Quelques résultats de complexité en algorithmique parallèle et systolique

Denis Trystram

▶ To cite this version:

Denis Trystram. Quelques résultats de complexité en algorithmique parallèle et systolique. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1988. tel-00009202v2

HAL Id: tel-00009202
https://theses.hal.science/tel-00009202v2

Submitted on 13 Oct 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

de DOCTORAT de l'INPG
(arrété ministériel du 5 juillet 1984)
Spécialité : INFORMATIQUE

présentée le 15 Avril 1988 par
Denis TRYSTRAM

QUELQUES RESULTATS DE COMPLEXITE EN ALGORITHMIQUE PARALLELE ET SYSTOLIQUE

jury :

J.P. VERJUS (président)
F. BACCELLI
M. COSNARD
G. MAZARE
P. QUINTON (rapporteur)
Y. SAAD (rapporteur)

préparée au sein du laboratoire TIM3

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

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M.MERMET
Directeur des Etudes et de la formation: Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

PROFESSEURS DE 1ère CATEGORIE

COINDE Alexandre	Gestion
GOUX Claude	Métallurgie
LEVY Jacques	Métallurgie
LOWYS Jean-Pierre	Physique
MATHON Albert	Gestion
RIEU Jean	Mécanique-Résistance des matériaux
SOUSTELLE Michel	Chimie
FORMERY Philippe	Mathématiques Appliquées

PROFESSEURS DE 2ème CATEGORIE

HABIB Michel	Informatique
PERRIN Michel	Géologie
VERCHERY Georges	Matériaux
TOUCHARD Bernard	Physique Industrielle

DIRECTEUR DE RECHERCHE

LESBATS Pierre	Métallurgie

MAITRE DE RECHERCHE

BISCONDI Michel	Métallurgie
DAVOINE Philippe	Géologie
FOURDEUX Angeline	Métallurgie
KOBYLANSKI André	Métallurgie
LALAUZE René	Chimie
LANCELOT Francis	Chimie
LE COZE Jean	Métallurgie
THEVENOT François	Chimie
TRAN MINH Canh	Chimie

Personalités habilitées à diriger des travaux de recherche

DRIVER Julian	Métallurgie
GUILHOT Bernard	Chimie
THOMAS Gérard	Chimie

Professeurs à l'UER de Sciences de Saint-Etienne

VERGNAUD Jean-Maurice	Chimie des Matériaux et Chimie Industrielle

Je voudrais exprimer ici toute ma reconnaissance aux membres du jury :

Jean Pierre Verjus, pour l'honneur qu'il me fait de présider ce jury,
Michel Cosnard, plus qu'un directeur de recherche, par ses encouragements, a su me prodiguer une motivation constante tout en me laissant l'initiative nécessaire à mon épanouissement. Il m'a transmis une certaine manière de penser et de porter la casquette,
François Baccelli, pour s'être déplacé depuis Nice, pour un exposé qui sera bientôt distribué,
Guy Mazaré pour les "bémols" qu'il m'a suggéré d'introduire dans le texte afin d'améliorer la mélodie,
Patrice Quinton, pour avoir su m'accorder un peu de son temps et accepté de juger ces pages indigestes,
Youcef Saad, pour l'intérêt qu'il a porté à mon travail, la confiance et l'amitié qu'il m'a toujours prodiguées.

Je tiens également à remercier :

Yves Robert, avec qui la recherche est devenue un jeu où chacun est tour à tour acteur et metteur en scène. Je suis heureux de l'amitié qu'il veut bien me témoigner,
Pascal Laurent-Gengoux, de qui j'ai pu apprécier la compétence et la patience lorsque nous partagions le même bureau à l'Ecole Centrale où nous avons conjugué le Gradient ensemble pendant trois années,
Pierre Comon, compagnon de toujours, pour tout ce que nous avons partagé,
Cyrille Comar, Pierre Hochard, Patrice Richard, Jean Luc Stehlé et dans cette ambiance le plus souvent masculine Grazyna Wojesko, membres du CREDI de l'Ecole Centrale qui m'a accueilli et au sein duquel j'ai pu organiser une petite cellule de recherche avec Philippe Boutin, le roi de la contrepétrie et du fer à souder et François Ptitgars-Vincent.
Jean Della Dora mon directeur bien aimé, François Robert qui m'a initié à la recherche - je souhaite à d'autres qu'il leur communique son enthousiasme -, Afonso Ferreira mon brésilien préféré, Jean Michel Muller mon voisin de bureau préféré, Bernard Tourancheau mon randonneur préféré et tous les autres membres de l'équipe "Algorithmique Parallèle et Calcul Formel", chercheurs et secrétaires que je retrouve souvent autour d'une tasse de café,
Jean Claude Konig et Jérôme Ryckbosch, pour leur compétence et leur gentillesse et Piero Sguazzero, qui m'a permi d'effectuer plusieurs séjours au Centre Scientifique d'IBM à Rome,
Enfin, Victor Prosciuto, personnage multiple et tous les autres, Antoine, pour ne pas m'avoir dérangé pendant la rédaction, Finger, pour sa conversation et Patrizia, pour des raisons qui n'ont rien à voir dans tout cela !

Nous participons tous
à l'écriture d'un grand livre

ORGANISATION GENERALE

Le lecteur trouvera dans ces quelques pages le résumé de l'ouvrage et les principales options choisies. J'espère qu'elles lui donneront le goût de poursuivre jusqu'au bout la lecture.

Dans ce travail, qui retrace le bilan de quatre années de recherches, il n'était pas raisonnable de tout présenter. J'ai choisi de privilégier les principes et méthodes plutôt que des résultats techniques fastidieux et des applications spécifiques. Cependant, la seconde partie de cette thèse est consacrée à un recueil de travaux personnels qui viennent compléter la présentation des idées de la première partie par des articles plus détaillés.

L'ouvrage ne suit pas l'ordre chronologique des recherches, il a été réorganisé pour une meilleure compréhension. Délibérément, j'ai choisi de présenter d'abord les résultats généraux, puis des directions plus particulières. Chacun des chapitres tente, de manière allusive (mais non simpliste), de montrer la constitution d'outils et de modèles pour l'étude des algorithmes sur les architectures parallèles, de faire le point sur ces méthodes, d'en proposer des extensions pour que le modèle se rapproche encore plus de la réalité et enfin, d'illustrer le propos par un tour d'horizon de quelques résultats plus concrets.

Le lecteur pourra se reporter à la fin pour une connaissance approfondie, plus technique et réellement opérationnelle des méthodes présentées.

Ces résultats ont été établis en collaboration avec P. Comon, M. Cosnard, J.C. Konig, P. Laurent-Gengoux, M. Marrakchi, J.M. Muller, Y. Robert et J. Ryckbosch. Je voudrais témoigner ici du grand plaisir que j'ai eu à travailler avec chacun d'eux et je souhaite vivement que les échanges fructueux que nous avons eus ensemble continuent dans l'avenir.

PLAN

L'introduction est consacrée aux rappels des concepts fondamentaux qui régissent la conception des architectures parallèles. Dans le second chapitre, nous présentons les résultats théoriques concernant l'étude de complexité des algorithmes parallèles, ainsi qu'une description de l'architecture de référence, qui est une machine MIMD à mémoire partagée. Nous y discutons la validité de ce modèle théorique. Le chapitre suivant est dédié à l'ensemble des résultats de complexité concernant des algorithmes de référence de l'algèbre linéaire (diagonalisation de Jordan et élimination de Gauss), il a pour but d'illustrer la méthodologie. Il existe plusieurs versions possibles de la méthode de Gauss en parallèle, qui conduisent principalement à deux grandes classes de graphes de précédence, conceptuellement différents : les graphes de type "glouton" et ceux du type "2-pas".

Ces types de graphes se rencontrent d'une manière plus générale dans d'autres problèmes d'algèbre linéaire ainsi que dans certaines méthodes non numériques de la théorie des graphes, comme la recherche des plus courtes distances ou la fermeture transitive. Nous développons ensuite les résultats de complexité concernant des algorithmes d'ordonnancement optimaux sur ces deux types de graphes pour les exemples les plus courants (versions "kji" et "kij" de Gauss en parallèle). Nous examinons ensuite comment adapter l'étude en prenant en compte les temps de communication entre les processeurs. Puis, nous présentons quelques résultats complémentaires sur les architectures à parallélisme massif : les architectures "systoliques" (en particulier, des implémentations systoliques du problème du chemin algébrique et du calcul de projections en filtrage adaptatif). Avant de conclure, nous présentons des résultats plus pratiques, de parallélisation effective sur des architectures existantes (le CRAY-XMP et l'IBM 3090 VF multi processeur) en nous intéressant plus particulièrement à l'algorithme du gradient conjugué.

SOMMAIRE

1. UNE INTRODUCTION AU PARALLELISME 9
1.1. Le besoin en puissance de calcul
1.2. Les solutions possibles
1.3. Les architectures parallèles

2. COMPLEXITE 23
2.1. Description du modèle de référence
2.2. Les outils d'analyse
2.3. Illustration sur des exemples simples
2.4. Un mot des algorithmes itératifs

3. GRAPHE DE PRECEDENCE 35
3.1. Diagonalisation de Jordan en parallèle
3.2. L'Elimination de Gauss

4. ORDONNANCEMENT 43
4.1. Le graphe Glouton
4.2. Le graphe 2-pas

5. PRISE EN COMPTE DES TEMPS DE COMMUNICATION 59
5.1. Analyse générale
5.2. Etude de version par blocs
5.3. Algorithme adaptatif
5.4. Vers un modèle général

6. RESEAUX SYSTOLIQUES 69
6.1. Présentation générale
6.2. Problème du chemin algébrique
6.3. Calcul de projections en filtrage adaptatif

7. LES SUPER CALCULATEURS 81
7.1. Ordinateurs vectoriels
7.2. Programmation du gradient conjugué

8. CONCLUSION 89

REFERENCES 91

RECUEIL D'ARTICLES

"Parallel solution of dense linear systems using diagonalization methods"
Internat. Journal of Comp. Math. 22, 1987
avec M. COSNARD et Y. ROBERT

"Parallel Gaussian elimination for an MIMD computer"
Parallel Computing, Vol. 6, N. 3, 1988
avec M. COSNARD, M. MARRAKCHI et Y. ROBERT

"Optimalité d'une classe d'algorithmes d'ordonnancement"
Comptes Rendus de l'Académie des Sciences de Paris N. 5, Série I, 1988
avec J.C. KONIG et Y. ROBERT

"Optimal Scheduling Algorithms for Parallel Gaussian Elimination"
Proceedings of EUCOPE 87, North Holland, à paraître
avec Y. ROBERT

"Communication Costs versus Computation Costs in Parallel Gaussian Elimination"
Internat. Workshop "Parallel Algorithm and Architecture", North Holland, 1986
avec M. COSNARD, J.M. MULLER et Y. ROBERT

"Comments on Scheduling Parallel Iterative methods on Multiprocessor Systems"
Parallel Computing Vol. 7, N. 1, 1988
avec Y. ROBERT

"An orthogonal systolic array for the Algebraic Path Problem"
Computing 39, 1987
avec Y. ROBERT

"Parallel implementation of the APP"
Proceedings of CONPAR - Springer Verlag, 1986
avec Y. ROBERT

"Mise en oeuvre systolique de projections"
Traitement du Signal Vol. 4, N. 1, 1987
avec P. COMON et Y. ROBERT

"Parallel conjugate gradient algorithm with local decomposition"
Rapport de Recherches IMAG I-689, 1987
avec P. LAURENT-GENGOUX

1.UNE INTRODUCTION AU PARALLELISME

Présentation générale des différentes formes de parallélisme. Description des architectures parallèles.

1.1. LE BESOIN EN PUISSANCE DE CALCUL

Aujourd'hui, la demande en puissance de calcul est sans cesse croissante dans le domaine des applications scientifiques. On mesure la vitesse d'un algorithme sur un ordinateur donné, en millions d'opérations qu'il est capable d'effectuer sur des nombres flottants en une seconde (en MFlops pour Mega FLoating point OPerations per Second). Les puissances actuelles sont loin de satisfaire la demande. En effet, la complexité du monde s'accroit, ainsi que la perception que l'on en a. Ainsi, les modèles mathématiques pour représenter la réalité se compliquent et il faudrait pouvoir intégrer de plus en plus de paramètres pour rendre compte précisément des phénomènes qui nous entourent. Ce facteur est confirmé par l'importance considérable que prend la simulation numérique dans les sciences de l'ingénieur.

Actuellement, les plus gros ordinateurs dont on dispose ont une puissance réelle de quelques dizaines de MFlops, dans le meilleur des cas (les constructeurs annoncent des puissances bien supérieures, mais il s'agit de résultats de pointe réalisés dans les cas les plus favorables avec une parfaite utilisation de toutes les ressources comme les pipelines ou la répartition entre les processeurs, ce problème est discuté dans [Her] par exemple). De nombreux travaux font mention de l'état de l'art actuel sur les calculateurs les plus performants, ils s'appuient sur des chiffres donnés par les constructeurs ou mesurés par des chercheurs, citons entre autres articles, ceux de [Comp], [Her], [Len], [MC], [OV], [Qui].

Les ordinateurs actuels utilisent largement une forme ou une autre de parallélisme, ils atteignent des puissances de plusieurs MFlops. Parmi les constructeurs les plus connus, citons : CRAY (avec l'ordinateur le plus célèbre au monde : le CRAY 1, la gamme modèle XMP et maintenant le CRAY 2), Control Data (Cyber 205, ETA), IBM qui se lance dans la course aux super calculateurs scientifiques avec le 3090 VF multi processeur, les constructeurs japonais (FUJITSU, HITACHI), NEC (SX), la "Connection machine", les divers "Cosmic Cube", les hypercubes de INTEL (les iPSC) et de FPS (la gamme des T) etc.. Sans compter les puissances considérables disponibles sur de plus petits systèmes pour un prix plus "raisonnable" (ALLIANT, CONVEX etc.). La modularité des machines comme l'Alliant peut permettre des puissances très élevées pour peu que l'on associe plusieurs modules comme dans le projet CEDAR développé à l'université de l'Illinois [GKLS].

L'avenir laisse présager, pour les prochaines années, des performances encore supérieures. Les architectures sous jacentes sont, pour la plupart, encore au stade du développement, même si certaines versions commerciales sont déjà disponibles. On peut espérer atteindre prochainement des puissances effectives de l'ordre du GFlops.

Quelques domaines concernés

De très nombreux domaines sont concernés par la demande en puissance de calcul. L'éventail des problèmes numériques très voraces en temps calcul est très large. Par exemple, en mécanique des milieux continus, la modélisation des phénomènes physiques conduit à des masses de calcul considérables.

Les puissances actuelles sont loin de répondre à ces besoins grandissant.

1.2. LES SOLUTIONS POSSIBLES

Face a cette demande considérable que nous venons d'évoquer, les solutions sont de trois natures pour arriver à des puissances de calcul convenables.

D'une part, 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.
De plus, les progrès constants de l'algorithmique ont également contribué beaucoup à l'amélioration des temps d'exécution.
Enfin, une des voies les plus prometteuses pour l'avenir vient de la remise en question de l'architecture classique des ordinateurs, avec principalement l'introduction du concept de "parallélisme".

Nous détaillons ci-dessous ces trois volets. Toutes ces solutions peuvent être combinées pour être utilisées simultanément.

1.2.1. AMELIORATIONS TECHNOLOGIQUES

La révolution des circuits intégrés.
A la base de toute opération de l'ordinateur, il y a les portes logiques, matérialisées par des transistors. Dans les années 1970, on assiste à un développement considérable de la technologie des circuits intégrés au Silicium. Les performances d'un ordinateur sont liées à la vitesse de ces composants. Principalement, on distingue deux critères pour obtenir une bonne rapidité :

Premièrement, la vitesse de commutation du transistor, qui permet de passer rapidement d'un état à l'autre. On peut l'améliorer en diminuant la taille du transistor afin de réduire le temps de charge.
D'autre part, la mobilité des électrons (caractéristique physique du matériau utilisé, elle est essentiellement liée à la nature du dopant et à sa concentration) permet également d'augmenter la vitesse de commutation. La vitesse de transfert peut augmenter si l'on réduit la taille des connexions entre les transistors sur un même composant.

Ces deux critères peuvent être satisfaits ensemble en augmentant la densité des circuits intégrés. Aujourd'hui, les composants basées sur la technologie au Silicium atteignent des tailles de l'ordre du micron, et la densité d'intégration est à peu près multipliée par un facteur deux tous les ans (selon [MC]). Gagner de la vitesse revient à réduire la grille du transistor (l'électrode centrale). On la réduira encore, sans nul doute, mais le plus significatif reste d'essayer

d'augmenter la mobilité des électrons dans les transistors, en changeant le cas échéant de technologie.

Certes, les performances d'un ordinateur sont d'abord liées à la vitesse des composants de base, mais il faut penser aussi à l'organisation interne générale de l'ordinateur. Il faut de plus relier les différents circuits entre eux pour obtenir l'ordinateur complet. Ces connexions sont significatives, on affirme même qu'elles sont à l'origine de près d'un tiers du temps global de traitement. Ce problème semble critique, on peut le palier partiellement grâce aux empilages multicouches par exemple.

Les limitations technologiques

En une nanoseconde (durée actuelle de commutation des transistors), un électron parcourt trente centimètres à la vitesse de la lumière (et en pratique, beaucoup moins dans les matériaux utilisés). La vitesse de la lumière constitue une limite technologique infranchissable. Le seul moyen pour gagner de la vitesse consiste donc à faire des connexions de plus en plus petites entre les composants.

De plus, à partir de certaines vitesses trop élevées, il apparait des effets perturbateurs. La miniaturisation des circuits intégrés semble limitée par divers phénomènes comme la diminution de la taille des motifs qui provient de la première étape de la gravure (lithographie), c'est à dire du transfert du dessin du masque sur la plaquette (il est en effet difficile d'obtenir une résolution inférieure à un micron, longueur du même ordre que longueur d'onde de la lumière utilisée pour l'insolation) ou comme la dissipation thermique qui s'accroit avec la densité des circuits (on observe alors une chute importante des performances des transistors) ou encore comme certains effets parasites (couplages capacitifs etc.).

Autant de raisons pour expliquer une limitation technologique notable du gain en performance des ordinateurs.

Quelques alternatives possibles

Les ordinateurs actuels, basés sur les circuits intégrés au Silicium, exploitent la technologie jusqu'à ses limites. On atteint des niveaux d'intégration très élevés, et les connexions entre circuits sont raccourcies au maximum. Ces acquis sont indispensables pour obtenir des ordinateurs performants, cependant, ils sont contrecarrés par certains phénomènes que nous venons d'évoquer au paragraphe précédent.

Le bilan reste insuffisant. Des progrés conséquents paraissent très difficiles, ils semblent en tout cas loin de pouvoir pallier seuls à la demande considérable en puissance de calcul.

Il existe cependant d'autres axes de recherches pour tenter de trouver d'autres technologies, comme de nouveaux semi-conducteurs, en particulier

l'Arséniure de Gallium (AsGa) ou l'utilisation des propriétés supraconductrices (Effet Josephson) ou encore la création de nouveaux types d'architecture de transistors tels que les transistors à base métallique ou à base perméable en technologie au Silicium.

La technologie AsGa est déjà une réalité, puisque certains composants existants sont déjà basés sur ce principe [Fre]. Elle permet de créer des composants beaucoup plus rapides. Tout se passe comme si la masse d'un électron qui traverse un cristal d'AsGa était inférieure à celle d'un électron dans un cristal de Silicium (ceci vient en fait du nombre de porteurs dans le cristal et de la mobilité des électrons). Donc, dans un champ électrique donné, les électrons dans l'AsGa sont accélérés davantage que dans du Silicium.

Cependant, on peut dire que les problèmes ne sont pas encore tous résolus, en particulier, ces technologies obligent à travailler à des températures très basses pour obtenir des bonnes propriétés (avec tous les problèmes que cela pose). De plus, le prix de revient est nettement plus élevé que dans le cadre de la technologie actuelle au Silicium. Les technologies qui s'appuient sur des matériaux composés sont beaucoup plus compliquées que dans le cas d'un unique constituant comme dans le Silicium. De plus, le Silicium forme très facilement un bon isolant avec l'oxygène (ce qui n'est pas le cas de l'AsGa) qui est indispensable dans les circuits intégrés.

On peut trouver dans [Ive] une description précise des 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 (à titre d'information, IBM a abandonné en 1983 son programme de recherches concernant la construction d'une machine utilisant l'effet Josephson). Les résultats récents dans le domaine des "supracon" (prix Nobel) paraîssent cependant assez encourageants. On parle également d'ordinateurs "optiques" qui pourraient contourner beaucoup de difficultés actuelles [Comp], on peut en effet imaginer des transmissions optiques de l'information qui constitue le goulot d'étranglement des ordinateurs.

On peut noter pour conclure ce paragraphe que le coût des recherches et réalisations dans ces domaines est considérable.

1.2.2. AMELIORATIONS ALGORITHMIQUES

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. En séquentiel, il permettait de gagner un ordre de grandeur sur ce calcul, effectuant la transformation avec une complexité en $O(n\log_2(n))$ contre $O(n^2)$ précédemment. Cet exemple montre combien la recherche algorithmique peut être importante dans la course au gain en vitesse d'exécution. Cependant, des progrès aussi spectaculaires paraissent impossibles.

La recherche sur les méthodes numériques performantes comme la FFT peuvent aider considérablement à résoudre des applications gourmandes en temps calcul. Elle est importante, et même si elle n'apparait pas comme primordiale, elle intervient en complément des autres voies que nous présentons ici.

1.2.3. UTILISATION DU PARALLELISME

L'idée qui préside au parallélisme est simple :
si différentes parties d'un algorithme sont indépendantes, il est possible d'envisager leur résolution simultanément pourvu que l'on dispose de plusieurs unités de traitement. On dit alors que l'on effectue la résolution en *parallèle*. Bien évidemment, il faut être sûr que les parties confiées à chacun des processeurs soient indépendantes, et qu'il n'y aura pas de conflits entre les données qu'elles utilisent.

Ce domaine s'est considérablement développé ces dernières années et selon une enquète récente sera amené à se développer encore (les prévisions de marché représentent plus de 6 milliards de dollars pour 1990 aux Etats Unis), il est de plus complémentaire des améliorations technologiques. C'est sans doute le plus prometteur pour obtenir des puissances élevées à moindre coût. Les architectures parallèles sont déjà une réalité, et ceci depuis fort longtemps comme nous allons le voir rapidement ci-dessous.

Bref historique
Les idées sous-jacentes ne sont pas nouvelles. On les retrouvent déjà dans les systèmes d'exploitation où il faut répartir des ressources entre plusieurs utilisateurs.

Le premier ordinateur parallèle disponible fut l'ILLIAC IV avec 64 processeurs [Bar], il était opérationnel dès le début des années 70 à la NASA. CDC livre en 1974 un STAR-100 au laboratoire Lawrence Livermore, le premier CRAY apparait en 1976 à Los Alamos, l'université de Carnegie Melon crée le Cmmp. A cette époque, les instructions était identiques sur chacun des

processeurs, seules les données différaient. Un peu plus tard, la firme Denelcor commercialise le premier ordinateur MIMD, le HEP (il avait alors 16 processeurs indépendants).

La plupart des ordinateurs qui voient le jour actuellement utilisent ce concept de répartition du travail en tâches indépendantes sur des unités séparées, que le contrôle soit global ou local. De nombreux articles et travaux permettent d'une part de rendre compte de l'état de l'art, mais aussi de montrer le considérable intérêt de ce domaine de recherche très prometteur [Agr], [Fly], [HJ], [HB], [Mir], [OV] etc..

On distingue plusieurs manières de traiter des processus en parallèle, chacune s'adaptant à certains types de problèmes.

Tout d'abord, il faut se poser la question de savoir comment connecter les processeurs entre eux. Il est également important de savoir comment est organisée la mémoire. Enfin, savoir combien de processeurs doivent être associés en parallèle.

Les recherches actuelles investissent à peu près toutes les combinaisons possibles. Certaines limites technologiques constituent un obstacle à une forme ou une autre. Par exemple, il est impossible de relier totalement tous les processeurs entre eux s'ils sont trop nombreux. Deux philosophies complémentaires semblent ainsi se dégager : d'une part, le développement de gros systèmes multi processeurs très puissants, et d'autre part, de plus petits systèmes utilisant des ressources en commun avec d'autres utilisateurs. Ces derniers systèmes décentralisés suivent une approche distribuée [Ray].

Comment connecter les processeurs entre eux ?

Les principales voies investies à l'heure actuelle sont les suivantes :

On peut connecter les processeurs en anneaux, où chaque noeud est en relation avec 2 voisins. La mémoire peut être seulement locale ou les processeurs peuvent partager une mémoire commune.

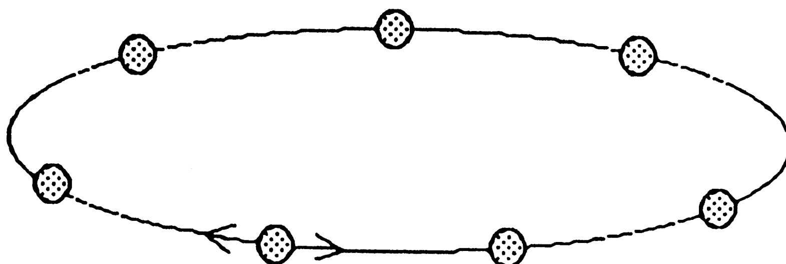


figure 1.1. connexion en anneau

Les processeurs peuvent appartenir à une grille régulière, orthogonale ou hexagonale. Ils peuvent également former un réseau complet, chaque processeur étant connecté à tous les autres. Cette solution devient impossible dans le cas de processeurs puissants dès que leur nombre augmente trop.

Un bon compromis semble être de choisir une topologie en hypercube. Dans un hypercube à 2^n sommets, chaque processeur est en relation avec n voisins. Ceci limite la longueur des chemins à parcourir pour relier 2 processeurs quelconques (cette solution permet d'imaginer des architectures à nombre assez élevé de processeurs). En ce sens, c'est meilleur qu'une topologie en anneaux à n processeurs où le chemin entre 2 processeurs quelconques peut nécessiter jusqu'à n/2 transferts. On trouvera dans [JH] des discussions détaillées sur ce sujet. Notons pour finir qu'en réorganisant un hypercube, on peut obtenir un anneau.

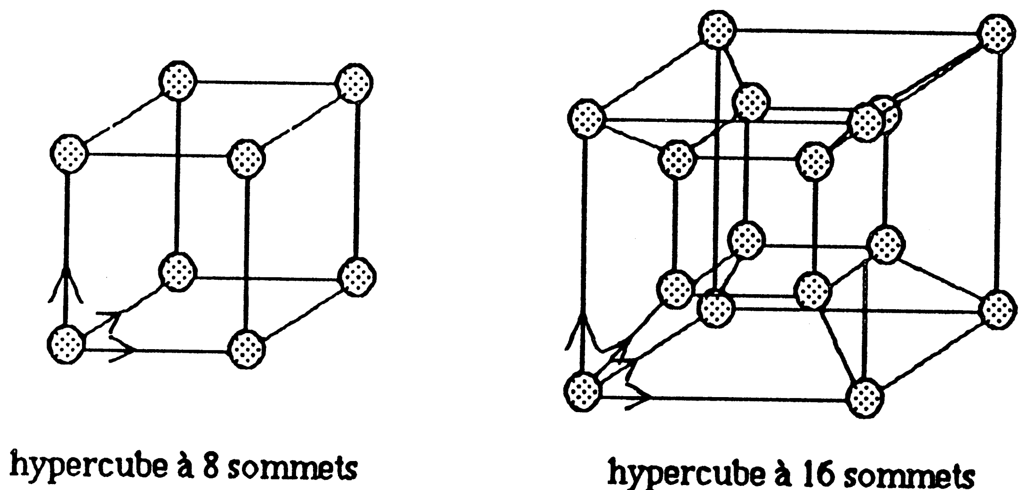


figure 1.2. connexion en hypercube

On peut imaginer toutes sortes d'autres relations entre les processeurs. Citons en particulier, la "Finite Element Machine" de la NASA avec ses 36 micro processeurs 16-bits, chaque noeud étant relié à ses 8 voisins les plus proches, plus un bus commun à tous. L'architecture MIDAS du laboratoire Lawrence Berkeley, organisée en arbre de processeurs. Le "Configurable Highly-Parallel Computer", avec son réseau de communication programmable et reconfigurable, on retrouve la même idée dans l'anneau reconfigurable du ICAP2 d'IBM-Yorktown etc. Il est à noter que cette "connectique" est un des points importants de la recherche sur le parallélisme. En effet, il ne sert à rien de chercher à obtenir des gains conséquents sur chaque processeur si l'on perd trop de temps dans les connexions externes !

Comment est organisée la mémoire ?

Dans tous les ordinateurs, les accés à la mémoire centrale constituent le talon d'Achille de l'architecture, c'est un goulot d'étranglement pour la manipulation des données. Les solutions sont nombreuses pour palier ce problème, citons principalement :

diminuer le nombre total des accés à la mémoire centrale (machines orientées "registres", accés associatifs etc..)
intercaler une mémoire locale (mémoire cache)
permettre les accés parallèles.

La mémoire peut être globale, partagée par tous les processeurs. Le passage d'un banc mémoire à l'autre se fait alors par l'intermédiaire d'un réseau d'interconnexion. Deux méthodes principales peuvent être utilisées pour relier des processeurs à la mémoire :

Les réseaux locaux qui sont plus particulièrement dédiés aux applications nécessitant des transferts peu fréquents d'informations (mais parfois très volumineux).

Les réseaux d'interconnexion utilisés pour des échanges fréquents de faibles quantités d'informations. La première solution qui vient à l'esprit consiste à relier les processeurs à la mémoire par un bus en temps partagé, mais les trop faibles débits des bus limitent rapidement le nombre de processeurs. On peut alors imaginer une topologie en grille orthogonale reliant chaque processeur à chaque banc mémoire. Ce type de réseau (cross-bar) assure le débit maximal, mais en pratique il est trop coûteux car le nombre de points de croisement croît très vite (il est proportionnel au produit du nombre de processeurs par le nombre de bancs mémoire). Entre ces deux extrêmes, il existe plusieurs solutions intermédiaires possibles (réseau de Benes, réseau Oméga ..) qui sont détaillées dans [AB], [LM] par exemple.

La mémoire peut être hiérarchisée (c'est-à-dire organisée en plusieurs niveaux de mémoires de plus en plus lentes à mesure que l'on s'éloigne du processeur).

Ou encore, la mémoire peut être complétement locale, chaque processeur ne pouvant accéder qu'aux données de ses voisins directs, comme dans les hypercubes à grands nombres de processeurs.

Il est clair que l'organisation de la mémoire influence l'écriture des algorithmes.

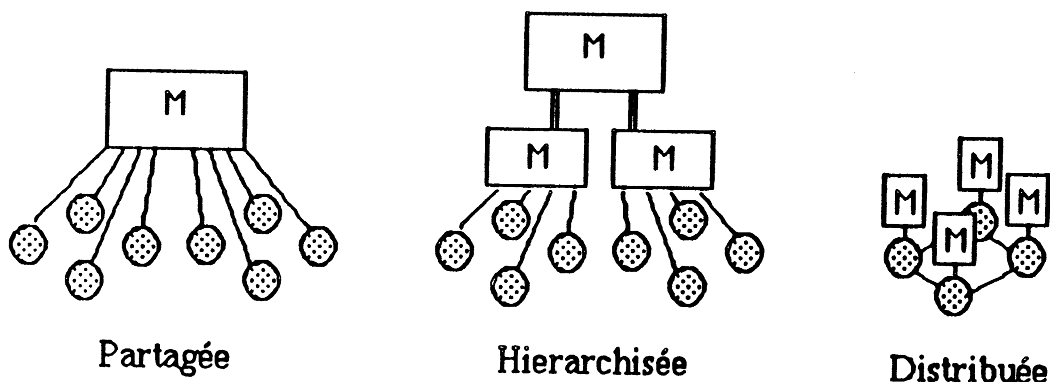


figure 1.3. organisation de la mémoire

Combien de processeurs doit on associer en parallèle ?
On peut connecter un petit nombre de processeurs puissants ou bien beaucoup de petits processeurs élémentaires.

Programmer sur les machines parallèles nécessite de réarranger les calculs pour exploiter au mieux l'architecture. Cela entraine une remise en question complète des algorithmes séquentiels classiques et même de notre manière de penser. Il est nécessaire d'avoir à notre disposition des outils pour permettre l'analyse des algorithmes sur ces types d'architectures parallèles. En particulier, l'analyse de la complexité (c'est-à-dire le décompte des opérations élémentaires que requiert un algorithme parallèle) va permettre de mesurer l'efficacité des algorithmes et de les comparer.

Dans le paragraphe suivant, nous détaillons les caractéristiques principales des différents types d'architectures parallèles.

1.3. LES ARCHITECTURES PARALLELES

Comme nous venons de l'évoquer, il existe plusieurs styles d'architectures parallèles, chacune appropriée à des applications particulières. 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 [Fly], [Sto], distinguant flots de données et flots d'instructions. Cependant, on peut aussi faire intervenir d'autres critères comme la manière de connecter les processeurs entre eux, ou étudier le couplage, les réseaux de connexion, les processus de synchronisation etc.

2.1.1. TENTATIVE DE CLASSIFICATION

Pipeline

Notons tout d'abord que le parallélisme peut intervenir à un niveau très élémentaire. La technique du "pipeline" (bitoduc, en français !) consiste à anticiper sur la lecture ou l'écriture de données pendant l'exécution de certaines instructions (addition par exemple) [Kog]. C'est en quelque sorte du parallélisme à un niveau très bas qui nécessite un amorçage et un vidage (dans le CYBER 205, il faut manipuler des vecteurs d'au moins 200 éléments pour atteindre la moitié de la vitesse de crête du pipe). Les opérations possibles sont figées une fois pour toute, cependant le pipeline a l'avantage d'être peu coûteux, car il ne nécessite pas la duplication de toutes les ressources. De plus, il est compatible (et complémentaire) avec les autres formes de parallélisme que nous détaillons ci-dessous.

Architecture de type SIMD (Single Instruction Multiple Data).

Ici, les processeurs peuvent intervenir complétement indépendemment. Ils sont chargés d'effectuer le même traitement sur des données différentes. L'ordinateur opère sur des vecteurs, mais l'approche est presque celle du séquentiel. Les manipulations simultanées sur des vecteurs deviennent alors des instructions élémentaires. Les premiers ordinateurs SIMD sont apparus il y a déjà longtemps (ILLIAC IV, STARAN, DAP etc..). En général, les instructions, opérateurs fonctionnels et données sont pipelinés pour permettre directement la manipulation de tableaux. On parle alors d'ordinateurs vectoriels.

Il est facile de gérer beaucoup de cellules élémentaires (approche de type "réseaux systoliques" [Kun1]) par un flot unique d'instructions. Dans ce dernier cas, l'initiative laissée aux processeurs est faible, et les machines restent dédiées à des types d'applications assez spécifiques.

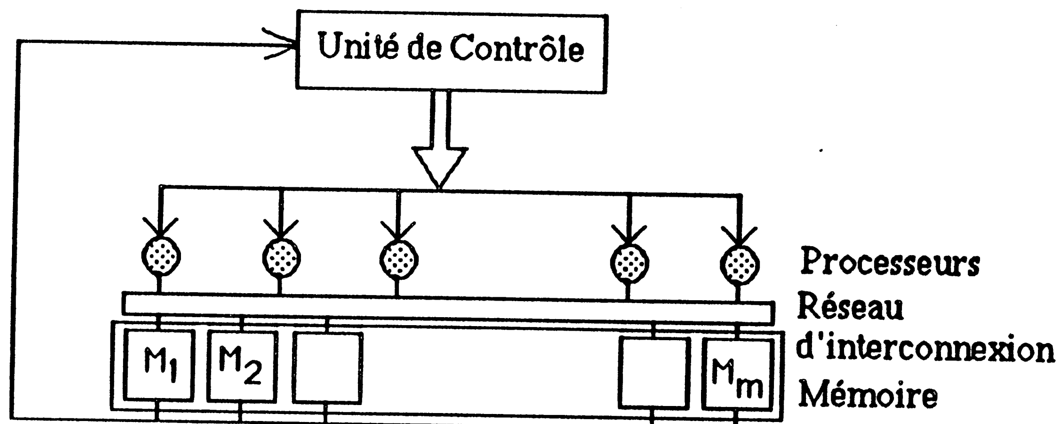


figure 1.4. Architecture SIMD

Notons pour terminer que ce type d'ordinateur est un peu tombé en désuétude. Pratiquement, il se limite aux "Array processors", unités spécialisées de calcul sur des tableaux que l'on adjoint à un processeur hôte ou des ordinateurs parallèles comme la "Connection Machine".

Architecture de type MIMD (Multiple Instructions Multiple Data).
Contrairement à précédemment, les processeurs peuvent être chargés d'exécuter des parties de code différentes. L'architecture se compose de plusieurs unités, d'une mémoire, d'un réseau d'interconnexion pour pouvoir passer d'une unité de contrôle à une autre ou d'un banc mémoire à un autre et enfin d'un réseau d'interruptions. On dit qu'un ordinateur MIMD est fortement couplé si les interactions entre les processeurs sont importantes. A titre d'exemple, citons le HEP de Denelcor, le CRAY 2 etc..

Les algorithmes sont alors asynchrones, l'architecture comporte un réseau qui gère les interruptions des processeurs. Pour les machines à mémoire partagée, le nombre de processeurs connectés reste assez faible. On peut envisager d'augmenter ce nombre en utilisant des mémoires hierarchisées ou distribuées.

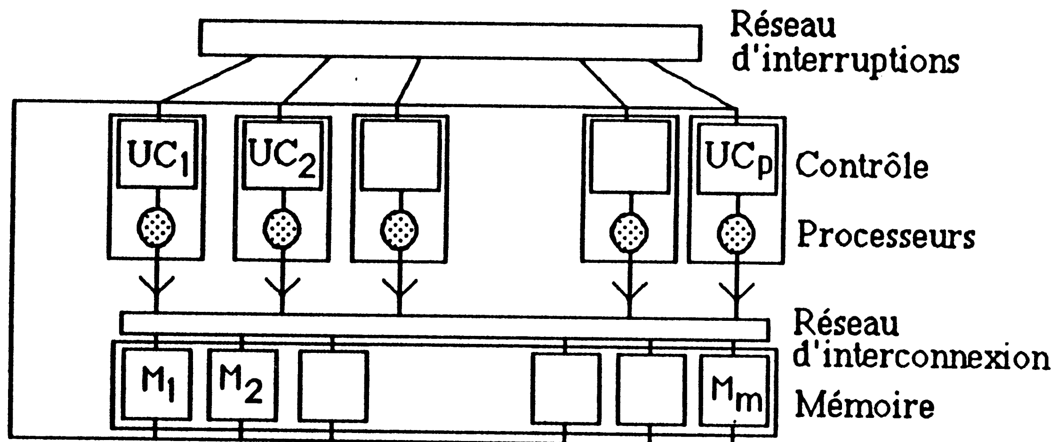


figure 1.5. Architecture MIMD

Cette classification est la plus ancienne et la plus connue, cependant elle est insuffisante. En effet, comment distinguer une machine MIMD avec et sans pipeline, faiblement ou fortement couplée ? Qu'est-ce qu'une machine MISD ? Où classer les architectures distribuées ? etc..

Plusieurs extensions ont été proposées depuis. On peut envisager d'autres critères tels que le nombre d'unités de traitement, l'organisation d'une mémoire commune ou de "l'Operating System", la longueur des mots, les intéractions matérielles ou logicielles à chaque niveau, les interruptions etc.. Ces problèmes sont discutés dans [Ens].

2. ANALYSE DE LA COMPLEXITE

Méthodologie d'étude des algorithmes parallèles. Définition des outils de base, étude d'exemples simples.

Les ordinateurs actuels utilisent largement le concept de parallélisme. Cette approche nous oblige à revoir l'algorithmique séquentielle classique, principalement de manière à éviter les conflits sur les données. On distingue plusieurs manières de traiter des processus en parallèle, chacune s'adaptant à certains types particuliers de problèmes. Le problème de l'architecture parallèle a été abordé au chapitre précédent, nous nous concentrons ici sur l'étude des méthodes d'analyse de la complexité sur des architectures MIMD à mémoire partagée.

2.1. DESCRIPTION DU MODELE DE REFERENCE

L'architecture parallèle qui nous intéresse ici, est de type MIMD fortement couplée à mémoire partagée. Elle est idéalisée. En effet, pour évaluer la vitesse d'un algorithme parallèle, il est d'usage de négliger les temps de transfert des données pour ne retenir que l'arithmétique. Cependant, des travaux récents proposent des modèles de complexité plus généraux qui permet de prendre en compte les temps de communication et de se rapprocher ainsi de la réalité, nous développerons ce point dans un chapitre ultérieur.

Le système est supposé capable de supporter des flots d'instructions multiples s'exécutant indépendamment et en parallèle sur plusieurs flots de données. Ceci suppose évidemment d'avoir bien défini les contraintes de chronologie entre les tâches dans l'algorithme (en d'autres termes, d'avoir réglé les problèmes de synchronisation). Nous négligeons les temps de communication nécessaires pour le transfert des données ou des résultats [CRT1]. Chaque processeur peut exécuter n'importe laquelle des quatre opérations arithmétiques élémentaires dans le même temps, temps que nous prendrons comme unité de référence dans le décompte final.

