'un petit réseau fera appel sans arrêt sur chaque processeur à des mécanismes de récupération de place mémoire au cours de l'algorithme (ramasse miettes). Sur un réseau de taille plus importante, la saturation de la pile dans l'espace mémoire n'interviendra presque jamais, procurant un gain de temps important sur la durée d'exécution de l'algorithme. Ce phénomène a déjà été remarqué par les chercheurs de l'équipe Calcul Formel du laboratoire. La prise en compte de la performance d'un logiciel, ici le système d'exploitation, est très délicate et laisse de grands problèmes ouverts pour les machines parallèles.

CONCLUSIONS

J'espère par ce travail avoir apporté une petite contribution à l'algorithmique parallèle et par là même avoir montré que les machines à mémoire distribuée sont intéressantes.

Un point important est la modularité de telles architectures qui permet d'étendre les résultats pour un plus grand nombre de processeurs.
Des algorithmes matriciels, typiquement l'élimination de Gauss, se trouvent dans la plupart des codes de calcul scientifique. Ils consistent en une imbrication de "boucles pour" ce qui caractérise un grand nombre d'algorithmes divers (des plus courts chemins d'un graphe à l'algèbre linéaire). Cela donne un intérêt pratique aux algorithmes étudiés sur des architectures parallèles à mémoires distribuée.

Les méthodes que nous proposons pour augmenter l'efficacité des algorithmes sur machines à mémoires distribuées peuvent s'élargir facilement en changeant les paramètres liés à la machine cible ou les formules analytiques de complexité des algorithmes étudiés.

Un grand nombre de problèmes restent ouverts, dans tous les domaines parcourus, dès que l'on relache un peu les contraintes des modèles, par exemple pour la répartition des données le choix entrelacé par blocs. Ou bien lorsque l'on augmente la complexité des solutions algorithmiques proposées, on ne sait plus alors modéliser les temps morts dûs aux synchronisations.

Pour finir, je dirai que le développement des cartes de Transputers pour les stations de travail encourage encore (si besoin est !) de telles études sur les systèmes, l'architecture et l'algorithmique parallèle.

RÉFÉRENCES

[Amd] G.M. AMDAHL, Validity of the single-processor approach to achieving large-scale computing capabilities, AFIPS 30, AFIPS Press (1967), p 483-485

[Ben] A. BENAINI, Thèse de Docteur de l'INP-Grenoble, (1988)

[Ber] C. BERGE, Graphes et Hypergraphes, Dunod Ed.

[BGWJ] M. BERRY, K. GALLIVAN, W. HARROD, W. JALBY, S. LO, U. MEIER, B. PHILIPPE, A. H. SAMEH, Parallel algorithms on the CEDAR system, COMPAR 86, G. Goos and J. Hartmanis eds., Lecture Notes in Computer Science 237, Springer Verlag (1986), p 25-39

[BKK] A. BOJANCZYK, R.P. BRENT, H.T. KUNG, Numerically stable solution of dense systems of linear equations using mesh-connected processors, Tech. Rep, Carnegie Mellon University (1981)

[Cap] P. R. CAPPELLO, Gaussian elimination on a hypercube automaton, Journal of Parallel and Distributed Computing 4 (1987), p 288-308

[CDMR] M. COSNARD, E.M. DAOUDI, J.M. MULLER, Y. ROBERT, On parallel and systolic Givens factorizations of dense matrices, Parallel Algorithms and Architectures, North-Holland (1986), p 245-258

[CDR] M. COSNARD, E.M. DAOUDI, Y. ROBERT, Complexity of the parallel Givens factorization on shared memory architectures, Rap. Rech.(1989), LIP-IMAG, ENS -Lyon

[CDT1] M. COSNARD, E.M. DAOUDI, B. TOURANCHEAU Communication dans les réseaux de processeurs et complexité d'algorithmes, Colloque C^3 (1987), Angoulème, A. Arnold e., p 121-143

[CGe] C. CHU, A. GEORGE, Gaussian elimination with partial pivoting and load balancing on a multiprocessor, Research Rep CS-85-48 (1985), Univ. of Waterloo, Ontario, Canada N2L 3G1

[Cha] R. M. CHAMBERLAIN, An alternative view of LU factorization on a hypercube multiprocessor, Hypercube Multiprocessors 1987, M.T. Heath ed., SIAM editions (1987), p 569-575

[CMR] M. COSNARD, J.M. MULLER, Y. ROBERT, Parallel QR decompostion of a rectangular matrix, Numerische Mathematik 48, (1986), p 239-249

[CMRT] M. COSNARD, M. MARRAKCHI, Y. ROBERT, D. TRYSTRAM, Parallel Gaussian elimination on an MIMD Computer, Parallel Computing 6 (1988), p 275-296

[CRo1] M. COSNARD, Y. ROBERT, Complexity of parallel QR factorization, J.A.C.M. 33,4 (1986), p 712-723

[CRo2] M. COSNARD, Y. ROBERT, Algorithmique parallèle: une étude de complexité, à paraître dans Technique et Science Informatiques

[CRo3] M. COSNARD, Y. ROBERT, Systolic Givens factorization of dense rectangular matrices, Internat. Journal of Comp. Math., vol 25 (1988), p 287-298

[CRRS] P. CARNEVALI, G. RADICATI, Y. ROBERT, P. SGUAZZERO, Efficient FORTRAN Implementation of the Gaussian elimination and Householder reduction algorithms on the IBM 3090 vector multiprocessor, IBM ECSEC Technical Report (1987)

[CRTo] M. COSNARD , Y. ROBERT, B. TOURANCHEAU, Evaluating speedups on distributed memory architectures, à paraître dans Parallel Computing.

[CRTr] M. COSNARD, Y. ROBERT, D. TRYSTRAM, Parallel solution of dense linear systems using diagonalization methods, Internat. Journal of Comp. Math. 22 (1987)

[CSa] T.F. CHAN, Y. SAAD, Multigrid algorithms on the hypercube multiprocessor, IEEE T.C. 35 (1986), p 969-977

[CTc] M. COSNARD, M. TCHUENTE, Designing systolic algorithms by top-down analysis, ICS 88, Boston (1988)

[CTo1] M. COSNARD, B. TOURANCHEAU, Linear systems solvers on processor networks, ICIAM 87, (1987), Paris.

[CTo2] M. COSNARD, B. TOURANCHEAU, Matrix decomposition on multiprocessors, International Conference On Parallel Processing and Applications, Septembre 1987, l'Aquilla, Italie, Ed. North Holland.

[CTT] M. COSNARD , M. TCHUENTE, B. TOURANCHEAU, Systolic Gauss-Jordan elimination for dense linear systems, à paraître dans Parallel Computing.

[CTV1] M. COSNARD, B. TOURANCHEAU, G. VILLARD, Gaussian Elimination on Message Passing Architecture, International Conference on Supercomputing (1987), Patras, Grece, Ed. Springer-Verlag.

[DDDH] J.J. DONGARRA, J. Du CROZ, I. DUFF, S. HAMMARLING, A proposal for a set of a Level 3 BLAS, Argonne National Laboratory Report MCS-TM-88 (1987)

[DIp] J.M. DELOSME, I.C.F. IPSEN, Systolic array synthesis, in Parallel Algorithms and Architectures North-Holland (1986), p 295-312

[DSo] J.J. DONGARRA and D.C. SORENSEN, Linear algebra on high-performance computers, in Paralel Computing 85, M. Feilmeier et al. eds., Elsevier Science Publishers B.V. (1986), p 221-230

[Dun] T.H. DUNIGAM, Hypercube Performance, Hypercube Multiprocessors 1987, Ed. M.T. Heath, SIAM (1987), p 178-192.