Nous supposons un environnement de programmation du type Fortran, c'est-à-dire où les éléments des matrices sont accédés par colonnes. Seulement deux types de transfert sont autorisés : chargement d'une colonne entière de la matrice depuis la mémoire centrale jusqu'au processeur et stockage dans l'autre sens, de la colonne dans la mémoire. Nous supposons également que la duplication d'une colonne a un coût nul, ce qui nous permet d'envisager le transfert simultané d'une même colonne vers plusieurs processeurs. Dans ce cas, aucun processeur n'est autorisé à modifier cette donnée. D'autre part, un processeur ne peut modifier une donnée que si elle réside dans sa propre mémoire.

Quelques remarques sur la validité du modèle

Un modèle dans lequel les temps de synchronisation et de transfert sont négligés peut paraître a priori insuffisant. Cependant, les méthodes que nous étudions pour des applications scientifiques (essentiellement les méthodes fondamentales de l'algèbre linéaire avec coefficients numériques ou non) reposent sur des opérations vectorielles. En conséquence, il est raisonnable de supposer que l'accés aux données est pipeliné, ainsi dans le cas où les données sont accédées par colonnes (dans un environnement de type Fortran par exemple), le transfert d'une même colonne vers tous les processeurs est simultané.

De plus, d'après la construction même du graphe de précédence, il est possible en première approximation, d'inclure des temps de communication dans le décompte des temps d'exécution. En effet, dans la plupart des algorithmes numériques, le nombre de données stockées ou chargées est à peu près proportionnel au nombre d'opérations flottantes effectuées. Nous verrons de plus au chapitre 5 comment définir un modèle général permettant de tenir compte des temps de communication dans l'analyse de la complexité.

2.2. LES OUTILS D'ANALYSE

Programmer sur une architecture parallèle donnée nécessite un réarrangement des calculs pour exploiter au mieux les caractéristiques de cette machine. Cela entraine une remise en question complète des algorithmes séquentiels classiques. Il est nécessaire d'avoir à notre disposition des outils pour permettre l'analyse des algorithmes sur ces nouveaux types d'architectures. En particulier, l'analyse de la complexité (c'est-à-dire le décompte des opérations élémentaires que requiert un algorithme parallèle sur différents processeurs) va permettre de mesurer l'efficacité des algorithmes et de les comparer. On dit qu'un algorithme est optimal si son temps d'exécution est minimal parmi tous les algorithmes possibles. Il peut exister plusieurs algorithmes nécessitant des nombres différents de processeurs qui s'exécutent en temps optimal.

Bref historique

Le problème général d'analyse de la complexité des algorithmes a de tout temps intéressé les informaticiens. C'est une mesure absolue des algorithmes qui permet de les comparer entre eux et de confronter les résultats pratiques à la théorie. Il est nécessaire de pouvoir se référer à un algorithme optimal (le meilleur pour la classe considérée). De plus, elle permet de définir des outils théoriques nouveaux et d'être utilisée comme aide à l'intuition dans la recherche d'autres algorithmes.

Les premiers résultats concernant les algorithmes parallèles datent des années 70. Ils reposaient sur des modèles de structures de type SIMD-MIMD ayant un nombre infini de processeurs, partageant une mémoire centrale et négligeant les temps de communication entre mémoire et unités de traitement. Cette approche traditionnelle est motivée principalement par le désir d'explorer les potentialités maximales du parallélisme. Elle semble aujourd'hui limitée et tombe quelque peu en désuétude au profit d'une analyse prenant plus en compte les particularités des architectures parallèles (systolique, architectures distribuées, faible de nombre de processeurs etc..). Dans un premier temps, nous allons présenter l'approche classique, puis nous discuterons les faiblesses de ce modèle et proposerons des solutions pour l'améliorer.

2.2.1. QUELQUES DEFINITIONS FONDAMENTALES

Notons p le nombre de processeurs. Nous supposerons ce nombre infini, cependant, on peut montrer que pour un problème d'algèbre linéaire de dimension n, il est limité par O(n). On montre en effet que dans le cas contraire, le coût des communications prédomine sur l'arithmétique [Saa]. Plus précisément, on pose :

$p = \alpha n$, avec $\alpha \leq 1$.

Il est commode d'utiliser la notion d'efficacité pour mesurer les performances d'un algorithme parallèle. On la définit comme le rapport du temps séquentiel (noté τ_{seq}) sur le temps d'exécution à p processeurs (τ_p), divisé par le nombre de processeurs. En clair,

$$e_p = \tau_{seq} / p\tau_p$$

L'efficacité est évidemment comprise entre 0 et 1 (il est impossible d'accélérer un algorithme parallèle avec p processeurs d'un facteur supérieur à p). Elle joue le rôle d'un rendement et permet de mesurer le taux moyen d'occupation des processeurs.

Toutes ces définitions sont usuelles, on trouvera des détails et précisions dans [Cos], [Hel], [Sam], [Sch].

Graphe de précédence

L'analyse de la complexité repose sur la notion de graphe de précédence des tâches [CD], [CRT1], [LKK]. Intuitivement, elle permet d'introduire un formalisme qui sous-tend l'étude des algorithmes parallèles, elle est décrite en détail dans [CRT1] ou [LKK] par exemple. Nous n'en donnons ici que la démarche essentielle.

La méthodologie d'étude des algorithmes parallèles, comporte principalement trois étapes.

Un algorithme est un processus fini, segmenté en un ensemble de tâches élémentaires. Une tâche étant une instruction (ou un bloc d'instructions) entièrement définie par son comportement extérieur, entrées, sorties et temps d'exécution. Comme nous l'avons évoqué dans l'introduction, le temps d'exécution d'une tâche élémentaire sera de durée de l'ordre de n de manière à ce que l'arithmétique prédomine sur le temps de transfert des données et sur le temps de synchronisation.

Ces tâches sont liées entre elles par des contraintes temporelles de chronologie. On définit pour les expliciter une relation d'ordre entre les tâches :

$T \ll T'$ signifie que la tâche T' ne peut commencer que lorsque T est terminée.

Cette étape se modélise commodément à l'aide d'un graphe (appelé "graphe de précédence") où les sommets représentent les tâches et les arêtes, les contraintes temporelles.

La dernière étape consiste alors à trouver un ordonnancement pour affecter au mieux les tâches aux processeurs, en respectant les contraintes de chronologie du graphe de précédence et les contraintes matérielles liées à l'architecture de la machine (l'accés limité aux données et les problèmes dûs à la synchronisation des

processeurs). Cette étape est sans aucun doute la plus délicate, elle permet en particulier de sélectionner parmi toutes les versions parallèles obtenues, la plus efficace au regard de l'ordinateur de référence.

Une même méthode peut conduire à plusieurs versions différentes en parallèle, suivant le type d'accés aux données, le découpage initial en tâches, l'affectation aux processeurs ou leur synchronisation. Certaines versions sont mieux adaptées à telle ou telle structure d'ordinateur. Un exemple complet est donné plus loin comme illustration.

2.2.2. POSITION DU PROBLEME

Le problème fondamental de la parallélisation d'un algorithme est de rechercher le degré maximal de parallélisme qu'il renferme.

La question est de trouver un ordonnancement optimal pour affecter les tâches aux processeurs en respect avec le graphe de précédence, de telle façon que l'algorithme parallèle s'exécute en temps minimal (noté τ_{opt}), et dans une phase ultérieure, déterminer le nombre minimum de processeurs nécessaires pour effectuer l'algorithme en τ_{opt}.

L'unité de temps est égale au temps d'exécution d'une des quatre opérations arithmétiques élémentaires.

Il faut de plus que l'algorithme d'ordonnancement trouvé soit descriptible et implantable le cas échéant sur une machine existante (ou au moins simulable). Ceci élimine les procédures trop compliquées et les opérations supplémentaires de prétraitement sur les données, qu'il est souvent difficile de mesurer. Enfin, étant donné un nombre p fixé de processeurs, on doit être en mesure de trouver un algorithme optimal s'exécutant avec p processeurs.

On pose $p=\alpha n$. Les résultats asymptotiques sont obtenus quand n tend vers l'infini, en particulier, l'efficacité asymptotique $e_{\infty,\alpha}$ est définie comme la limite de e_p quand $n \to \infty$. On dit qu'un algorithme est asymptotiquement optimal quand $e_{\infty,\alpha}$ est maximum.

Une analyse asymptotique peut paraître a priori contradictoire avec le problème pratique lié à un nombre fixé, faible, de processeurs, elle peut sembler insuffisante. Comme nous l'avons déjà souligné, notre propos est d'étudier le degré de parallélisme maximal d'un algorithme donné avec un nombre infini de processeurs sur une architecture idéalisée. Les résultats obtenus sont généraux et théoriques. Au

sur une architecture idéalisée. Les résultats obtenus sont généraux et théoriques. Au chapitre 5 nous montrerons comment les utiliser dans la pratique. En particulier, pour la version kji modifiée de l'élimination de Gauss en parallèle qui est décrite plus loin, le graphe obtenu est de type glouton [CMa1] dont on connait un algorithme d'ordonnancement asymptotiquement optimal. Cet algorithme est simple à décrire, mais il existe ponctuellement une meilleure répartition des processeurs pour une taille de matrice et un nombre de processeurs donnés que celle de l'ordonnancement glouton, pourtant asymptotiquement optimal. Cette répartition est particulière au problème, à dimension et nombre de processeurs fixés, elle n'enlève rien à l'intérêt général de l'ordonnacement Glouton.

Cet exemple n'est pas unique, il est possible de trouver, au cas par cas et pour une taille donnée (pas trop grande en général), un ordonnancement meilleur. Le problème de la description formelle générale se pose pour ce genre d'algorithmes dont il n'existe pas de stratégies constructives systématiques (à notre connaissance !).

Notre but est de concevoir des algorithmes d'ordonnancement généraux.

Le problème général d'un ordonnancement quelconque des tâches est NP-complet. En ce qui concerne les problèmes numériques que nous nous proposons de résoudre ici, il existe certaines régularités dans le passage d'un niveau du graphe de précédence à un autre et dans l'expression du temps d'exécution d'une tâche au temps de la suivante dans un niveau donné.

De nombreux résultats théoriques ont été obtenus dans le cas d'algorithmes ayant des tâches de même temps d'exécution (pour deux processeurs et pour les graphes qui sont des arbres). Les principaux sont résumés dans [Cos].

2.3. ILLUSTRATION DE LA METHODOLOGIE SUR QUELQUES EXEMPLES SIMPLES

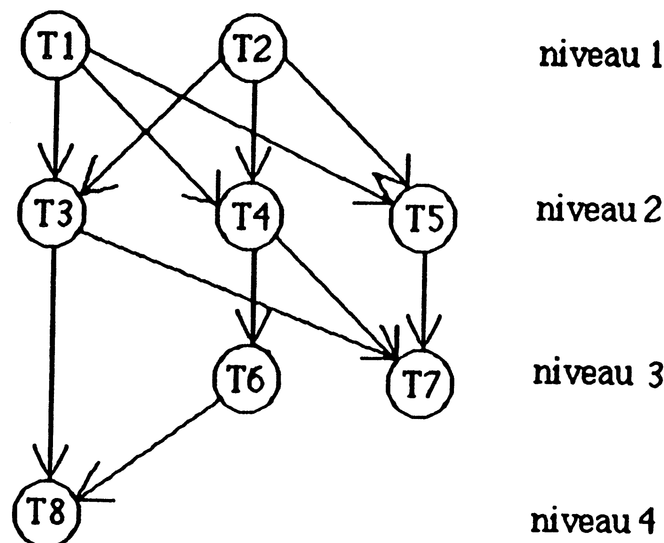
Etude d'un algorithme simple

Le premier exemple devenu un classique du genre, est tiré de [LKK]. Il illustre la notion de graphe des tâches et d'algorithme d'ordonnancement.

Soit un algorithme dont les huit tâches élémentaires, de mêmes durées d'exécution (que nous prendrons pour simplifier, égales à 1 unité), sont liées par les contraintes de précédence suivantes :

T1<<T3 T1<<T4 T1<<T5 T1<<T8
T2<<T3 T2<<T4 T2<<T5 T2<<T7
T3<<T7 T3<<T8
T4<<T6 T4<<T7 T4<<T8
T5<<T7
T6<<T8

On obtient le graphe de précédence, après élimination des contraintes redondantes. Cette représentation suit implicitement l'ordre naturel de la décomposition par prédécesseurs.



Cherchons la réponse aux questions fondamentales que nous avons présentées précédemment.

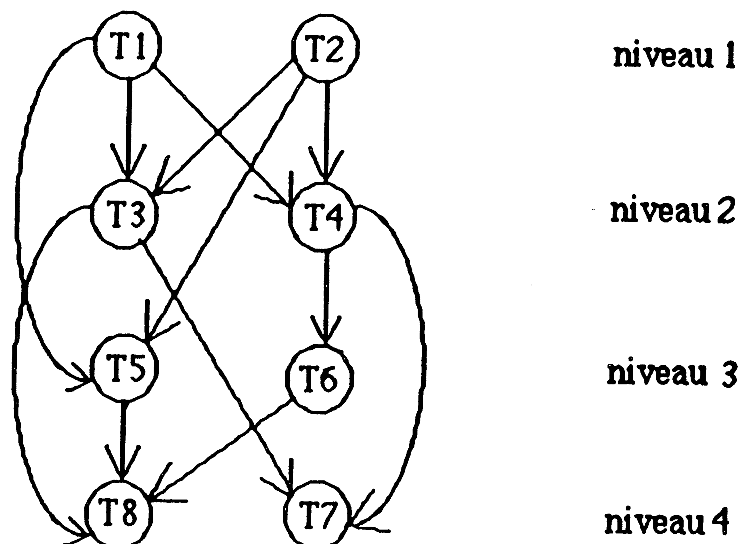
1) Quel est le temps τ_{opt}, d'un algorithme optimal ?

C'est le chemin le plus long du graphe, soit trivialement $\tau_{opt} = 4$.

2) Trouver un ordonnancement qui réalise l'algorithme en temps τ_{opt}, et si possible avec le minimum de processeurs.

Là encore, la réponse n'est pas difficile. On peut par exemple affecter les tâches niveaux par niveaux aux processeurs, auquel cas, 3 processeurs conduisent à une solution. Le calcul rapide de l'efficacité conduit à $e_3=8/(3.4)=2/3$.

Cependant, une autre décomposition est possible, elle fait apparaitre directement la largeur du graphe. Nous l'avons représentée graphiquement ci-dessous.



On remarque qu'il est possible d'obtenir τ_{opt} avec seulement 2 processeurs, en procédant niveau par niveau.

Le calcul de l'efficacité est trivial :

$$e_2 = \tau_{seq} / 2\, \tau_{opt}, \text{ soit } e_2 = 1$$

Ceci est confirmé par la remarque que les processeurs sont toujours tous utilisés.

Compliquons un peu la règle du jeu : Imaginons que les tâches n'aient plus toutes la même durée d'exécution. La recherche du temps optimal n'est plus aussi simple, il faut connaître le temps de tous les chemins possibles. De même, la répartition des tâches aux processeurs est obtenue en passant en revue toutes les possibilités pour

chaque valeur du nombre de processeurs. On pourra toujours trouver la solution dans le cas de l'algorithme précédent à 8 tâches, mais le problème devient impossible à résoudre pour un problème quelconque avec un trop grand nombre de tâches.

Un exemple en dimension n :
le produit de matrices

La multiplication de matrices est une procédure que l'on utilise à la base de la résolution de nombreux problèmes. Elle est intrinséquement parallélisable et va nous servir à illustrer l'analyse de la complexité.

Le calcul du produit A des matrices B et C de tailles n par n, s'écrit :

$$A_{ij} = \Sigma_{k=1,n} B_{ik} * C_{kj} \quad \text{pour } i,j = 1, 2, .., n$$

L'algorithme nécessite l'emploi de trois boucles, suivant chacun des indices i,j et k. Selon l'accés aux éléments de la matrice, des permutations sont possibles pour réarranger ces trois boucles. La manière d'accéder aux éléments conditionne l'algorithme sur une architecture parallèle donnée. Les six permutations possibles sont les suivantes :

```
c   version 1, ordre ijk
    pour i = 1,n faire
        pour j = 1,n faire
            Aij = 0
            pour k = 1,n faire
                Aij = Aij + Bik* Ckj

c   version 2, ordre jik
    pour j = 1,n faire
        pour i = 1,n faire
            Aij = 0
            pour k = 1,n faire
                Aij = Aij + Bik* Ckj

c   version 3, ordre kij
    pour i = 1,n faire
        pour j = 1,n faire
            Aij = 0
    pour k = 1,n faire
        pour i = 1,n faire
```

```
            pour j = 1,n faire
                A_ij = A_ij + B_ik * C_kj

c   version 4, ordre kji
    pour j = 1,n faire
        pour i = 1,n faire
            A_ij = 0
    pour k = 1,n faire
        pour j = 1,n faire
            pour i = 1,n faire
                A_ij = A_ij + B_ik * C_kj

c   version 5, ordre ikj
    pour i = 1,n faire
        pour j = 1,n faire
            A_ij = 0
        pour k = 1,n faire
            pour j = 1,n faire
                A_ij = A_ij + B_ik * C_kj

c   version 6, ordre jki
    pour j = 1,n faire
        pour i = 1,n faire
            A_ij = 0
        pour k = 1,n faire
            pour i = 1,n faire
                A_ij = A_ij + B_ik * C_kj
```

Tous ces algorithmes exécutent le même calcul. Cependant, dès qu'on les implémente pratiquement sur une machine parallèle donnée, les performances peuvent varier grandement. Par exemple, dans un environnement de type Fortran les matrices sont accédées par lignes, ce qui élimine d'office certaines versions [DGK]. Parmi les versions restantes, certaines sont équivalentes. On peut les regrouper en deux classes selon le type des calculs effectués dans la boucle la plus interne :

```
c   somme avec accumulation - DOT
    pour m = 1,n faire
        S = S + B_m * C_m
```

c somme "dynamique" - SAXPY
pour m = 1,n **faire**
$S_m = S_m + B_m * C_m$

Ces deux versions sont conceptuellement différentes, elles conduisent à deux graphes distincts. L'analyse de complexité est très simple à cause de la grande régularité des tâches élémentaires.

2.4. UN MOT DES ALGORITHMES ITERATIFS

Le modèle que nous venons de présenter ne permet pas de traiter le cas d'algorithmes comportant des instructions conditionnelles (en d'autre termes, notre méthodologie n'accepte pas les circuits dans le graphe de précédence). Tant que nous restons dans les algorithmes de base de l'algèbre linéaire comme le produit de matrices ou les méthodes directes (diagonalisation de Jordan, Elimination de Gauss, Rotations de Givens ..), ceci n'est pas génant.

Cependant, les algorithmes itératifs sont un cas particulier très utilisé d'algorithmes qui comportent des conditionnelles. La forme générale est la suivante.

pour j = 1,... **faire**
calculer un produit matrice-vecteur
résoudre un système élémentaire
si convergence **alors** arret

Il y a deux manières d'envisager la parallélisation [Kro].

Parallélisation à l'intérieur d'une itération

D'une part, on peut voir un algorithme itératif comme un traitement synchrone où l'on parallélise le contenu d'une itération. C'est le cas de la plupart des solutions proposées, et en particulier pour l'algorithme du gradient conjugué que nous étudions au chapitre 7. Il suffit de répartir la charge des processeurs le plus égalitairement possible. Nous avons développé des résultats concernant la méthode de Gauss-Seidel en utilisant cette approche [RT4].

On cherche à résoudre le système Ax=b, M désigne la partie triangulaire inférieure de A et N sa partie triangulaire supérieure stricte.

c Gauss-Seidel

pour j = 1,... **faire**

calculer le vecteur b' = b + Nx

résoudre le système triangulaire Mx = b'

si convergence **alors** arret

On décompose l'itération courante en 2 étapes : le calcul du second membre (qui peut être trivialement parallélisé avec une efficacité 1) et la résolution du système triangulaire. Pour ce dernier, le graphe de précédence est de type "2-pas" avec des valeurs particulières pour le temps d'exécution des tâches élémentaires [RT4]. Nous traitons en détail un exemple de parallélisation de l'algorithme itératif du gradient conjugué dans le chapitre 7 avec ce type de technique.

Parallélisation sur plusieurs itérations

D'autre part, on peut voir l'algorithme comme un vaste processus asynchrone. Par exemple, il est possible d'anticiper la résolution de l'itération suivante avant que l'itération courante ne soit complétement terminée dès qu'un processeur a terminé. On rencontre cette situation dans le cas où le travail à effectuer à l'intérieur d'une itération est mal réparti.

Cette technique est très difficile à mettre en oeuvre en pratique pour un algorithme quelconque, cependant on peut imaginer la même idée que précédemment : une répartition sur un nombre fixe d'itérations (2 ou 3 par exemple). On retrouve les méthodes de réductions cycliques ou comme dans le cas de l'algorithme du gradient conjugué, lorsqu'il existe une écriture possible sur 2 pas, presque "naturelle" (on retombe alors sur la méthode semi-itérative de Tchebytchev [GVL]).

Dans ce dernier cas, pour une matrice quelconque, une autre idée peut venir à l'esprit : on calcule des solutions intermédiaires sur chaque processeur et on tient compte de ces valeurs dès que les calculs sont terminés. C'est une sorte de relaxation asynchrone sur le temps d'exécution des tâches. La difficulté est de prouver théoriquement la convergence de ces "nouvelles" méthodes itératives.

3. GRAPHE DE PRECEDENCE

Illustration du graphe de précédence sur la diagonalisation de Jordan et l'Elimination de Gauss. On obtient les graphes Glouton et 2-pas.

Les algorithmes fondamentaux de l'algèbre linéaire sont à la base de la plupart des problèmes numériques qu'ont à résoudre les ingénieurs. Leur implantation sur des architectures parallèles doit donc être effectuée le plus efficacement possible, de plus leur étude est d'une grande importance théorique dans la constitution des méthodes d'analyse de la complexité. Nous faisons référence dans ce chapitre aux travaux [CRT1], [CMaR1] menés en commun avec M. Cosnard, M. Marrakchi et Y. Robert.

3.1. DIAGONALISATION DE JORDAN EN PARALLELE

Supposons que l'on veuille résoudre un système linéaire dense Ax = b de taille n.

Le principe de la diagonalisation de Jordan est de transformer la matrice initiale A en une matrice diagonale D. On procède par prémultiplications successives de manière à éliminer les éléments hors diagonaux des colonnes. La forme naturelle consiste à écrire les n étapes k suivantes : pour toutes les colonnes à partir de k, préparer l'élément pivot pour l'élimination et enfin éliminer sur toutes les lignes. En pratique, on inclut le vecteur second membre b dans A (qui devient une matrice (A,b) de dimension n par (n+1)). L'algorithme correspondant est donné ci-dessous :

```
c    Jordan - forme kji
     pour k = 1,n faire
          pour j = k+1,n+1 faire
c              exécuter la tâche T_kj
               a_kj = - a_kj / a_kk
               pour i = 1,n faire  i≠k
                  a_ij = a_ij +  a_ik * a_kj ;
```

Toutes les tâches T_{kj} (pour $j \geq k+1$) s'exécutent dans le même temps : 2n-1. On peut y inclure les temps d'accés aux données, 2(n-1)+2 chargements et n-1 stockages.

Les contraintes de précédence sont de 2 types. D'une part il faut qu'un niveau soit terminé pour passer au suivant et d'autre part que le pivot soit calculé à un niveau donné pour pouvoir effectuer les éliminations sur ce niveau. Soit,

$T_{kj} << T_{k+1,j}$ pour tout $j>k$

$T_{k,k+1} << T_{kj}$ pour tout $j>k+1$

Le graphe de précédence pour n=6 a l'allure suivante :

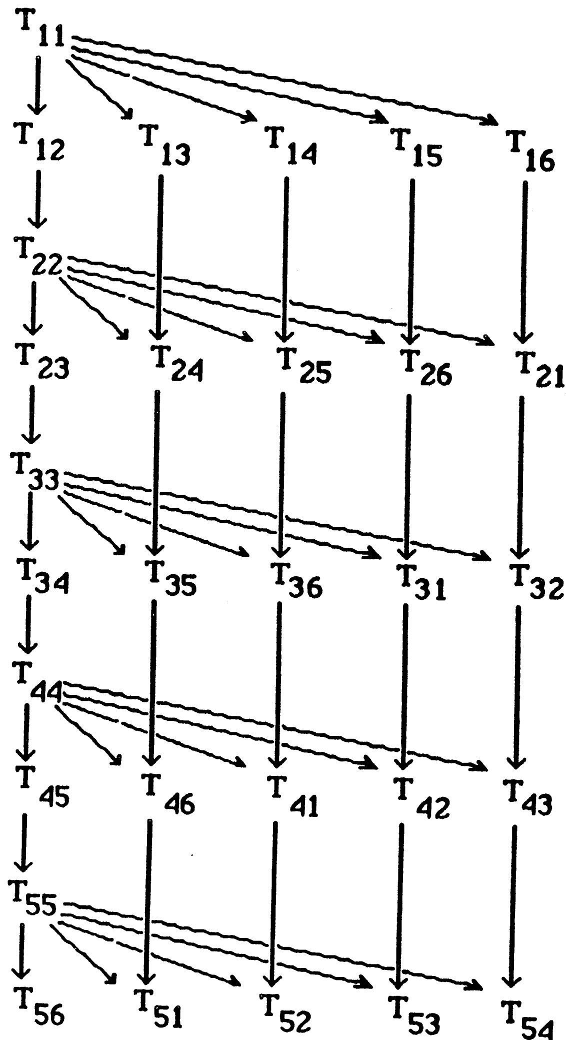


figure 3.1. Graphe de Jordan par colonnes pour n=6

Cette écriture présuppose que l'on accéde aux éléments de A par colonnes. Si l'on autorise un accés aux données par lignes, l'algorithme séquentiel s'écrit alors :

```
c    Jordan - forme kij
     pour k = 1,n faire
c         exécuter la tâche T_kk
          c = 1 / a_kk
          pour j = k+1,n+1 faire
               a_kj = a_kj * c
          pour i = 1,n faire  i≠k
c              exécuter la tâche T_ki
               pour j = k+1,n+1 faire
                  a_ij = a_ij -  a_ik * a_kj ;
```

Le temps de la tâche T_{kk} est de n-k+2 unités, celui de T_{ki} de 2(n-k+1).

Les contraintes de précédence imposent à T_{kk} d'être exécutée avant les T_{ki} qui doivent être exécutées avant que le niveau niveau suivant ne commence, c'est-à-dire avant le début des tâches $T_{k+1,i}$. En résumé :

$T_{kk} << T_{ki}$ pour tout $i \neq k$

$T_{ki} << T_{k+1,i}$ pour tout $i \neq k$

Ce qui conduit au graphe de précédence suivant (donné pour n=6) :

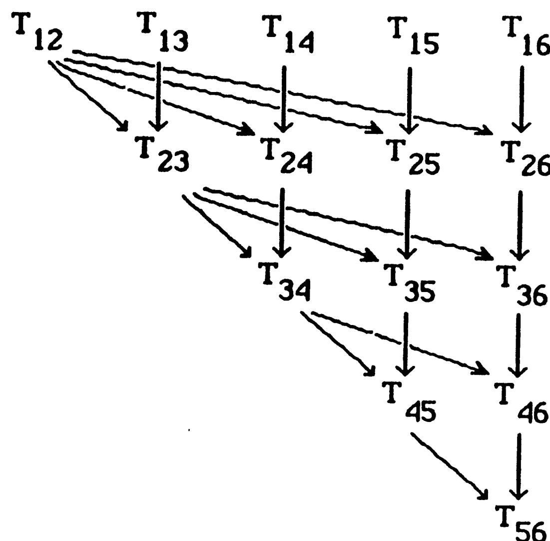


figure 3.2. Graphe de Jordan par lignes pour n=6

3.2. L'ELIMINATION DE GAUSS

L'ensemble des résultats que nous présentons dans ce chapitre est tiré essentiellement de [CMa1]. La parallélisation de l'algorithme d'élimination de Gauss se fait suivant le processus décrit au chapitre précédent : on définit la décomposition en tâches élémentaires et on construit le graphe de précédence associé. On obtient au total dix versions différentes, suivant l'accés aux données et la variante considérée. Ces graphes peuvent être regroupés en quatre catégories : les graphes triangulaires, les graphes à 2 pas, les doubles graphes triangulaires et doubles 2-pas. L'analyse de la complexité de ces derniers se déduit facilement de celle des graphes triangulaires et 2-pas, aussi ne présentons nous que les résultats concernant les deux premiers types de graphes. Notons pour finir que ces graphes se retrouvent dans bien d'autres problèmes d'algèbre linéaire, comme dans les rotations de Givens pour la factorisation QR d'une matrice ou encore dans le problème du chemin algébrique (que nous aborderons au chapitre 6 dans un cadre systolique) etc..

L'élimination de Gauss est un algorithme bien connu [GVL]. En général, un pivotage est nécessaire pour accroître la stabilité numérique de ces méthodes. Il est possible de prendre en compte le surcroît de calcul engendré par cette opération en augmentant par exemple la tâche diagonale T_{kk} de la procédure de recherche du maximum et de l'échange des éléments.

3.2.1. LES DIFFERENTES ECRITURES DE GAUSS EN PARALLELE

Les principales versions que l'on peut obtenir à partir de la méthode de Gauss ont été décrites dans [DGK] dans le cas d'une architecture vectorielle. Le même travail a été effectué dans [CMa1] sur les machines parallèles, elles sont les suivantes.

```
c     forme "naturelle " kji - SAXPY simple (sans pivotage)
      pour k = 1,n-1 faire
c          exécuter la tâche Tkk
           pour i = k+1,n faire
                akj = - akj / akk
           pour j = k+1,n faire
c               exécuter la tâche Tkj
                pour i = k+1,n faire
                     aij = aij + aik * akj ;
```

La tâche T_{kk} dure $t_k = (n-k)$ unités de temps et T_{kj} dure $2t_k$ unités de temps.

Pour la même version avec pivotage partiel, étudiée dans [MR1] ou [RT5] par exemple, l'algorithme est le suivant :

```
c    forme kji - avec pivotage
     pour k = 1,n-1 faire
c         exécuter la tâche T_kk
          chercher_pivot (k,p)
          échanger (a_kk,a_pk)
          pour i = k+1,n faire
               a_kj = - a_ik / a_kk
          pour j = k+1,n faire
c              exécuter la tâche T_kj
               échanger (a_kj,a_pj)
               pour i = k+1,n faire
                  a_ij = a_ij + a_ik * a_kj ;
```

Si comme dans [LKK] ou [MR1], on suppose que le coût d'une multiplication suivie d'une addition est égal au coût d'une multiplication et d'une comparaison, la tâche T_{kk} dure t_k = (n-k+1) unités de temps et T_{kj} dure (n-k) unités de temps.

Les contraintes de chronologie entre les tâches sont détaillées dans [CMa1]. Tous les algorithmes précédents sont des variantes du même type de graphe de précédence. La version suivante est obtenue en échangeant les boucles d'indices i et j dans la forme SAXPY usuelle, elle conduit à un graphe de nature différente :

```
c    forme kji - SAXPY modifiée
     pour k = 1,n-1 faire
          pour j = k+1,n faire
c              exécuter la tâche T_kj
               a_kj = - a_kj / a_kk
               pour i = k+1,n faire
                  a_ij = a_ij + a_ik * a_kj ;
```

Les tâches T_{kj} durent $t_k = 2(n-k)+1$ unités de temps.

3.2.2. FORMULATION GENERALE DES GRAPHES

Les différents algorithmes que nous venons de présenter peuvent être regroupés dans les deux formulations générales suivantes :

c Graphe triangulaire général

```
pour k = 1,n-1 faire
    pour j = k+1,n faire
        exécuter la tâche T_kj
```

les tâches T_{kj} coûtent t_k unités de temps. Les contraintes de précédence sont de deux types :

à k fixé, $T_{k,k+1} << T_{k+1,j}$ pour tout $j \geq k+2$
à k fixé, $T_{kj} << T_{k+1,j}$ pour tout $j \geq k+2$

Le graphe de précédence associé est le suivant :

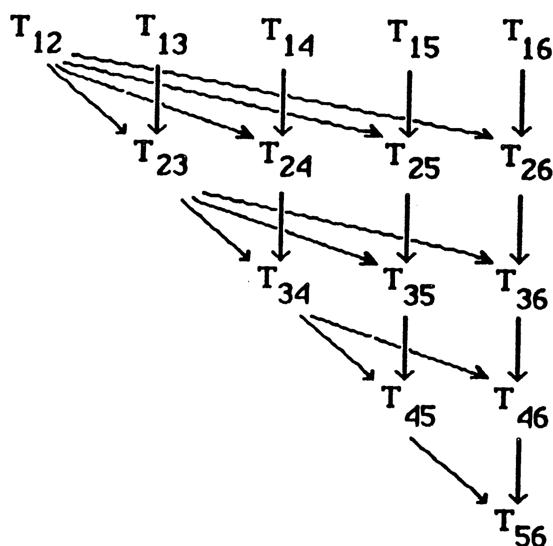


figure 3.3. Graphe triangulaire pour n=6

c Graphe 2-pas général

```
pour k = 1,n-1 faire
    executer la tâche T_kk
    pour j = k+1,n faire
        exécuter la tâche T_kj
```

où la tâche T_{kk} coûte at_k unités de temps et T_{kj} coûte bt_k (a et b sont des rationnels).

Les contraintes de précédence sont de deux types :

à k fixé, $T_{kk} << T_{kj}$ pour tout $j \geq k$
à k fixé, $T_{kj} << T_{k+1\,j}$ pour tout $j \geq k$

dont le graphe est représenté ci-dessous.

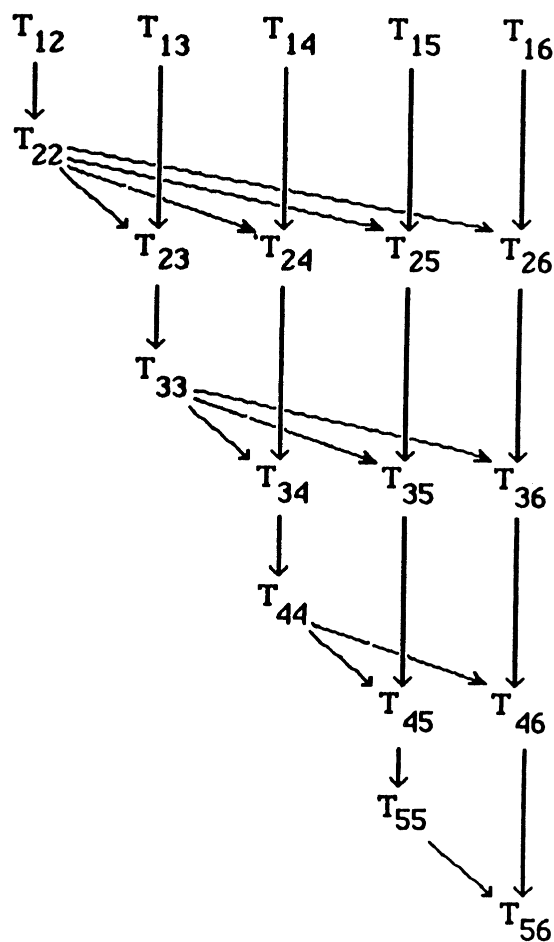


figure 3.4. Graphe "2-pas" pour n=6

Nous allons étudier maintenant les algorithmes d'ordonnancement généraux pour chacun de ces deux types de graphes.

4. ORDONNANCEMENT

Nous présentons l'affectation des tâches aux processeurs pour les graphes de type glouton et à 2-pas. Nous donnons des ordonnancements optimaux pour ces deux graphes.

Le problème difficile de l'ordonnancement des tâches aux processeurs est abordé dans ce chapitre sur les exemples des graphes triangulaires et des graphes à 2-pas. Les résultats de ce chapitre ont été établis en collaboration avec M. Cosnard, M. Marrakchi et Y. Robert [CMa1] pour la première partie et J.C. Konig et Y. Robert [KRT], [RT5] pour la deuxième partie. Ils concernent l'élimination de Gauss.

Etant donnés un nombre de processeurs et un graphe de précédence (qui rend compte des contraintes de chronologie liant les tâches dans un algorithme), il faut trouver le meilleur ordonnancement possible qui permette d'affecter les tâches aux processeurs, en respectant les contraintes. De nombreux auteurs se sont posés ce problème général de répartition des ressources sur un nombre donné de postes, citons parmi les plus référencés les travaux de Coffman [Cof], Coffman et Denning [CD], Graham [Gra], Kohler [Kho]. Certains travaux (dont ceux de Ulman [Ulm]) suggèrent même que le problème général n'a pas de solutions systématiques.

En ce qui concerne l'application de ces techniques à l'étude de la complexité des méthodes de l'algèbre linéaire, on peut citer (outre les notres !) les travaux de [BP], [Kho], [Kum], [LKK], [Mis], [Vel] etc..

4.1. GRAPHE GLOUTON

L'algorithme "Glouton" correspond à un ordonnancement du graphe triangulaire précédent dont la forme générale est donnée ci-dessous :

```
pour k = 1, n-1 faire
    pour j = k+1, n faire
        1. préparer une transformation en utilisant la colonne k
        call PREP (..., k, ...)
        2. appliquer cette transformation à la colonne j
        call APPL (..., k, j, ...)
```

Supposons que le temps d'exécution des tâches T_{kj} soit égal à t_k unités de temps. Les contraintes de précédence sont :

(A) : $T_{k,k+1} << T_{k+1,j}$ pour $1 \leq k \leq n-1$, $k+2 \leq j \leq n$

(B) : $T_{k,j} << T_{k+1,j}$ pour $1 \leq k \leq n-1$, $k+1 \leq j \leq n$

4.1.1. PRESENTATION INFORMELLE DU GLOUTON

Le graphe de précédence triangulaire comporte (n-1) niveaux, le niveau k se compose de l'ensemble des tâches T_{kj} à k fixé. L'algorithme glouton exécute les tâches niveau par niveau et de gauche à droite sur chaque niveau. Il effectue à tout moment le maximum de tâches possible (d'où le nom de glouton). Plus précisément, l'algorithme effectue les tâches dans l'ordre suivant :

$$T_{1,2} - T_{1,3} - T_{1,4} - \ldots - T_{1,n} - T_{2,3} - T_{2,4} - \ldots - T_{2,n} - T_{3,4} - \ldots - T_{n-1,n}$$

Au départ, l'algorithme glouton commence l'exécution des p premières tâches T_{12} -T_{13} - ... - T_{1p}. Pour tous les temps ultérieurs, si q processeurs sont libres, il les affecte aux q tâches suivantes sous réserve de vérifier les contraintes de type (B), dans le cas contraire, il en exécute le maximum possible.

4.1.2. OPTIMALITE ASYMPTOTIQUE

L'optimalité asymptotique de l'algorithme glouton repose sur le lemme suivant dont la démonstration complète se trouve dans [CRT3].

Lemme 4.1.
Il existe un algorithme d'ordonnancement optimal qui satisfait à la contrainte supplémentaire :

$$(C) : T_{k,j} \ll T_{k,j+1} \quad \text{pour } j \geq k+1$$

Nous étudions maintenant le résultat de complexité général qui concerne tous les graphes triangulaires dont le temps des tâches est constant par niveau et de l'ordre de n. Nous présentons le théorème suivant avec la version kji modifiée de l'élimination de Gauss, c'est-à-dire pour le cas où le temps est de $t_k=a(n-k)$ unités. a est un paramètre réel positif.

Théorème 4.2.
L'algorithme Glouton est asymptotiquement optimal. Pour un nombre $p=\alpha n$ processeurs, son efficacté est égale à :

$$e_{opt,\alpha} = 1/(1+\alpha^3/2)$$

Preuve.
En utilisant le lemme précédent, il suffit de démontrer que l'algorithme glouton est asymptotiquement optimal dans la classe des algorithmes qui satisfont à la contrainte supplémentaire (C). Soit τ_0, le temps minimum pour arriver à la tâche $T_{n-p,p}$ avec n'importe quel algorithme de la classe précédente. Les contraintes de chronologie exigent que toutes les tâches des niveaux 1 à (n-p) soient effectuées avant la tâche $T_{n-p,p}$, c'est à dire :

$$\tau_0 \geq [\, \Sigma_{k=1,n-p} (n-k).t_k \,]/p \quad \text{avec } t_k = a\,(n-k)$$
$$\tau_0 \geq a\,(2n^3/6 - 2p^3/6)/p + O(n) \quad \text{en posant } p=\alpha n$$
$$\tau_0 \geq 2an^2\,(1-\alpha^3)/(6\alpha) + O(n)$$

La progression des tâches restantes est bloquée par le parcours des tâches de la dernière colonne :

$$T_{n-p+1,n} - T_{n-p+2,n} - \dots - T_{n-1,n}$$

ce qui donne une borne inférieure du temps d'exécution d'un algorithme optimal :

$$\tau_{opt,\alpha} \geq \tau_0 + \sum_{k=n-p+1,n} t_k$$

$$\tau_{opt,\alpha} \geq 2an^2 (1 - \alpha^3) /(6\alpha) + a\, n^2 \alpha^2/2 + O(n)$$

L'algorithme glouton est d'efficacité 1 jusqu'à ce qu'il y ait moins de p tâches indépendantes à effectuer, c'est à dire jusqu'au niveau (n-p). Le temps qui sépare l'algorithme glouton de la borne τ_0 est un facteur linéaire, de l'ordre de O(n). Cela montre que la borne précédente est valable asymptotiquement pour l'algorithme glouton à un facteur linéaire près, le temps est le suivant :

$$\tau_{opt,\alpha} = an^2 (2 + \alpha^3) /(6\alpha) + O(n)$$

En terme d'efficacité $e_{opt,\alpha} = \tau_{seq} / p\, \tau_{opt,\alpha}$, comme $\tau_{seq} = an^3 /3$, on obtient :

$$e_{opt,\alpha} = 1 / (1 + \alpha^3/2)$$

De plus, sous les hypothèses du théorème, la détermination de p_{opt} est triviale.

$$\tau_{opt} = \sum_{k=1,n-1} a(n-k) \quad \text{et} \quad p_{opt} = n-1$$

Remarque.
Les résultats sont asymptotiques. Comme nous l'avons fait remarquer précédemment, il est parfois possible de trouver un ordonnancement meilleur pour une valeur fixée de n et p. G. Rote nous a communiqué un contrexemple qui est consigné dans [Cma1], mais cet ordonnancement ne semble pas être applicable systématiquement pour d'autres valeurs de n et p. De plus, il a été amélioré à son tour par J.C. Konig, en appliquant une heuristique du chemin critique.