[Fly] M.J. FLYNN, Very high-speed computing systems, Proc. IEEE 54, 1901-1909 (1966)

[FPS] FLOATING POINT SYSTEMS, T Series Manual Release C, (1988)

[Gei] G.A. GEIST, Efficient parallel LU factorisation with pivoting on a hypercube multiprocessor, Oak Ridge National Laboratory Tech. Rep. 6211 (1985)

[Gen] W.M. GENTLEMAN, Some complexity results for matrix computations on parallel processors, JACM 25, no 1 (1978), p 112-115

[GHe] G.A.GEIST, M.T.HEATH, Matrix Factorization on a hypercube multiprocessor, Hypercube Multiprocessors 1986, M.T. Heath ed., SIAM Ed (1986), p 161-180

[GHS] J.L. GUSTAFSON, S. HAWKINSON, K. SCOTT, The architecture of a homogeneous vector supercomputer, Proceedings of ICCP 86, IEEE Computer Science Press (1986), 649-652

[GJMS] K. GALLIVAN, W. JALBY, U. MEIER, A. SAMEH, The impact of hierarchical memory systems on linear algebra algorithmic design, CSRD Technical Report 625, The University of Illinois (1987)

[GKu] W.M GENTLEMAN, H.T. KUNG, Matrix triangularisation by systolic arrays, in Proceedings SPIE (Society of Photo-Optical Instrumentation Engineers), vol 298, Real-time Signal Processing IV (1981), San Diego, Califomia,p 19-26

[GLo] G.H. GOLUB, C.F. VAN LOAN, Matrix Computations, The John Hopkins University Press (1983)

[GNe] A. GERASOULIS, I. NELKEN, Gaussian elimination and Gauss-Jordan on MIMD architectures, Report LCSR-TR-105, Department of Computer Science, Rutgers University (1988)

[GPe] D.D. GAJSKI, J.K. PEIR, Essential issues in multiprocessors systems, IEEE Computer, (1985), p 9-27

[GRo] D. GANNON, J. VAN ROSENDALE, On the impact of communication in the design of parallel algorithms, IEEE Trans. Comput. 33, 12 (1984), p 1180-1194

[Gus1] J.L. GUSTAFSON, The scaled-sized model: a revision of Amdahl's law, ICS Supercomputing'88, L.P. Kartashev and S.I. Kartashev eds., International Supercomputing Institute Inc. (1988), vol. II, p 130-133

[Gus2] J.L. GUSTAFSON, Reevaluating Amdahl's law, Communications of the ACM 31, 5 (1988), p 532-533

[HBr] K. HWANG, F. BRIGGS, Parallel processing and computer architecture, Mc Graw Hill (1984)

[Hel] D. HELLER, A survey of parallel algorithms in numerical linear algebra, Siam Review 20 (1978), p 740-777

[Her] A. HERSCOVICI, Introduction aux grands calculateurs scientifiques, Ed. Eyrolles, (1986)

[Hip] P. G. HIPES, A Gauss-Jordan decomposition and system solver for distributed memory multicomputers, CONPAR 88, (1988), Manchester, U.K., C.H. Jesshope and K.D. Rheinhartz eds.

[HJo] C.T. HO, S.L. JOHNSSON, Distributed routing algorithms for broadcasting and personalized communication in hypercubes, in Proceedings of ICCP 86, IEEE Computer Science Press (1986), p 640-648

[Hoa] C. HOARE,Communicating sequential process, CACM vol 21-8 (1978), p 666-677

[HRo] M.C HEATH, C.H. ROMINE, Parallel solution of triangular systems on distributed memory multiprocessors, SIAM J. Sci. Statist. Comput., vol 9 no 3 (1988)

[IBM] IBM Europe Institute, University Supercomputing, Oberlech, Austria, 1-5 août (1988)

[Inm1] INMOS product description, IMS T414 transputer.

[Inm2] INMOS product description, IMS T800 transputer.

[ISS] I.C.F. IPSEN, Y. SAAD, M.H. SCHULTZ, Complexity of dense linear system solution on a multiprocessor ring, Lin. Alg. Appl. 77 (1986), p 205-239

[Joh] S.L. JOHNSSON, Communication efficient basic linear algebra computations on hypercube architectures, Journal of paral. and distrib. comput. 4, (1987), p 133-172

[KLe] H.T. KUNG, C.E. LEISERSON, Systolic arrays for VLSI, Symposium on Sparse Matrices Computations, Knoxville, Tennessy, (1978), p 256-282

[Kun] H.T. KUNG, Why systolic architectures, IEEE Computer 15, 1 (1982), p 37-46

[LCo] G. LI, T.F. COLEMAN, A parallel triangular solver for a Hypercube multiprocessor, SIAM Journal of Sci. Statist. Comput., vol 9 no3, (1988)

[LKe] R. LEE, Z. M. KEDEM, Synthesizing linear array algorithms fron nested for loop algorithms, IEEE Transac. on Comp., vol 37, no 12, (1988)

[LKK] R.E. LORD, J.S. KOWALIK, S.P. KUMAR, Solving linear algebraic equations on an MIMD computer, J. ACM 30 (1), 1983, p 103-117

[MCl] J.J. MODI, M.R.B. CLARKE, An alternative Givens ordering, Num. Math. 43 (1984), p 83-90

[Mel] R. MELHEM, Parallel Gauss-Jordan elimination for the solution of dense linear systems, Parallel Computing 4 (1987) 339-343

[Mol] C. MOLER, Matrix computations on distributed memory multiprocessors, Hypercube Multiprocessors 1986, M.T. Heath ed., SIAM editions (1986), p 161-180

[MRo] M. MARRACHI, Y. ROBERT, Optimal Algorithms for Gaussian Elimination on a MIMD Computer, à paraître dans Parallel Computing, (1989)

[MTj] N.M. MISSIRLIS, F. TJAFERIS, Parallel matrix factorizations on a shared memory MIMD computer, Supercomputig, E.N. Houstis et al. ed., Lecture Notes in Comp. Science 297, Springer verlag (1988), p 926-937

[ORo] J.M. ORTEGA, C.H. ROMINE, The ijk forms of factorization methods II. Parallel systems, Parallel Computing 7 (1988), p 149-162

[QRo] P. QUINTON, Y. ROBERT, Algorithmes et architectures systoliques, Masson Ed. (1989)

[Rat] J. RATTNER, Concurrent processing: a new direction in scientific computing, in AFIPS Conference Proceedings 54, AFIPS Press (1985), p 159-166

[Rom] C.H. ROMINE, The parallel solution of triangular systems on a hypercube, Hypercube multiprocessors 1987, M.T. Heath ed., SIAM (1987), Philadelphia, p 552-559

[RRS] G. RADICATI, Y. ROBERT, P. SGUAZZERO, Block processing in linear algebra on the IBM 3090 Vector Multiprocessor, Supercomputer 23, vol V-1 (1988), p 15-25

[RTc] Y. ROBERT, M. TCHUENTE, Résolution systolique de systèmes linéaires denses, MAN 2 19, 2(1985), p 315-326

[RTo1] Y. ROBERT, B. TOURANCHEAU, "LU and QR factorization on the FPS T Series hypercube", CONPAR 88, Septembre 1988, Manchester, U.K., C.H. Jesshope and K.D. Rheinhartz eds.

[RTo2] Y. ROBERT, B. TOURANCHEAU, "Block Gaussian Elimination on a Hypercube Vector Multiprocessor", Revista de Mathematicas Aplicadas 10, (1988), p 77-89, Universidad de Chile.

[RTo3] Y. ROBERT, B. TOURANCHEAU, "Linear algorithms on distributed memory machines", Mathematics in signal processing, Warwick (UK), 13-15 Déc.(1988).

[RTo4] Y. ROBERT, B. TOURANCHEAU, "Impact of the architecture topology on data allocation strategies for Gaussian elimination on the hypercube", The fourth Conference on Hypercube Concurrent Computer and Applications (HCCA4), Monterey (Cal.), USA, (1989).

[RTr] Y. ROBERT, D. TRYSTRAM, Optimal scheduling algorithms for parallel Gaussian elimination, EUCOPE 87, North Holland (1987).

[RTV1] Y. ROBERT, B. TOURANCHEAU, G. VILLARD, "Algorithmes de Gauss et Jordan sur un anneau de processeurs", CRAS 88, Paris France

[RTV2] Y. ROBERT, B. TOURANCHEAU, G. VILLARD "Data allocation strategies for the Gauss and Jordan algorithms on a ring of processors", à paraître dans Information Processing Letters.