4.2. GRAPHE 2-PAS

Nous venons de voir en détail les résultats de complexité concernant les graphes triangulaires. Pour la plupart des méthodes d'élimination, il correspond une version complémentaire (correspondant aux graphes à 2-pas) qui peut s'écrire sous la forme compacte suivante :

pour k = 1, n-1 **faire**
    1. préparer une transformation en utilisant la colonne k
    call PREP (..., k, ...)
    pour j = k+1, n **faire**
        2. appliquer cette transformation à la colonne j
        call APPL (..., k, j, ...)

On note T_{kk} la tâche de préparation et T_{kj} la tâche d'application de la transformation à la colonne j. Dans le cas général, les temps d'exécution sont respectivement a(n-k)+c1 et b(n-k)+c2 où a, b ,c1, c2 sont des paramètres rationnels qui dépendent de la version étudiée. Pour des raisons de simplicité, nous nous plaçons ici dans le cas a=b=1 et c1=c2=0, une étude complète est donnée dans [MR2].

Ce cas correspond par exemple à la version kji de Gauss. Le temps d'exécution de l'algorithme séquentiel est égal à $n^3/3 + O(n^2)$. Le plus long chemin du graphe est composé des tâches :

$$T_{11} - T_{12} - T_{22} - T_{23} - \dots - T_{kk} - T_{k,k+1} - \dots - T_{n-1,n-1} - T_{n-1,n}.$$

Dans ce cas, les tâches diagonales ont même durée que les autres (n-k), la longueur du graphe est $\tau_{opt} = n^2-1$. C'est une borne inférieure du temps d'exécution d'un algorithme d'ordonnancement quelconque.

On cherche les meilleures stratégies possibles pour affecter les tâches aux processeurs.

Quelques idées préliminaires simples

Il est trivial de définir un ordonnancement avec n processeurs en temps τ_{opt} (un processeur s'occupant d'une colonne donnée). L'efficacité est alors 1/3.

Remarquons tout d'abord que nous pouvons isoler la première tâche et l'affecter à l'un quelconque des processeurs.

L'idée première d'un ordonnancement (donnée par Lord, Kowalik et Kumar [LKK]), est d'utiliser une stratégie de type Glouton. On considère globalement deux

phases distinctes de manière à exécuter les tâches niveau par niveau en efficacité 1 maximale, jusqu'à ce qu'il reste moins de p tâches à exécuter (c'est-à-dire jusqu'au niveau n-p). La deuxième phase commence alors, contrainte de suivre le plus long chemin restant. Un ordonnancement de ce type, nous conduit à une première remarque : dans la dernière phase, les processeurs travaillent de moins en moins. Au cours de cette phase, on aurait donc intérêt à garder en réserve des tâches qui ne retarderont pas son temps, au prix d'un autre ordonnancement. Plus précisément ces tâches sont celles dont le plus long chemin restant à parcourir est inférieur à celui correspondant à la tâche $T_{n-p,n-p}$ (à partir de laquelle il reste moins de p tâches à exécuter, où l'on ne peut plus aller en efficacité 1), la première phase étant toujours effectuée en efficacité 1. On regroupe alors les tâches suivant le chemin qui leur reste à parcourir. La figure suivante schématise la différence entre les deux stratégies. Les parties achurées s'exécutent dans le même temps (celui du chemin qui mène de $T_{n-p,n-p}$ à la fin) et les autres parties sont d'efficacité 1.

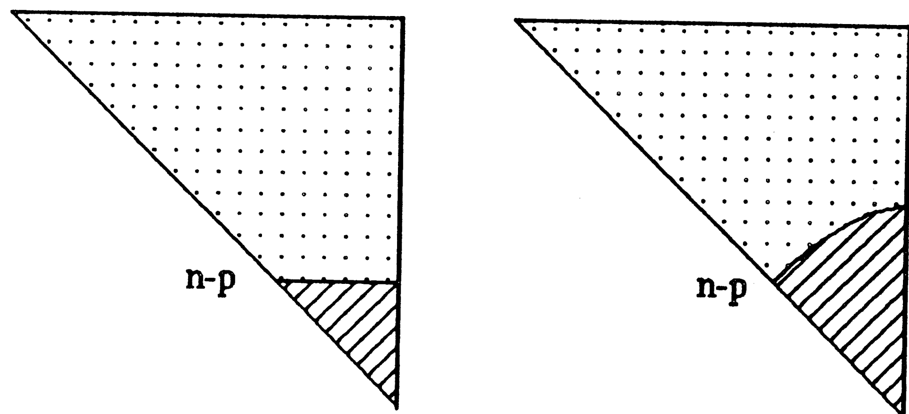


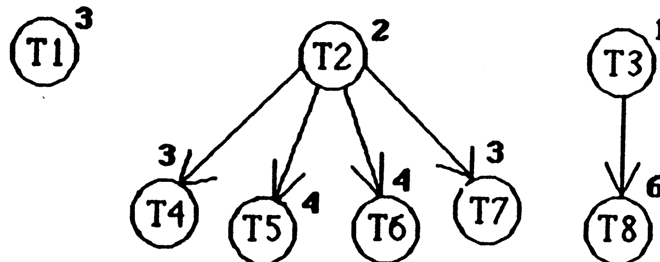
figure 4.1. Schématisation des ordonnancements

4.2.1. OPTIMALITE ASYMPTOTIQUE DU GRAPHE 2-PAS

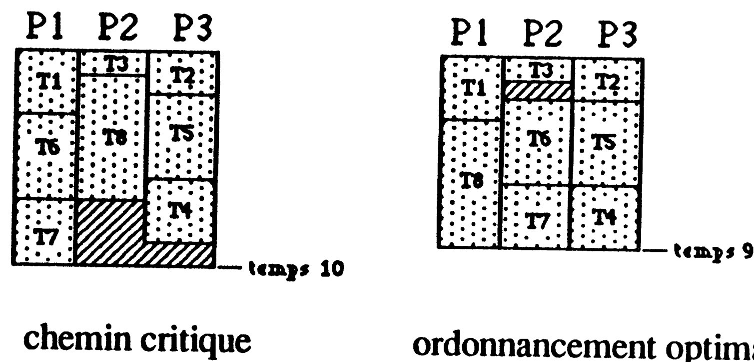
On définit la fonction ch (T) qui, à une tâche T, associe son chemin critique (défini comme le plus long restant à parcourir) et les lignes de niveaux correspondantes qui regroupent, sur chaque colonne, les tâches de même chemin critique (à une constante près). Certains travaux traitent de cet ordonnancement [Koh], mais le résultat l'optimalité est faux pour un graphe quelconque.

L'heuristique du chemin critique

En effet, si l'on considère l'exemple suivant du graphe à 8 tâches à répartir sur 3 processeurs, l'ordonnancement du chemin critique n'est pas le meilleur. Le temps des tâches est indiqué sur la figure.



La répartition des tâches aux processeurs est la suivante :



chemin critique ordonnancement optimal

figure 4.2. Le chemin critique n'est pas optimal en général

On démontre dans le paragraphe suivant, les bornes de l'optimalité asymptotique pour le graphe à 2-pas où le temps des tâches est constant par niveau, égal à n-k.

Théorème 4.3.
Un algorithme asymptotiquement optimal s'exécutant avec $p=\alpha n$ processeurs est d'efficacité inférieure à :

$e_{opt,\alpha} = 1/(1+\alpha^3)$ pour $\alpha \leq \alpha_0$

et $e_{opt,\alpha} = 1/(3\alpha)$ pour $\alpha \geq \alpha_0$

(où α_0 est solution du polynôme $\alpha^3 - 3\alpha + 1 = 0$).

Preuve :
La démonstration suivante peut paraître un peu lourde pour qui connait le résultat basé sur les surface de calcul et surface de repos suivant la méthodologie décrite dans [LKK], mais elle a l'avantage de donner l'intuition de la construction

d'un ordonnancement optimal issu du chemin critique. De plus, nous généralisons le résultat. L'idée de [LKK] est d'introduire les surfaces d'activité SC (somme des temps d'exécution effectifs de chaque processeurs) et d'inactivité forcée SR des processeurs, par référence au temps d'exécution séquentiel (on a en effet la relation $SR+SC=p.\tau_{opt}$), on obtient facilement l'efficacité d'un algorithme asymptotiquement optimal. Cette méthode donne des résultats théoriques et ne conduit pas, en général, à des ordonnancements constructibles.

Imaginons un algorithme, noté (Alg), qui effectue les tâches en deux phases : en efficacité 1 au dessus de la courbe de niveau associée à $T_{n-p,n-p}$ (que nous noterons (C) dans la suite) et en temps du plus long chemin au dessous de (C). Montrons qu'un tel algorithme est optimal.

Soit (Alg'), un algorithme d'ordonnancement quelconque. On considère la courbe (C') de la frontière des tâches effectuées après l'exécution de $T_{n-p,n-p}$. Trois cas peuvent se présenter, ils sont schématisés ci-dessous. On note S_1 (respectivement S_2), la somme des temps séparant les courbes (C) et (C') dans la zone 1 (respectivement dans la zone 2).

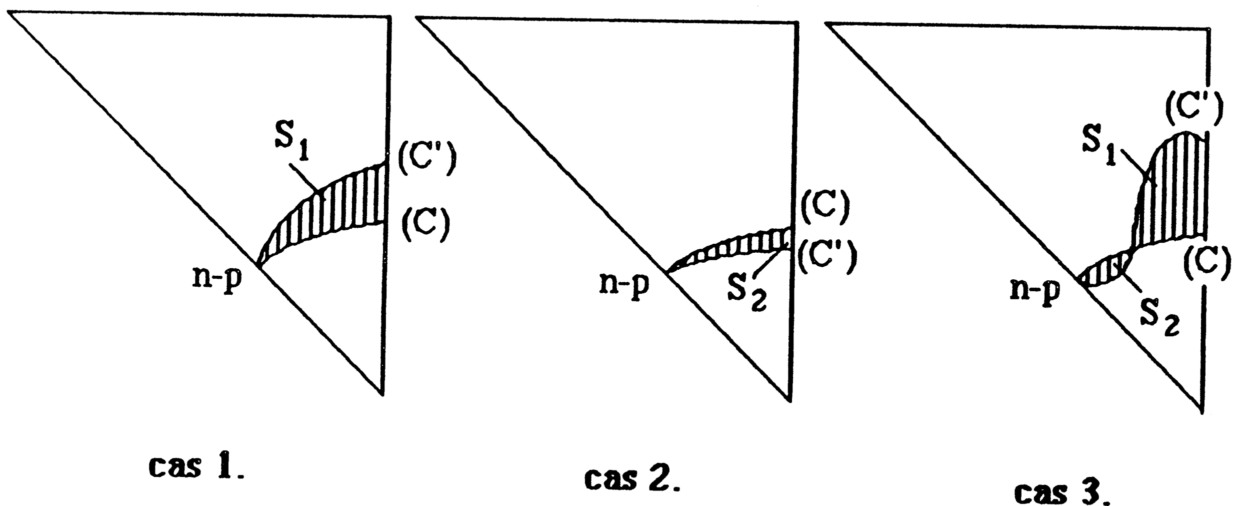


figure 4.3. Comparaison de (Alg) avec un algorithme quelconque

cas 1.

Le gain de (Alg') dans la première phase est inférieur ou égal à S_1/p. Par contre, le temps de la phase 2 est au moins rallongé de la distance maximum pour aller de (C') à (C) qui est minorée par la moyenne S_1/p. Donc, (Alg') est au mieux équivalent à (Alg).

cas 2.

(Alg') est moins bon dans la phase 1 (il y a plus de tâches effectuées) et le temps de la phase 2 est toujours égal à $ch(T_{n-p,n-p})$.

cas 3.

Le gain de (Alg') dans la phase 1 est inférieur ou égal à $(S_1-S_2)/p$. Par contre, la phase 2 est rallongée d'au moins S_1/p comme dans le cas 1. (Alg) est donc meilleur d'au moins S_2/p.

On en déduit les efficacités correspondantes. Le temps de la phase 2 est $ch(T_{n-p,n-p})$.

$$\tau_2 = \Sigma_{i=n-p,n} (n-i) + \Sigma_{i=n-p,n} (n-i) - 1 = p(p+1) - 1$$

Le temps de la phase 1, d'efficacité 1, est obtenu par $1/p\ (\tau_{seq} - \tau_{2\text{-}seq})$.

$$\tau_{2\text{-}seq} = \Sigma_{j=1,p} [\ p(p+1) - j(j+1)\ ] = 2p^3/3 + O(p^2).$$

De plus, le temps séquentiel est $\tau_{seq} = n^3/3 + O(n^2)$, donc la somme des tâches exécutées lors de la première phase est $\tau_{seq} - \tau_{2\text{-}seq}$

soit $\tau_1 \geq n^2 (1 - 2\alpha^3) / (3\alpha) + O(n)$.

ce qui donne : $e_{opt,\alpha} = 1 / (1 + \alpha^3)$

Donc, si un algorithme est optimal, il a cette forme.

Signification de α_0

Le calcul précédent de l'efficacité sous-entendait d'assurer une efficacité 1 au cours de la première phase. La masse totale minimum de travail effectuée durant cette phase ne doit pas excéder la quantité de travail pour atteindre la courbe (C). En d'autres termes, il faut que :

$p\ (p^2 - n^2) \leq \tau_{seq} - \tau_{2\text{-}seq}$ c'est-à-dire

$p\ (1-\alpha^2)\ n^2 \leq (1-2\alpha^3)\ n^3/3$

soit, $\alpha^3 - 3\alpha + 1 \geq 0$

ce polynôme en α est décroissant sur [0,1] et s'annule pour $\alpha_0 = 0.3473$.
Donc, la condition est vraie pour $\alpha \leq \alpha_0$.

Pour $\alpha \geq \alpha_0$, on se trouve à la fin de l'exécution de la tâche $T_{n-p,n-p}$ sous la courbe (C). Le temps parallèle d'exécution de l'algorithme τ_p sera la somme de l'exécution de la phase 1 et du chemin ch (n-p,n-p). Soit au total :

$$ch(1,1) = n^2 + O(n).$$

En passant à l'efficacité, on obtient : $e_{opt,\alpha} = 1 / (3\alpha)$.
Remarquons pour finir que la condition de continuité est vérifiée en $\alpha = \alpha_0$.

Il nous reste à construire de tels ordonnancements.

4.2.2. REVUE DES PRINCIPAUX RESULTATS CONNUS

Il est très facile de donner un algorithme d'ordonnancement avec n processeurs exécutant les tâches dans le temps $\tau_{opt} = 2n-1$. L'efficacité asymptotique est alors $e_{\infty,1} = 1/3$.

Lord, Kowalik et Kumar ont proposé un ordonnancement réalisant τ_{opt} avec seulement n/2 processeurs [LKK]. Son efficacité asymptotique est $e_{\infty,1/2} = 2/3$. Cet algorithme est asymptotiquement optimal en temps. Le principe est de regrouper les tâches par blocs de 2 dans un niveau k donné, de façon à équilibrer le temps des tâches diagonales T_{kk} et $T_{k,k+1}$ avec les tâches du niveau considéré.

Veldhorst [Vel] donne un algorithme d'ordonnancement de même temps d'exécution τ_{opt}, n'utilisant qu'un nombre $p = (\sqrt{2}/4)n$ de processeurs (soit $\alpha \approx 0.354$, borne améliorant le résultat $\alpha = 1/2$ de [LKK] et très proche de la borne optimale théorique $\alpha_0 = 0.3473$ établi dans le paragraphe précédent).

On sait plus généralement qu'il existe des algorithmes parallèles qui s'exécutent en τ_{opt} avec une efficacité $e_{\infty,\alpha} = 1/(3\alpha)$ pour $\alpha \geq \alpha_0$.

Pour les petites valeurs de α, aucun ordonnancement optimal n'est connu. L'optimalité est plus ardue à montrer dans ce cas, pourtant bien plus utile en pratique (les problèmes à traiter comportent en effet de grandes tailles pour des nombres relativement faibles de processeurs).

Les premiers résultats jusqu'alors étaient dûs encore à Lord, Kowalik et Kumar [LKK]. On peut généraliser l'algorithme évoqué précédemment pour $\alpha = 1/2$ à un nombre $p = \alpha n$ plus faible de processeurs, on obtient alors une efficacité :

$$e_{\infty j} = 1/3((j+1)\alpha - 4\alpha^3 \Sigma_{i=1,j}\, i^2) \quad \text{pour } 1/2(j+1) \leq \alpha \leq 1/2j$$

Nous avons donné dans [CMa2] un ordonnancement meilleur, réduisant l'efficacité à $1/(1+4\alpha^3)$ pour $\alpha \leq \alpha_0$. L'idée de cet ordonnancement est de type "Glouton", on procède niveau par niveau jusqu'au niveau n-2p à partir duquel certains processeurs seront inactifs (en effet, un niveau est caractérisé par l'exécution séquentielle des deux tâches $T_{k,k}$ et $T_{k,k+1}$).

Ces résultats ont été améliorés à $1/(1+2\alpha^3)$ dans [MR] pour des valeurs de α inférieures à 1/3.

Nous proposons ci-dessous un ordonnancement asymptotiquement optimal, c'est à dire s'exécutant en efficacité $1/(1+\alpha^3)$ pour $\alpha \leq 0.305$.

4.2.3. ORDONNANCEMENT OPTIMAL

Théorème 4.4.
Pour un problème de taille n avec $p = \alpha n$ processeurs où $\alpha \leq 2/(\sqrt{43}) \approx 0.305$, il existe un algorithme asymptotiquement optimal d'efficacité $e_\alpha = 1/(1+\alpha^3)$.

Décrivons informellement l'algorithme en insistant sur les idées importantes.
Le graphe est partitionné en quatre régions, comme le montre la figure suivante. La borne $(\sqrt{43})/2$ vient de cette division.

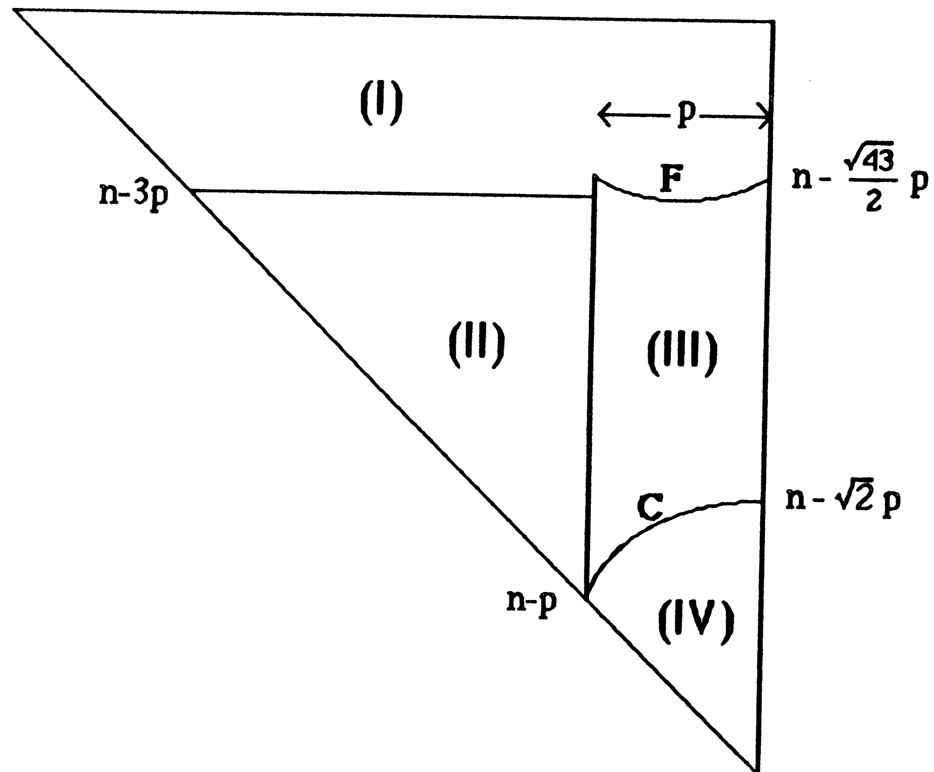


figure 4.5. Schématisation de l'algorithme optimal

L'idée est de garder le maximum de processeurs actifs le plus longtemps possible. Il est clair que dès que l'on atteint la tâche $T_{n-p,n-p}$, il reste moins de p tâches à exécuter en parallèle dans le graphe, et certains processeurs sont nécessairement inactifs. Les tâches diagonales doivent être exécutées séquentiellement. On cherche à maximiser le travail que les processeurs peuvent effectuer durant ce temps, cela correspond à la région (IV) de la figure.

On voudrait pouvoir exécuter les régions (I), (II) et (III) en efficacité maximale. Certes, on pert un peu d'efficacité dans la région (IV), mais on en pert le minimum. En conséquence, l'algorithme sera optimal (pourvu que les contraintes soient vérifiées !).

Description de l'algorithme

On appelle niveau k, la ligne du graphe de précédence composée des tâches $\{T_{k,k+1}, ..., T_{k,n}\}$. La région (I) comprend les tâches des niveaux 1 à n-3p+1. L'explication de la courbe frontière (F) comprise entre les régions (I) et (III) est donnée plus loin.

L'algorithme procède en trois phases séquentielles :

1. Dans la première phase, les tâches de la région (I) sont exécutées niveau par niveau suivant un schéma glouton.

2. La seconde phase concerne les régions (II) et (III). On commence à effectuer les tâches de (II) et dès qu'un processeur se libère, on l'affecte à la région (III).

3. Dans la dernière phase, il suffit d'affecter un processeur à chaque colonne.

La difficulté réside dans la phase 2. Détaillons la :

A partir du niveau n-3p, les processeurs commencent à se libérer, à raison d'un nouveau tous les deux niveaux (le premier processeur au niveau n-3p-2, le k-ième au niveau n-3p+2k). Notons, P le groupe de processeurs restant dans la région (II) et Q celui de la région (III). L'idée est d'affecter un couple processeur k et p-k au pool P ($k \leq p/2$), en colonne n-p+k et n-k, dès qu'il se libère, pendant un temps correspondant à leur inactivité dans (II). Ainsi, le premier processeur reste inactif $8p^2$ (temps pour finir : $9p^2$ moins le temps de la région (IV) : p^2) et le dernier un temps nul. Cela correspond au niveau $n-(\sqrt{17})p$.

Pour un couple de processeurs quelconque, le principe est identique. Le processeur k se libère avant le (p-k)-ième. Il s'occupe alternativement des tâches des colonnes n-p+k et n-k jusqu'au niveau h(k) à partir duquel il reste confiné dans la colonne n-p+k. h est tel que la distance à la courbe (C) est égale au temps libre du processeur p-k. Lorsque ce dernier se libère, il effectue les tâches de la colonne n-k jusqu'à la ligne frontière (C), qu'ils atteignent simultanément, par construction.

Nous avons schématisé le processus ci-dessous.

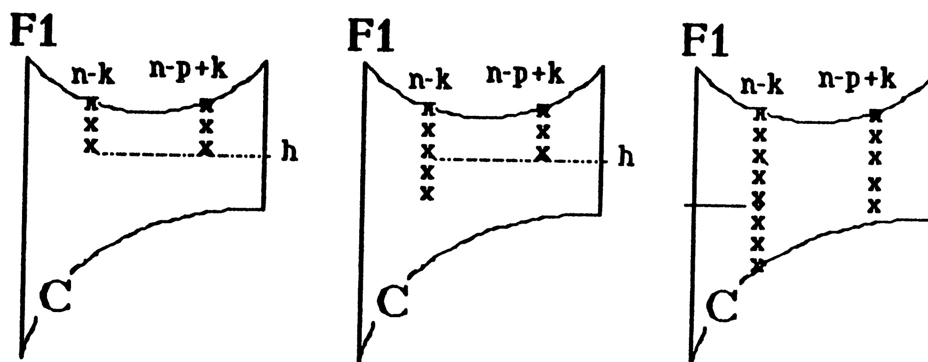


figure 4.6. Activité des processeurs k et p-k sur 2 colonnes :

(1) - processeur k est alternativement sur les 2 colonnes
(2) - il termine sur la colonne n-k à partir de h
(3) - dès que p-k se libère, il termine chacun leur colonne

Cela conduit pour tous les processeurs à la courbe F1 de la figure suivante. Cependant, une partie des tâches (celle correspondant à (S)), située au dessus de cette courbe, n'ont pas été exécutées, il est donc nécessaire de "remonter" cette courbe par une équipotentielle (en temps) comme il est indiqué sur la figure.

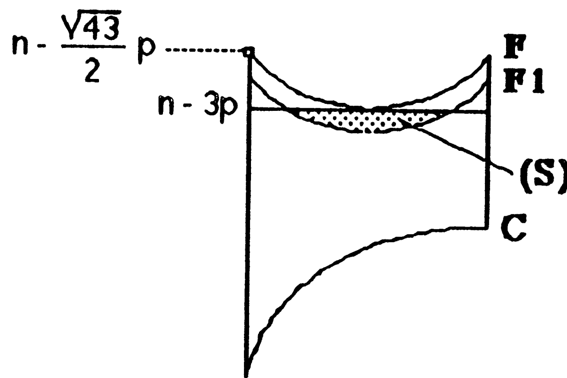


figure 4.7. obtention de la courbe F

Les contraintes de précédence sont bien satisfaites, l'ordonnancement que nous venons de présenter est d'efficacité 1 dans les régions (I), (II) et (III) [RoT4]. De plus, on peut calculer le temps de l'algorithme. On en déduit la valeur de l'efficacité asymptotique, on trouve $e_{\infty,\alpha} = 1/(1+\alpha^3)$.

D'après le théorème d'optimalité 4.3., ces résultats montrent que l'ordonnancement que nous venons de présenter est constructible et asymptotiquement optimal.

On sait donc construire un ordonnancement asymptotiquement optimal du graphe 2-pas pour $\alpha \leq 0.305$ et $\alpha \geq \alpha_0$ (α_0=0.3473).

Le chemin critique pour toute valeur de α ?

Nous conjecturons de plus que dans l'intervalle]0.305,0.3473[, un ordonnancement basé sur l'algorithme du chemin critique (c'est-à-dire qui exécute à tout instant les tâches de plus long chemin restant à parcourir) est aussi asymptotiquement optimal. Nous avons simulé l'efficacité de cet ordonnancement (en calculant son temps) et l'avons comparé à la valeur optimale $1/(1+\alpha^3)$. La courbe suivante confirme expérimentalement cette conjecture.

Nous avons mesuré l'efficacité en faisant varier n à α fixé égal 1/3.

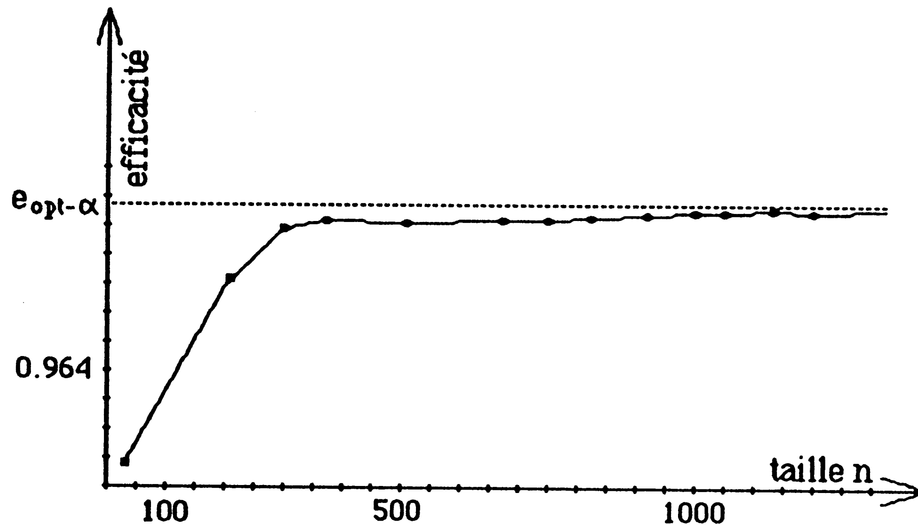


figure 4.8. comparaison de l'efficacité du chemin critique et de la valeur optimale pour α =1/3 en fonction de la taille de la matrice

Cela montre que même pour les faibles valeurs de n, l'ordonnancement selon le chemin critique a un excellent comportement. Malheureusement, il est très difficile d'en dériver des résultats théoriques car on ne sait pas actuellement comment vérifier les contraintes de précédence.

En ce qui concerne l'optimalité dans l'intervalle]0.305,0.3473[, Marrakchi et Robert ont récemment démontré l'optimalité asymptotique [MR2] en utilisant la même idée de découpage en régions que je viens de présenter, mais en se limitant à des frontières horizontales de manière à obtenir la même répartition des processeurs sur toutes les colonnes (il y a alors plus de régions à prendre en compte).

Pour finir, notons que ce type de graphe se retrouve dans l'algorithme de Gauss-Seidel avec un temps des tâches constant. Les résultats précédents s'étendent à cette application [RoT5].

5. PRISE EN COMPTE DES TEMPS DE COMMUNICATION

Tentative d'extension du modèle d'analyse de la complexité en tenant compte des temps de communication.

5.1. ANALYSE GENERALE

Dans ce chapitre, nous reprenons l'analyse de la parallélisation de la version kji modifiée de l'élimination de Gauss, en incluant dans l'étude de complexité algorithmique, le coût des échanges de données entre les processeurs et la mémoire centrale, partie la plus lente de l'architecture. En effet, bien que dépendant de la structure considérée, les coûts de communication ne sont en général pas négligeables par rapport aux coûts des calculs. Nous avons été confrontés à ce problème lors d'un séjour au centre scientifique ECSEC d'IBM à Rome [CRT3]. La comparaison des résultats théoriques de complexité avec les temps expérimentaux a montré une mauvaise adéquation du modèle à la réalité. La raison est qu'à faible granularité des tâches (la notion de "granularité" permet de rendre compte du rapport entre le coût de l'arithmétique interne aux tâches et des transferts de données), les communications ne sont plus négligeables. Une première idée a donc été d'augmenter la granularité, ce qui nous a conduit à envisager des méthodes par blocs.

Nous donnons dans ce chapitre un algorithme d'ordonnancement par blocs, en discutant les stratégies de choix de la taille des blocs (ceci nous conduit à un algorithme où il faut adapter à chaque étape la meilleure taille des blocs). Puis, en reprenant comme modèle de référence l'architecture à mémoire partagée présentée dans l'introduction, nous étudions théoriquement l'influence du temps de communication sur le temps d'exécution pour l'ordonnancement Glouton.

L'architecture parallèle sous jacente est composée d'un ensemble de p processeurs, possédant chacun une mémoire locale et partageant une mémoire centrale commune. Les échanges entre cette mémoire partagée et les processeurs se

font par l'intermédiaire d'un réseau d'interconnexion, de sorte qu'en une unité de communication, p données peuvent être transférées simultanément. Pour simplifier, nous supposerons qu'un processeur ne peut débuter l'exécution d'une tâche que lorsque toutes les données qu'il utilise sont stockées dans sa mémoire locale. De même, les résultats ne peuvent être transférer en mémoire partagée qu'à l'issue du calcul.

Nous appellerons respectivement τ_{exec} et τ_{com} les unités de temps de calcul et de temps de communication (en supposant qu'un chargement mémoire s'effectue dans le même temps qu'un rangement). Sans perte de généralité, nous prendrons τ_{exec} constant pour chacune des opérations arithmétiques de base. En revanche, τ_{com} dépend fortement de l'architecture considérée, c'est une fonction croissante de la taille du problème n et du nombre de processeurs p.

L'algorithme qui nous sert de support ici est la version kji modifiée de Gauss, dont l'ordonnancement est de type glouton.

```
c    Version kji  modifiée
     pour k = 1, n-1
          pour j = k+1, n
c         exécuter la tâche T_kj
               a_kj = - a_kj / a_kk
               pour i = k+1, n
                    a_ij = a_ij +  a_ik * a_kj
```

Le coût des tâches T_{kj} en cacul est $(2(n-k)+1)\ \tau_{exec}$.

Le coût en communication est $(3(n-k)+2)\ \tau_{com}$ (en effet, chaque appel à un élément de la matrice nécessite une communication avec la mémoire, on dénombre au total 2(n-k)+2 chargements et (n-k) rangements).

Notons de plus que le fait de prendre en compte les temps de communication ne modifie pas les relations de précédence entre les tâches. Nous obtenons la même forme de graphe triangulaire pour lequel nous avons montré que l'algorithme glouton est asymptotiquement optimal. Cependant, une hypothèse du théorème précédent n'est plus vérifiée : dans le décompte des temps d'exécution des tâches, les communications ne sont plus négligeables (de l'ordre de O(n)) puisque τ_{com} dépend en fait de n et de p. En reprenant la démonstration, il n'est pas difficile de voir que l'on peut facilement intégrer cette condition dans le calcul. Ces remarques conduisent au théorème suivant :

Théorème 5.1.
Pour un graphe des tâches triangulaire tel que les temps d'exécution des tâches soient constants pour k fixé et décroissants avec les niveaux, l'algorithme glouton est asymptotiquement optimal et son temps d'exécution est égal à :

$$\tau_{opt} = n^2 \,[(4+ 2\alpha^3)\, \tau_{exec} + (6+ 3\alpha^3)\, \tau_{com}\,] / 6\alpha$$

Preuve.
Le principe de la démonstration est identique à précédemment. Comme les temps d'exécution total des tâches T_{kj} (notés t_k) sont décroissants avec k, le temps d'exécution global peut être calculé à partir de la formule suivante :

$$\tau_{opt} = [\Sigma_{k=1,n-p} (n-k)\, t_k]/p + \Sigma_{k=n-p+1,n-1}\, t_k$$

où $t_k = (2(n-k)+1)\, \tau_{exec}$ et $(3(n-k)+2)\, \tau_{com}$ on obtient, en posant $p = \alpha n$:

$$\tau_{opt} = n^2\, [2(1 - \alpha^3)/ 3\alpha + \alpha^2]\, \tau_{exec} + n^2\, [(1 - \alpha^3)/ \alpha + 3\alpha^2/2]\, \tau_{com} + O(n)$$

$$\tau_{opt} = n^2 \,[(4+ 2\alpha^3)\, \tau_{exec} + (6+ 3\alpha^3)\, \tau_{com}\,] / 6\alpha$$

5.2. ETUDE DE VERSIONS PAR BLOCS

Posons n=qr et décomposons la matrice A en q^2 blocs de tailles r. On peut alors écrire la forme kji modifiée par blocs en remplaçant les opérations scalaires par des opérations sur des blocs de manière à augmenter la granularité des tâches (c'est à dire le poids relatif de l'arithmétique sur les communications).

```
c    Version par blocs kji  modifiée
     pour k = 1, q
          pour j = k+1, q
c         exécuter la tâche TBkj
               Akj = Akk^-1* Akj
               pour i = k+1, q
                    Aij = Aij -  Aik * Akj
```

Décompte des durées des tâches

Le calcul de $A_{kj} = A_{kk}^{-1*} A_{kj}$ revient à résoudre r systèmes de taille r de matrice A_{kk} (ce qui requiert $8r^3/3$ opérations arithmétiques). De plus, le calcul de $A_{ij} = A_{ij} - A_{ik} * A_{kj}$ correspond à (q-k) multiplications et additions matriciels, sur des blocs rxr, soit un temps de $2r^3(q-k)\ \tau_{exec}$.

Donc, au total, le coût de l'exécution de TB_{kj} est :

$$2r^3[(q-k)+4/3]\tau_{exec}.$$

D'autre part, ces calculs nécessitent la connaissance de A_{kk} A_{ij} A_{ik} et A_{kj} pour i variant de k+1 jusqu'à q. Ainsi, on peut évaluer facilement le temps de communication :

$$2r^2[(q-k)+1]\tau_{com}.$$

Si τ_{excc} et τ_{com} sont du même ordre de grandeur, il est clair que les temps de communication sont négligeables devant le calcul (en r^3 contre r^2).

Donc, implémenter cet algorithme parallèle par blocs selon le schéma d'ordonnancement que nous avons développé précédemment devient réaliste.

Analyse de glouton par blocs

Le résultat est simple à établir car le graphe a la même allure (les contraintes de précédence sont identiques). Il suffit donc de transposer la démonstration précédente avec des nouvelles valeurs pour le temps des tâches. Cela conduit au théorème suivant :

Théorème 5.2. L'algorithme glouton avec p=αq processeurs est asymptotiquement optimal pour la version kji modifiée de Gauss par blocs. Son temps d'exécution est égal à :

$$\tau_{opt} = n^3(2+\alpha^3) / 3\alpha q$$

Preuve.
Le temps d'exécution global peut être calculé comme précédemment :

$$\tau_{opt} = [\Sigma_{k=1,q-p} (q-k)\, t_k]/p + \Sigma_{k=q-p+1,q-1}\, t_k$$

où $t_k = 2r^3[(q-k)+4/3]\tau_{exec}$

$$\tau_{opt} = 2r^3/ 3\alpha q\, (q^3-p^3) + r^3p^2$$

on obtient le résultat en posant $p = \alpha q$:

$$\tau_{opt} = r^3q^3\, [2(1-\alpha^3)+3\alpha^3] / 3\alpha q$$

$$\tau_{opt} = n^3(2+\alpha^3) / 3\alpha q$$

Ce qui conduit à une efficacité égale à :

$$e_{opt,\alpha} = 1/(2+\alpha^3)$$

Notons que le temps est minimum en α=1. Comparons pour conclure les efficacité des versions par points et par blocs.

En l'absence de temps de communication, la version par points est meilleure. Ceci peut paraître paradoxal, mais nous avons vu que ce cas n'est pas vraiment réaliste. En supposant que les communications sont du même ordre que le calcul, la version par blocs est meilleure. Nous aborderons dans le paragraphe 5.4. une comparaison plus systèmatique en fonction de l'architecture sous-jacente.

5.3. ALGORITHME ADAPTATIF

Description de l'algorithme glouton adaptatif

Dans les versions de l'élimination de Gauss par points, les transferts de données ne se sont plus négligeables par rapport à l'arithmétique. Nous venons de donner une solution qui consiste à envisager des stratégies par blocs pour augmenter le poids du calcul par rapport aux transferts de données. Cependant, il y a de moins en moins de tâches à effectuer (la version par points garde une pleine efficacité plus longtemps). Donc, vers la fin de l'éminination, les processeurs ne travaillerons pas au mieux et l'on va perdre ainsi de l'efficacité.

Ces remarques nous ont conduits à chercher des solutions mixtes, c'est-à-dire des versions adaptatives où la taille des blocs est déterminée au mieux, au fur et à mesure, de façon à garder le plus longtemps possible une pleine efficacité.

Le principe de l'algorithme adaptatif est le suivant :

Fixons tout d'abord q=p. On décompose la matrice initiale en q^2 blocs, comme précédemment, et l'on applique la première étape l'élimination par blocs. Une fois le premier bloc de colonnes éliminé, on réitère la première étape d'une élimination par blocs sur la sous matrice qu'il reste à traiter, et ainsi de suite jusqu'à une dernière étape où il reste moins de tâches que de processeurs que l'on traite par l'algorithme par points. La figure 5.1. suivante schématise l'algorithme.

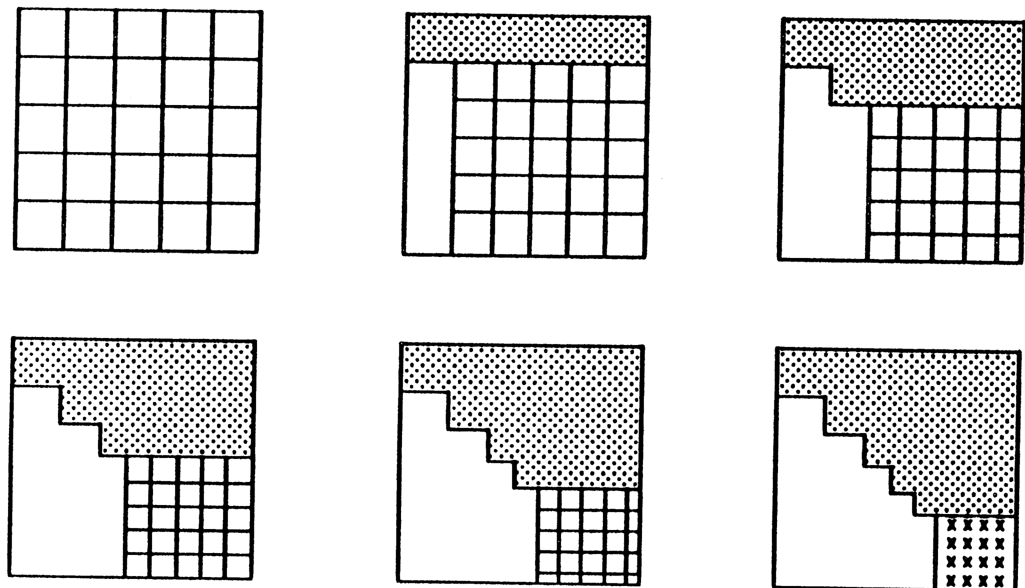


figure 5.1. étapes de l'algorithme adaptatif avec 5 processeurs

Conclusion provisoire

Les performances comparatives de ces trois séries d'algorithmes (par points, par blocs et adaptatif) sont consignés dans [CMuR]. Pour la plupart des cas, c'est le dernier algorithme qui est le meilleur, comme le montre le tableau suivant représente les temps d'exécution (en milliard d'opérations flottantes) des trois algorithmes en tenant compte des communications (avec un facteur constant) pour une matrice de dimension 5000. La matrice est divisée exactement en p^2 blocs (où p est le nombre de processeurs). Il s'agit d'une simulation.

Nombre de proc.	kji par points	bloc kji	kji adaptatif
9	18.5	14.2	9.6
19	8.7	6.8	4.5
39	4.2	3.3	2.2
49	3.4	2.6	1.8
99	1.6	1.3	0.89
249	0.67	0.52	0.38
499	0.33	0.27	0.22

Jusqu'ici, nous avons supposé que le temps des communications élémentaires étaient soit nul, soit constants du même ordre que le temps d'exécution des opérations arithmétiques. Dans la pratique, les temps d'accés aux données en mémoire centrale sont nettement supérieurs (il est vrai que l'on dispose souvent de mémoires caches plus petites mais beaucoup plus rapides ou de tampons de données qui réduisent les temps d'accés, il n'est alors pas déraisonnable de considérer des temps voisins) de plus, ils dépendent du nombre de processeurs et de la taille du problème.

Examinons maintenant un modèle plus général.

5.4. VERS UN MODELE GENERAL

Avant de présenter le modèle général, faisons une brève présentation des réseaux d'interconnexion.

Réseaux d'interconnexion

La structure générale des réseaux d'interconnexion est présentée dans [AB].

Un réseau de connexion permet de relier p processeurs à b bancs mémoire. Le temps d'échanges avec la mémoire est en fait relié aux nombres de bancs, c'est à dire indirectement à la taille du problème. Un réseau complet (cross-bar) est composé de b.p commutateurs. Cette solution idéale, dans le sens où elle autorise toujours toutes les permutations possibles, n'est pas réaliste dès que l'on augmente sensiblement le nombre de processeurs ou le nombre de bancs. Un réseau moins coûteux peut être obtenu en supprimant quelques commutateurs, mais en permettant toujours tous les passages (au prix éventuellement d'un réarrangement du routage des données). C'est le cas du réseau de Benes par exemple qui est composé de 2pLn(p) commutateurs. Le réseau Omega possède quant à lui moitié moins de commutateurs mais certaines permutations peuvent nécessiter plusieurs passages à travers le réseau (on dit alors que le réseau est blocant). En supposant les temps de commutation constants, nous déduisons que τ_{com} peut prendre des valeurs qui ont l'allure suivante, selon le réseau considéré :

$\tau_{com} = \lambda$	indépendant de n et de p
$\tau_{com} = \lambda\, f(n)$	croissant en n
$\tau_{com} = \lambda\, g(n,p)$	croissant en n et p

Cas 1 : $\tau_{com} = \lambda$

Pour l'analyse, on reprend la formule générale établie au paragraphe 5.1. Fixons pour simplifier $\tau_{exec} = 1$. Dans la pratique, λ peut atteindre plusieurs dizaines, cependant, si l'on dispose d'une mémoire cache ou de registres de données (et que l'on anticipe les transferts de données), λ peut être de l'ordre de l'unité.

Etudions tout d'abord le cas simple d'un temps de communication constant. Pour l'algorithme kji par points, le coût de communication devient $\lambda[n^3/p + p^2/2]$, il est du même ordre que le coût de calcul : $[(2/3)\, n^3/p + p^2/3]$. On obtient alors :

$$\tau_{opt} = [n^3\,(3\lambda + 2) + p^3\,(\lambda/2 + 1)]\,/\,3p$$

En ce qui concerne les versions par blocs et adaptatives, le coût des communications est négligeable. Pour donner une idée plus pratique, on peut conclure ce cas en comparant les résultats globaux sous l'hypothèse où $r=q=p=\sqrt{n}$:

$$\tau_{pt} = (2/3+\lambda)\, n^{5/2}$$

$$\tau_{bl} = n^{5/2}$$

$$\tau_{ad} = (2/3)\, n^{5/2}$$

Pour les deux cas suivants, les communications sont prépondérantes sur l'arithmétique pour l'algorithme par points. Aussi, nous étudierons la version adaptative (dont nous avons montré qu'elle était toujours meilleure).

Cas 2 : $\tau_{com} = \lambda\, f(n)$

Ce cas n'est pas bien différent du précédent sur le principe général, cependant, suivant la forme de la fonction croissante f, les communications vont prédominer. Notons ici que f est pratiquement à croissance lente. L'expression générale est la suivante :

$$\tau_{ad} = (2/3)n^3/p + (3\lambda/2)\, n^2 f(n)$$

ce qui nous permet de conclure pour ce cas :

Si $p=O(n/f(n))$ alors les communications et l'exécution sont du même ordre
Si $p>O(n/f(n))$ alors les communications prédominent
Si $p<O(n/f(n))$ alors l'arithmétique prédomine

Cas 3 : $\tau_{com} = \lambda\, g(n,p)$

Dans ce cas, l'expression générale du temps de l'algorithme adaptatif est la suivante :

$$\tau_{ad} = (2/3)n^3/p + (3\lambda/2)\, n^2 g(n,p)$$

ce qui donne :

Si $p=O(n/g(n,p))$ alors les communications et l'exécution sont du même ordre
Si $p>O(n/g(n,p))$ alors les communications prédominent
Si $p<O(n/g(n,p))$ alors l'arithmétique prédomine

Contrairement au cas précédent, le temps d'exécution ne décroît plus strictement lorsque p croît. Il existe une valeur optimale du nombre de processeurs p, que l'on peut obtenir à partir des formules précédentes. On peut trouver un calcul détaillé dans [CT] pour $g(n,p)=\sqrt{np}$.

6. ARCHITECTURES SYSTOLIQUES

Présentation du modèle systolique. Etude de deux implantations : le problème du chemin algébrique et le calcul de projections en filtrage adaptatif.

6.1. PRESENTATION GENERALE

Le modèle systolique a été introduit en 1978 par Kung et Leiserson [KL] pour répondre à la demande sans cesse croissante d'une plus grande puissance de calcul à moindre frais. Il s'est révélé être un outil très efficace pour la conception de processeurs intégrés spécialisés.

Une architecture systolique est agencée en forme de réseau composé d'un grand nombre de cellules élémentaires identiques et localement interconnectées (linéairement, orthogonalement, hexagonalement ..). Chaque cellule reçoit des données provenant de ses cellules voisines, effectue un calcul simple, puis transmet à son tour les résultats aux cellules voisines, un temps de cycle plus tard. Pour fixer un ordre de grandeur, chaque cellule a au plus la complexité d'un petit micro processeur.

Les cellules évoluent en parallèle, sous le contrôle d'une horloge globale : plusieurs calculs sont effectués simultanément sur le réseau et on peut anticiper la résolution de plusieurs instances du même problème sur un même réseau.

La dénomination "systolique" provient d'une analogie avec la circulation du flot sanguin dans le corps humain, sous un contrôle global synchronisé par le coeur. Notons que ce type d'architectures est déjà une réalité, en effet il existe plusieurs réalisations commerciales parmi lesquelles le GAPP [Lef] ou le WARP.

Nous développons ci-dessous deux exemples d'implémentation systolique : le problème du chemin algébrique et le calcul de projections en filtrage adaptatif.

6.2. PROBLEME DU CHEMIN ALGEBRIQUE

Sous le concept de "chemin algébrique", on retrouve unifiés des algorithmes issus de domaines variés comme l'inversion d'une matrice réelle, la fermeture réflexive et transistive d'une relation binaire ou encore la recherche des plus courtes distances dans un graphe pondéré.

Quelques définitions préliminaires

Etant donné un graphe pondéré $G = (V,E,\omega)$
où V est l'ensemble fini des sommets $(v_i)_{i=1,...,n}$
E l'ensemble des arcs
ω une application à poids de E dans le semi-anneau $(H, \oplus, \otimes, *)$
H est de zéro (0) (également absorbant pour $\otimes$), d'unité (1)
$\otimes$ est distributive par rapport à $\oplus$
* est définie par : $c^* = \oplus_{i\geq 0} c^i = c \oplus (c \otimes c) \oplus (c \otimes c \otimes c) \oplus (...)$

Formellement, le problème du chemin algébrique (APP pour Algebraic Path Problem) est défini par :

> pour tout $c \in H$, trouver pour toutes les paires de sommets (v_i, v_j), les valeurs $d_{ij} = \oplus \{\omega(p), p \in M_{ij}\}$
> où M_{ij} est l'ensemble des chemins de v_i à v_j

Principe général

On associe au graphe $G = (V,E,\omega)$ une matrice de poids A définie par :

$a_{ij} = \omega(i,j)$ si $(i,j) \in$ E et 0 sinon.

Cette matrice a la propriété remarquable suivante :

$A^2 = a_{ij}^{(2)}$ où $a_{ij}^{(2)} = \oplus_{k=1,n}\, a_{ik} \otimes a_{kj}$

$A^m = a_{ij}^{(m)}$ où $a_{ij}^{(m)} = \oplus_{k=1,n} \{\omega(p), p \in M_{ij}^{(k)}$ comportant m arcs$\}$

Donc, la matrice cherchée D des paires de sommets possédant un chemin d'un arc quelconque à un autre, vérifie la relation $D = A^*$ (cumul de toutes les puissances successives de A).

On construit successivement les ensembles $M_{ij}^{(k)}$ de tous les chemins de v_i à v_j dont les sommets intérieurs sont d'indices inférieurs à k.

Partant de $A^{(0)} = a_{ij}$, on calcule alors $a_{ij}^{(k)} = \oplus \{\omega(p), p \in M_{ij}^{(k)}\}$ pour k=1, .., n. A la fin, on retrouve $a_{ij}^{(n)} = d_{ij}$. L'algorithme est le suivant :

pour k=1,n **faire**
  $a_{kk}^{(k)} = (a_{kk}^{(k-1)})^*$
  pour i=1,n **faire** i≠k
    $a_{ik}^{(k)} = a_{ik}^{(k-1)} \otimes a_{kk}^{(k)}$
  pour j=1,n **faire** j≠k
    pour i=1,n **faire** i≠k
      $a_{ij}^{(k)} = a_{ij}^{(k-1)} \oplus a_{ik}^{(k)} \otimes a_{kj}^{(k-1)}$
    $a_{kj}^{(k)} = a_{kk}^{(k)} \otimes a_{kj}^{(k-1)}$

Les différentes instances de l'APP s'obtiennent en spécifiant les opérations ⊕, ⊗ et * dans les semi-anneaux adéquats :

Inversion d'une matrice à coefficients réels. Les deux opérations sont les opérations arithmétiques usuelles dans IR, * est définie par :

$c^* = 1/(1-c)$ si $c \neq 1$

(en pratique, l'algorithme délivre en sortie l'inverse de (I-A), mais on peut obtenir A^{-1} facilement par des transformations élémentaires).

Fermeture reflexive et transitive d'une relation binaire. Les poids a_{ij} sont des booléens, ⊕ et ⊗ sont respectivement le "et" et le "ou" logique et * est l'opération (non commutative) de mise à "vrai".

Recherche des plus courtes distances dans un graphe pondéré. Ici, les a_{ij} sont choisis dans IR∪{-∞,+∞}. ⊕ est le minimum (de zéro +∞), ⊗ est l'addition sur les réels, étendue à H et enfin * est définie par :

si $c \geq 0$ alors $c^* = 0$ sinon $c^* = -\infty$

Réalisation

Le réseau que nous proposons est décrit sur la figure 6.1. C'est un réseau orthogonal de n(n-1) cellules. La ligne k du réseau effectue le k-ième pas de l'algorithme. Le fonctionnement détaillé est dans [RT1].

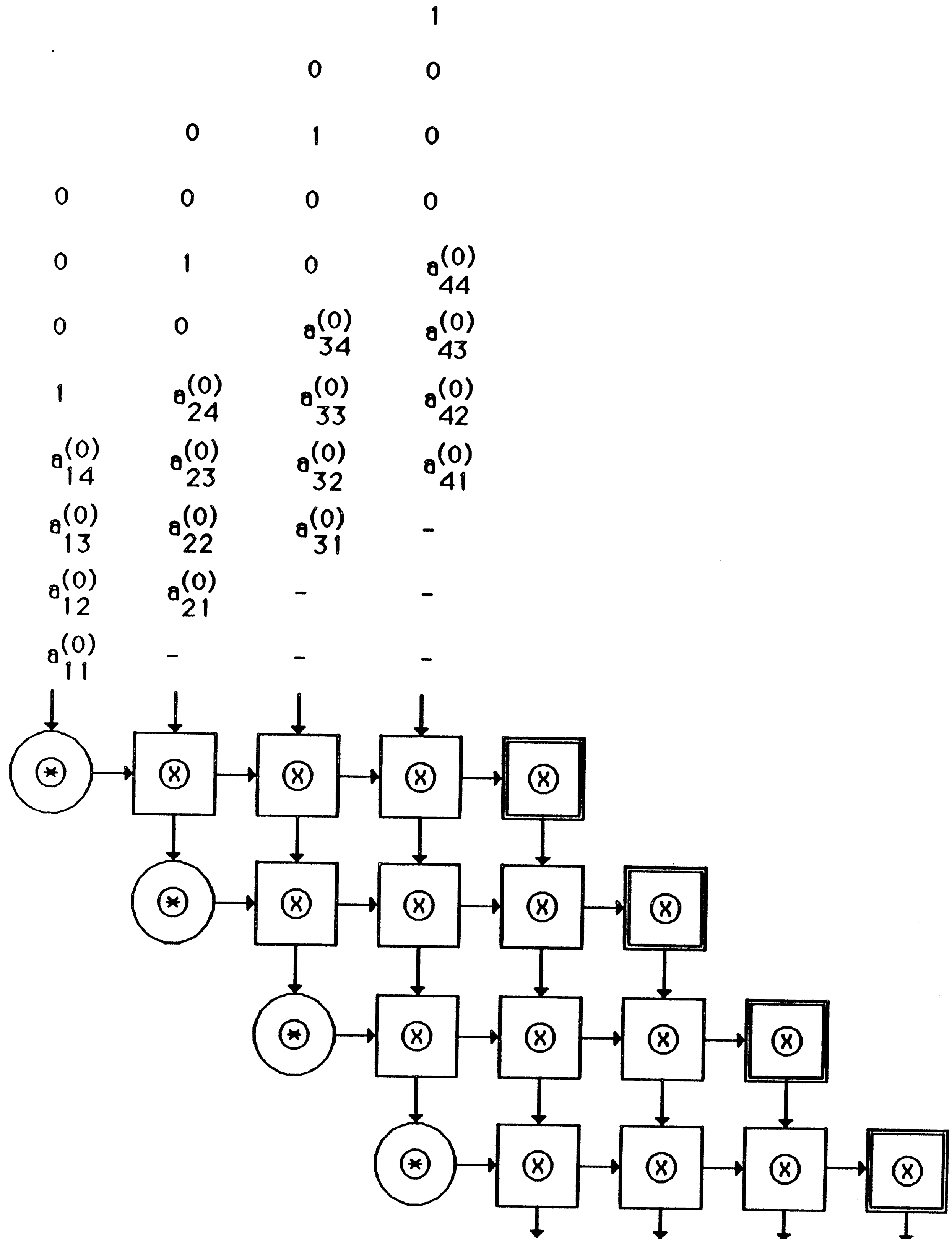


figure 6.1. réseau systolique pour l'APP (n=4)

Comme l'indique la figure, on distingue trois types de cellules : les cellules diagonales rondes, qui correspondent à l'opération d'inversion * de leur première entrée puis de transfert des suivantes, les cellules simples carrées qui stockent leur première entrée puis effectuent un pas élémentaire de "produit scalaire" et les cellules doubles carrées du bord droit qui fonctionnent de la même manière avec une initialisation différente de leur registre interne. Leur fonctionnement est schématisé dans la figure 6.2.

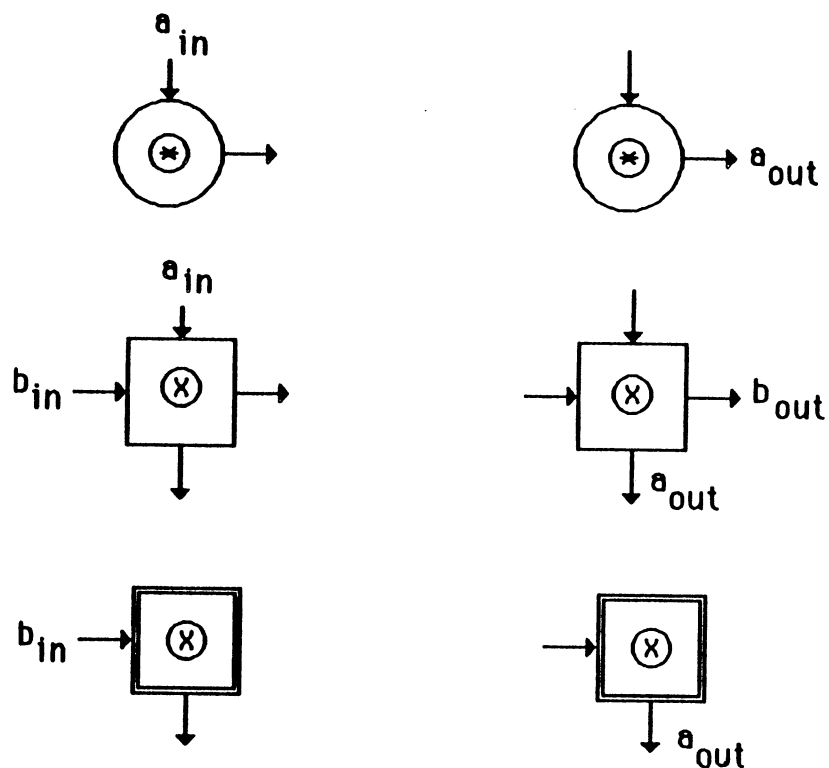


figure 6.2. détail du fonctionnement des cellules

Dans l'implantation que nous proposons, on entre la matrice $C=(A,I_n)$ pour ne pas écraser la matrice $A^{(k-1)}$ lors du calcul de $A^{(k)}$ à l'étape k.

La solution est obtenue en 5n-2 unités de temps pour le problème général, il est optimal et améliore toutes les solutions existantes. L'argument de l'optimalité peut être intuité par les deux remarques suivantes :

Premièrement, le chemin le plus long est de longueur 3n-2 (il correspond au calcul séquentiel de $c_{11}^{(1)}$, $c_{21}^{(1)}$, $c_{22}^{(1)}$, $c_{22}^{(2)}$, ..., $c_{k,k-1}^{(k-1)}$, $c_{k,k}^{(k-1)}$, $c_{k,k}^{(k)}$,, ..., $c_{nn}^{(n)}$, soit n unités pour calculer les $c_{k,k}^{(k)}$, et 2(n-1) pour les autres). De plus, les 2(n-1) coefficients de la n-ième ligne et de la n-ième colonne doivent se rencontrer pour acquérir leur valeur finale, ce qui nécessite 2(n-1) unités de temps. Donc, le temps

optimal de résolution d'un problème APP de taille n est 5n-4. Ce qui prouve que notre réseau en 5n-2 est optimal.

Notons pour finir que ce problème a fait l'objet d'une implantation en parallèle [RT2].

6.3. CALCUL DE PROJECTIONS EN FILTRAGE ADAPTATIF

Nous nous intéressons ici à l'implémentation systolique du calcul des projections intervenant dans le traitement des signaux numériques. Le problème général peut être posé de la manière suivante [CoRT] :

Calculer successivement le produit $A_i^{-1}B_i$ pour i=1,2, ...

où A_i et B_i sont respectivement des matrices de tailles nxn et nxp, elles sont issues de A_{i-1} et B_{i-1} par des modifications de rang k ou des modifications de rang plein et de petites normes.

Nous avons proposé dans [CoRT] une étude générale pour calculer $A^{-1}B$ par décomposition en sous problèmes matriciels élémentaires. Nous faisons un rapide tour d'horizon des meilleurs réseaux existants qui conduisent au produit $A^{-1}B$ par combinaisons des réseaux de base, puis nous proposons un réseau unique, basé sur la décomposition de Gauss-Jordan, qui permet de calculer directement $A^{-1}B$. Enfin, nous donnons la description d'un réseau systolique qui permet de tenir compte des éléments de A_{i-1} et B_{i-1} dans le calcul de $A_i^{-1}B_i$. C'est un réseau original qui inclut la mise à jour des matrices.

Il est amusant de constater que ce réseau est une adaptation du réseau précédent qui fournissait la solution du problème du chemin algébrique.

Position du problème du calcul des projections

Les calculs qui doivent être implémentés sur ce réseau sont ceux de projecteurs dans lesquelles les matrices de covariance sont soit recalculées à chaque pas, soit calculées récursivement [CoRT].

Dans [CR], une implémentation systolique performante de projections a été présentée dans le cas d'un calcul non récursif, et de matrices de covariance hermitiennes, ou symétriques réelles, sans structure imposée. En revanche, nous allons nous intéresser ici aux calculs de projections régis par une récurrence exponentielle [CoRT], résumés comme suit :

$$A_j = (1-\alpha)\ A_{j-1} + \ \alpha\ u_j u_j^t\ ;$$
$$B_j = (1-\alpha)\ B_{j-1} + \ \alpha\ u_j v_j^t\ ;\ \text{pour } 0< \alpha <1$$

où les vecteurs u_j et v_j sont donnés.

Dans un premier temps, aucune structure particulière ne sera imposée aux matrices de covariance si ce n'est l'hermicité (ou la symétrie dans le cas réel) et la non négativité.

6.3.1. IMPLANTATION SYSTOLIQUE DE $A^{-1}B$

Première solution possible

On peut évaluer le temps de calcul global du produit $A^{-1}B$ à partir des meilleurs réseaux systoliques élémentaires existants pour une matrice A de taille nxn.

On obtient au mieux la triangularisation en 3n unités de temps (asymptotiquement) [GK], l'inversion d'une matrice triangulaire en 2n [LW] et la multiplication en 4n [PV]. Soit 11n unités de temps pour obtenir la matrice A^{-1} (décomposition LU, inversion de U et L puis multiplication $U^{-1}L^{-1}$), donc 15n au total pour obtenir $A^{-1}B$. De plus, le flot des entrées n'est pas compatible d'une application à l'autre (entrée des matrices par colonnes ou par diagonales). Il faudrait donc en toute rigueur rajouter au temps de calcul, le temps de réordonnancement des données. La solution qui consiste à coupler des réseaux élémentaires est donc à rejeter puisque trop coûteuse.

Calcul direct de $A^{-1}B$

Le calcul direct de $A^{-1}B$ peut être vu comme la résolution de p systèmes linéaires de même matrice :

$$A\, x_i = b_i \text{ pour } 1 \leq i \leq p$$

On note B la matrice de taille nxp formée des p vecteurs seconds membres b_i et C la matrice cumulée (A,B) de taille n par (n+p).

Le réseau systolique orthogonal de Gentleman et Kung [GK], permet la résolution d'un système linéaire. Nous proposons ici une solution qui consiste à remplacer le système triangulaire par un système diagonal en utilisant le même réseau. On remplace simplement la phase de vidage par une phase de calcul supplémentaire.

Ce réseau orthogonal comporte n(n+1) cellules et permet le calcul de $A^{-1}B$ en un temps asymptotique de 4n+p unités.

Le calcul de $A^{-1}B$ s'effectue à partir de la diagonalisation de Jordan. Si la matrice A n'a pas les propriétés de stabilité requises, il faut mixer l'algorithme d'élimination avec un algorithme de factorisation orthogonale. Supposons pour l'instant que A est non singulière, et préoccupons nous exclusivement de l'implémentation systolique de la méthode, sans souci d'ordre numérique.

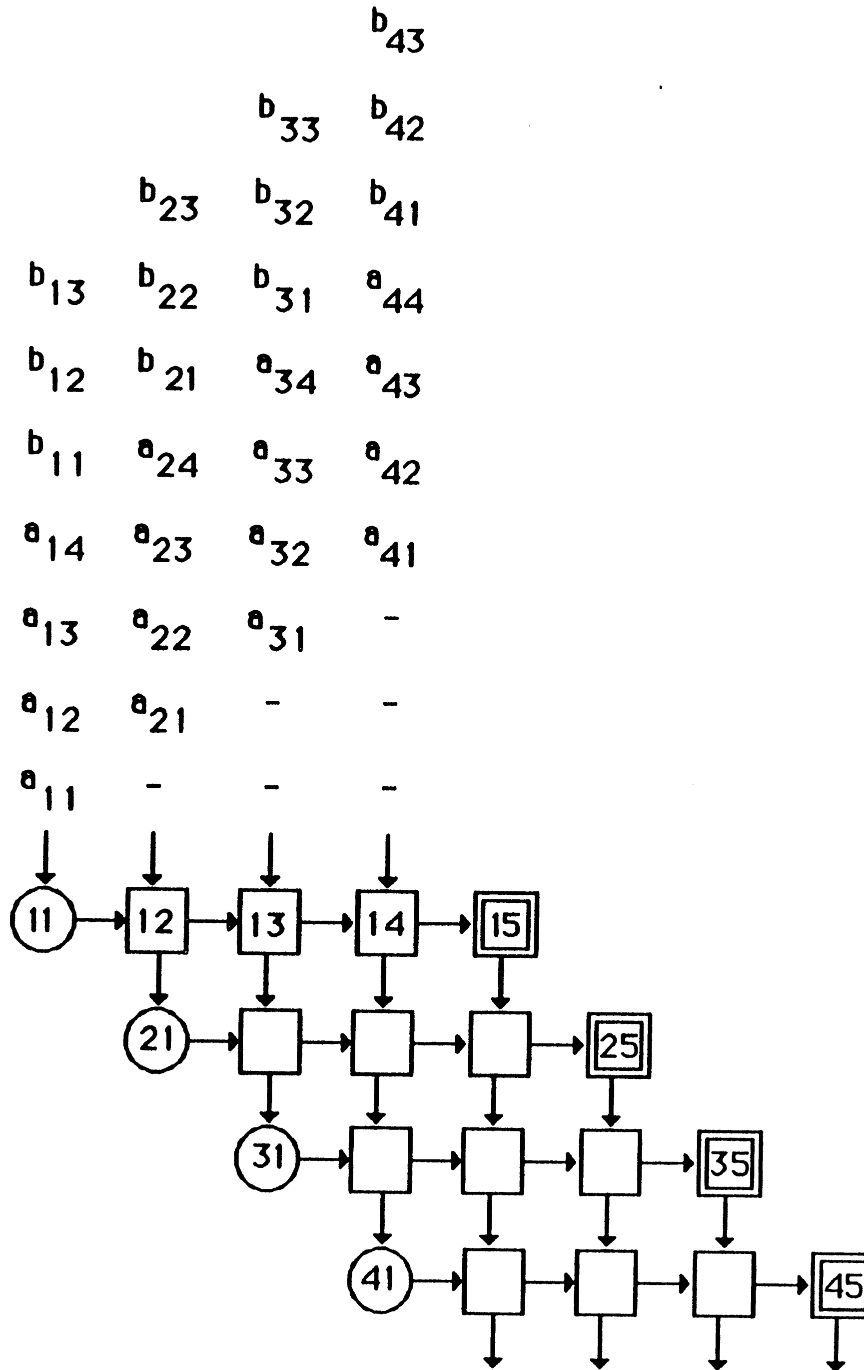


Figure 6.3 : Calcul direct de $A^{-1}B$

Avant de décrire le fonctionnement de ce réseau, il convient de donner un mot d'explication informelle : dans le réseau de Gentleman et Kung [GK], les lignes de la matrice sont stockées dans les processeurs, et les matrices M_{ik} circulent à travers le réseau. Ici, le fonctionnement est dual : les lignes circulent, et les matrices M_{ik} sont

matrices M_{ik} sont stockées dans les cellules. En effet, dans l'algorithme de Jordan chaque ligne doit, quand elle devient la ligne pivot, rencontrer toutes les autres, e non plus seulement celles d'indices supérieurs. Plutôt que de faire repartir à travers le réseau une ligne déjà stockée, comme nous l'avons fait précédemment il est préférable ici de faire circuler toutes les lignes de la matrice C. Ceci explique aussi que la matrice A entre dans le réseau sous forme transposée pa rapport au réseau de Gentleman et Kung.

Il y a n étapes dans l'algorithme de Jordan. Nous implémentons chacune d'entre elles sur une ligne du réseau. Pour bien comprendre le fonctionnement de ce réseau, il est important de noter que les lignes de la matrice initiale sont permutées circulairement en traversant le réseau, de manière à ce que la ligne pivot à l'étape k puisse être combinée avec toutes les autres dans les cellules P_{k1}, ..., $P_{k,n+1}$.

Description des cellules

De la même manière que précédemment, on distingue trois types de cellules :

• type 1 : les cellules rondes calculent l'inverse de leur première entrée valide, puis elles agissent simplement comme des cellules de délai.

• type 2 : les cellules simples carrées initialisent d'abord leur registre interne en stockant (après modification) leur première entrée valide. Puis elles agissent comme des cellules IPS (un pas élémentaire de produit scalaire).

• type 3 : les cellules doubles carrées opèrent en fait comme les cellules carrées, mais leur registre interne n'est pas initialisé de la même façon.

Résumons maintenant les performances de ce réseau :

On calcule $A^{-1}B$ en 4n+p-2 unités de temps sur un réseau de n(n+1) cellules où A et B sont des matrices denses de tailles respectives nxn et nxp.

6.3.2. RESEAU ADAPTATIF

Nous proposons maintenant une petite adaptation qui permet de calculer directement le produit $A_{i+1}^{-1}B_{i+1}$, en fonction de A_i et B_i au pas précédent en tenant compte des modifications de rang 1. Partant du réseau que nous venons de décrire, il suffit de lui adjoindre en entrée un réseau de préconditionnement de mise à jour des entrées. On suppose que le réseau a déjà été initialisé lors d'une étape précédente, en d'autres termes que les coefficients sont stockés dans les cellules.

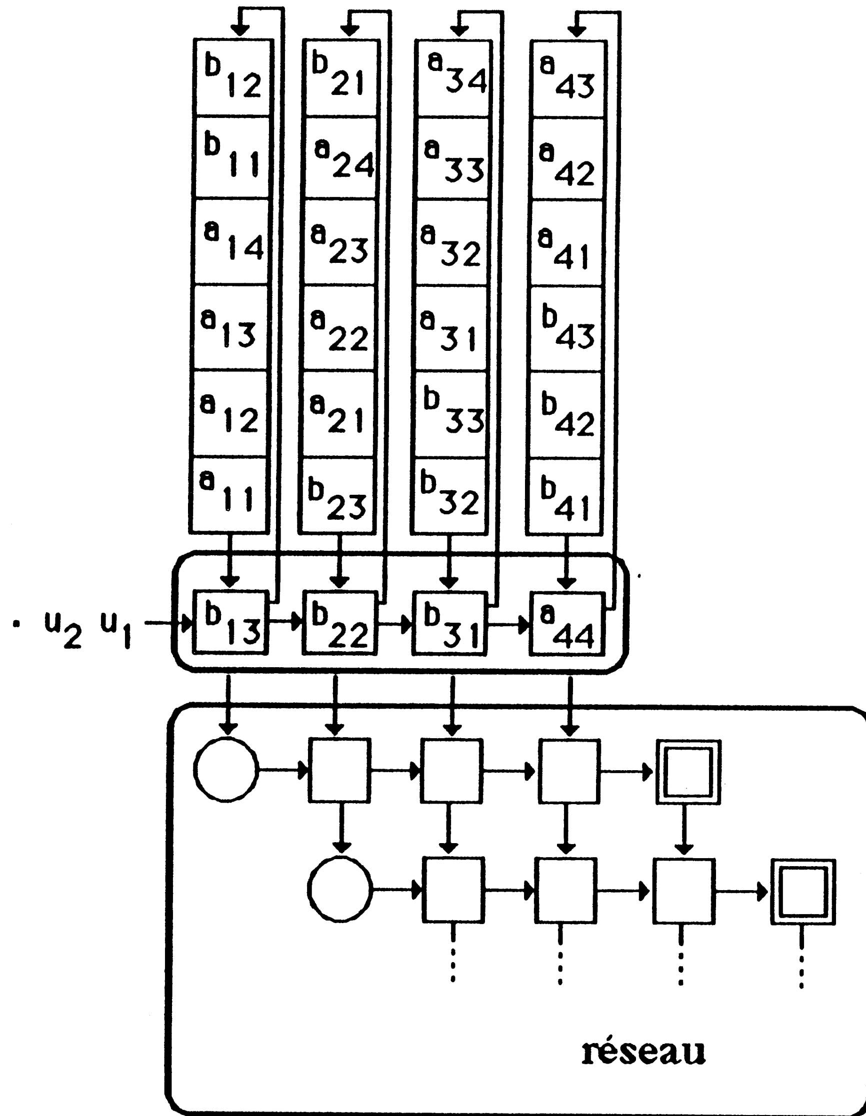


Figure 6.4 : fonctionnement du réseau de préconditionnement

7. LES SUPER CALCULATEURS

Etude d'exemples d'implantation parallèle sur des architectures existantes de type super calculateurs : le Gradient Conjugué sur CRAY XMP et IBM-3090 VF.

Nous avons présenté dans le chapitre précédent des implantations d'algorithmes sur des réseaux systoliques comportant de nombreuses cellules élémentaires. Une approche duale consiste à coupler peu de processeurs très puissants, c'est l'alternative de type "super calculateurs" que nous nous proposons de développer ici.

7.1 ORDINATEURS VECTORIELS

Le concept général de parallélisme a été largement présenté dans les chapitres précédents. Dans les applications scientifiques, la plupart des données numériques traitées ont la forme de vecteurs (au sens large). Dans un ordinateur vectoriel, les composantes indépendantes d'un même vecteur sont manipulées collectivement et effectuent la même opération (ou la même série d'opérations).

C'est un forme assez élémentaire de parallélisme, il s'agit d'une simultanéïté "logique" qui nécessite un amorçage. Un nouveau résultat est calculé tous les cycles. Il n'y a pas de perte de temps dû au chargement (sauf au début), une nouvelle donnée arrive tous les cycles d'horloge, les autres données étant soit en cours de chargement depuis un organe de mémoire, soit en train de circuler dans les étages opératoires successifs.

On distingue principalement deux formes de vectoriel : des unités spécialisées, séparées, que l'on adjoint comme périphérique à un ordinateur classique et des processeurs vectoriels intégrés à un système. Dans ce dernier cas, une unité vectorielle peut être vue simplement comme une adjonction à la partie exécutive, d'un jeu d'instructions supplémentaire qui contient des registres vectoriels pour

stocker les éléments consécutifs à traiter et des circuits spécialisés pour exécuter des opérations arithmétiques et logiques sur un flot de données.

Vu le délai d'amorçage, on a intérêt à travailler sur des vecteurs assez longs. Ils sont segmentés pour être contenus dans les registres vectoriels. L'accés aux données et les opérations sur les sections successives du vecteur peuvent être simultanés.

Pour résumer, on a donc séparation matérielle des ressources, adjonction d'unités spécialisées pour le traitement de vecteurs et anticipation dans la manipulation des éléments de ces vecteurs de manière à obtenir un rendement optimal (après un temps de démarrage).

La figure suivante schématise une unité vectorielle.

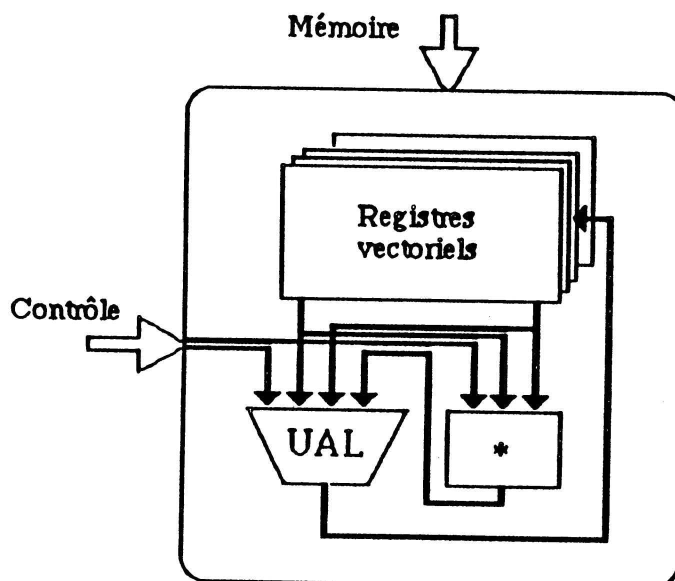


figure 7.1. Schéma d'une unité vectorielle

Ces super ordinateurs vectoriels sont d'un usage très général, ils sont souvent assez difficiles à manipuler, mais acceptent la résolution de problèmes très variés. Ce type de parallélisme a déjà largement investi l'industrie. Il existe des compilateurs qui détectent automatiquement des instructions élémentaires pouvant être exécutées simultanément. Il est possible d'aborder les problèmes comme nous l'avons fait jusqu'ici, en cherchant un parallélisme à un degré élémentaire (en utilisant le cas échéant des versions par blocs des algorithmes pour réduire les communications coûteuses) via l'étude théorique de l'ordonnancement sur les graphes de précédence. Mais dans la plupart des cas, les utilisateurs se contentent de découper leurs problèmes à un niveau macroscopique et d'adresser à chaque processeur des procédures entières, indépendantes.

7.2. PROGRAMMATION DU GRADIENT CONJUGUE

7.2.1. PRESENTATION GENERALE

L'ensemble des résultats présentés dans ce chapitre sont tirés des travaux menés en commun avec J. Ryckbosch et F. Vincent au centre de recherches d'EDF à Clamart [RTV] et P. Laurent-Gengoux [LT2] au centre scientifique ECSEC de IBM à Rome. L'importance considérable de l'algorithme du gradient conjugué pour la résolution de nombreux problèmes numériques (grands systèmes linéaires creux, recherche de modes propres, optimisation sous contraintes, analyse non linéaire etc..) n'est plus à démontrer [LT1]. L'intérêt majeur de cet algorithme est désormais acquis par l'ensemble de la communauté des numériciens.

Nous nous intéressons ici à la mise en oeuvre du gradient conjugué pour la résolution de grands systèmes linéaires Ax=b où la matrice A est de dimension n, symétrique définie positive. On montre facilement [GV] que ce problème est équivalent à la recherche du minimum de la fonctionnelle quadratique dans IR^n :

$$f(x) = 1/2\ x^t.Ax - x^t.b$$

Tel quel, cette méthode n'est pas d'une grande efficacité, bien que la convergence soit assurer théoriquement (aux erreurs machine près..) en moins de n itérations [GV], elle est garantie par les relations de conjugaison. La vitesse de convergence est fortement influencée par le conditionnement (c'est-à-dire, dans le cas symétrique par le rapport des valeurs propres extrêmes), plus il est proche de 1, meilleure est la convergence. On peut d'ailleurs aussi montrer que la répartition des valeurs propres agit sur la convergence.

On peut améliorer le conditionnement en substituant au système initial Ax=b le nouveau système $L^{-1}AL^{-t}(L^tx)= L^{-1}b$ de telle sorte que la matrice $M=LL^t$ soit une "approximation" de A. On dit alors que l'on "préconditionne" le système [Try1].

A chaque étape k, on doit résoudre le système Mz = r. Le problème est de choisir "correctement" la matrice M. Un bon préconditionnement consiste à prendre une matrice M assez proche de A et dont le système associé soit facile à résoudre. On trouve dans la littérature de nombreuses stratégies de préconditionnement, chacune adaptée à un problème spécifique.

L'algorithme du gradient conjugué préconditionné est donné ci-dessous :

```
procédure GCP (A,M,x,b,n) ;
début
    Initialiser (x,r,z,old,d) ;
    pour k := 1,2, ..., n  répèter
        v := Ad ;
        rho := old / vt.d ;
        x := x - rho*d ;
        r := r - rho*v ;
        résoudre Mz = r ;
        new := rt.z ;
        beta := new / old ;
        d := z - beta*d ;
        old := new ;
    jusqu'à  norme (r) < precision donnée
fin ;
```

Dans le cas d'une matrice pas trop creuse (c'est-à-dire comportant un nombre significatif d'éléments non nuls, la complexité de l'algorithme est, pour une itération, le coût du produit matrice-vecteur et de la résolution du système préconditionné (les produits scalaires étant négligeables, surtout si l'on utilise des ordinateurs avec des facilités vectorielles, dans lesquels les opérations de type produits scalaires sont des primitives élémentaires).

7.2.2. COMMENT PARALLELISER LE GRADIENT CONJUGUE ?

Les voies sont multiples.

On peut tout d'abord découper la matrice en blocs (c'est toujours possible). Dans ce cas, le préconditionnement le mieux adapté semble être de prendre pour M la partie bloc-diagonale de A. On peut pousser un peu plus loin en approximant A par sa factorisation incomplète par blocs. Il est également possible de remplacer chaque bloc par une approximation simple. Si les problèmes viennent d'une modélisation par Eléments Finis, on peut également choisir des blocs élémentaires qui se recouvrent en utilisant directement les matrices de raideurs, on rejoint alors les techniques dites de "multisplitting". Une autre approche consiste à découper le domaine [LT2]. Le préconditionnement correspond alors à un problème local, et la parallélisation devient naturelle car on peut réduire les principales opérations à chaque sous domaine.

7.2.3. UN EXEMPLE SIMPLE DE RESOLUTION

Nous présentons ici la résolution d'un problème de modélisation des réseaux électriques sur CRAY XMP, le problème complet est traité dans [Ryc].

Pour bâtir un système informatique d'aide au choix de la topologie du réseau électrique, il faut disposer : d'une heuristique générant un grand nombre de topologies susceptibles de rendre les topologies admissibles, d'un algorithme capable de vérifier trés rapidement si oui ou non les transits sont corrects et enfin de critères susceptibles de fournir à l'exploitant des éléments de choix des topologies finalement retenues. Nous ne traitons ici que du deuxième point.

Ceci dit, en ajoutant que tout ce qui suit se place dans l'approximation linéaire classique du "courant continu", donnons la formule de calcul des transits :

$$t = Au + a_0$$

A et a_0 sont une matrice et un vecteur constant, calculés une fois pour toute. u est un vecteur qu'il faut calculer pour chaque topogie, ses composantes représentent en effet les différences de phase aux extrémités des éléments susceptibles d'être connectés ou déconnectés (car des calculs préliminaires permettent de représenter les diverses topologies comme la connexion ou la déconnexion d'un ensemble de dipôles, éventuellement fictifs et soumis alors à des lois électriques hétérodoxes [Ryc]). Ce vecteur u, de dimension de l'ordre de 80, s'obtient par résolution d'un système linéaire :

$$Bu = b_0$$

La matrice B est pleine, symétrique définie positive (mais pas forcément strictement), différente pour chaque topologie ; il est courant, lors de la recherche d'une correction de topologie de résoudre mille de ces systèmes, ce qui justifie pleinement un effort spécial à leur résolution. On peut remarquer pour conclure que les systèmes à résoudre sont fréquemment singuliers. En effet, certaines manoeuvres de topologie séparent le réseau en plusieurs composantes connexes, pour lesquelles l'équilibre production-consommation n'est pas respecté. Il n'y a alors pas de solution.

Choix d'une résolution par gradient conjugué

Ici, l'emploi d'une méthode usuelle comme la méthode de Cholesky a été abandonnée après expérimentation : les coefficients des matrices présentent en effet de fortes disparités d'ordre de grandeur (qui peuvent aller jusqu'à 10^{-15}), et les matrices sont souvent singulières. En cas d'échec lors d'une factorisation de matrice, il est donc assez délicat d'arbitrer entre les erreurs d'arrondi d'une part et l'existence effective d'une valeur propre nulle d'autre part.

C'est pourquoi nous avons choisi d'utiliser l'algorithme du gradient conjugué avec préconditionnement par la diagonale. Cette méthode est efficace pour notre

exemple et intrinsèquement très vectorielle (multiplication matrice-vecteur, produits scalaires et mise à jour globale de vecteurs).

Implantation

Nous avons implanté la résolution sur le CRAY XMP de la DER-EDF. Il a été utilisé en monoprocesseur en utilisant les facilités vectorielles, c'est à dire essentiellement une procédure de multiplication matrice-vecteur écrite en assembleur exécutée diagonale par diagonale de façon à obtenir des données consécutives dans les registres vectoriels.

La figure suivante représente le temps d'exécution de l'algorithme en fonction de la taille des vecteurs.

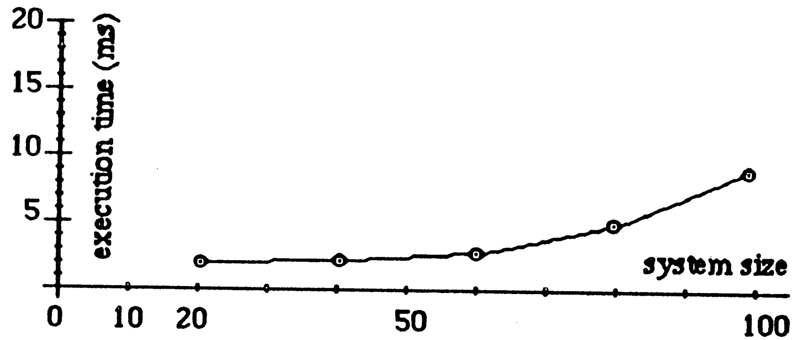


figure 7.2. vectorisation du gradient conjugué préconditionné diagonal pour la résolution du problème de réseau

7.2.4. DECOMPOSITION DE DOMAINES

Principe

Les problèmes que l'on se propose de résoudre ici sont axés sur la modélisation du comportement de solides composés de matériaux différents, qui conduisent à des grandes inhomogénéités dans les matrices (typiquement, un solide fait de plusieurs matériaux composites). On découpe le domaine en sous-domaines, regroupés par zones cohérentes. Les produits matrices-vecteurs sont décomposables trivialement sur chaque sous-domaine, le choix du préconditionnement correspond à une résolution locale, obtenue en annulant les noeuds extérieurs à un sous-domaine. Cela revient à résoudre un système cohérent sur chaque sous-domaine, par une méthode de Gauss par exemple. Ici, on peut imaginer sur chaque processeur, un algorithme très efficace écrit en assembleur et utilisant au mieux l'architecture interne du processeur (facilités vectorielles etc..).

La méthode est extrêmement efficace, même en séquentiel, pour des problèmes ayant un couplage intrinsèque en matériaux de natures différentes. La solution est obtenue plus rapidement et plus précisément qu'avec toute autre méthode. Pour s'en convaincre, nous avons comparé ce préconditionnement avec un préconditionnement diagonal, qui suffit souvent pour obtenir une solution (comme nous venons de le voir). Le gain de temps est supérieur à un facteur 5.

On peut réécrire l'algorithme précédent avec les adaptations adéquates :

début
    Initialiser (x,r,z,d) ;
    pour k := 1,2, ..., n **répèter**
        pour chaque sous-domaine Ω_i **faire**
            $v_i := A_i d_i$;
        reconstituer v à partir des v_i ;
        mise_à_jour (x,r) ;
        pour chaque sous-domaine Ω_i **faire**
            résoudre $M_i z_i = r_i$;
        reconstituer z à partir des z_i ;
        mise_à_jour (d) ;
    jusqu'à norme (r) < precision donnée
fin ;

Implantation de la méthode

Les résultats numériques suivants ont été obtenus sur l'IBM 3090-VF de l'ECSEC, on trouvera une description détaillée de la machine dans [IBM]. Il s'agit d'une architecture MIMD à plusieurs niveaux de mémoires (un cache de 64 Kmots), pouvant comporter jusqu'à 6 processeurs vectoriels.

On représente ci-dessous l'efficacité de la méthode proposée pour résoudre le problème numérique de l'analyse des déformations sur un matériau composite. La matrice du système comporte de fortes discontinuités entre ses coefficients (ce qui rend d'ailleurs pratiquement impossible la résolution globale du problème sans regrouper les zones de manière cohérente). La solution a été obtenue avec 2 processeurs et simulée pour 4 et 6 processeurs. A l'heure actuelle, nous attendons les résultats pratiques définitifs.

La résolution nécessite un temps de l'ordre de quelques secondes seulement pour un système de taille supérieure à 10000, avec 6 processeurs.

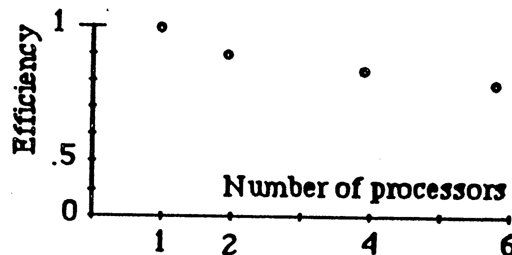


figure 7.3. Efficacité du Préconditionnement local

8. CONCLUSION

Où le héros peut enfin se reposer après avoir surmonté de nombreuses difficultés.

L'objet d'une conclusion usuelle est de dégager du texte les idées principales, de reprendre synthétiquement les résultats essentiels afin de dresser un bilan. L'histoire a montré de tout temps combien la vérité scientifique était relative. Cette remarque est d'autant plus aigue à notre époque et encore plus dans le secteur de l'informatique. Les résultats obtenus sont provisoires, sujets à être brûlés demain (il est de notoriété publique que les ordinateurs qui sont commercialisés sont déjà périmées à leur sortie !).

Le travail présenté dans cette thèse est centré autour du parallélisme et des différentes formes qu'il peut prendre. Notre intérêt s'est porté sur la recherche du degré maximal de parallélisme contenu dans une classe d'algorithmes donnée (les procédures fondamentales de l'algèbre linéaire et en particulier l'élimination de Gauss). Les motivations de l'analyse théorique de la complexité ont été largement débattue et vous a convaincu, je l'espère. De plus, nous avons développé à la fin de l'ouvrage quelques résultats concrets de parallélisation sur des architectures existantes.

Je voudrais pour finir entrouvrir quelques portes rencontrées au fil de cette thèse.

Tout d'abord, une étude complète sur les réseaux d'interconnexion permettrait une modélisation théorique précise des coût de communication dans les architectures parallèles à mémoires partagées. Il serait alors possible d'obtenir un modèle (dépendant de l'architecture) qui rende compte au mieux de la réalité. L'analyse de la complexité proposée au chapitre 5 reste valide. Il existe également des travaux portant sur les problèmes de synchronisation. Dans notre démarche, ils sont inclus implicitement dans le graphe des tâches, en réalité, il existe des synchronisations matérielles. Il faudrait pouvoir en tenir compte pour connaître exactement leur influence. Ajoutons pour finir que ce peut être fait de la même manière que pour les communications.

L'étude de problèmes issus de la modélisation de phénomènes physiques est une source enrichissante pour les informaticiens. Les études de complexité développées ici portent sur des procédures élémentaires d'algèbre linéaire. Les problèmes que les physiciens et numériciens ont à résoudre font appel à ces outils, que l'on assemble comme s'imbriquent les pièces d'un puzzle. Ainsi, la plupart des algorithmes utilisés pour des problèmes pratiques sont à base de produits matrice-vecteur et de résolutions de systèmes élémentaires (comme dans le cas du gradient conjugué), il faut organiser les rencontres des données, découpler en sous-problèmes indépendants. De nombreux travaux actuels se tournent vers ces aspects très concrets du parallélisme.

Enfin, l'extension de l'analyse de la complexité à des architectures distribuées (en particulier à celles du type hypercube) est primordiale. C'est une des voies les plus prometteuses qui permet de prendre en compte un grand nombre de processeurs. Il faut pouvoir intégrer au modèle de complexité que nous avons développé des coûts de communication variables, qui dépendent de la distance entre les processeurs. Cela consiste en fait à étendre le graphe de précédence en un graphe valué, chaque arête ayant un poids qui représente la distance maximale de transfert des données pour relier les tâches dans une topologie donnée (anneau ou hypercube). Il existe des recherches en cours sur ce problème de l'allocation dynamique de tâches à des processeurs.

REFERENCES

[Agr] D.P. AGRAWAL
Advanced Computer Architectures, Tutorial IEEE, North Holland (1986)

[ADM] H.M. AHMED, J.M. DELOSME, M. MORF
Highly concurrent computing structures for matrix arithmetic and signal processing, IEEE Computer magazine n° 15, 1, pp 65-82 (1982)

[Alm] G.S. ALMASI
Overview of Parallel Processing, Parallel Computing 2, North Holland, pp 191-203 (1985),

[AFQ] F. ANDRE, P. FRISON, P. QUINTON
Algorithmes systoliques : de la théorie à la pratique, RR n° 214 INRIA Roquencourt (1983)

[AB] M. AUGUIN, F. BOERI
Etude comparative de réseaux d'interconnexions dans une architecture MIMD, Congrès sur les nouvelles architectures, Ed. J.P. VERJUS, Eyrolles (1984)

[BW] BELL, WULF
Cmmp a multiprocessor, in Proc. AFIPS Fall Joint Comp., pp 765-777 (1972)

[BBK] A. BOJANCZYK, R.P. BRENT, H.T. KUNG
Numerically stable solution of dense linear systems of linear equations using mesh-connected processors, Technical Report CMU (1981)

[BKL] R.P. BRENT, H.T. KUNG, F.T. LUK
Some linear-time algorithms for systolic arrays, Technical Report CS 82-541Université de Cornell (1982)

[Car] R. CARASSO
Une machine systolique intégrée en CMOS : le GAPP de NCR, Minis et Micros 226 pp 15-16 (1985)

[CKS] S.C. CHEN, D.J. KUCK, A.H. SAMEH
Practical parallel band triangular system solvers, ACM Trans. Math. Software n° 4, 3, pp 270-277 (1978)

[CD] E.G. COFFMAN, P.J. DENNING
Operating systems theory, Prentice Hall (1972)

[CR] P. COMON, Y. ROBERT
A systolic array for computing BA^{-1}, IEEE on ASSP (1987)

[CoRT] P. COMON, Y. ROBERT, D. TRYSTRAM
Mise en oeuvre systolique de Projections , Traitement du Signal Vol. 4, n° 1, pp 73-85 (1987)

[CoT] P. COMON, D. TRYSTRAM
An Incomplete Factorization Algorithm for Adaptive Filtering, Signal Processing 13, pp 353-360 (1987)

[Com] COMPUTING, Integrated Optical Computing (1987)

[Cos] M. COSNARD
Cours INRIA-CNRS Algorithmique distribuée, La Colle sur Loup, (1987)

[CMa1] M. COSNARD, M. MARRAKCHI, Y. ROBERT, D. TRYSTRAM
Parallel Gaussian Elimination on an MIMD Computer, PARALLEL COMPUTING (1987)

[CMa2] M. COSNARD, M. MARRAKCHI, Y. ROBERT, D. TRYSTRAM
Gauss elimination algorithms for MIMD computers, Proc. CONPAR 86, Händler et al. eds., Lecture Notes in Computer Science 237, Springer Verlag pp 247-254 (1986)

[CMuR] M. COSNARD, J.M. MULLER, Y. ROBERT, D. TRYSTRAM
Communication Costs Versus Computation Costs in Parallel Gaussian Elimination, International Workshop : "Parallel Algorithms and architectures", Luminy, Ed. North Holland (1986)

[CR] M. COSNARD, Y. ROBERT
Algorithmique parallèle: une étude de complexité, TSI 6, 2, pp 115-125 (1987)

[CRT1] M. COSNARD, Y. ROBERT, D. TRYSTRAM
Résolution parallèle de systèmes linéaires denses par diagonalisation, Bulletin EDF, Série C n° 2 pp 67-87 (1986)

[CRT2] M.COSNARD, Y. ROBERT, D. TRYSTRAM
Parallel Solution of Dense Linear Systems using Diagonalization Methods , International Journal of Computer Mathematics (1987)

[CTV] M. COSNARD, B. TOURANCHEAU, G. VILLARD
Presentation de l'Hypercube FPS T20, Bigre n° 56 pp 12-17, (1987)

[CT] M. COSNARD, D. TRYSTRAM
Communication complexity of Gaussian elimination on MIMD shared memory computers, Technical Report Ecole Centrale Paris (1986)

[DGK] J.J. DONGARRA, F.G. GUSTAVSON, A. KARP
Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, SIAM Review 26, 1 pp 91-112, (1984)

[Erh] J. EHREL
Parallélisation d'algorithmes numériques, thèse de 3-ième cycle Université d'Orsay (1982)

[EJLT] J. EHREL, W. JALBY, A. LICHNEWSKI, F. THOMASSET
Quelques progrès en calcul parallèle et vectoriel, 6-ième colloque internat. sur les méthodes de calcul scientifique (1983)

[Ens] P.H. ENSLOW
Multiprocessors and other Parallel Systems - An introduction an Overview, Computer Architecture Workshop, Ed. Handler Springer Verlag pp 133-198, (1975)

[ED] D.J. EVANS, R.C. DUNBAR
The parallel solution of triangular systems of equations, IEEE Trans. Computers, C-32 pp 201-204, (1983)

[Fei] M. FEILMEIER
Parallel Computers - Parallel Mathematics, IMACS North Holland (1977)

[Fen] T.Y. FENG
A survey of interconnection networks, Computer (1981)

[Fly1] M.J. FLYNN
Very high-speed computing systems, Proc. IEEE 54 pp 1901-1909, (1966)

[Fly2] M.J. FLYNN
Some computer organisations and their effectiveness, IEEE Trans. on Computers C-21, 9 pp 948-960, (1972)

[FK] M.J. FOSTER, H.T. KUNG
The design of special purpose VLSI chips, IEEE Comuter magazine n° 13, 1 pp 26-40, (1980)

[Fre] W. FRENSLEY
Les transistors en Arseniure de Gallium, Pour La Science n° 120 (1987)

[GKLS] D.D. GAJSKI, D. KUCK, D. LAWRIE, A. SAMEH
CEDAR - A large scale Multiprocessor, Proceedings of International Conference on Parallel Processing pp 524-529, (1983)

[GP1] D.D. GAJSKI, J.K. PEIR
Parallel Processing: Problems and solutions, Technical Report Université d'Illinois (1984)

[GP2] D.D. GAJSKI, J.K. PEIR
Essential issues in multiprocessors systems, IEEE Computer magazine pp 9-27, (1985)

[GJMS] K. GALLIVAN, W. JALBY, U. MEIER, A. SAMEH
The impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design, RR. CSRD, Univ. of Illionois at Urbana-Champaign (1987)

[GK] W.M. GENTLEMAN, H.T. KUNG
Matrix triangularisation by systolic arrays, Proc. of SPIE 298, Real-time processing IV (1981)

[GV] G. H. GOLUB, C. F. VAN LOAN
Matrix computations, The Johns Hopkins University Press (1983)

[Han] W. HANDLER
The impact of Classification Schemes on Computer Architecture, The Proceedings of the International Conference on Parallel, pp 7-15 (1977)

[Hel] D. HELLER
A survey of Parallel Algorithms in Numerical Linear Algebra, SIAM Review 20 pp 740-777 (1978)

[Her] A. HERSCOVICI
Introduction aux grands ordinateurs scientifiques, Eyrolles (1986)

[HJ] R.W. HOCKNEY, C.R. JESSHOPE
Parallel computers: architectures, programming and algorithms, Adam Hilger, Bristol (1981)

[HZ] E. HOROWITZ, A. ZORAT
Divide and conquer for parallel processing, IEEE Trans. on Computers C-32, 6 (1983)

[HB] K. HWANG, F. BRIGGS
Parallel processing and computer architecture, Mac Graw Hill (1984)

[IBM] IBM Systems Journal, Vol. 25, N. 1 (1986)

[Ive] W.R. IVERSEN
Superconductor R&D moves ahead on several world fronts, Electronics (1987)

[Kog] P.M. KOGGE
The architecture of pipelined computers, Mac Graw Hill, NY (1981)

[KRT] J.C. KONIG, Y. ROBERT, D. TRYSTRAM
Optimalité d'une classe d'algorithmes d'ordonnancement pour le graphe 2-pas, C.R.A.S. N. 5, Série 1, pp. 295-298 (1988)

[Kro] L. KRONSJO
Computational complexity of sequential and parallel algorithms, John WILEY & Sons (1985)

[Kuc1] D.J. KUCK
High Speed Machines and their Compilers, Proceedings of the CREST Parallel Processing Systems Course, Cambridge Univ. Press (1980)

[Kuc2] D.J. KUCK
The structure of computers and computations, Wiley & Sons, NY (1978)

[KY] A.V. KULKARNI, D.W. YEN
Systolic processing and an implementation for signal and image processing, IEEE Trans. on Computers C-31, 10 pp 1000-1009 (1982)

[Kum] S.P. KUMAR
Parallel algorithms for solving linear equations on MIMD computers, PhD. Thesis, Washington State University (1982)

[KK] S.P. KUMAR, J.S. KOWALIK
Parallel factorization of a positive definite matrix on an MIMD computer, Proc. Int. Conf. ICCD 84 pp 410-416 (1984)

[Kun1] H.T. KUNG
The structure of parallel algorithms, Advances in Computers n°19 pp 65-112 (1980)

[Kun2] H.T. KUNG
Use of VLSI in algebraic computation: some suggestions, in Proc. SYMSAC 81, Ed. ACM, NY (1981)

[Kun3] H.T. KUNG
Why systolic architectures ?, IEEE Computer magazine n° 15, 1 pp 37-46 (1982)

[KLa] H.T. KUNG, M.S. LAM
Fault-tolerance and two-level pipeliningin VLSI systolic arrays, Technica
Report université de Carnegie Melon (1983)

[KLeh] H.T. KUNG, P.L. LEHMAN
Systolic VLSI array for relational database operations, Proc. of th
ACM/SIGMOD Internat. Conf. on management of data, Los Angeles (1980)

[KLei] H.T. KUNG, C.E. LEISERSON
Systolic arrays for VLSI, in Proc. of the symposium on Sparse Matrice
Computations, Knoxville, Tennessy pp 256-282 (1978)

[LT1] P. LAURENT-GENGOUX, D. TRYSTRAM
Practical Aspects from Linear Systems Solvers on Parallel Computers , Conf
ICIAM, Paris, La Villette (1987)

[LT2] P. LAURENT-GENGOUX, D. TRYSTRAM
Parallel Conjugate Gradient Algorithm with local Decomposition, à paraître

[LS] D.H. LAWRIE, A.H. SAMEH
The Computation and Communication Complexity of a Parallel Bandec
System Solver, ACM Trans. on Math. Software, vol 10, n° 2, pp 185-195
(1984)

[LS] C.E. LEISERSON, J.B. SAXE
Optimizing synchronous systems, Proc. of the 22-th annual symposium on
foundations of Computer Science, IEEE pp 23-36 (1981)

[Lef] P. LEFEVRE
Le GAPP - microprocesseur matriciel à architecture systolique, bulletin de la
DER/EDF C-3 pp 5-30 (1986)

[Len] J. LENFANT
Quelle architecture pour les gigaflops ?, bulletin de liaison INRIA n. 4, spécial
parallélisme pp 2-10 (1984)

[LM] J. LENFANT, C. MICHEL
Les réseaux d'interconnexions, Congrès sur de nouvelles architectures,
Eyrolles (1984)

[LW] G.G. LI, B.W. WAH
The design of optimal systolic arrays, IEEE Trans. on Computers 34, N. 1, pp
66-77 (1985)

[LKK] R.E. LORD, J.S. KOWALIK, S.P. KUMAR
Solving linear algebraic equations on an MIMD computer, J. ACM 30, 1 pp 103-117 (1983)

[Mir] MIRANKER
A Survey of Parallelism in Numerical Analysis, SIAM Review 13 pp 524-547 (1971)

[MR1] M. MARRAKCHI, Y. ROBERT
Un algorithme parallèle pour la méthode de Gauss. C.R. Acad. Sc. Paris 303, série I, pp 425-429 (1986)

[MR2] M. MARRAKCHI, Y. ROBERT
Optimal Algorithms for Gaussian Elimination on a MIMD Computer, Rapport de Recherches IMAG-RT34 (1988)

[MC] C. MEAD, L. CONWAY
Introduction aux systèmes VLSI, InterEditions (1983)

[MK] J. MISKLOSKO, V. KOTOV
Algorithms, Software and Hardware of parallel computers, Springer Verlag (1985)

[OV] J.M. ORTEGA, R.G. VOIGT
Solution of Partial Differential Equations on vector and parallel computers, SIAM Review n° 27, 2 pp 149-240 (1985)

[PV] F.P. PREPARATA, J. VUILLEMIN
Area-time optimal VLSI Networks for multiplying matrices, Information Proceedings letters 11, N. 2, pp 77-80 (1980)

[Ray] M. RAYNAL
Algorithmes Distribués & Protocoles, Eyrolles (1986)

[Rob] Y. ROBERT
Algorithmique Parallèle : réseaux d'automates, architectures systoliques, machines MIMD, Thèse d'état INP Grenoble (1986)

[RT1] Y. ROBERT, D. TRYSTRAM
An orthogonal systolic array for the Algebraic Path Problem , Computing n° 39 pp 187-199 (1987)

[RT2] Y. ROBERT, D. TRYSTRAM
Parallel Implementation of the APP, Proc. CONPAR 86, Händler et al. eds., Lecture Notes in Computer Science 237, Springer Verlag, pp 247-254 (1986)

[RT3] Y. ROBERT, D. TRYSTRAM
Systolic Solution of the Algebraic Path Problem, International Workshop on Systolic Arrays, OXFORD Ed. W. MOORE chez Adam HILGER (1986)

[RT4] Y. ROBERT, D. TRYSTRAM
Comments on Scheduling Parallel Iterative Methods on Multiprocessor systems, Parallel Computing, à paraître

[RT5] Y. ROBERT, D. TRYSTRAM
Optimal Sheduling Algorithms for Parallel Gaussian Elimination, Proceedings of EUCOPE 87, North Holland à paraître

[Rot] G. ROTE, A systolic array algorithm for the Algebraic Path Problem, Computing n° 34 pp 191-219 (1985)

[RV] G. ROUCAIROL, J.P. VERJUS
Parallélisme, communication et synchronisation , Ed. CNRS (1985)

[Rus] R.M. RUSSELL
The CRAY-1 Computer System, Comm. of the ACM pp 63-72 (1978)

[Ryc] J. RYCKBOSCH
A Method for Solving the Problem of Optimal Switching, Proceedings of IFAC, Pekin (1987)

[RTV] J. RYCKBOSCH, D. TRYSTRAM, F. VINCENT
Résolution d'un problème de réseau par Gradient Conjugué, à paraître au Bulletin EDF-DER

[Saa] Y. SAAD
Communication complexity of the Gaussian elimination algorithm on multiprocessors, Report DCS/348, Yale University (1985)

[Sam1] A. H. SAMEH
Numerical parallel algorithms - a survey, in "High-speed Computer and Algorithm Organization", Academic Press pp 207-228 (1977)

[Sam2] A. H. SAMEH
An overview of parallel algorithms, bulletin DER-EDF C1 pp 129-134 (1983)

[SK] A. H. SAMEH, D.J. KUCK
On stable parallel linear system solvers, J. of ACM 25, 1 pp 81-91 (1978)

[Sche] U. SCHENDEL
Introduction to Numerical Methods for Parallel Computers, Ellis Horwood Series, Wiley & Sons, NY (1984)

[Schr] R. SCHREIBER
Systolic arrays: high performance parallel machines for matrix computation, Proc. Elliptic Problem Solvers II, Academic Press pp 187-194 (1984)

[Sri] M. SRINIVAS
Optimal parallel scheduling of Gaussian elimination DAG's, IEEE Trans. on Computers C32 pp 1109-1117 (1983)

[Sto] H.S. STONE
Introduction to Computer Architecture, Science Res. Associates, Inc. (1983)

[Tch] M. TCHUENTE
Contribution à l'étude des méthodes de calcul pour des systèmes de type coopératif, Thèse d'état université de Grenoble (1982)

[Tho] F. THOMASSET
Cours de DEA Université d'Orsay (1984)

[TBH] P.C. TRELEAVEN, D.R. BROWNBRIDGE, R.P. HOPKINS
Data-driven and demand-driven computer architecture, ACM Comp. Surveys 14, pp 93-143 (1982)

[Try1] D. TRYSTRAM
Expérimentation d'algorithmes de préconditionnement pour des grands systèmes creux, Thèse de Docteur Ingénieur de l'INPG (1984)

[Try2] D. TRYSTRAM
On the solution of dense linear systems by diagonalization methods with parallel computers, conférence "Future Trends of Computing", Grenoble, Ed. CHENIN et al. Masson-Wiley (1985)

[Ull] J.D. ULLMAN
Computational aspects of VLSI, Computer Science Press, Rockville, Maryland (1984)

[Vel] M. VELDHORST
A note on Gaussian elimination with partial pivoting on an MIMD computer, Technical Report RUU-CS-84-14, University of Utrecht, NL (1984)

[Vui] J. VUILLEMIN
Nouvelles structures d'ordinateurs, Proc. VLSI Algorithms and Architectures, Elsevier Science pp 137-151 (1985)

[WS] C. WHITBY-STREVENS
The Transputer, Proceedings of the Symposium on Computer Architecture (1984), 151-156

RECUEIL D'ARTICLES

Intern. J. Computer. Math., 1987, Vol. 00, pp. 000-000
Photocopying permitted by license only
© 1987 Gordon and Breach, Science Publishers, Inc.
Printed in Great Britain

Ms B1

Parallel Solution of Dense Linears Systems using Diagonalization Methods

M. COSNARD and Y. ROBERT

C.N.R.S.-Laboratoire TIM3, I.N.P.G., 38031 Grenoble Cedex, France

and

D. TRYSTRAM

Ecole Centrale Paris, Grande voie des vignes, 92295 Chatenay Malabry Cedex, France

(Received March 1986; in final form December 1986)

We study the parallel implementation of two diagonalization methods for solving dense linear systems: the well known Gauss-Jordan method and a new one introduced by Huard. The number of arithmetic operations performed by the Huard method is the same as for Gaussian elimination, namely $2n^3/3$, less than for the Jordan method, namely $n^3/3$. We introduce parallel versions of these methods, compare their performances and study their complexity. We assume a shared memory computer with a number of processors p of the order of n, the size of the problem to be solved. We show that the best parallel version for Jordan's method is by rows whereas the best one for Huard's method is by columns. Our main result states that for a small number of processors the parallel Huard method is faster than the parallel Jordan method and slower otherwise. The separation is obtained for $p = 0.44n$.

KEY WORDS: Parallel numerical algorithms, linear system solver, Jordan method, diagonalization methods, parallel computational complexity.

C.R. CATEGORIES: G.1.0, F.2.1.

This work has been supported by the Centre National de la Recherche Scientifique through the GRECO C3.

0. INTRODUCTION

The well-known Gaussian elimination method is the most commonly used algorithm to solve linear systems of equations on sequential computers. It consists of two distinct steps: triangularizing the matrix, and then solving a triangular system. Another way to proceed is to gather these two steps into a single one: diagonalizing the matrix, usually via the Jordan method. But the Jordan method is more costly than that of Gauss ($n^3+O(n^2)$ arithmetic operations against $2n^3/3+O(n^2)$). No theoretical justification of this difference has been given for a long time.

Recently, Huard [10] has introduced another method which can be viewed as a variation of Gaussian elimination that computes the solution of the system while triangularizing it, hence leading to the diagonalization of the matrix within the same number of arithmetic operations.

Most often, pivoting of rows or columns (or both) is used for stability reasons. However we shall not consider here the overhead due to pivoting. A stability analysis of the Jordan method can be found in Golub and Van Loan [8]. In the case of symmetric definite positive matrices, the Huard method has been proven to be equivalent to the conjugate directions method by Laurent-Gengoux and Trystram [14].

Rather, we concentrate on evaluating the amount of arithmetic operations which are required by the different algorithms, and we neglect the time needed for data accessing, retrieving and exchanging. Synchronization in parallel algorithms is also important, but is machine dependant and hence behind the scope of this paper: compare parallel implementations of Jordan and Huard methods from a computational complexity view point.

When we have more than one processor, a method can lead to various parallel versions. The number of processors is not the only parameter since the constraints imposed by the structure of the machine (see the various taxonomies of Flynn [6], Gajski [7], Treleaven [22], the systolic model, Kung [13], Robert [17]...) can modify the performances of the method.

However, we shall assume in the following that we can access the elements of the matrix A, either by rows or by columns. Then the two allowed transfer operations will be loading and storing a row or

a column. The duplication of a row or a column will have a zero cost, i.e. we assume that it is possible to transfer simultaneously the same data to various processors. In this case, no processor can modify this data. Conversely, a processor can modify a data if it is the only one to possess the data. We shall not study the technical mechanism which solves the problem of data allocation without conflict. We remark simply that the preceding constraints and the associated mechanisms lead to an unambiguous execution of the algorithm.

Once the constraints defined, the first step in the parallelization of a method is the definition of the elementary tasks and their precedence graph. This graph shows the temporal dependency of the operations of the algorithm. This warrants the absence of conflict, the temporal integrity of the data and a perfect correspondance between the method and its parallel version. The tasks are then assigned to the available processors according to the precedence graph.

The task allocation problem is difficult and leads to optimality and complexity studies. The parallelization of Gaussian elimination has been studied by Lord, Kowalik and Kumar [15] in case of an unbounded number of processors and by Srinivas [21] in case of a fixed number of processors. Parallel algorithms for the QR decomposition have been proposed in case of $[n/2]$ processors ([ ] is the floor function) by Cosnard and Robert [3] for square matrices and Cosnard, Muller and Robert [5] in the rectangular case; see also the papers of Modi and Clarke [16] and of Sameh and Kuck [18].

In the contrary, no work has been done on the parallelization of the diagonalization methods. In this paper, we introduce parallel algorithms for this class of methods, we compare their performances, and we study their complexity. We shall assume that the computational cost of each elementary arithmetic operation is the same: this will be our basic unit.

1. BACKGROUND: JORDAN AND HUARD METHODS

In what follows we suppose that we have to solve a dense linear systems $Ax=b$. Hence we shall work with a matrix A of order n by $n+1$.

1.1. Jordan method

The idea is the same for the Jordan method as for that of Gauss, except that we are looking for a diagonal matrix at the end. In matrix notation, the Jordan method can be written as $J_n J_{n-1} \dots J_1 A = D$ (D is a diagonal matrix).

```
(*Jordan method*)
For k←1 to n
  (*updating row k*)

  c←1/a_kk

  For j←k+1 to n+1

    a_kj←a_kj*c

  (*zeroing out the off-diagonal elements in column k*)
  For i←1 to n,  i≠k
    For j←k+1 to n+1

      a_ij←a_ij−a_ik*a_kj
```

1.2. Huard method

As in the Jordan method, the matrix A is transformed into a diagonal matrix. But the off-diagonal elements are annihilated in a different order: at each step k the first $k-1$ elements of both row and column k are eliminated. The diagonal matrix which is obtained at the end is the same as in the Jordan method: we have $P_n P_{n-1} \dots P_1 A = D$. Hence the Huard method can be seen as a variation of the Gaussian elimination method. The sequential algorithm is:

```
(*Huard method*)
For k←1 to n
  (*zeroing part of row k up to the main diagonal*)
  For i←1 to k−1
    For j←k to n+1
```

$a_{kj} \leftarrow a_{kj} - a_{ki} * a_{ij}$

(*updating row k*)

$c \leftarrow 1/a_{kk}$

For $j \leftarrow k+1$ to $n+1$

$a_{kj} \leftarrow a_{kj} * c$

(*zeroing part of column k up to the main diagonal*)
For $i \leftarrow 1$ to $k-1$
For $j \leftarrow k+1$ to $n+1$

$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

More precisely, we can compare the costs of these sequential methods:

	div.	multiplications	additions
Gauss	n	$n^3/3+n^2-n/3$	$n^3/3+n^2/2-5n/6$
Jordan	n	$n^3/2+n^2/2$	$n^3/2-n/2$
Huard	n	$n^3/3+n^2-n/3$	$n^3/3+n^2/2-5n/6$

This table shows that the Gaussian elimination and the Huard methods have exactly the same cost in arithmetic operations. Jordan's method is more expensive.

2. AN OPTIMALITY RESULT

A sequential algorithm can lead to various parallel versions. Following Lord et al. [15], we define a task system from the original sequential algorithm and deduce the precedence constraints. A parallel algorithm is obtained by assigning these tasks to the processors respecting the constraints. The processors execute their tasks concurrently. Hence two parallel versions can have different execution times. Clearly an algorithm is optimal if it requires the least number of operations.

In this section, we prove an optimality result that will be useful for the parallelization of several algorithms later on. We consider the parallel solution, with p processors, of a task system $\{T_{kj}, 1 \leq k \leq n, k+1 \leq j \leq n+1\}$. We let the relation $\ll$ be the precedence constraint (see [15]) which means that if $T \ll T'$, then task T is to be completed before task T' can start its execution. We assume that the constraints in the precedence graph are the following:

$$T_{k,k+1} \ll T_{k+1,j} (j \geq k+2) \qquad \text{(A)}$$

$$T_{k,j} \ll T_{k+1,j} (j > k) \qquad \text{(B)}$$

For a given value of k, there are $n+1-k$ tasks to be executed. We shall make the additional assumption that all the tasks $T_{kj}(j>k)$ of a same level k require the same execution time t_k. The precedence graph is figured below:

$T_{12} T_{13} T_{14} \dots T_{1,n+1}$ level 1

$T_{23} T_{24} \dots T_{2,n+1}$ level 2

... ...

$T_{n,n+1}$ level n

Let $T \leqq T'$ denote the following precedence constraint: the execution of T' can not begin before that of T (but a simultaneous execution is allowed). We have the

PROPOSITION 1 *There exists optimal parallel algorithms which satisfy the additional precedence constraints* (C):

$$T_{k,j} \underline{\ll} T_{k,j+1} (j \geq k+1) \qquad \text{(C)}$$

Proof Let M be any parallel algorithm. We construct a parallel algorithm M^* which has the same execution time as M and satisfies the constraint (C):

—if at time t, the algorithm M starts the execution of q tasks of level k, M^* starts the execution of q tasks of level k too, but it starts

the first q not yet executed tasks of level k (according to the ordering imposed by the constraint (C))

—if at time t, the algorithm M does not start the execution of any task level k, the same holds for M^*.

Hence at any time during the execution, M^* has executed the same number of tasks of each level as M. We still have to prove that the precedence constraints (A) and (B) are met by M^*. For (A) this is evident since $T_{k,k+1}$ is the first task of level k executed by M^*.

For the constraint (B), let us assume that M starts the execution of q tasks of level $k+1$ at time t. Let s_k and s_{k+1} be respectively the numbers of tasks of level k and $k+1$ having been processed by M (and M^*) at time t. Since M satisfies the constraints (A) and (B), $s_k \geq s_{k+1}+q+1$. At time t, M^* starts the execution of the tasks $T_{k+1,k+j}$, where $s_{k+1}+1 \leq j \leq s_{k+1}+q+1$. Since M^* has already completed the tasks $T_{k,k+j}$, where $1 \leq j \leq s_k$, it does satisfy the precedence constraint (B). □

Now we introduce the greedy algorithm G, that we can informally define as follows: the algorithm G executes the tasks from one row of the precedence graph to another, from left to right in each row, and it starts the maximum number of tasks at each time (hence the name of greedy). More precisely, the algorithm G executes the tasks in the following order:

$$T_{12} \ll T_{13} \ll \cdots \ll T_{1,n+1} \ll T_{23} \ll T_{24} \ll T_{2,n+1} \ll T_{3,4} \ll \cdots \ll T_{n,n+1}$$

At time $t=1$, the algorithm G starts the execution of the tasks T_{12}, $T_{13}, \ldots, T_{1p}$. For any $t \geq 1$, if $q \leq p$ processors are idle, the algorithm G assigns these q processors to the execution of the following q tasks, provided that the constraint (B) makes it possible (otherwise, the algorithm executes the maximum number of tasks).

PROPOSITION 2 *If $t_k - t_{k-1} = O(1)$, then the greedy algorithm G is asymptotically optimal. Its execution time is equal to*

$$TG = (1/p) \sum_{i=1}^{n+1-p} (n+1-i).t_i + \sum_{i=n-p}^{n} t_i + O(t_{n-p+1})$$

Proof Using Proposition 1, we show that the greedy algorithm is asymptotically optimal in the class of the algorithms which satisfy to the precedence constraints (A), (B) and (C).

We first construct a lower bound for the optimal execution time by noting that:

—in order to execute $T_{n-p+1,n+1}$ all the tasks of the preceding levels must have been executed,

—from level $n-p$ up to level n, there is less than p tasks per level; the precedence constraints imply that these levels must be executed one after the other.

Hence the best that can be done is to execute level 1 to $n-p$ with efficiency one and after proceed level by level. From this, we deduce that:

$$T \geq (1/p) \sum_{i=1}^{n+1-p} (n+1-i).t_i + \sum_{i=n-p}^{n} t_i$$

In the greedy algorithm, we notice that all the p processors are activated without any latency until the execution of the task $T_{n+1-p,n+1}$ since there are more than p tasks per level up to level $n+1-p$. The tasks $T_{n+2-p,n+1}$, $T_{n+3-p,n+1}, \ldots, T_{n,n+1}$ are then executed sequentially. Let

$$T_1 = (1/p) \sum_{i=1}^{n+1-p} (n+1-i).t_i \quad \text{and} \quad T_2 = \sum_{i=n-p}^{n} t_i$$

From the hypothesis, we deduce that the execution of $T_{n+1-p,n+1}$ is terminated before $T_1 + 0(t_{n-p+1})$ time units. The remaining tasks can be executed in less than T_2 units. Since t_{n-p+1} is a lower order term with respect to $T_1 + T_2$, this concludes the proof. □

3. PARALLEL JORDAN

Data access lead to various task decomposition of the method.

```
{JORDAN by rows}
For k←1 to n
  For j←k+1 to n+1
```

```
{JORDAN by columns}
For k←1 to n
  For j←k+1 to n+1
```

```
do a_kj <- a_kj/a_kk
For i <- 1 to n,   i ≠ k
  For j <- k+1 to n+1
    do a_ij <- a_ij - a_ik * a_kj
```

```
do a_kj <- a_kj/a_kk
For i <- 1 to n,   i ≠ k
  do a_ij <- a_ij - a_ik * a_kj
```

3.1. Jordan by rows

We can define the following tasks:

$\bullet JR_{kk}$: ⟨**For** $j \leftarrow k+1$ **to** $n+1$
do $a_{kj} \leftarrow a_{kj}/a_{kk}$⟩

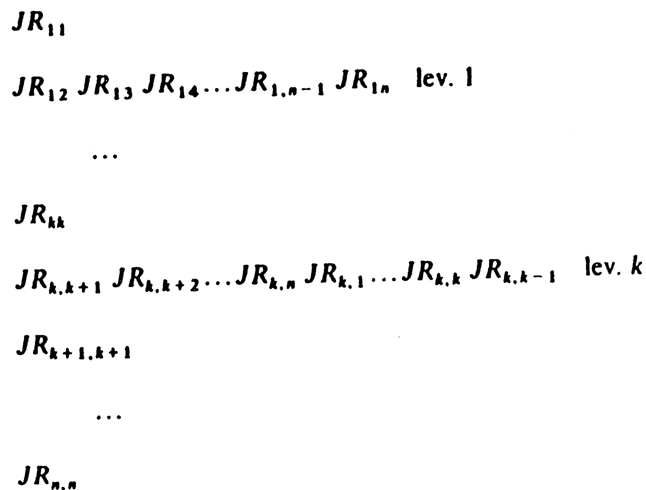
$\bullet JR_{ki}$: ⟨**For** $j \leftarrow k+1$ **to** $n+1$, $i \neq k$
do $a_{ij} \leftarrow a_{ij} - a_{ij} * a_{kj}$⟩

JR_{kk} can be processed in $n+1-k$ units and JR_{ki}, $i \neq k$, in $2(n+1-k)$ units. The precedence constraints are:

$$JR_{kk} \ll JR_{ki} \quad \text{for} \quad i \neq k \tag{A}$$

$$JR_{ki} \ll JR_{k+1,i} \quad \text{for} \quad i \neq k \tag{B}$$

which leads to the precedence graph:



$JR_{n,1}JR_{n,2}\ldots JR_{n,n-1}$ lev. n

i) computation of JR_{opt}
The longest path in the graph is:

$$\{JR_{11}, JR_{12}, JR_{22}, JR_{23}, \ldots, JR_{kk}, JR_{k,k+1}, \ldots, JR_{n,n} JR_{n,1}\}$$

and hence $JR_{opt} = 3n(n+1)/2$.

ii) asymptotic efficiency with $n-1$ processors.

It is easy to construct an algorithm using $p=n-1$ processors in time JR_{opt}. The asymptotic efficiency is $EJR_{n-1,\infty} = n^3/n(3n^2/2) = 2/3$. Recall that the efficiency $E_{p,n}$ is the ratio between the sequential execution time and the execution time with p processors. The asymptotic efficiency is the limit of $E_{p,n}$ when n goes to infinity.

iii) optimal execution time $p \leq (2n/3)$ processors.

We assume that $p = \alpha \cdot n$, with $\alpha \leq 2/3$. We introduce an asymptotically optimal algorithm and evaluate its execution time. Without loss of generality we set $p = 2q+1$.

We assign all the tasks of the longest path to the first processor which process them as soon as possible. Then we divide the remaining tasks into q blocks if $r=(n-2)/q$ consecutive columns of the precedence graph and we assign their execution to a pair of processors. For example, processors 2 and 3 process the tasks corresponding to columns 1 to r and processors $2i$ and $2i+1$ the tasks of columns $(i-1)r+1, (i-1)\,r+2, \ldots, ir$.

We shall present the scheduling of the first pair of processors. The remaining pairs operate in a similar way, using a simple change of the indices. $P2$ and $P3$ process columns 1 to r. For simplicity, we assume that $r = 3s$ (hence s is asymptotically equal to $2n/(3p)$).

For $s=1$ (which corresponds to $\alpha = 2/3$), the processors operate along the following scheme (ij in column Ps means that processor Ps processes the task JR_{ij}):

Since the execution time of JR_{kl} is twice this of JR_{kk}, we divided (conceptually) their execution into two parts. Remark that $P1$, $P2$ *and* $P3$ are unemployed during at most one time unit when they process a task. More precisely, we loose 2 time units per level with respect to an algorithm using JR_{opt}. We deduce that the execution time is:

P1	P2	P3	Allocated time
11			$n+1$[a]
12	13	—	$n+1$[a]
12	13	14	n
22	15	14	n[b]
23	15	24	n[b]
23	25	24	$n-1$
33	25	26	$n-1$[b]
34	35	26	$n-1$[b]
34	35	36	$n-2$
44	37	36	$n-2$[b]
45	37	46	$n-2$[b]
45	47	46	$n-3$
55	47	48	$n-3$[b]
56	57	48	$n-3$[b]
...	...	...	...

[a]Although n time units would be sufficient to process these tasks, we allow $n+1$ time units for their execution, in order to respect the general formalism of the algorithm.
[b]Means that at least one processor is idle during one time unit.

$$JR_{alg} = JT_{opt} + 2n$$

Hence it is asymptotically optimal and its efficiency is:

$$EJR_{2n/3,\infty} = n^3/(2n/3) \cdot (3n^2/2) = 1$$

Now, we deal with the general case ($s \geqq 1$). $P2$ and $P3$ begin processing the same tasks in levels 1 and 2 as precedently, but before processing levels 3 and 4, they finish the execution of the tasks of levels 1 and 2, by blocks of 3 columns, as indicated in the following example for $s=2$.

The execution rate of the processors (except $P1$) is asymptotically equal to 1, which shows that the algorithm is of asymptotic efficiency 1 and hence asymptotically optimal. Its execution time is now equal to s times JR_{alg}:

$$s \cdot JR_{alg} = (2n/3p) \cdot (3n^2/2) = n^3/p = n^2/a$$

$P1$	$P2$	$P3$	Allocated time
11			$n+1$[b]
12	13	—	$n+1$[b]
12	13	14	n
22	15	14	n[b]
23	15	24	n[b]
23	25	24	$n-1$
33	25	26	$n-1$[b]
—	16	26	n[b]
—	16	17	n
—	18	17	n
—	18	27	n[b]
—	28	27	$n-1$
—	28	29	$n-1$
34	35	29	$n-1$[b]
34	35	36	$n-2$
44	37	36	$n-2$[b]
45	37	46	$n-2$[b]
45	47	46	$n-3$
55	47	48	$n-3$[b]
—	38	48	$n-1$[b]
—	38	39	$n-2$
—	3, 10	39	$n-2$
—	3, 10	49	$n-2$[b]
—	4, 10	49	$n-3$
—	4, 10	4, 11	$n-3$
56	57	4, 11	$n-3$[b]
...	...	...	...

Remark that it is not useful to have more than $2n/3$ processors (although the maximum number of tasks per level is $n-1$): there exists an algorithm of time T_{opt} using only $2n/3$ processors. Indeed, we know that $\alpha=2/3$ is the minimal value in order to obtain an algorithm of time equivalent to T_{opt} using $p=\alpha\cdot n$ processors: for $\alpha<2/3$, the execution time of an optimal algorithm is n^2/α.

Moreover, since the algorithms are asymptotically optimal, the various evaluations of the efficiency are complexity results on the parallelization of this method. We deduce the following result:

PROPOSITION 1

i) $JR_{opt}=3n^2/2$

ii) $EJR_{n,\infty}=2/3$

iii) for $p \leqq 2n/3$, $p=\alpha \cdot n$, *there exists an asymptotically optimal algorithm of efficiency* 1*; its execution time is* n^2/α.

iv) the minimal value of α *in order to obtain an algorithm of time* JR_{opt} *is* $\alpha=2/3$.

v) for $p \geqq 2n/3$, $p=\alpha \cdot n$, $EJR_{p,\infty}=2/3\alpha$.

3.2. JORDAN by columns

The tasks are defined in the following way:

$$\bullet JC_{kj}: \langle a_{kj} \leftarrow a_{kj}/a_{kk}$$

$$\textbf{For } i \leftarrow 1 \textbf{ to } n, \quad i \neq k$$

$$\textbf{do } a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj} \rangle$$

The execution time of JC_{kj} is $2n-1$.

The precedence constraints are:

$$JC_{k,k+1} \ll JC_{kj} \quad \text{for} \quad j>k+1 \tag{A}$$

$$JC_{kj} \ll JC_{k+1,j} \quad \text{for} \quad j>k \tag{B}$$

which leads to the following graph:

$$JC_{12}\ JC_{13}\ JC_{14} \ldots JC_{1,n+1} \quad \text{lev. 1}$$

$$JC_{23}\ JC_{24} \ldots JC_{2,n+1} \quad \text{lev. 2}$$

$$\ldots JC_{n,n+1} \quad \text{lev. } n$$

This graph satisfies the hypotheses of section 2.

PROPOSITION 4

i) There exists optimal algorithms which satisfy the additional precedence constraint (C):

$$JC_{k,j} \ll JC_{k,j+1} (j \geqq k+1) \quad \text{(C)}$$

ii) The greedy algorithm is optimal.

iii) For $p=\alpha \cdot n$, $JC_p=(1+\alpha^2)n^2/\alpha+O(n)$ *and* $EJC_p=1/(1+\alpha^2)$

Proof. Following the proof of the Proposition 2 we can show that, if t_k is independant of k, the greedy algorithm is indeed optimal (since it is synchronous). The formula of Section 2 is then:

$$JC_p=(1/p)\sum_{i=1}^{n+1-p}(n+1-i)\cdot(2n-1)+\sum_{i=n+2-p}^{n} 2n-1$$

which gives the result:

$$JC_p=((n+1-p)(n+p)/2p+p-1)^*(2n-1)$$

Let $p=\alpha \cdot n$. We obtain:

$JC_p=((1+\alpha^2)/\alpha)n^2+o(n)$ □

4. PARALLEL HUARD

The Huard method can be decomposed in two steps per iteration:

—annihilation of the $k-1$ first elements in row k and corresponding updating of the $n+2-k$ following elements: descent step

—annihilation of the $k-1$ first elements in column k and corresponding updating of the $n+1-k$ last elements of the $k-1$ first rows: climb step.

The access to the elements of the matrix leads to two different versions of each steps: by rows or by columns. Hence we obtain four different algorithms as follows:

```
(*descent by rows*)
For i←1 to k−1
  For j←k to n+1
    a_kj←a_kj−a_ki*a_ij
```

```
(*descent by columns*)
For j←k to n+1
  For i←1 to k−1
    a_kj←a_kj−a_ki*a_ij
```

```
(*climb by rows*)
For i←1 to k−1
  For j←k+1 to n+1
    a_ij←a_ij−a_ik*a_kj
```

```
(*climb by columns*)
For j←k+1 to n+1
  For i←1 to k−1
    a_ij←a_ij−a_ik*a_kj
```

The descent by rows cannot be parallelized since, for each value of i, the $n+2-k$ elements a_{ki} are updated. Parallel updating of these elements would contradict our assumptions that a given element cannot be modified simultaneously by various processors. Hence two versions are allowed: descent columns—climb rows and descent columns—climb columns.

4.1. Columns—rows parallel Huard

```
(*descent columns—climb rows*)
For k←1 to n
  (*descent columns*)
  For j←k to n+1
    For i←1 to k−1
      a_kj←a_kj−a_ki*a_ij

  (*preparing row k*)
  For j←K+1 to n+1
    a_kj←a_kj/a_kk

  (*climb rows*)
  For i←1 to k−1
    For j←k+1 to n+1
      a_ij←a_ij−a_ik*a_kj
```

Three different tasks can be considered:

—D_{kj}: For $i \leftarrow 1$ to $k-1$
  $a_{kj} \leftarrow a_{kj} - a_{ki} * a_{ij}$

—C_{kj}: $a_{kj} \leftarrow a_{kj}/a_{kk}$
—R_{ik}: For $j \leftarrow k+1$ to $n+1$
  $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

The execution times of the $n+2-k$ tasks D_{kj}, of the $n+1-kC_{kj}$ and of the $k-1R_{ik}$ are respectively $2(k-1)$, 1 and $2(n+1-k)$ units. The precedence constraints are the following:

—D_{kk} and D_{kj} precedes C_{kj}

—for all i and j, C_{kj} precedes R_{ik}

—for all i and j, R_{ik} precedes D_{k+1}, j.

Hence we obtain the task graph:

$$
\begin{array}{llllll}
C_{12} & C_{13} & C_{14} & \cdots & C_{1,n+1} \\
D_{22} & D_{23} & D_{24} & \cdots & D_{2,n+1} \\
 & C_{23} & C_{24} & \cdots & C_{2,n+1} \\
R_{12} & & & & \\
 & \cdots & \cdots & \cdots & \cdots \\
 & D_{kk} & D_{kk+1} & \cdots & D_{k,n+1} \\
 & & C_{kk+1} & \cdots & C_{k,n+1} \\
R_{1k} \quad R_{2k} \cdots R_{k-1,k} & & & & \cdots \\
\cdots & \cdots & \cdots & &
\end{array}
$$

Call $HCRP_p$ the execution time of an optimal algorithm using p processors and $EHCR_p$ the corresponding asymptotic efficiency.

PROPOSITION 5 *For*

$$p=\alpha n, \; HCR_p \geqq n^2(2+6\alpha^2-2\alpha^3)/(3\alpha)+O(n/\alpha)$$

$$EHCR_p \leqq 1/(1+3\alpha^2-\alpha^3)$$

Proof The constraints imply that the execution of the tasks R_{ik} cannot begin before the end of C_{kj} and D_{kj}, given k, for all i and j. Hence we can consider independently the execution of C_{kj} and D_{kj} and of R_{ik}.

Let us first compute the optimal time T_1 for executing all the C_{kj} and D_{kj}. Clearly T_1 is the sum of the t_k corresponding to the execution time of the C_{kj} and D_{kj} for a given k.

An obvious lower bound for t_k is obtained by assuming that C_{kj} has 0 cost:

$$t_k \geqq [(n+2-k/p](2k-2)$$

The following algorithm produces an upper bound:

Processor $q+1$ executes $D_{k,k+q+rp}$ and $C_{k,k+q+rp}$

with $q=0,\ldots,p-1$, $r=0,\ldots,[(n+2-k-p)/p]$ and $C_{k,k}$ an empty task of cost 1. We obtain:

$$t_k \leqq [(n+2-k)/p](2k-1)$$

and deduce:

$$T_1 = \sum_{k=2}^{n} [(n+2-k)/p](2k-1) + O(n^2/p)$$

The preceding formula is difficult to evaluate in the general case. We shall only derive a realistic lower bound. For this we divide the sum in two parts:

$$T_1 = \sum_{k=2}^{n-p+1} [(n+2-k)/p](2k-1) + \sum_{k=n-p+2}^{n} (2k-1) + O(n^2/p)$$

Hence we obtain:

$$T_1 \geqq \sum_{k=2}^{n-p+1} (n+2-k)(2k-1)/p + \sum_{k=n-p+2}^{n} (2k-1) + O(n^2/p)$$

$$T_1 \geqq (n-p)^2(n+2p)/3p + p(2n-p) + O(n^2/p)$$

We can now compute the optimal time T_2 for executing all the R_{ik}. Once again T_2 is the sum of the r_k corresponding to the execution time of the R_{ik} for a given k. In this case the following greedy algorithm is optimal:

Processor q executes $R_{q+rp,k}$ with $q=1,\ldots,p$ and $r=0,\ldots,[(k-p)/p]$.

This gives:

$$r_k = [(k-1)/p]2(n+1-k)$$

For $i \leftarrow 1$ to k

$a_{k+1,j} \leftarrow a_{k+1,j} - a_{k+1,i} * a_{ij}$

We consider the following tasks P_{kj}.

$P_{kj}: a_{kj} \leftarrow a_{kj}/a_{kk}$ $\quad 1 \leq k \leq n-1$

For $i \leftarrow 1$ to $k-1$ $\quad k+1 \leq j \leq n+1$

$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

For $i \leftarrow 1$ to k

$a_{k+1} \leftarrow a_{k+1j} - a_{k+1i} * a_{ij}$

The constraints are the same as for Jordan by columns and the tasks graph is the following:

$$\begin{array}{ccccc} P_{12} & P_{13} & P_{14} & \dots & P_{1,n+1} \\ & P_{23} & P_{24} & \dots & P_{2,n+1} \\ & & & \dots & \\ & & & & P_{n,n+1} \end{array}$$

In this case the execution time of P_{kj} is $(4k-1)$ and the constraints are:

(A) $P_{k,k+1}$ must be completed before the beginning of $P_{k+1,j} (j \geq k+2)$

(B) for fixed $j > k, P_{k,j}$ must be completed before the beginning of $P_{k+1,j}$

Once again the tasks graph satisfies the assumptions of Section 3.

PROPOSITION 6

i) The greedy algorithm is asymptotically optimal for the columns—columns parallel Huard method.

ii) For $p = \alpha n$,

$$HCC_p = n^2(2 + 6\alpha^2 - 2\alpha^3)/3\alpha + O(n/\alpha)$$

$$EHCC_p = 1/(1 + 3\alpha^2 - \alpha^3)$$

Proof Given k the execution time is the same for all the P_{kj}: $4k-1$. The formula given in Section 2 is transformed in:

and

$$T_2 = \sum_{k=2}^{n} [(k-1)/p]2(n+1-k) = 2\sum_{k=1}^{n-1} [k/p](n-k)$$

This formula is also difficult to evaluate. Therefore we proceed as in the preceding case:

$$T_2 = 2\sum_{k=1}^{p} (n-k) + 2\sum_{k=p+1}^{n-1} [k/p](n-k)$$

Hence we obtain the following lower bound:

$$T_2 \geqq 2\sum_{k=1}^{p} (n-k) + 2\sum_{k=p+1}^{n-1} k(n-k)/p$$

$$T_2 \geqq n^3/3p + p(n-p/3) + O(n^2/p)$$

Gathering the two results, we obtain:

$$HCR_p \geqq (n^3 + (n-p)^2(n+2p))/3p + p(3n-4p/3) + O(n^2/p)$$

Let

$$p = \alpha n,\ HCR_p \geqq n^2(2+6\alpha^2-2\alpha^3)/(3\alpha) + O(n/\alpha)$$

$$EHCR_p \leqq 1/(1+3\alpha^2-\alpha^3) \quad \square$$

4.2. Columns—columns parallel Huard

(*descent columns—climb columns*)
For $k \leftarrow 1$ **to** n
  For $j \leftarrow k+1$ **to** $n+1$
    $a_{kj} \leftarrow a_{kj}/a_{kk}$

    For $i \leftarrow 1$ **to** $k-1$
      $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

For $i \leftarrow 1$ to k

$a_{k+1,j} \leftarrow a_{k+1,j} - a_{k+1,i} * a_{ij}$

We consider the following tasks P_{kj}.

$P_{kj}: a_{kj} \leftarrow a_{kj}/a_{kk}$ $\quad 1 \leq k \leq n-1$

For $i \leftarrow 1$ to $k-1$ $\quad k+1 \leq j \leq n+1$

$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

For $i \leftarrow 1$ to k

$a_{k+1} \leftarrow a_{k+1j} - a_{k+1i} * a_{ij}$

The constraints are the same as for Jordan by columns and the tasks graph is the following:

$$\begin{matrix} P_{12} & P_{13} & P_{14} & \cdots & P_{1,n+1} \\ & P_{23} & P_{24} & \cdots & P_{2,n+1} \\ & & \cdot & \cdots & \\ & & & & P_{n,n+1} \end{matrix}$$

In this case the execution time of P_{kj} is $(4k-1)$ and the constraints are:

(A) $P_{k,k+1}$ must be completed before the beginning of $P_{k+1,j}(j \geq k+2)$

(B) for fixed $j > k, P_{k,j}$ must be completed before the beginning of $P_{k+1,j}$

Once again the tasks graph satisfies the assumptions of Section 3.

PROPOSITION 6

i) The greedy algorithm is asymptotically optimal for the columns—columns parallel Huard method.

ii) For $p = \alpha n$,

$$HCC_p = n^2(2 + 6\alpha^2 - 2\alpha^3)/3\alpha + O(n/\alpha)$$

$$EHCC_p = 1/(1 + 3\alpha^2 - \alpha^3)$$

Proof Given k the execution time is the same for all the P_{kj}: $4k-1$. The formula given in Section 2 is transformed in:

$$HCC_p=(1/p)\sum_{i=1}^{n+1-p}(n+1-i)\cdot(4i-1)+\sum_{i=n+1-p+1}^{n}4i-1$$

which leads to the result:

$$HCC_p=((n+1-p)/p)[(4n+5)(n+2-p)/2-2(n+2-p)$$
$$\times(2n+3-2p)/3-(n+1)]$$
$$+p(4n-3-2p)$$

Let $p=\alpha\cdot n$, we obtain

$$HCC_p=((2+6\alpha^2-2\alpha^3)/3\alpha)n^2+O(n/\alpha)\quad\square$$

5. PARALLEL JORDAN *VERSUS* PARALLEL HUARD

For $p=\alpha n$ processors $(0<\alpha\leqq 1)$, the formulas of the following array give the execution times of the algorithms divided by n^2.

	$\alpha\leqq 1/2$	$\alpha\geqq 1/2$
JR_p	$1/\alpha$	$\leqq(2\alpha+1)/2\alpha$
JC_p	$(\alpha^2+1)/\alpha$	
HCC_p	$(-2\alpha^3+6\alpha^2+2)/3$	
HCR_p	$\geqq(-2\alpha^3+6\alpha^2+2)/3\alpha$	

PROPOSITION 7

i) $JR_p\leqq JC_p$, *for all* α.
ii) $HCC_p\leqq HCR_p$, *for all* α.
iii) *For* $\alpha\leqq 0.44$, $HCC_p<JR_p$
For $\alpha>0.45$, $JR_p<HCC_p$

Proof (i) and (ii) are obvious. (iii) We compare Jordan by rows and Huard columns—columns.

For $\alpha \leqq 1/2$, we have to find the roots of the equation:

$$-2\alpha^3 + 6\alpha^2 - 1 = 0$$

There exists a single root between 0 and $1: \alpha_0 = 0.442125301668$.

JR_p is faster than HCC_p if $\alpha \leqq \alpha_0$ slower otherwise.

For $\alpha \geqq 1/2$, the equation is $-2\alpha^3 + 6\alpha^2 - 6\alpha + 1 = 0$ which has a unique real root: 2.4.2 Hence $JR_p < HCC_p$ in this case. □

The best parallel version of Jordan method is by rows whereas the best parallel version of Huard method is by columns. The Jordan method can be well parallelized. Hence for a small number of processors the parallel Huard method is faster than the parallel Jordan method and slower otherwise. The separation is obtained for $\alpha = \alpha_0$ (α_0 close to 0.44).

References

[1] H. M. Ahmed, J. M. Delosme and M. Morf, Highly concurrent computing structures for matrix arithmetic and signal processing, *Computer Magazine* (1982), 65–82.

[2] M. Cosnard and Y. Robert, Complexité de la décomposition QR en parallèle, *C.R. Acad. Sc. Paris* 297(A) (1983), 549–552.

[3] M. Cosnard and Y. Robert, Complexity of parallel QR factorization, to appear in Journal of ACM.

[4] M. Cosnard, J. M. Muller and Y. Robert, Parallel QR decomposition of a rectangular matrix, *Numerische Math.* 48(2) (1986), 239.

[5] M. Feilmeier, *Parallel Computers-Parallel Mathematics*, IMACS North-Holland, 1977.

[6] M. J. Flynn, Very high-speed computing systems, *Proc. IEEE* 54 (1966), 1901–1909.

[7] D. D. Gajski and J. K. Peir, *Parallel Processing: Problems and Solutions*, Preprint Univ. Illinois, 1984.

[8] G. H. Golub and C. F. Van Loan, *Matrix Computation*, The Johns Hopkins Univ. Press, 1983.

[9] D. Heller, A survey of parallel algorithms in numerical linear algebra, *SIAM Review* 20 (1978), 740–777.

[10] P. Huard, La méthode Simplexe sans inverse explicite, *bulletin EDF* (1979), 79–98.

[11] K. Hwang and F. Briggs, *Parallel Processing and Computer Architecture*, McGraw Hill, 1984.

[12] D. J. Kuck, *The Structure of Computers and Computations*, J. Wiley & Sons, New York, 1978.
[13] H. T. Kung, Why systolic architectures?, *IEEE Computer* 15 (1982), 37–46.
[14] P. Laurent-Gengoux and D. Trystram, *On the Equivalence of Two Algorithms for Solving Linear Systems*, Prep. Ecole Centrale Paris, 1986.
[15] R. E. Lord, J. S. Kowalik and S. P. K.:mar, Solving linear algebraic equations on an MIMD computer, *J. ACM* 30(1) (1983), 103–117.
[16] J. J. Modi and M. R. B. Clarke, An alternative Givens ordering, *Numerische Math.* 43 (1984), 83–90.
[17] Y. Robert, Block LU decomposition of a band matrix on a systolic array, *Int. J. Computer Math.*, 17(3) (1985), 295–316.
[18] A. Sameh and D. J. Kuck, On stable parallel linear system solvers, *J. ACM* 25(1) (1978), 81–91.
[19] A. Sameh, An overview of parallel algorithms, *Bulletin EDF* (1983), 129–134.
[20] U. Schendel, *Introduction to Numerical Methods for Parallel Computers Ellis Horwood Series*, J. Wiley & Sons, New York, 1984.
[21] M. Srinivas, Optimal parallel scheduling of Gaussian elimination DAG's, *IEEE T.C.* 32 (1983) 1109–1117.
[22] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, Data-driven and demand-driven computers, *ACM Computing Surveys*, 14 (1982), 93–143.

Parallel Computing 6 (1988) 275–296
North-Holland

Parallel Gaussian elimination on an MIMD computer *

M. COSNARD, M. MARRAKCHI and Y. ROBERT
C.N.R.S. – Laboratoire TIM3-INPG, F-38031 Grenoble Cedex, France

D. TRYSTRAM
Ecole Centrale Paris, F-92295 Chatenay Malabry Cedex, France

Received February 1987

Abstract. This paper introduces a graph-theoretic approach to analyse the performances of several parallel Gaussian-like triangularization algorithms on an MIMD computer. We show that the SAXPY, GAXPY and DOT algorithms of Dongarra, Gustavson and Karp, as well as parallel versions of the LDM^t, LDL^t, Doolittle and Cholesky algorithms, can be classified into four task graph models. We derive new complexity results and compare the asymptotic performances of these parallel versions.

Keywords. Parallel algorithms, task graphs, linear algebra, Gaussian elimination, computational complexity.

1. Introduction

The well-known Gaussian elimination method is the most commonly used algorithm to solve linear systems of equations on sequential computers. Six different versions for a vector pipeline machine have been considered in [3]. Three implementations where data are accessed column-wise have been discussed in detail, each of them corresponding to a given permutation of the loop indices i, j, k of the sequential algorithm: namely the SAXPY version (form kji), the GAXPY version (form jki) and the DOT version (form ijk). In this paper, we deal with the design of MIMD versions of these algorithms (see [5,6,9,10] for a classification of parallel computers).

A parallel MIMD version of the Gaussian elimination algorithm with partial pivoting has been discussed in [11,13], and an implementation of the LDL^t decomposition algorithm in [12]. The performance analysis is based on the task graph model represented in [11]. Informally, algorithms are splitted into elementary tasks, whose execution ordering is directed by precedence constraints. The task graph model, which can be constructed directly from these precedence constraints, is the basic tool of our theoretical analysis.

Together with MIMD versions of the [3] algorithms, we analyse several Gaussian-like elimination algorithms: parallel versions of the LDM^t, LDL^t, Doolittle and Cholesky algorithms.

In Section 3, we present sequential versions of the algorithms, define the tasks together with their precedence constraints, and construct the associated task graph or precedence graph. We remark that these graphs can be classified into four categories which we name greedy, 2-steps, double greedy and double 2-steps graphs.

* This work has been supported by the Centre National de la Recherche Scientifique through the GRECO C3.

0167-8191/88/$3.50 © 1988, Elsevier Science Publishers B.V. (North-Holland)

Section 4 is devoted to the theoretical analysis of these graphs. Assuming that the number of processors is proportional to the problem size, we design parallel algorithms, assigning tasks to processors according to the precedence constraints. We then establish complexity results.

In Section 5, we compute the execution times of the parallel algorithms corresponding to the ten sequential algorithms, and we compare their performances. A modified version of the KJI-SAXPY of [3] appears to be the most suitable to parallelization.

2. Restriction to the model

Similarly to [12] we state that

(i) Precise details of the underlying architecture are unimportant, but we assume a system which is capable of supporting multiple instruction streams executing independently and in parallel on multiple data streams [6,9,10,17]. We assume that there are means to synchronize the solution process, i.e. enforce temporal precedence constraints which are imposed by the nature of implemented algorithms.

(ii) Throughout the paper, p denotes the number of processors, and E_p is the efficiency of the parallel algorithm considered [4,8,16,17].

(iii) The cost analysis of the derived algorithms is based on the following assumptions:

- each processor can perform any of the four arithmetic operations in an unit of time,
- there are no memory conflicts or data communication delays.

To be more realistic, we add the following hypothesis when triangularizing a dense $n \times n$ matrix:

(iv) elementary tasks should be of length $O(n)$, so that synchronization and data communication do not predomine over arithmetic. Hence we do not allow tasks of constant length (independent of n), contrarily to [12].

(v) The number of processors is limited to $O(n)$. It is shown in [15] that processor communication costs overcome arithmetic when $O(n^2)$ processors are used. More precisely, we set $p = \alpha n$, with $0 < \alpha < 1$.

Most often, pivoting of rows or columns (or both) is used for stability reasons. However, we shall not consider here the overhead due to pivoting. Rather, we concentrate on evaluating the amount of arithmetic operations which are required by the different algorithms, and we neglect the time needed for data accessing, retrieving and exchanging.

Moreover, we shall assume in the following that we can access the elements of the matrix A by columns to be more realistic and close to a FORTRAN programming style environment. Then the two allowed transfer operations will be loading and storing a column. The duplication of a column will have a zero cost, i.e. we assume that it is possible to transfer simultaneously the same data to various processors. In this case, no processor can modify this data. Conversely, a processor can modify a data if it is the only one to possess it. Remark that the preceding constraints and the associated mechanisms lead to an unambiguous execution of the algorithms.

Once the constraints are defined, the first step in the parallelization of a method is the definition of the elementary tasks and their precedence graph. This graph shows the temporal dependency of the operations of the algorithm. This warrants the absence of conflict, the temporal integrity of the data and a perfect correspondence between the method and its parallel version. The tasks are then assigned to the available processors according to the precedence graph.

Throughout the paper, the relation $T \ll T'$ denotes the precedence constraint and means that task T is to be completed before task T' can start its execution [11,13]. All other notations of graph theory will be consistent with [13].

3. The algorithms

We consider the following algorithms:

(A) {Generic Gaussian elimination algorithm}
{Form KJI-SAXPY of [3]}
For $k = 1$ **to** $n - 1$
  execute T_{kk}: ⟨**For** $i = k + 1$ **to** n
      do $a_{ik} := a_{ik}/a_{kk}$⟩
  For $j = k + 1$ **to** n
    execute T_{kj}: ⟨**For** $i = k + 1$ **to** n
      do $a_{ij} := a_{ij} - a_{ik} * a_{kj}$⟩

T_{kk}: $n - k$ arithmetic operations, $1 \leqslant k \leqslant n - 1$,
T_{kj}: $2(n - k)$ arithmetic operations, $k + 1 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n - 1$.
Total number of arithmetic operations: $2n^3/3 + O(n^2)$.
Precedence constraints:
$T_{kk} \ll T_{kj}$, $k + 1 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n - 1$,
$T_{kj} \ll T_{k+1,j}$ $k + 1 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n - 1$.

(See Fig. 1.)

(B) {Generic Gaussian elimination algorithm}
{Form JKI-GAXPY of [3]
For $j = 1$ **to** n
  For $k = 1$ **to** $j - 1$
    execute T_{kj}: ⟨**For** $i = k + 1$ **to** n
      do $a_{ij} := a_{ij} - a_{ik} * a_{kj}$⟩
  execute T_{jj}: ⟨**For** $i = j + 1$ **to** n
      do $a_{ij} := a_{ij}/a_{jj}$⟩

T_{kj}: $2(n - k)$ arithmetic operations, $1 \leqslant k \leqslant j - 1$, $1 \leqslant j \leqslant n$,
T_{jj} : $n - j$ arithmetic operations, $1 \leqslant j \leqslant n$.
Total number of arithmetic operations: $2n^3/3 + O(n^2)$.
Precedence constraints:
$T_{kj} \ll T_{k+1,j}$, $1 \leqslant k \leqslant j - 1$, $1 \leqslant j \leqslant n$,
$T_{jj} \ll T_{jk}$, $j + 1 \leqslant k \leqslant n$, $1 \leqslant j \leqslant n$.

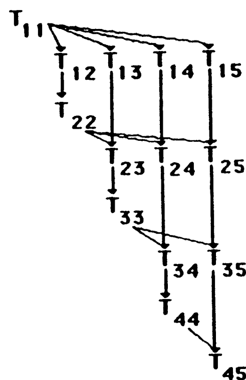


Fig. 1. Precedence graph of Gaussian elimination (form KJI) for a matrix of size 5.

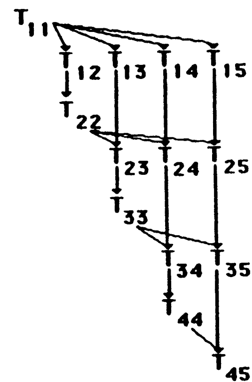


Fig. 2. Precedence graph of Gaussian elimination (form JKI) for a matrix of size 5.

We remark that the precedence graph (see Fig. 2) is the same as that of the form KJI-SAXPY of Generic Gaussian elimination algorithm (A). This is due to the fact that for a given j, the tasks T_{kj}, $1 \leqslant k \leqslant j$, are to be processed sequentially. In the contrary, for a given k, the tasks T_{kj}, $k+1 \leqslant j \leqslant n$, can be processed in parallel. Hence this shows that two different sequential versions of an algorithm can lead to the same parallel implementation.

(C) {Generic Gaussian elimination algorithm}
{Form IJK-DOT of [3]}
For $i = 2$ **to** n
  For $j = 2$ **to** i
    execute T_{ij}: $\langle a_{i,j-1} := a_{i,j-1}/a_{j-1,j-1}$
      For $k = 1$ **to** $j-1$
        do $a_{ij} := a_{ij} - a_{ik} * a_{kj}\rangle$
  For $j = i+1$ **to** n
  execute U_{ij}: $\langle$**For** $k = 1$ **to** $i-1$
    do $a_{ij} := a_{ij} - a_{ik} * a_{kj}\rangle$

T_{ij}: $2j-1$ arithmetic operations, $2 \leqslant j \leqslant i$, $3 \leqslant i \leqslant n$,.
U_{ij}: $2(i-1)$ arithmetic operations, $i+1 \leqslant j \leqslant n$, $2 \leqslant i \leqslant n$.

Total number of arithmetic operations: $2n^3/3 + O(n^2)$.
Precedence constraints:
$T_{ij} \ll T_{i,j+1}$, $2 \leqslant j \leqslant i-1$, $3 \leqslant i \leqslant n$,
$T_{ii} \ll U_{ij}$, $i+1 \leqslant j \leqslant n$, $2 \leqslant i \leqslant n-1$,
$U_{ij} \ll U_{i+1,j}$, $i+2 \leqslant j \leqslant n$, $2 \leqslant i \leqslant n-2$,
$U_{j,j+1} \ll T_{i,j+1}$, $i \leqslant j \leqslant n-1$, $2 \leqslant i \leqslant n$.
(See Fig. 3.)

(D) {Generic Gaussian elimination algorithm}
{Form KJI-SAXPY modified}
For $k = 1$ **to** $n-1$
  For $j = k+1$ **to** n
    execute T_{kj}: $\langle a_{kj} := a_{kj}/a_{kk}$
      For $i = k+1$ **to** n
        do $a_{ij} := a_{ij} - a_{ik} * a_{kj}\rangle$

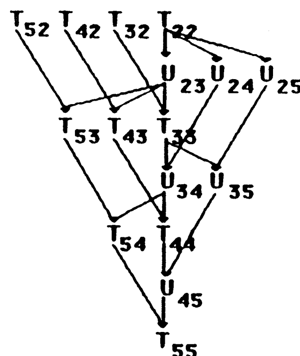


Fig. 3. Precedence graph of Gaussian elimination (form IJK) for a matrix of size 5.

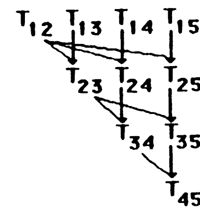


Fig. 4. Precedence graph of Gaussian elimination (form KJI modified) for a matrix of size 5.

T_{kj}: $2(n-k)+1$ arithmetic operations, $k+1 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n-1$.
Total number of arithmetic operations: $2n^3/3 + O(n^2)$.
Precedence constraints:

$T_{k,k+1} \ll T_{k+1,j}$, $k+2 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n-2$,
$T_{kj} \ll T_{k+1,j}$, $k+2 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n-2$.

(See Fig. 4.)

(E) {Doolittle reduction [7]}

For $k=1$ **to** n
  For $j=k$ **to** n
    execute T_{kj}: ⟨**For** $p=1$ **to** $k-1$
      do $a_{kj} := a_{kj} - a_{kp} * a_{pj}$⟩
  For $i=k+1$ **to** n
    execute U_{ki}: ⟨**For** $p=1$ **to** $k-1$
      do $a_{ik} := a_{ik} - a_{ip} * a_{pk}$
      $a_{ik} := a_{ik}/a_{kk}$⟩

T_{kj} : $2(k-1)$ arithmetic operations, $k \leqslant j \leqslant n$, $1 \leqslant k \leqslant n$,
U_{ki} : $2k-1$ arithmetic operations, $k+1 \leqslant i \leqslant n$, $1 \leqslant k \leqslant n-1$.

Total number of arithmetic operations: $2n^3/3 + O(n^2)$.
Precedence constraints:

$T_{kj} \ll T_{k+1,j}$, $k+1 \leqslant j \leqslant n$, $2 \leqslant k \leqslant n-1$,
$T_{kk} \ll U_{kj}$, $k+1 \leqslant j \leqslant n$, $2 \leqslant k \leqslant n-1$,
$U_{ki} \ll U_{k+1,i}$, $k+2 \leqslant i \leqslant n$, $2 \leqslant k \leqslant n-2$,
$U_{k,k+1} \ll T_{k+1,j}$ $k+1 \leqslant j \leqslant n$, $2 \leqslant k \leqslant n-1$.

(See Fig. 5.)

(F) {LDM^t decomposition [7]}
{Non-optimized code}

For $k=1$ **to** n
  execute T_{kk}: ⟨**For** $p=1$ **to** $k-1$
    do $a_{kk} := a_{kk} - a_{kp} * a_{pp} * a_{pk}$⟩
  For $i=k+1$ **to** n
    execute T_{ik}: ⟨**For** $p=1$ **to** $k-1$
      do $a_{ik} := a_{ik} - a_{ik} - a_{ip} * a_{pp} * a_{pk}$;
      $a_{ik} := a_{ik}/a_{kk}$⟩

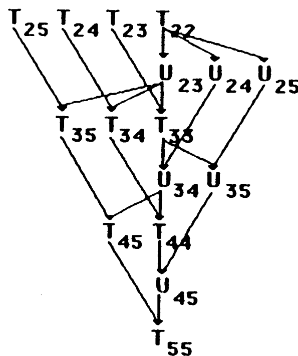


Fig. 5. Precedence graph of Doolittle reduction for a matrix of size 5.

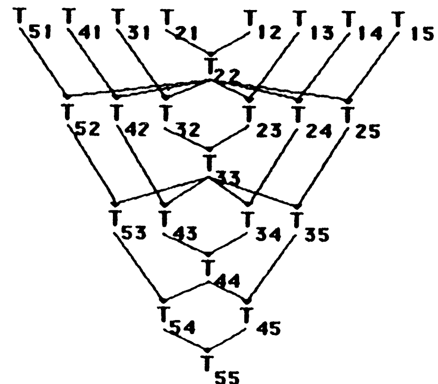


Fig. 6. Precedence graph of LDM^t decomposition for a matrix of size 5.

For $j = k+1$ **to** n
  execute T_{kj}: ⟨**For** $p = 1$ **to** $k-1$
    do $a_{kj} := a_{kj} - a_{kp} * a_{pp} * a_{pj}$
    $a_{kj} := a_{kj}/a_{kk}$⟩

T_{kk}: $3(k-1)$ arithmetic operations, $1 \leqslant k \leqslant n$,
T_{ik} : $3k-2$ arithmetic operations, $k+1 \leqslant i \leqslant n$, $1 \leqslant k \leqslant n-1$,
T_{kj}: $3k-2$ arithmetic operations, $k+1 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n-1$.

Total number of arithmetic operations: $n^3 + O(n^2)$.
Precedence constraints:
$T_{kk} \ll T_{kj}$, $k+1 \leqslant j \leqslant n$, $2 \leqslant k \leqslant n-1$,
$T_{kk} \ll T_{ik}$, $k+1 \leqslant i \leqslant n$, $2 \leqslant k \leqslant n-1$,
$T_{ik} \ll T_{i,k+1}$, $k+1 \leqslant i \leqslant n$, $1 \leqslant k \leqslant n-1$,
$T_{kj} \ll T_{k+1,j}$, $k+1 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n-1$.
(See Fig. 6.)

(G) {LDM^t decomposition [7]}
{Optimized code}
For $k = 1$ **to** n
  execute T_{kk}: ⟨**For** $p = 1$ **to** $k-1$
    do begin $r_p := d_p * a_{pk}$; $w_p := a_{kp} * d_p$;
    $a_{kk} := a_{kk} - a_{kp} * r_p$; **end**⟩
  For $i = k+1$ **to** n
    execute T_{ik}: ⟨**For** $p = 1$ **to** $k-1$
      do $a_{ik} := a_{ik} - a_{ip} * r_p$
      $a_{ik} := a_{ik}/a_{kk}$⟩
  For $j = k+1$ **to** n
    execute T_{kj}: ⟨**For** $p = 1$ **to** $k-1$
      do $a_{kj} := a_{kj} - w_p * a_{pj}$
      $a_{kj} := a_{kj}/a_{kk}$⟩

T_{kk}: $4(k-1)$ arithmetic operations, $1 \leqslant k \leqslant n$,
T_{ik} : $2k-1$ arithmetic operations, $k+1 \leqslant i \leqslant n$, $1 \leqslant k \leqslant n-1$,
T_{kj}: $2k-1$ arithmetic operations, $k+1 \leqslant j \leqslant n$, $1 \leqslant k \leqslant n-1$.

Total number of arithmetic operations: $2n^3/3 + O(n^2)$.

The precedence constraints and the precedence graph are the same as for the non-optimize LDM^t decomposition (See Fig. 6). The execution times of the tasks are different.

(H) {LDL^t decomposition [7]}
{Non-optimized code—see [12]}
For $k = 1$ **to** n
  execute T_{kk}: ⟨**For** $p = 1$ **to** $k-1$
    do $a_{kk} := a_{kk} - a_{kp} * a_{pp} * a_{kp}$⟩
  For $i = k+1$ **to** n
    execute T_{ik}: ⟨**For** $p = 1$ **to** $k-1$
      do $a_{ik} := a_{ik} - a_{ip} * a_{pp} * a_{kp}$;
      $a_{kk} := a_{ik}/a_{kk}$⟩

T_{kk}: $3(k-1)$ arithmetic operations, $1 \leqslant k \leqslant n$,

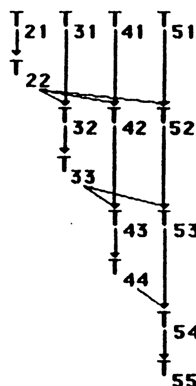


Fig. 7. Precedence graph of LDL^t decomposition for a matrix of size 5.

T_{ik} : $3k-2$ arithmetic operations, $\quad k+1 \leqslant i \leqslant n,\ 1 \leqslant k \leqslant n-1.$

Total number of arithmetic operations: $n^3/2 + O(n^2)$.
Precedence constraints:

$T_{kk} \ll T_{ik}, \qquad k+1 \leqslant i \leqslant n,\ 2 \leqslant k \leqslant n-1,$
$T_{ik} \ll T_{i,k+1}, \qquad k+1 \leqslant i \leqslant n,\ 2 \leqslant k \leqslant n-1.$

(See Fig. 7.)

(I)
```
{LDLᵗ decomposition [7]}
{Optimized code}
For k = 1 to n
  execute T_kk: ⟨For p = 1 to k − 1
                   do begin r_p := d_p * a_pk:
                     a_kk := a_kk − a_kp * r_p}; end⟩
  For i = k + 1 to n
    execute T_ik: ⟨For p = 1 to k − 1
                     do a_ik := a_ik − a_ip * r_p
                   a_ik := a_ik/a_kk⟩
```

T_{kk}: $3(k-1)$ arithmetic operations, $\quad 1 \leqslant k \leqslant n,$
T_{ik} : $2k-1$ arithmetic operations, $\quad k+1 \leqslant i \leqslant n,\ 1 \leqslant k \leqslant n-1.$

Total number of arithmetic operations: $n^3/3 + O(n^2)$.

The precedence constraints and the precedence graph are the sam as for the non-optimized LDL^t decomposition (see Fig. 7). The execution times of the tasks are different.

(J)
```
{Cholesky decomposition A = GGᵀ [7]}
For k = 1 to n
  execute T_kk: ⟨For p = 1 to k − 1
                   do a_kk := a_kk − a_kp * a_kp
                 a_kk := sqrt(a_kk)⟩
  For i = k + 1 to n
    execute T_ik: ⟨For p = 1 to k − 1
                     do a_ik := a_ik − a_ip * a_kp
                   a_ik := a_ik/a_kk⟩
```

T_{kk}: $2k-1$ arithmetic operations, $1 \leqslant k \leqslant n$,
T_{ik} : $2k-1$ arithmetic operations, $k+1 \leqslant i \leqslant n$, $1 \leqslant k \leqslant n-1$.

Total number of arithmetic operations: $n^3/3 + O(n^2)$
(we assume that a square root takes a unit of time too).

The precedence constraints and the precedence graph are the same as for the non-optimized LDL^t decomposition (see Fig. 7). The execution times of the tasks are different.

4. Graph-theoretical analysis

We have listed ten algorithms in the previous section, which have led to seven task graphs (Figs. 1–7). In fact, a closer analysis shows that there are four kinds of graphs. We name the first two as follows:
- 'greedy' graph: the task graph of algorithm (D),
- '2-steps' graph: the task graphs of algorithms (A), (B), (H), (I), (J).

The two others can be viewed as the result of some interleaving of two greedy graphs, or two 2-steps graphs:
- 'double greedy' graph: the task graphs of algorithms (C) and (E),
- 'double 2-steps' graph: the task graphs of algorithm (F).

We first concentrate in this section on the analysis of the greedy and 2-steps graphs. We design parallel algorithms and establish complexity results for these two kinds of graphs. We will make use of this analysis to derive the analysis of the double greedy and double 2-steps graphs.

As we said in Section 2, we limit the number of processors to be $p = O(n)$ when triangularizing a matrix of order n. More precisely, we let $p = \alpha n$, and derive asymptotic results when n goes to infinity.

4.1. Greedy graphs

{General form of the greedy algorithm}
For $k = 1$ **to** $n-1$
  For $i = k+1$ **to** n
    Execute T_{ki}

Such a graph is also encountered when solving linear systems by diagonalization methods [1,2]. We assume that the elementary task T_{ki} (for $i > k$) can be processed in bt_k units where b is an integer and $t_k = k$ for all k or $t_k = n-k$ for all k. The precedence constraints are

(A) $\quad T_{k,k+1} \ll T_{k+1,j}, \quad 1 \leqslant k \leqslant n-1,\ k+2 \leqslant j \leqslant n,$
(B) $\quad T_{k,j} \ll T_{k+1,j}, \quad 1 \leqslant k \leqslant n-1,\ k+2 \leqslant j \leqslant n.$

For a given value of k, there are $n-k$ tasks to be executed. The precedence graph is given in Fig. 8 (see also Fig. 4).

Let $T \leqslant\leqslant T'$ denote the following precedence constraint: the execution of T' cannot begin before that of T (but a simultaneous execution is allowed). We have

Proposition 4.1. *There exist optimal parallel algorithms which satisfy the additional constraint* (C)

(C) $\quad T_{k,j} \leqslant\leqslant T_{k,j+1}, \quad j \geqslant k+1.$

Proof. Let M be any parallel algorithm. We construct a parallel algorithm M^* which has the same execution time as M and satisfies the constraint (C):

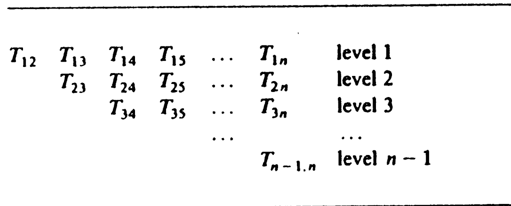


Fig. 8. Greedy graph.

Greedy algorithm

A better algorithm

Fig. 9. The greedy algorithm is not optimal.

– if at time t, the algorithm M starts the execution of q tasks of level k, M^* starts the execution of q tasks of level k too, but it starts the first q not yet executed tasks of level k (according to the ordering imposed by the constraint (C)),
– if at time t, the algorithm M does not start the execution of any task level k, the same holds for M^*.

Hence at any time during the execution, M^* has executed the same number of tasks of each level as M. We still have to prove that the precedence constraints (A) and (B) are met by M^*. For (A) this is evident since $T_{k,k-1}$ is the first task of level k executed by M^*.

For the constraint (B), let us assume that M starts the execution of q tasks of level $k+1$ at time t. Let s_k and s_{k+1} be respectively the number of tasks of level k and $k+1$ having been processed by M (and M^*) at time t. Since M satisfies the constraints (A) and (B), $s_k \geqslant s_{k+1} + q + 1$. At time, t, M^* starts the execution of the tasks $T_{k+1,k+j}$, where $s_{k+1} + 1 \leqslant j \leqslant s_{k+1} + q$. Since M^* has already completed the tasks $T_{k,k+j}$, where $1 \leqslant j \leqslant s_k$, it does satisfy the constraint (B). □

Now we introduce the greedy algorithm G, that we can informally define as follows: the algorithm G executes the tasks from one level of the precedence graph to another, from left to right in each level, and it starts the maximum number of tasks at each time (hence the name of greedy). More precisely, the algorithm G executes the tasks in the following order:

$$T_{12} \leqslant\leqslant T_{13} \leqslant\leqslant \cdots \leqslant\leqslant T_{1n} \leqslant\leqslant T_{23} \leqslant\leqslant T_{24} \leqslant \cdots \leqslant T_{2n} \leqslant\leqslant T_{34} \leqslant\leqslant \cdots \leqslant\leqslant T_{n-1,n}.$$

At time $t = 1$, the algorithm G starts the execution of the tasks $T_{12}, T_{13}, \ldots, T_{1p}$. For any $t \geqslant 1$, if $q \leqslant p$ processors are idle, the algorithm G assigns these q processors to the execution of the following q tasks, provided that constraint (B) makes it possible (otherwise, the algorithm executes the maximum number of tasks).

Proposition 4.2. *The greedy algorithm G is asymptotically optimal. Its execution time is equal to*

$$T_G = \frac{b}{p} \sum_{i=1}^{n-p} (n-i)t_i + b \sum_{i=n-p+1}^{n-1} t_i + O(n).$$

Proof. Using Proposition 4.1, we show that the greedy algorithm is asymptotically optimal i the class of the algorithms which satisfy to the precedence constraints (A), (B) and (C). W know that a task T_{kj} is to be executed before a task $T_{k,j+1}$ in this particular class c algorithms. Let t_0 be the minimum time at which $T_{n-p,n}$ is completed by any algorithm of thi class. We have

$$t_0 \geqslant \frac{b}{p} \sum_{i=1}^{n-p} (n-i)t_i + O(n),$$

since all the tasks of level 1 to $n-p$ must have been executed before $T_{n-p,p}$.

The task $\{T_{n-p+1,n}, \dots, T_{n-1,n}\}$ must be processed sequentially. Hence the execution tim T_{opt} of an optimal algorithm is bounded below by

$$T_{\text{opt}} \geqslant \frac{b}{p} \sum_{i=1}^{n-p} (n-i)t_i + b \sum_{i=n-p+1}^{n-1} t_i.$$

The greedy algorithm is of asymptotic efficiency 1 up to level $n-p$, since there are mo than p independent tasks to be processed up to this level. Thus the time t_1 at which the greed algorithm completes the execution of $T_{n-p,p}$ only differs from t_0 by a linear factor $O(n)$. Th shows that the greedy algorithm meets the preceding bound up to a linear factor, theret establishing the proposition. □

With $p = \alpha n$ processors, this leads to the following execution times:

(i) $t_k = k$, then $T_G = bn^2[(1-\alpha)/(6\alpha) + \frac{1}{2}] + O(n)$,

(ii) $t_k = n-k$, then $T_G = bn^2[(2+\alpha^3)/(6\alpha)] + O(n)$.

Contrarily to the intuitive feeling, the greedy algorithm is not optimal, as can be shown t the following example communicated by G. Rote [14]: let $n = 8$, $p = 3$, $b = 1$ and $t_k = n -$ the execution time of the greedy is 50 but there exists an algorithm of time 49 (see Fig. 9).

4.2. 2-steps graphs

```
{General form of the 2-steps algorithm}
For k = 1 to n − 1
  For i = k + 1 to n
    Execute T_ki
  Execute T_k+1,k+1
```

We assume that the elementary tasks T_{ki} (for $i > k$) can be processed in bt_k units a $T_{k+1,k+1}$ in at_k units, where a and b are integers and $t_k = k$ for all k or $t_k = n-k$ for all The precedence constraints are

$$T_{kk} \ll T_{ki}, \qquad 1 \leqslant k \leqslant n-1,\ k+1 \leqslant i \leqslant n,$$
$$T_{ki} \ll T_{k+1,i} \qquad 1 \leqslant k \leqslant n-1,\ k+1 \leqslant i \leqslant n$$

which leads to the precedence graph (see also Figs 1, 2 and 7) of Fig. 10.

We shall now define an algorithm to assign the tasks to the processors according to t precedence constraints. Informally, this algorithm is divided into three different parts. The fi part corresponds to a greedy scheme whose asymptotic efficiency is equal to 1. As soon as reach a level with not enough tasks for all the p processors to be activated simultaneously, t algorithm goes further level by level, in a synchronous fashion. The difference between t second and third phases only lies in the time needed to complete the execution of a given le We set $s = \lfloor a/b \rfloor + 1$ and for an integer q, wet let $q^+ = \max(1, q)$.

In fact, in order to simplify the presentation, we assume the existence of an extra proces denoted P_0. Since the total number of processors is proportional to n, this hypothesis does t modify our asymptotic performance evaluation.

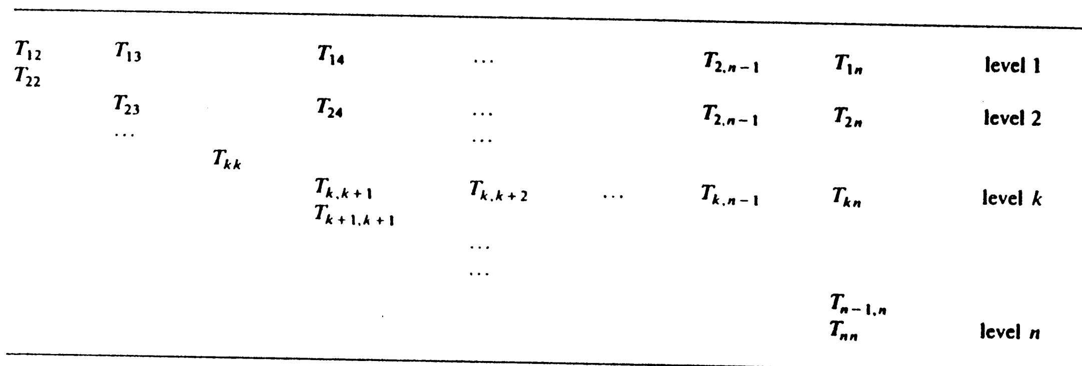


Fig. 10. 2-steps graph.

The first part of the algorithm corresponds to the execution of level 1 up to level $(n-sp)^+$. We assign all the tasks T_{kk}, $2 \leqslant k \leqslant (n-sp)^+$, to the processor P_0. The remaining processors will execute the tasks T_{ki}, $i \geqslant k+1$, in a greedy manner, level by level and in each level from left to right. The only difference with Section 4.1 is due to the presence of the tasks processed by P_0. We let P_0 begin the execution of the task $T_{k+1,k+1}$ as soon as the task $T_{k,k+1}$ is completed. In order to obtain an asymptotic efficiency equal to 1 in this phase, we must ensure that there are enough tasks to be executed at each time-step so that no processor P_i, $i \geqslant 1$, is idle.

Let t be the time at which the execution of $T_{k,k+1}$ starts. P_0 begins the execution of task $T_{k+1,k+1}$ at time $t+bt_k$ and completes at time $t+(a+b)t_k$. Since the tasks in a given level are executed from left to right, the execution of no task of level k has begun at time $t-1$. Hence, no more than $\lceil (a+b)/b \rceil p = sp$ tasks of level k can have been processed at time $t+(a+b)t_k$. We have $k > (n-sp)^+$ by hypothesis, hence there are more than sp tasks in level k. Therefore this ensures that P_0 can complete the execution of $T_{k+1,k+1}$ while the other processors are still processing the tasks of level k. This proves both that no precedence constraint is violated (hence the correctness of the algorithm) and that no processor P_i, $i \geqslant 1$, is idle, thereby establishing our claim.

To evaluate the execution time of phase 1, we use the fact that the asymptotic efficiency of the algorithm is 1, since the time lost by processor P_0 can be neglected ($p = \alpha n$). We can compute the execution time τ_1 of this phase as

$$\tau_1 = \frac{b}{p} \sum_{k=1}^{(n-sp)^+} (n-k)t_k + \mathrm{O}(n).$$

Let us now consider the second phase of the algorithm: from level $(n-sp)^+$ to level $(n-(s-1)p)^+$. In this phase the algorithm has not asymptotic efficiency 1. We choose to synchronize the processors so that they all begin simultaneously the execution of new tasks. The algorithm proceeds level by level, but now the processing of a new level does not begin before the completion of the preceding one. The tasks in a given level are processed from left to right.

Let us detail the execution of level k, which is composed of $n-k$ tasks $T_{k,k+1},\ldots,T_{k,n}$. We let $n-k=(s-1)p+r_k$, $0<r_k<p$. At time t, $P_1,\ldots,P_p$ starts the execution of $T_{k,k+1}, T_{k,k+2},\ldots,T_{k,k+p}$. At time $t+bt_k$, $P_1,\ldots,P_p$ start the execution of the following p tasks if $s>2$ and the execution of the last r_k tasks if $s=2$ (note that $s=\lceil a/b \rceil + 1 \geqslant 2$). More precisely, $P_1,\ldots,P_p$ start the execution of a new block of p tasks every bt_k time-steps, until time $t+(s-1)bt_k$, where they start the execution of the last r_k tasks. So they complete the execution of level k at time $t+sbt_k$.

P_0 starts the execution of $T_{k+1,k+1}$ as soon as possible, that is at time $t + bt_k$, when $T_{k,k+1}$ i completed (and P_0 does not process any non-diagonal task). The execution of $T_{k+1,k+1}$ i completed at time $(t + bt_k) + at_k$. Since $(b + a)t_k \leqslant sbt_k$, the processor P_0 terminates its tas before the end of the level k.

Hence we can compute the execution time τ_2 of this phase as

$$\tau_2 = \sum_{k=(n-sp+1)^+}^{(n-(s-1)p)^+} sbt_k + O(n).$$

In this last phase: from level $(n-(s-1)p)^+$ to level n, the processors operate in a ve similar fashion as in phase 2, the only difference lies in the fact that $P_1, \ldots, P_p$ complete tl execution of a given level k before P_0 completes the execution of $T_{k+1,k+1}$. Hence tl processing of level $k+1$ starts only when P_0 has finished its work.

We can compute the execution time τ_3 of this phase as

$$\tau_3 = \sum_{k=(n-(s-1)p+1)^+}^{n} (a+b)t_k + O(n).$$

The total execution time of the algorithm is: $T_{alg} = \tau_1 + \tau_2 + \tau_3$.

We do not know whether this algorithm is asymptotically optimal. However, we can obtain lower bound for the execution time.

Consider an optimal algorithm. Let t be the time where the last task of level $(n - sp)^+$ completed. Let $T_{(n-sp)^+,j}$ denote this task. The precedence constraints imply that all the tas of the preceding levels have been executed, therefore $t \geqslant \tau_1 - O(n)$, since τ_1 corresponds to : algorithm whose asymptotic efficiency is equal to 1.

The path $\{T_{(n-sp)^++1,j}, T_{(n-sp)^++2,j}, \ldots, T_{jj}, T_{j,j+1}, T_{j+1,j+1}, \ldots, T_{n,n}\}$ is composed of tas that must be processed sequentially. The shortest path is obtained for $j = n$. Its length is

$$\tau_4 = \sum_{k=(n-sp+1)^+}^{n} bt_k.$$

Hence, the optimal execution time T_{2S} must be greater than or equal to the sum of τ_1 and

$$T_{2S} \geqslant T_{inf} = \tau_1 + \tau_4.$$

We point out that if $p \geqslant n/(s-1)$, the first two phases of the algorithm are empty. T precedence constraints imply that our algorithm is optimal in this case, since its time execution corresponds to the longest path of the tasks graph.

There is another case worth pointing out. When a is a multiple of b, $sb = a + b$, and ı second phase of the algorithm is executed in the time of the longest path. This shows that ı algorithm in this case for $p \geqslant n/s$.

We can gather all these results in the following proposition:

Proposition 4.3. *Let* $s = \lceil a/b \rceil + 1$.

(1) *The execution time* T_{alg} *of the previous algorithm is given by*

$$T_{alg} = \tau_1 + \tau_2 + \tau_3.$$

(2) *If* $p \geqslant n/(s-1)$, *the algorithm is asymptotically optimal*:

$$T_{alg} = T_{2S} = \tau_3.$$

(3) *If* $p \geqslant n/s$ *and* a/b *is an integer, the algorithm is asymptotically optimal*:

$$T_{alg} = T_{2S} = \tau_2 + \tau_3.$$

(4) *If* $p < n/(s-1)$, *the optimal execution time* T_{2S} *is bounded by*

$$\tau_1 + \tau_4 = T_{inf} \leqslant T_{2S} \leqslant T_{alg}.$$

P_0	P_1	P_2	P_3	
	12	13	14	Phase 1
	12	13	14	
22	15	16	17	
	15	16	17	
	18	19	23	
	18	19	23	
33	24	25	26	
	24	25	26	
	27	28	29	
	27	28	29	
	34	35	36	Phase 2
	34	35	36	
44	37	38	39	
	37	38	39	
	45	46	47	
	45	46	47	
55	48	49	–	
	48	49	–	
	56	57	58	
	56	57	58	
66	59	–	–	
	59	–	–	
	67	68	69	Phase 3
	67	68	69	
77	–	–	–	
	78	79	–	
	78	79	–	
88	–	–	–	
	89	–	–	
	89	–	–	

Fig. 11. Table of execution.

For a better comprehension, let us consider now an example. We take $a = 1$, $b = 2$, $s = 2$, $p = 3$ and $n = 9$. We recall that, during phase 1, the processors work asynchronously and with full efficiency. In Fig. 11 ij in column k means that task T_{ij} is processed by processor P_k. Since the value of b is twice that of a, the execution time of T_{ij} is twice that of T_{jj}. To ease the comprehension we conceptually split T_{ij} in two. The execution of this example is detailed in the Fig. 11.

With $p = \alpha n$ processors this leads to the following execution times:

(i) $t_k = k$:

$$\text{(a)} \quad \tau_1 = \begin{cases} bn^2\left[(1-\alpha s)^2(1+2\alpha s)/(6\alpha)\right] + \mathrm{O}(n) & \text{for } \alpha < 1/s, \\ 0 & \text{otherwise;} \end{cases} \qquad (\tau_1')$$

(b) $$\tau_2 = \begin{cases} bn^2\left[s\alpha(2+(1-2s)\alpha)/2\right] + O(n) & \text{for } \alpha < 1/2, \qquad (\tau_2' \\ bn^2\left[s(1-(s-1)\alpha)^2/2\right] + O(n) & \text{for } 1/s < \alpha < 1/(s-1), \qquad (\tau_2'' \\ 0 & \text{otherwise;} \end{cases}$$

(c) $$\tau_3 = \begin{cases} (a+b)n^2\left[(s-1)(2-(s-1)\alpha)\alpha/2\right] + O(n) & \text{for } \alpha < 1/(s-1) \\ (a+b)n^2/2 & \text{otherwise;} \end{cases}$$

$(\tau_3'$

$(\tau_3''$

(d) $$\tau_4 = \begin{cases} bn^2\left[s(2-s\alpha)g\alpha/2\right] + O(n) & \text{for } \alpha < 1/s, \qquad (\tau_4' \\ bn^2/2 & \text{for } 1/s < \alpha < 1/(s-1); \qquad (\tau_4'' \end{cases}$$

(ii) $t_k = n - k$;

(a) $$\tau_1 = \begin{cases} bn^2\left[(1-\alpha^3 s^3)/(3\alpha)\right] + O(n) & \text{for } \alpha < 1/s, \\ 0 & \text{otherwise;} \qquad (\tau_1' \end{cases}$$

(b) $$\tau_2 = \begin{cases} bn^2\left[s(2s-1)\alpha^2/2\right] + O(n) & \text{for } \alpha < 1/s, \qquad (\tau_2' \\ bn^2\left[s(1-(s-1)^2\alpha^2)/2\right] + O(n) & \text{for } 1/s < \alpha < 1(s-1), \qquad (\tau_2'' \\ 0 & \text{otherwise;} \end{cases}$$

(c) $$\tau_3 = \begin{cases} (a+b)n^2\left[(s-1)^2\alpha^2/2\right] + O(n) & \text{for } \alpha < 1/(s-1), \qquad (\tau_3' \\ (a+b)n^2/2 & \text{otherwise;} \qquad (\tau_3'' \end{cases}$$

(d) $$\tau_4 = \begin{cases} bn^2\left[s^2\alpha^2/2\right] + O(n) & \text{for } \alpha < 1/s, \qquad (\tau_4' \\ bn^2/2 & \text{for } 1/s < \alpha < 1(s-1). \qquad (\tau_4'' \end{cases}$$

We can sum up our results in Table 1. Note that for $s = 2$, the rightmost part of the tabl vanishes.

4.3. Comparison of optimal algorithms

In this section, we compare the performances of optimal algorithms for greedy graphs an 2-steps graphs. Remark that we know neither how to design an optimal algorithm, nor how t compute the optimal time for these two graphs. However, the greedy algorithm is asymptot cally optimal for the greedy graph.

We consider a sequential problem which can be parallelized in two ways, leading either to greedy graph or to a 2-steps graph. This is the case for the generic Gaussian elimination whic leads to a greedy graph (version (D) from KJI-modified) or to a 2-steps graph (version (A) froı KJI and version (B) form JKI).

Table 1

α	0		$1/s$		$1/(s-1)$		1
T_{alg}		$\tau_1' + \tau_2' + \tau_3'$		$\tau_2'' + \tau_3'$		τ_3''	
T_{inf}		$\tau_1' + \tau_4'$		τ_4''			
T_{2S}		?		?		τ_3''	

Let (P) be a problem which can be parallelized as follows:

{Greedy algorithm}	{2-steps algorithm}
For $k = 1$ **to** $n - 1$	**For** $k = 1$ **to** $n - 1$
For $i = k + 1$ **to** n	**For** $i = k + 1$ **to** n
Execute T_{ki}	**Execute** U_{ki}
	Execute $U_{k+1,k+1}$

We assume that the elementary tasks T_{ki} and U_{ki} (for $i > k$) can be processed in bt_k units, and $U_{k+1,k+1}$ in at_k units, where a and b are integers and $t_k = k$ for all k or $t_k = n - k$ for all k. The precedence constraints are those defined above.

Clearly, the total amount of operations is not the same in the two cases, which may look surprising! However, the difference is proportional to n, and this will not modify our analysis.

Let T_G and T_{2S} denote the execution time of optimal algorithms, as above. The following proposition shows that the greedy graph is more suitable to parallelization than the 2-steps graph:

Proposition 4.4. $T_{2S} \geqslant T_G - O(n)$.

Proof. The proof is similar to the computation of the lower bound τ_4 in the previous section. Let t be the time where the greedy algorithm completes the level $n - p$. When an optimal algorithm for the 2-steps graph completes the level $n - p$ at time t', it must have executed all the tasks of previous levels (due to the precedence constraints). Since the total number of operations from level 1 up to level $n - p$ is the same in the two graphs (up to a linear factor), and since the greedy algorithm works with efficiency 1, we deduce that $t' \geqslant t - O(n)$.

Let $T_{n-p,j}$ be the last executed task level $n - p$ by the 2-steps optimal algorithm. The path $\{T_{(n-p)+1,j}, T_{(n-p)+2,j}, \ldots, T_{jj}, T_{j,j+1}, T_{j+1,j+1}, \ldots, T_{nn}\}$ is composed of task that must be processed sequentially. The shortest path is obtained for $j = n$. Its length is greater than

$$\sum_{k=n-p+1}^{n} bt_k.$$

Together with Proposition 4.2, we deduce that $T_{2S} \geqslant T_G - O(n)$. □

4.4. Double graphs

The design and analysis of parallel algorithms for double greedy and double 2-steps graphs are derived from the previous results in a very straightforward manner. The key-idea is to gather two levels of the two interleaved graphs into a single one, leading to a graph that we have already studied.

4.4.1. Double greedy graphs

{General form of the double greedy algorithm}
For $k = 1$ **to** n
  For $j = k$ **to** n
    Execute T_{kj}
  For $i = k + 1$ **to** n
    Execute U_{ki}

We assume that the elementary tasks T_{kj} (for $j \geqslant k$) can be processed in bt_k unit, that the elementary tasks U_{ki} (for $i > k$) can be processed in ct_k units, where b and c are integers, and $t_k = k$ for all k or $t_k = n - k$ for all k. The precedence constraints are

$$\begin{aligned}
&T_{kj} \ll T_{k+1,j}, && k+1 \leqslant j \leqslant n,\ 2 \leqslant k \leqslant n-1,\\
&T_{kk} \ll U_{k,j}, && k+1 \leqslant j \leqslant n,\ 2 \leqslant k \leqslant n-1,\\
&U_{ki} \ll U_{k+1,i}, && k+2 \leqslant i \leqslant n,\ 2 \leqslant k \leqslant n-2,\\
&U_{k,k+1} \ll T_{k+1,j}, && k+1 \leqslant j \leqslant n,\ 2 \leqslant k \leqslant n-1.
\end{aligned}$$

T_{1n}	...	T_{14}	T_{13}	T_{12}	T_{11}						level 1
	T_{2n}	...	T_{24}	T_{23}	U_{12} T_{22}	U_{13}	U_{14}	U_{15}	...	U_{1n}	level 2
		T_{3n}	...	T_{34}	U_{23} T_{33}	U_{24}	U_{25}	...	U_{2n}		level 3
				...	U_{34}	U_{35}	...	U_{3n}			
							...				
		$T_{n-1,n}$		$T_{n-1,n-1}$							level $n-1$
					$U_{n-1,n}$ T_{nn}						level n

Fig. 12. Double greedy graph.

For a given value of k, there are $2(n-k)+1$ tasks to be executed. The precedence graph i given in Fig. 12 (see Fig. 3).

We construct a double greedy algorithm for this double greedy graph in the same way as w have defined a greedy algorithm for the greedy graph. From level 1 up to level $n-p$, the task are executed in a greedy manner in the following order:

$$T_{kk}, T_{k,k+1}, \ldots, T_{k,n}, U_{k,k+1}, U_{k,k+2}, \ldots, U_{kn}.$$

From level $n-p+1$ to level n, the number of tasks T and U is respectively less than or equ: to p. Hence, the algorithm proceeds level by level, and in each level in two stages, executin first the tasks T and after that the tasks U. Applying the same analysis as for the greed algorithm, it is not difficult to show that the precedence constraints are verified by the doubl greedy algorithm.

However, we point out that we cannot prove here the optimality of this greedy algorithn since there could exist more efficient algorithms which do not gather the tasks. Let T_{alg} deno the execution time of this double greedy algorithm:

(i) if $t_k = k$, then $T_{alg} = (b+c)n^2[(1+3\alpha^2-\alpha^3)/(6\alpha)] + O(n)$,

(ii) if $t_k = n-k$, then $T_{alg} = (b+c)n^2[(2+\alpha^3)/(6\alpha)] + O(n)$.

We do not know whether this algorithm is asymptotically optimal. However we can obtain lower bound for the execution time in a similar way as in Sections 4.2 and 4.3.

Consider an optimal algorithm. Let t be the time where the last task of level $(n-p)$ completed. The tasks of the preceding levels 1 to $n-p$ could have been executed with fu asymptotic efficiency. The execution time of the remaining tasks is a greater than or equal the time of the shortest path, leading to the following formula:

$$T_{DG} \geq T_{inf} = \frac{1}{p}\sum_{k=1}^{n-p}(b+c)(n-k)t_k + \min(b,c)\sum_{k=n-p+1}^{n} t_k.$$

4.4.2. Double 2-steps graphs

```
{General form of the double 2 steps algorithm}
For k = 1 to n - 1
  For j = k + 1 to n
    Execute T_kj
  For i = k + 1 to n
    Execute U_ik
  Execute S_k+1,k+1
```

We assume that the elementary tasks T_{kj} (for $j > k$) and U_{ik} (for $i > k$) can be processed bt_k units, and that $S_{k+1,k+1}$ can be processed in at_k time units, where a and b are intege

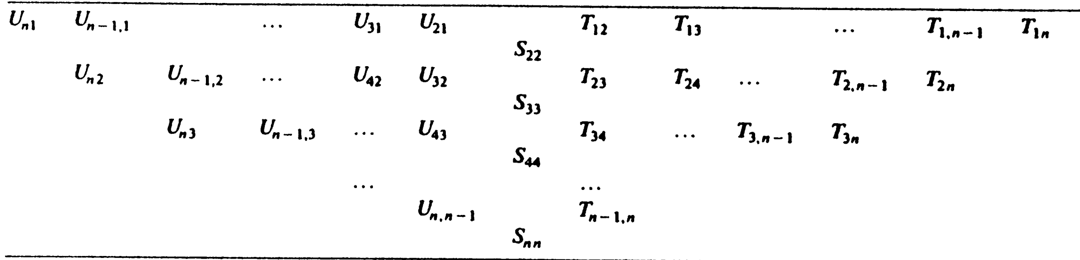


Fig. 13. Double 2-steps graph.

and $t_k = k$ for all k or $t_k = n - k$ for all k. The precedence constraints are

$$\begin{aligned}
&S_{kk} \ll T_{kj}, && k+1 \leqslant j \leqslant n,\ 2 \leqslant k \leqslant n-1,\\
&S_{kk} \ll U_{ik}, && k+1 \leqslant i \leqslant n,\ 2 \leqslant k \leqslant n-1,\\
&T_{k,k+1} \ll S_{k+1,k+1}, && 1 \leqslant k \leqslant n-1,\\
&T_{kj} \ll T_{k+1,j}, && k+2 \leqslant j \leqslant n,\ 1 \leqslant k \leqslant n-2,\\
&U_{k+1,k} \ll S_{k+1,k+1}, && 1 \leqslant k \leqslant n-1,\\
&U_{ik} \ll U_{i,k+1}, && k+2 \leqslant i \leqslant n,\ 1 \leqslant k \leqslant n-2.
\end{aligned}$$

This leads the following graph of Fig. 13 (see Fig. 6).

The algorithm that we propose proceeds in three phases, just as for the 2-steps graph. The only difference lies in the levels where the different phases terminate. There is twice as many tasks per level in the double graph.

Let p be even without loss of generality. Phase 1 is asynchronous and corresponds to level 1 up to level $(n - sp/2)^+$. Phase 2 and 3 are synchronous and correspond to level $(n - sp/2)^+ + 1$ up to $(n(s-1)p/2)^+$, and to level $(n(s-1)p/2)^+ + 1$ up to n respectively.

This leads to the following execution times:

$$\tau_1 = 1/p \sum_{k=1}^{(n-sp/2)^+} 2(n-k)bt_k + \mathrm{O}(n),$$

$$\tau_2 = \sum_{k=(n-sp/2)^+ + 1}^{(n-(s-1)p/2)^+} sbt_k + \mathrm{O}(n),$$

$$\tau_3 = \sum_{k=(n-(s-1)p/2)^+ + 1}^{n} (a+b)t_k + \mathrm{O}(n).$$

The lower bound τ_4 can be derived in a similar way,

$$\tau_4 = \sum_{k=(n-sp/2)^+ + 1}^{n} bt_k + \mathrm{O}(n).$$

The execution time of the parallel algorithm can be computed just as in Section 4.2. Letting $p = \alpha n$, the cases to be discussed are respectively $0 < \alpha < 2/s$, $2/s < \alpha < 2/(s-1)$, and $2/(s-1) < \alpha < 1$. We refer to the analysis of algorithm (F) and (G) (LDM^t decomposition) in the next section.

5. Performances evaluation

In this section, we compute the execution times of the parallel algorithms that we propose for the ten versions listed in Section 3. Then we compare their performances.

5.1. Computing the execution times

The preceding analysis leads to the following results where we present the leading coeff cient, corresponding to n^2, and omit linear terms. We also tabulate the asymptotic efficienc Let us recall that the asymptotic efficiency is $E_\alpha = T_1/(\alpha n T_{\alpha n})$ where T_1 is the sequenti execution time and $T_{\alpha n}$ is the execution time with αn processors.

(A) *Generic Gaussian elimination*: *Form KJI-SAXPY*. The precedence graph is a two-ste graph with $a = 1$, $b = 2$, $s = 2$ and $t_k = n - k$. We point out that there is only one case, name $\alpha = 1$, for which the asymptotically optimal time is known, see Table 2.

(B) *Generic Gaussian elimination*: *Form JKI-GAXPY*. The precedence graph and the perfo mances evaluation are the same as those of (A).

(C) *Generic Gaussian elimination*: *Form IJK-DOT*. The precedence graph is a double gree graph with $b = c = 2$ and $t_k = k$. We point out that in this case the asymptotically optimal tin is unknown, see Table 3.

(D) *Generic Gaussian elimination*: *Form KJI-greedy SAXPY*. The precedence graph is greedy graph with $b = 2$ and $t_k = n - k$. We point out that in this case the time we give is t asymptotically optimal time, see Table 4.

(E) *Doolittle reduction*. The precedence graph and the performances evaluation are the sar as those of (C).

(F) *LDM' decomposition*: *non-optimized code*. The precedence graph is a double two-ste graph with $a = 3$, $b = c = 3$, $s = 2$ and $t_k = k$; see Table 5.

(G) *LDM' decomposition*: *optimized code*. The precedence graph is a double two-steps gra with $a = 4$, $b = c = 2$, $s = 3$ and $t_k = k$. We point out that since $a = 2b$, our algorithm asymptotically optimal for $\alpha > 2/3$; see Table 6.

(H) *LDL' decomposition*: *non-optimized code*. The precedence graph is a two-steps gra with $a = b = 3$, $s = 2$ and $t_k = k$. We point out that since $a = b$, our algorithm is asymptotica optimal for $\alpha > 1/s$; see Table 7.

Table 2

α	0		1/2		1
T_{alg}		$2/(3\alpha)+13\alpha^2/6$		$2-\alpha^2/2$	
T_{inf}		$2/(3\alpha)-4\alpha^2/3$		1	
T_{2S}		?		?	3/2
E_α		$1/(1+13\alpha^3/4)$		$4/(12\alpha-3\alpha^3)$	4/9

Table 3

α	0		1
T_{alg}		$2/(3\alpha)+2\alpha-2\alpha^2/3$	
T_{inf}		$2/(3\alpha)+\alpha^2/3$	
T_{DG}		?	2
E_α		$1/(1+3\alpha^2-\alpha^3)$	

Table 4

α	0		1
T_G		$2/(3\alpha)+\alpha^2/3$	
E_α		$1/(1+\alpha^3/2)$	

Table 5

α	0		1
T_{alg}		$1/\alpha+3\alpha-\alpha^2$	
T_{inf}		$1/\alpha+\alpha^2/2$	
T_{D2S}		?	
E_α		$1/(1+3\alpha^2-\alpha^3)$	

Table 6

α	0		2/3		1
T_{alg}		$2/(3\alpha)+9\alpha/2-9\alpha^2/4$		3	
T_{inf}		$2/(3\alpha)-3\alpha/2+9\alpha^2/4$		1	
T_{D2S}		?		3	
E_α		$1/(1+27\alpha^2/4-27\alpha^3/8)$		$2/9\alpha$	

Table 7

α	0		1/2		1
T_{alg}		$1/(2\alpha)+6\alpha-4\alpha^2$		3	
T_{inf}		$1/(2\alpha)+2\alpha^2$		3/2	
T_{2S}		?		3	
E_α		$1/(1+12\alpha^2-8\alpha^3)$		$1/(6\alpha)$	

(I) *LDL^t decomposition: optimized code*. The precedence graph is a two-steps graph with $a=3$, $b=2$, $s=3$ and $t_k=k$. We point out that since $a=2b$, our algorithm is asymptotically optimal for $\alpha>1/s$; see Table 8.

(J) *Cholesky decomposition: GG^T*. The precedence graph is a two-steps graph with $a=b=2$, $s=2$ and $t_k=k$. We point out that since $a=b$, our algorithm is asymptotically optimal for $\alpha>1/s$; see Table 9.

Table 8

α	0		1/2		1
T_{alg}		$1/(3\alpha)+9\alpha-9\alpha^2$		3	
T_{inf}		$1/(3\alpha)-3\alpha+9\alpha^2$		1	
T_{2S}		?		3	
E_α		$1/(1+27\alpha^2-27\alpha^3)$		$1/(9\alpha)$	

Table 9

α	0		1/2		1
T_{alg}		$1/(3\alpha)+4\alpha-8\alpha^2/3$		2	
T_{inf}		$1/(3\alpha)+4\alpha^2/3$		1	
T_{2S}		?		2	
E_α		$1/(1+12\alpha^2-8\alpha^3)$		$1/(6\alpha)$	

5.2. Comparison

In this section, $T_{(X)}$ denotes the asymptotic execution time of the parallel algorithm that we propose for the version (X) (we omit linear terms).

5.2.1. Optimized versus non-optimized codes

We first check that the optimized versions of the LDM^t and LDL^t decompositions lead to better parallel algorithms than the non-optimized ones, just as in the sequential case:

(i) comparing algorithms (F) and (G):

for $\alpha < 2/3$, $T_{(F)} - T_{(G)} = n^2(4 - 18\alpha^2 + 15\alpha^3)/(12\alpha)$.
for $\alpha \geqslant 2/3$, $T_{(F)} - T_{(G)} = n^2(1-\alpha)^3/\alpha$,
in both cases, $T_{(F)} - T_{(G)} \geqslant 0$;

(ii) comparing algorithms (H) and (I):

for $\alpha < 1/3$, $T_{(H)} - T_{(I)} = n^2(1 - 18\alpha^2 + 30\alpha^3)/(6\alpha)$,
for $1/3 \leqslant \alpha < 1/2$, $T_{(H)} - T_{(I)} = n^2(1-2\alpha)^3/(2\alpha)$,
for $1/2 \leqslant \alpha$, $T_{(H)} - T_{(I)} = 0$,
in all cases, $T_{(H)} - T_{(I)} \geqslant 0$.

5.2.2. Comparing Gaussian-like elimination algorithms

There are six versions with the same sequential time ($2n^3/3$ arithmetic operations) to be compared. We point out that comparing the execution times is equivalent to comparing the efficiency in this case. These six versions are

(A) form KJI-SAXPY,
(B) form JKI-GAXPY,
(C) form IJK-DOT,
(D) form KJI-greedy SAXPY,
(E) Doolittle reduction,
(G) optimized LDM^t decomposition.

We already know that $T_{(A)} = T_{(B)}$ and $T_{(C)} = T_{(F)}$.

(i) comparing algorithms (A) and (D):

for $\alpha < 1/2$, $T_{(A)} - T_{(D)} = n^2(11\alpha^2/6)$,
for $1/2 \leqslant \alpha$, $T_{(A)} - T_{(D)} = n^2(-4 + 12\alpha - 5\alpha^3)/(6\alpha)$,
in both cases, $T_{(A)} - T_{(D)} \geqslant 0$;

(ii) comparing algorithms (C) and (A):

for $\alpha < 1/2$, $T_{(C)} - T_{(A)} = n^2(12 - 17\alpha)\alpha/6$,
for $1/2 \leqslant \alpha$, $T_{(C)} - T_{(A)} = n^2(4 - 12\alpha + 12\alpha^2 - \alpha^3)/(6\alpha)$,
in both cases, $T_{(C)} - T_{(A)} \geqslant 0$;

(iii) comparing algorithms (G) and (A):

for $\alpha < 1/2$, $T_{(G)} - T_{(A)} = n^2(54 - 53\alpha)\alpha/(12\alpha)$,
for $1/2 \leqslant \alpha < 2/3$, $T_{(G)} - T_{(A)} = n^2(8 - 24\alpha + 54\alpha^2 - 21\alpha^3)/(12\alpha)$,
for $2/3 \leqslant \alpha$, $T_{(G)} - T_{(A)} = n^2(1 + \alpha^2/2)$,
in all cases, $T_{(G)} - T_{(A)} \geqslant 0$;

(iv) comparing algorithms (G) and (C):

for $\alpha < 2/3$, $T_{(G)} - T_{(C)} = n^2(30 - 19\alpha)\alpha/12$,
for $2/3 \leqslant \alpha$, $T_{(G)} - T_{(C)} = n^2(-2 + 9\alpha - 6\alpha^2 + 2\alpha^3)/(3\alpha)$,
in all cases, $T_{(G)} - T_{(C)} > 0$.

We sum up these comparisons in the following proposition:

Proposition 5.1. (i) $T_{(A)} = T_{(B)}$ *and* $T_{(C)} = T_{(E)}$.
(ii) $T_{(G)} \geqslant T_{(C)} \geqslant T_{(A)} \geqslant T_{(D)}$.

Among the six versions, the form KJI-greedy SAXPY has the best performances, followed by the forms KJI-SAXPY and JKI-GAXPY, whatever the number of processors.

The last parallelizable versions are IJK-DOT, Doolittle reduction and optimized LDM^t decomposition.

Let us compare now the optimal asymptotic performances of the six versions. Let $T_{(X)opt}$ denote the time of an optimal algorithm for the version (X). The greedy algorithm that we propose for version (D) is optimal, hence we know $T_{(D)opt}$. We do not know $T_{(A)opt} = T_{(B)opt}$, $T_{(C)opt} = T_{(E)opt}$ nor $T_{(G)opt}$. Proposition 4.4 shows that $T_{(A)opt} = T_{(B)opt} \geqslant T_{(D)opt}$. Moreover, we obtain lower bounds for $T_{(C)opt}$ and $T_{(G)opt}$ which lead to

- $T_{(C)opt} \geqslant T_{(D)opt}$ for $\alpha \leqslant 1/3$,
- $T_{(G)opt} \geqslant T_{(D)opt}$ for $\alpha \geqslant 13/23$.

We conjecture that version (D) is the most efficient for any $\alpha \in [0,1]$.

5.2.3. Comparing algorithms for symmetric matrices

We compare version (I) optimized LDL^t and version (J) Cholesky decomposition:

for $\alpha < 1/3$, $T_{(I)} - T_{(J)} = n^2(15 - 19\alpha)\alpha/3$,
for $1/3 \leqslant \alpha < 1/2$, $T_{(I)} - T_{(J)} = n^2(-1 + 9\alpha - 12\alpha^2 + 8\alpha^3)/(3\alpha)$,
for $1/2 \leqslant \alpha$, $T_{(I)} - T_{(J)} = n^2$,
in all cases, $T_{(I)} - T_{(J)} \geqslant 0$.

6. Concluding remarks

We have presented ten parallel versions of three triangularization algorithms for solving dense linear systems. We have used a graph-theoretical approach to analyse and compared their performances.

The precedence graphs of the ten versions reduce to four classes of graphs. Using $p = \alpha n$ processors, where $\alpha \leqslant 1$, we design for each class a parallel algorithm and obtain either asymptotically optimal execution times of lower bounds thereof.

It is worth pointing out that designing parallel algorithms is quite different from designing vector pipeline algorithms. For instance, the two vector versions KJI-SAXPY and JKI-GAXPY of [3] lead to the same precedence graph, hence to the same parallel implementation. Contrarily to the vector case where the JKI-GAXPY appears to be the best form [3], the KJI-greedy SAXPY is the most suitable for parallelization on an MIMD machine.

We have concentrated on triangularization algorithms without pivoting. It is possible to include the possibility of partial pivoting in our model, assuming that the comparison of two reals also requires an unit of time. For instance, the KJI version with partial pivoting of [13] leads to a 2-steps graph with $a = b = 2$.

Finally, the computer organization that we have assumed consists of sequential processors which communicate through a shared memory. Looking forward, choosing as a unit of time any

of the four arithmetic operations does not take into account the possibility for a given task to be pipelined itself on a vector processor. The model should be refined to deal with asynchronous vector multiprocessors.

References

[1] M. Cosnard, Y. Robert and D. Trystram, Résolution parallèle de systèmes linéaires denses par diagonalisation, *Bulletin EDF C* **2** (1986) 67–87.

[2] M. Cosnard, Y. Robert and D. Trystram, Comparison des méthodes parallèles de diagonalisation pour la résolution de systèmes linéaires denses, *C.R. Acad. Sci. Paris Ser. I 301* (16) (1985) 781–784.

[3] J.J. Dongarra, F.G. Gustavson and A. Karp, Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, *SIAM Rev.* **26** (1) (1984) 91–112.

[4] M. Feilmeier (Ed)., *Parallel Computers — Parallel Mathematics* (North-Holland, Amsterdam, 1977).

[5] M.J. Flynn, Very high-speed computing systems, *Proc. IEEE* **54** (1966) 1901–1909.

[6] D.D. Gajski and J.K. Peir, Essential issues in multiprocessors systems, *IEEE Comput.* (June 1985) 9–27.

[7] G.H. Golub and C.F. Van Loan, *Matrix Computation* (The Johns Hopkins University Press, Baltimore, MD, 1983).

[8] D. Heller, A survey of parallel algorithms in numerical linear algebra, *SIAM Rev.* **20** (1978) 740–777.

[9] R.W. Hockney and C.R. Jesshope, *Parallel Computers: Architectures, Programming and Algorithms* (Adam Hilger, Bristol, 1981).

[10] K. Hwang and F. Briggs, *Parallel Processing and Computer Architecture* (McGraw-Hill, New York, 1984).

[11] S.P. Kumar, Parallel algorithms for solving linear equations on MIMD computers, PhD. Thesis, Washington State University, 1982.

[12] S.P. Kumar and J.S. Kowalik, Parallel factorization of a positive definite matrix on an MIMD computer, *Proc. ICCD 84* (1984) 410–416.

[13] R.E. Lord, J.S. Kowalik and S.P. Kumar, Solving linear algebraic equations on an MIMD computer, *J. ACM* **30** (1) (1983) 103–117.

[14] G. Rote, Private communication.

[15] Y. Saad, Communication complexity of the Gaussian elimination algorithm on multiprocessors, Report DCS/348, Yale University, 1985.

[16] A. Sameh, An overview of parallel algorithms, *Bulletin EDF* (1983) 129–134.

[17] U. Schendel, *Introduction to Numerical Methods for Parallel Computers* (Wiley, New York, 1984).

C. R. Acad. Sci. Paris, t. 306, Série I, p. 295-298, 1988

Informatique théorique/*Computer Science*

Optimalité d'une classe d'algorithmes d'ordonnancement pour la méthode de Gauss en parallèle

Jean-Claude KONIG, Yves ROBERT et Denis TRYSTRAM

Résumé – Nous présentons un algorithme parallèle asymptotiquement optimal pour la résolution par la méthode de Gauss d'un système linéaire dense. L'architecture sous-jacente est une structure SIMD/MIMD à mémoire partagée. Pour un problème de taille n et un nombre de processus $p=\alpha n$, $\alpha \leqq 1/(2+\sqrt{2})$, l'algorithme proposé est d'efficacité $e=1/(1+\alpha^3)$.

Optimality of a class of scheduling algorithms for parallel Gaussian elimination

Abstract – We present an asymptotically optimal algorithm for parallel Gaussian elimination of dense linear systems on a shared memory SIMD/MIMD architecture. For a problem of size n, with $p=\alpha n$ processors, $\alpha \leqq 1/(2+\sqrt{2})$, the efficiency of the algorithm is $e=1/(1+\alpha^3)$.

1. INTRODUCTION. – Pour résoudre un système linéaire dense $A x=b$ sur un ordinateur séquentiel classique, la méthode la plus utilisée est celle de Gauss avec pivotage partiel. Nous étudions ici la parallélisation de cet algorithme d'élimination sur une machine de type SIMD/MIMD à mémoire partagée. Pour un problème de taille n et un nombre de processeurs $p=\alpha n$, $\alpha \leqq 1/(2+\sqrt{2})$, nous proposons un algorithme asymptotiquement optimal, d'efficacité $e=1/(1+\alpha^3)$.

2. UN MODÈLE THÉORIQUE. – Nous faisons les hypothèses et les restrictions suivantes :

(i) On note e_p l'efficacité de l'algorithme parallèle considéré, définie comme le rapport $T_1/(p T_p)$ [3], où p est le nombre de processeurs, T_p le temps d'exécution de l'algorithme avec p processeurs, et T_1 le temps d'exécution de l'algorithme séquentiel.

(ii) Nous considérons un système capable de supporter plusieurs flux d'instructions s'exécutant indépendamment et en parallèle sur plusieurs flux de données ([3], [8]). Nous supposons qu'il existe des mécanismes permettant de synchroniser le processus de résolution, dont le coût sera négligé devant celui des opérations arithmétiques. Le temps d'accès en mémoire centrale pour lire ou stocker une donnée sera considéré comme nul. Pour rester réalistes, ces hypothèses appellent les restrictions suivantes :

– la communication entre processeurs se fait par l'intermédiaire d'une mémoire partagée plutôt que par un bus local;

– pour un problème de taille n, le nombre de processeurs p est limité à $O(n)$.

Saad [7] montre que les temps de communications prédominent pour résoudre un problème matriciel de taille n avec n^2 processeurs. Plus précisément, nous poserons $p=\alpha n$, avec $\alpha \leqq 1$.

(iii) Nous suppposons que chaque processeur peut réaliser en une unité de temps une multiplication suivie d'une soustraction, ou une comparaison suivie d'une multiplication, ou encore une division.

(iv) L'accès aux éléments d'une matrice A s'effectue par colonnes. Les deux opérations de transfert possibles seront la transmission d'une colonne de A de l'organe de stockage vers un organe de traitement et l'opération inverse. La duplication d'une colonne aura un coût nul : nous supposons qu'il est possible de transférer simultanément une même donnée vers plusieurs processeurs. Dans ce cas, aucun processeur ne pourra modifier

---

Note présentée par Jacques-Louis LIONS.

0249-6291/88/03060295 $ 2.00 © Académie des Sciences

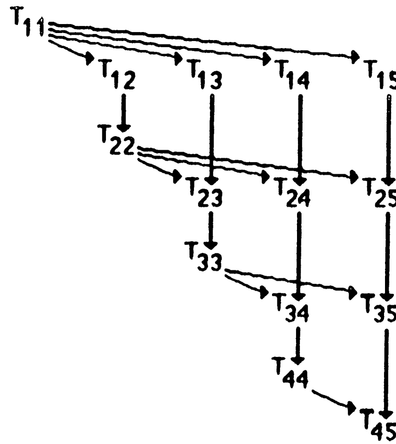


Fig. 1. — Graphe des tâches pour $n=5$.
Fig. 1. — Task graph for $n=5$.

cette donnée. Inversement, un processeur ne pourra modifier une donnée que s'il est le seul à en posséder un exemplaire.

(v) Nous utilisons le concept de graphe d'ordonnancement ou graphe des tâches ([4], [5]), où une tâche élémentaire T est une unité indivisible de traitement spécifiée uniquement en termes de son comportement extérieur : entrées, sorties et temps d'exécution, noté Ex(T). La relation de précédence $T \ll T'$ signifie que l'exécution de la tâche T doit être terminée avant que la tâche T′ ne puisse commencer. Pour un problème de taille n, le temps moyen d'exécution des tâches élémentaires doit être de l'ordre de n, afin de pouvoir effectivement négliger les coûts de synchronisation et de communication des données devant les opérations arithmétiques.

3. Graphe des taches. — Nous rappelons l'algorithme de Gauss avec pivotage partiel :

pour $k := 1$ **à** $n-1$ **faire**
  exécuter T_{kk}
    < trouver p tel que $|a_{pk}| = \max\{a_{kk}, \ldots, a_{nk}\}$
    $\mathrm{piv}(k) := p$ (*choix de la ligne p comme ligne pivot *)
    échanger $a_{\mathrm{piv}(k),\,k}$ et a_{kk}
    $c := 1/a_{kk}$
    pour $i := k+1$ **à** n **faire**
      $a_{ik} := a_{ik} * c$ >
  pour $j := k+1$ **à** n **faire**
    exécuter T_{kj}
      < échanger $a_{\mathrm{piv}(k),\,j}$ et a_{kj}
      pour $i := k+1$ **à** n **faire**
        $a_{ij} := a_{ij} - a_{ik} * a_{kj}$ >

Les contraintes d'ordonnancement sont :

(A) $\qquad T_{kk} \ll T_{kj}, \qquad k+1 \leqq j \leqq n, \qquad 1 \leqq k \leqq n-1$

(B) $\qquad T_{kj} \ll T_{k+1,\,j}, \qquad k+1 \leqq j \leqq n, \qquad 1 \leqq k \leqq n-1.$

Le graphe des tâches est décrit par la figure 1. On a $\mathrm{Ex}(T_{kk}) = n+1-k$ et $\mathrm{Ex}(T_{kj}) = n-k$ si $j \neq k$. Le temps d'exécution séquentiel est $T_1 = n^3/3 + O(n^2)$. Le plus long chemin du graphe est constitué des tâches $T_{11}, T_{12}, T_{22}, \ldots, T_{kk}, T_{k,\,k+1}, \ldots, T_{n-1,\,n-1}, T_{n-1,\,n}$. Sa longueur est $T_{\mathrm{opt}} = n^2 - 1$, et il est facile de construire un algorithme s'exécutant en temps T_{opt} (nécessairement optimal) si l'on dispose de $n/2$ processeurs [5]. En fait, un temps d'exécution T_{opt} peut être obtenu avec seulement $n\sqrt{2}/4$ processeurs [9]. Pour $\alpha \geqq \alpha_0 = n\sqrt{2}/4$, l'efficacité asymptotique est alors $e_\alpha = 1/(3\alpha)$. Aucun résultat d'optimalité

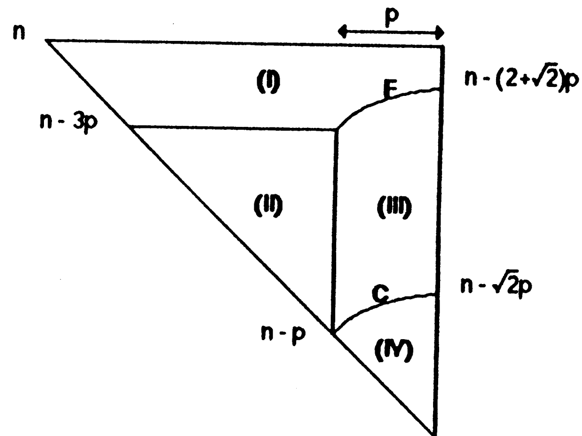


Fig. 2. — Partitionnement du graphe des tâches.
Fig. 2. — Partioning the task graph.

n'est connu pour les valeurs de $\alpha<\alpha_0$. Le meilleur algorithme conduit à une efficacité $e_\alpha=1/(1+2\alpha^3)$ si $\alpha\leqq 1/3$ [6].

4. UN ALGORITHME OPTIMAL. — Nous décrivons un algorithme asymptotiquement optimal s'exécutant avec $p=\alpha n$ processeurs, où $\alpha\leqq 1/(2+\sqrt{2})$. Nous partageons le graphe d'ordonnancement en quatre parties comme indiqué figure 2. La ligne critique C est la ligne de niveau qui passe par la tâche $T_{n-p,\, n-p}$: elle est composée des tâches $T_{i(j),\, j}$ avec $n-p\leqq j\leqq n$, $i(j)=\max\{i \mid \mathrm{ch}(T_{ij})\geqq \mathrm{ch}(T_{n-p,\, n-p})\}$, où $\mathrm{ch}(T_{ij})$ désigne le chemin critique de la tâche T_{ij} (plus long chemin dans le graphe restant à parcourir). La courbe frontière F est la parallèle à C qui passe par la tâche $T_{n-3p,\, n-p}$.

L'exécution de l'algorithme est divisée en trois phases séquentielles :

1. dans une première phase, les tâches de la partie (I) sont exécutées selon un schéma de type glouton [1];

2. les tâches de la partie (II) s'effectuent suivant l'algorithme décrit dans [1] et [6]; dès qu'un processeur se libère, il exécute les tâches de la partie (III) selon un schéma glouton, suivant les parallèles successives de F;

3. pour l'exécution des tâches de la partie (IV), on affecte simplement un processeur par colonne.

THÉORÈME. — *L'algorithme proposé pour un problème de taille n avec $p=\alpha n$ processeurs, $\alpha\leqq 1/(2+\sqrt{2})$, est asymptotiquement optimal, d'efficacité $e_\alpha=1/(1+\alpha^3)$.*

La démonstration s'effectue en plusieurs étapes :

LEMME 1. — *Le schéma d'exécution précédent respecte les contraintes d'ordonnancement.*

La seule difficulté est de montrer que les contraintes (A) sont respectées lors de l'exécution de la phase 2. La preuve est tout à fait similaire à celle donnée dans [6].

LEMME 2. — *Un schéma d'exécution quelconque pour un problème de taille n avec $p=\alpha n$ processeurs est d'efficacité asymptotique $e_\alpha\leqq 1/(1+\alpha^3)$.*

Soient $SC(p)$ et $SR(p)$ les surfaces de calcul et de repos introduites dans [5]. On a la relation $e_p\leqq 1/(1+SR(p)/SC(p))$. Pour l'algorithme de Gauss, $SC(p)=n^3/3$ et $SR(p)\geqq p^3/3$, d'où la relation cherchée.

LEMME 3. — *Le schéma d'exécution précédent est d'efficacité asymptotique $e_\alpha=1/(1+\alpha^3)$.*

On montre que tous les processeurs sont en permanence actifs lors de l'exécution des phases 1 et 2. Le temps d'exécution de ces phases est donc égal à la somme des surfaces

des parties (I), (II) et (III) divisée par p. La partie (IV) s'exécute en temps ch $(T_{n-p,\ n-p})$. Calculant la surface de la partie (IV), égale à $2p^3/3+O(n^2)$, on obtient ainsi le temps d'exécution $T_p=(1+\alpha^3)\,n^2/(3\alpha)+O(n)$, ce qui conduit au résultat.

5. CONCLUSION. — Le résultat d'optimalité présenté dans cette Note est important, puisqu'il peut s'appliquer à une large classe d'algorithmes de l'algèbre numérique linéaire (factorisation de Householder, ...). Plus précisément, notre analyse s'étend aux problèmes conduisant à des graphes d'ordonnancement comme celui de la figure 1, avec $\mathrm{Ex}(T_{kk})=a(n-k)+c_1$ et $\mathrm{Ex}(T_{kj})=b(n-k)+c_2$, où a, b, c_1 et c_2 sont quatre paramètres réels. Notons également que l'efficacité e_α de l'algorithme proposé pour $\alpha\leqq 1/(2+\sqrt{2})$ est supérieure à 0,975, ce qui montre le fort degré de parallélisme réalisé. Enfin, la détermination d'un algorithme optimal dans l'intervalle $\alpha\in]1/(2+\sqrt{2}),\ \sqrt{2}/4[\ \approx]0,29,\ 0,35[$ reste un problème ouvert.

Note reçue le 1er décembre 1987, acceptée le 7 décembre 1987.

RÉFÉRENCES BIBLIOGRAPHIQUES

[1] M. COSNARD, M. MARRAKCHI, Y. ROBERT et D. TRYSTRAM, *Parallel Computing* (à paraître).
[2] J. J. DONGARRA, F. GUSTAVSON et A. KARP, *S.I.A.M. Review*, 26, 1984, p. 91-112.
[3] K. HWANG et F. BRIGGS, *Computer architecture and parallel processing*, MacGraw Hill, 1984.
[4] S. P. KUMAR, *Ph. D. Thesis*, Washington State University, 1982.
[5] R. E. LORD, J. S. KOWALIK et S. P. KUMAR, *J. A. C. M.*, 30, n° 1, 1983, p. 103-117.
[6] M. MARRAKCHI et Y. ROBERT, *C. R. Acad. Sci. Paris*, 303, série I, 1986, p. 425-429.
[7] Y. SAAD, Report DCS/348, Yale University, 1985.
[8] A. SAMEH et D. J. KUCK, *J. A. C. M.*, 25, n° 1, 1978, p. 81-91.
[9] M. VELDHORST, Rapport RUU-CS-84-14, University of Utrecht, The Netherlands.

J.-C. K. : *E.N.S.I.A.*, 1, *rue des Olympiades*, 91300 *Massy;*
Y. R. et D. T. : *C.N.R.S.*, *Laboratoire TIM3*, *I.N.P.G.*, 38031 *Grenoble Cedex*.

OPTIMAL SCHEDULING ALGORITHMS FOR PARALLEL GAUSSIAN ELIMINATION

Yves ROBERT and Denis TRYSTRAM

CNRS, Laboratoire TIM3 - INPG
46, ave F. Viallet
38031 Grenoble Cedex, France

This paper uses a graph-theoretic approach to derive asympotically optimal algorithms for parallel Gaussian elimination on SIMD/MIMD computers with a shared memory system. Given a problem of size n and using $p=\alpha n$ processors, where $\alpha \leq 2/\sqrt{43} \simeq 0.305$, the asymptotically optimal algorithms are of efficiency $e_\alpha = 1/(1+\alpha^3) \geq 0.972$. This evidences the high degree of parallelism that can be achieved.

1. INTRODUCTION

This paper uses a graph-theoretic approach to provide a general framework for expressing the parallelism inherent into various numerical methods for solving dense linear systems of algebraic equations. Let A x = b be a linear system of size n. Most solution methods can be expressed in the general compact form:

```
DO k = 1, n-1
    1. prepare a transformation using column k of A
    call PREP(...,k,...)
    2. apply this transformation to columns j from k+1 to n
    DO j = k+1, n
        call APPLY(...,k,j,...)
```

This paper is restricted to the Gaussian elimination algorithm with partial pivoting, which clearly fits into this formalism. However, the results can be easily extended to a wide class of solution methods (e.g. Householder reduction).

We base our analysis of the parallel implementation on the task graph model presented in [1] [9] [11]. Informally, sequential algorithms are splitted into elementary tasks, whose execution ordering is directed by precedence constraints: independant tasks can be processed simultaneously. The task graph model which can be constructed directly from these precedence constraints, is the basic tool of our theoretical analysis.

Informally, let T_{kk} denote the task of preparing the k-th transformation and T_{kj} the task of applying it to update column j. The precedence constraints are the following:

- T_{kk} precedes T_{kj} for all $j \geq k+1$ (the generation of the transformation must be completed before it can be applied)
- T_{kj} precedes $T_{k+1,j}$ for all $j \geq k+1$ (the updating of column j at step k+1 can only begin when its updating from step k is terminated).

This work has been supported by the CNRS through the GRECO C3

These constraints lead to the generic task graph depicted in figure 1. The design and analysis of parallel scheduling algorithms for this task graph is the subject of this paper. We first set in a more precise way the basic definitions and assumptions for the graph-theoretic model, and we make some general remarks about its applicability. Then we recall results from the literature dealing with the study of the task graph of figure 1. We prove optimality results in the last section. More precisely, for Gaussian elimination with partial pivoting, given a system of size n and using $p = \alpha n$ processors, where $\alpha \leq 2/\sqrt{43} \sim 0.305$, then we derive asymptotically optimal algorithms of efficiency $e_{\infty,\alpha} = 1/(1+\alpha^3) \geq 0.972$. This evidences the high degree of parallelism that can be achieved.

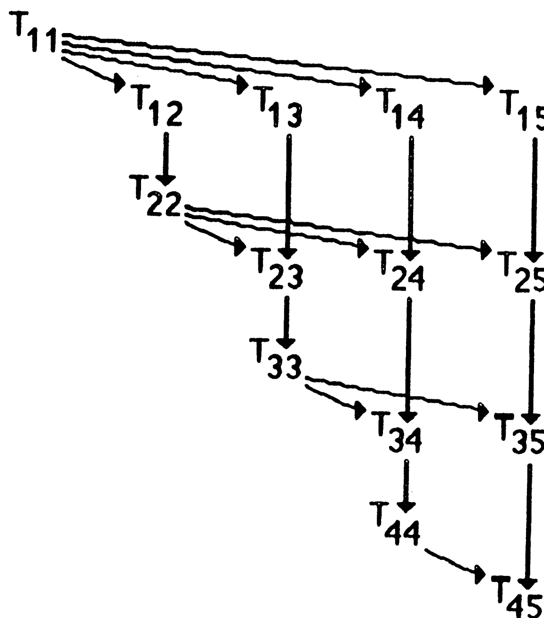


FIGURE 1
Task graph for n=5

2. THE THEORETICAL MODEL

We first present some definitions, together with some assumptions on the target architecture:

(i) We let e_p denote the efficiency of a parallel algorithm, defined as the ratio $T_1 / (pT_p)$ [8], where p is the number of processors, T_p the execution time with p processors, and T_1 the execution time of the sequential algorithm.

(ii) We assume a system which is able to support multiple instruction streams executing independently and in parallel on multiple data streams [7] [8] [14], and that there are means to impose temporal precedence constraints between the tasks of the implemented algorithms [3] [11]. We will neglect the cost for synchronization, and we suppose that each processor can perform any of the four arithmetic operations in an unit of time and that there are no memory conflicts nor data communication delays. For the model to be realistic, we need the following hypotheses:
- the processors communicate via a central shared memory rather than using a local bus between their private memories
- for a problem of size n, the number p of processors is limited to be $p = O(n)$. Saad [13] shows that communication costs dominate the arithmetic to solve a matricial problem of order n with n^2 processors. More precisely, we let $p = \alpha n$, with $\alpha \leq 1$.

(iii) Assuming a FORTRAN-like programming environment, the elements of a matrix A are accessed by columns. Only two transfert operations are allowed: loading a column of A from the central storage to a processor, and storing a column of A in memory. Duplicating a column will be of zero cost, which means that we allow for the simultaneous transfer of a given data to several processors. In such a case, no processor will be authorized to modify this data. On the other hand, a processor can modify a data only if it is resident in its own memory.

(iv) We employ the notion of task system [1] [3] [9] [10] [11], where an elementary task T is an indivisible unit of computational activity, specified only in terms of its external behavior: inputs, outputs, and execution time Ex(T). Throughout the paper, the relation $T \ll T'$ denotes the precedence constraint and means that task T is to be completed before task T' can start its execution. Elementary tasks will be of length O(n), so that synchronization overhead and data transfer costs do not predomine over arithmetic.
A model where communication delays are neglected could appear insufficient. However, we deal with methods for which the elements of a given matrix are accessed columnwise. Hence we can assume a stride-one accessing of data as prevalent in vector machines and cache-based architectures, so that data loading and unloading can be pipelined, and overlapped with arithmetic.
Another important situation where the model is quite reasonable is the following: looking back at our generic task graph, we can include the communication cost into the execution time of the tasks whenever the number of data loads and stores is proportional to the number of floating-point operations. This is the case for most numerical algorithms (we detail this point for Gaussian elimination in the next section).

3. TASK GRAPH FOR PARALLEL GAUSSIAN ELIMINATION

We recall the Gaussian elimination algorithm with partial pivoting [5] [6]:

DO k = 1, n-1
  execute T_{kk}
      $<$ determine p such that $|a_{pk}| = \max\{ a_{kk}, \dots, a_{nk} \}$
      pivot(k) = p
      interchange $a_{pivot(k),k}$ and a_{kk}
      $c = 1 / a_{kk}$
      DO i = k+1, n
          $a_{ik} = a_{ik} * c >$
  DO j = k+1, n
      execute T_{kj}
          $<$ interchange $a_{pivot(k),j}$ and a_{kj}
          DO i = k+1, n
              $a_{ij} = a_{ij} - a_{ik} * a_{kj} >$

The precedence constraints are:
(A) $T_{kk} \ll T_{kj}$ $\quad k+1 \le j \le n,\ 1 \le k \le n-1$
(B) $T_{kj} \ll T_{k+1,j}$ $\quad k+1 \le j \le n,\ 1 \le k \le n-1.$

The task graph is the one of figure 1. Assuming that comparing or interchanching two real numbers requires a unit of time, we have $Ex(T_{kk}) = 2(n-k) + 3$, and $Ex(T_{kj}) = 2(n-k) + 1$ if $j \neq k$. For T_{kk} there are n-k+1 data loads, and as many stores. For T_{kj}, $j \neq k$, there are 2(n-k+1) loads and as many stores. We can multiply these quantities by a relative communication cost unit τ_c to include them in the execution time, so as to take communication into account. A detailed study of the influence of communication costs has been made in [4].
This example motivates the general study of a task graph similar to the one of figure 1,

with $Ex(T_{kk}) = a(n-k) + c_1$ and $Ex(T_{kj}) = b(n-k) + c_2$ for $j \neq k$, where a, b, c_1 and c_2 are 4 rational parameters. For the sake of simplicity, we concentrate in this paper on the case $a = b = 1$. We also let $c_1 = c_2 = 0$, which does not modify the asymptotical analysis.

4. COMPLEXITY RESULTS

In this section, we consider the task graph of figure 1 with $Ex(T_{kk}) = Ex(T_{kj}) = n-k$, for $1 \leq k \leq n-1$, $k+1 \leq j \leq n$. We recall the precedence constraints:

(A) $T_{kk} \ll T_{kj}$ for all $j \geq k+1$, $1 \leq k \leq n-1$
(B) $T_{kk} \ll T_{k+1,j}$ for all $j \geq k+1$, $1 \leq k \leq n-1$

The sequential execution time is $T_1 = n^3/3 + O(n^2)$. The longest path of the graph is composed of the tasks $T_{11}, T_{12}, T_{22}, \ldots, T_{kk}, T_{k,k+1}, \ldots, T_{n-1,n-1}, T_{n-1,n}$. Its length is $T_{opt} = n^2-1$: this is a lower bound for the execution time of any algorithm.

Given a number of processors $p = \alpha\, n$, a parallel algorithm will have an execution time $T_p = O(n^2)$. We let n go to infinity to derive asymptotical results. The asymptotical efficiency is defined as $e_{\infty,\alpha} = \lim_{n \to \infty} e_p$. In particular, an algorithm is said to be asymptotically optimal if $e_{\infty,\alpha}$ is minimum.

It is obvious to design a parallel algorithm with $p = n$ processors whose execution time is (equivalent to) T_{opt}. The asymptotic efficiency of such an algorithm is $e_{\infty,1} = 1/3$. Lord, Kowalik and Kumar [11] propose a parallel algorithm with $p = n/2$ ($\alpha = 1/2$) which achieves the same execution time T_{opt}. Its asymptotic efficiency is thus $e_{\infty,1/2} = 2/3$. This algorithm is clearly asymptotically optimal.

This result has been improved by Veldhorst [15], who designs an algorithm with the same execution time T_{opt} and using only $p = (\sqrt{2}/4)n$ processeurs. For $\alpha \geq \sqrt{2}/4 \approx 0.354$, we deduce that there exist algorithms whose execution time is T_{opt}, of efficiency $e_{\infty,\alpha} = 1/(3\alpha)$, and this value is optimal. The value of $\sqrt{2}/4$ is very close to the lower bound derived in [11]: to execute an algorithm in time (equivalent to) T_{opt}, we need at least αn processors, where $\alpha \geq \alpha_0 \approx 0.347$ (α_0 is the solution of the equation $3\alpha-\alpha^3=1$).

For smaller values of α, no optimal algorithm is available. Optimality in this case is more difficult to prove, since the execution time of an optimal algorithm is unknown. However, this case is the most important in practice, since we usually solve large problems with a relatively small numbers of processors. The first algorithms for small values of α were due to Lord, Kowalik and Kumar [11]. The best result in the literature (to our knowledge) is the following: if $\alpha \leq 1/3$, there exists an algorithm of asymptotic efficiency $e_{\infty,\alpha} = 1/(1+2\alpha^3)$ [12]

We state now the main result of this paper:

Theorem : For a problem of size n, with $p = \alpha\, n$ processors, where $\alpha \leq 2/\sqrt{43} \approx 0.305$, we construct below an algorithm which is asymptotically optimal. Its efficiency is $e_{\infty,\alpha} = 1/(1+\alpha^3)$.

We first introduce the algorithm informally, pointing out the key-ideas of the design. We partition the task graph into four regions, as indicated figure 2. The bound $\alpha \leq 2/\sqrt{43}$ comes from this partition.

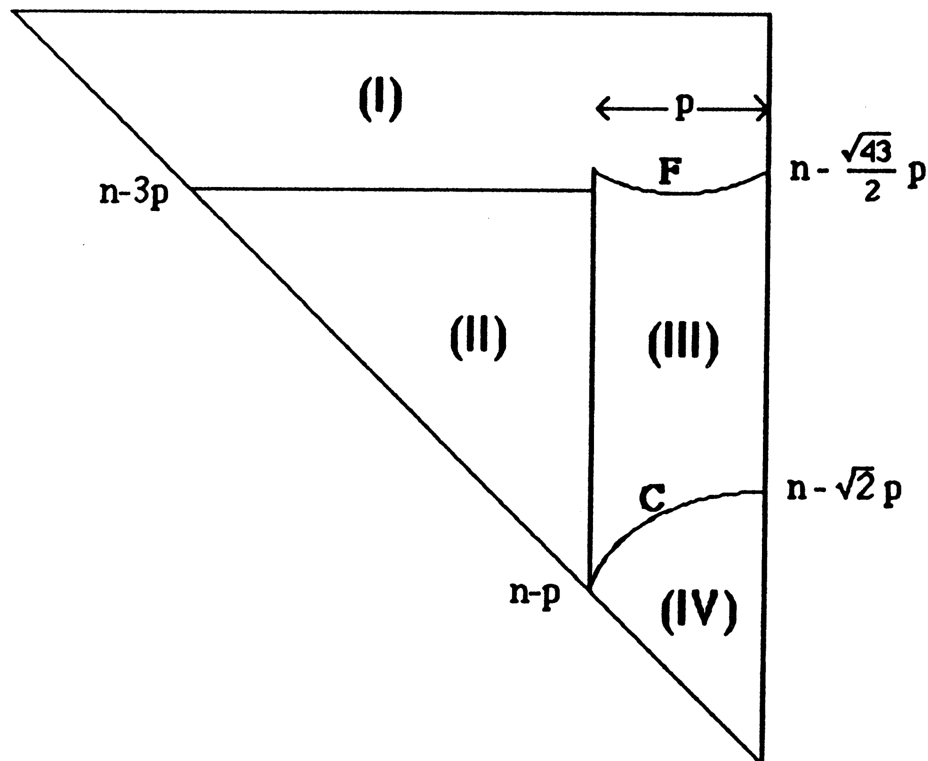


FIGURE 2
Partitioning the task graph

When designing a parallel algorithm, one would like to keep all the processors active as long as possible. However, this is clearly impossible in the bottom part of the graph. When $T_{n-p,n-p}$ has been executed (at time t, say), there are less than p tasks in the graph that can be executed in parallel, so that some processors will necessarily be inactive. All the tasks of the path { $T_{n-p+1,n-p}$, $T_{n-p+1,n-p+1}$, ..., $T_{n-1,n-2}$, $T_{n-1,n-1}$, $T_{n-1,n}$ } have to be executed sequentially. Let τ be the length of this path. It will take at least a time τ to complete the execution. The idea is to maximize the work that the p processors can do in these τ units of time. This corresponds exactly to the region (IV) of figure 2.

The challenge is then to execute all the other regions (I), (II) and (III) in full parallelism, with all the processors being active. Then we loose some parallelism when executing region (IV), but we loose the minimum. As a consequence, the resulting algorithm will be optimal.

4.1. Description of the algorithm

We call level k the row of the task graph composed of { $T_{k,k+1}$, ..., $T_{k,n}$ }. Region (I) comprises tasks of level 1 to n-3p+1. The boundary curve C between regions (III) and (IV) is the equipotential curve which contains task $T_{n-p,n-p}$: it is composed of the tasks $T_{i(j),j}$, $n-p \le j \le n$, $i(j) = \max \{ i \,/\, cp(T_{ij}) \ge cp(T_{n-p,n-p+1}) \}$, where $cp(T_{ij})$ is the length of the critical path from task T_{ij}, defined as the longest path in the graph from T_{ij} to $T_{n-1,n}$.

The boundary curve F between regions (I) and (III) will be introduced later. Technically, it is a quadratic curve which intersects level n-3p in column n-p/2 and level n-($\sqrt{43}/2$)p in column n.

The algorithm proceeds in three sequential phases:

1. in the first phase, tasks of region (I) are executed with a greedy algorithm, level by level
2. in the second phase, tasks of regions (II) and (III) are executed. We start the execution of region (II), and when a processor is released, it starts executing tasks of region (III).
3. in the third phase, for the execution of region (IV), we simply affect one processor to each column

We explicit below the execution of all phases in further details.

Phase 1

We do not consider T_{11}, which only affects the execution by a linear quantity O(n), thereby not modifying the asymptotical result. For the sake of simplicity, we assume the existence of an extra processor which performs only the diagonal tasks T_{22}, T_{33}, ... as soon as possible. Again, this does not affect the asymptotical analysis.

All the processors execute the non-diagonal tasks from one level of the graph to another, from left to right in each level, starting the execution as soon as possible (hence the name "greedy") [2]. Precedence constraints are satisfied because each processor executes at least two tasks in each level (in region (I) the number of tasks per level is at least twice the number of processors), which gives enough time to the extra processor to perform the diagonal task before the other processors start the execution of the next level.
We synchronize all the processors at the end of the phase, loosing a linear quantity O(n). Region (I) is executed in full parallelism, and phase 1 is executed in time (asymptotically) equal to the sequential time (the surface of the region) divided by p.

Phase 2

This phase is the most complicated. There are two pools P and Q of processors: processors of pool P are affected to region (II), processors of pool Q to region (III). P et Q evolve dynamically. At the beginning, all processors belong to P, at the end they are all in Q. More precisely, one processor moves from P to Q each time two new