[Saa1] Y. SAAD, Communication complexity of the Gaussian elimination algorithm on multiprocessors, Lin. Alg. Appl. 77 (1986), p 315-340

[Saa2] Y. SAAD, Gaussian elimination on hypercubes, in Parallel Algorithms and Architectures, M. Cosnard et al. eds., North-Holland (1986), p 5-18

[Sam1] A. SAMEH, Numerical parallel algorithms - a survey, in "High Speed Computer and Algorithm Organization", D.Kuck, D.Lawrie and A.Sameh eds, Academic Press (1977), p 207-228

[Sam2] A. SAMEH, Solving the linear least squares problem on a linear array of processors, Proc. Purdue Workshop on algorithmically-specialized computer organizations, W. Lafayettte, Indiana, (1982)

[Sam3] A. SAMEH, An overview of parallel algorithms, Bull. EDF, C1, (1983), p 129-134

[Sam4] A. SAMEH, On some parallel algorithms on a ring of processors, Computer Phys. Com. 37 (1985), p 159-166

[Sch] U. SCHENDEL, Introduction to Numerical Methods for Parallel Computers, Ellis Horwood Series, J. Wiley & Sons, New York, (1984)

[SKu] A. SAMEH, D.J. KUCK, On stable parallel linear system solvers, JACM 25, 1 (1978), p 81-91

[Sor] D.C. SORENSEN, Analysis of pairwise pivoting in Gaussian elimination, Tech. Rep., Argonne Nat. Lab., MCS-TM-26, (1984)

[Sri] M.K. SRIDHAR, A new algorithm for parallel solution of linear equations, Information Proc. Let. 24 (1987), p 407-412

[SSc1] Y. SAAD, M.H. SCHULTZ, Topological properties of hypercubes, IEEE Trans. Computers 37, 7 (1988), p 867-872

[SSc2] Y. SAAD, M.H. SCHULTZ, Data communication in hypercubes, Research Rep. YALEU/DCS/RR-428 (1985)

[SSc3] Y. SAAD, M.H. SCHULTZ, Data communication in parallel architectures, Research Rep. YALEU/DCS/RR-461 (1986)

[Ste] G.W. STEWART, Introduction to Matrix Computations, Academic Press (1973)

[STo1] S. KUPPUSWAMI, B. TOURANCHEAU, "Evaluating the performances of the FPS T Series hypercube computer", (1987), Rap. Rech. Info. Nu. 708, IMA Grenoble.

[STo2] S. KUPPUSWAMI, B. TOURANCHEAU, "Evaluating the performances of transputer based hypercube vector computer", Mars 1988, Rap. Int

[Tou1] B. TOURANCHEAU, "LU factorization on the FPS T Series hypercube", Parallel and distributed algorithms, Bonas, Octobre 1988, M. Cosnard et al ed., North Holland.

[Tou2] B. TOURANCHEAU, Mémoires distribuées: répartition des données pour l'algorithme de Gauss, Proc. 3ème Colloque C3, Angoulème (France), A. Arnold Ed. (1988)

[TVi1] B. TOURANCHEAU, G. VILLARD, "Manuel d'utilisation de l'hypercube FPS T20 (VMS, Occam)", Octobre 1987, Rapport Interne, IMA.Grenoble.

[TVi2] B. TOURANCHEAU, G. VILLARD, "Manuel d'utilisation de l'hypercube FPS T20 (Ultrix, C, Fortran)", Mars 1988, Rapport Interne, IMA Grenoble.

[Ull] J.D. ULLMAN, Computational Aspects of VLSI, Computer Science Press (1983)

[Vil1] G. VILLARD, Calcul formel et parallélisme Résolution de systèmes linéaires, Thèse INP-Grenoble, 23 Déc. (1988)

[Vil2] G. VILLARD, Parallel general solution of rational linear systems using p-adic expansions, IFIP 10.3 (1988), Pise-Italie

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs

. J. Cl. BERMOND, Directeur de Recherche CNRS

. Y. ROBERT, Professeur

Monsieur TOURANCHEAU Bernard

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " Informatique "

Fait à Grenoble, le 2 février 1989

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU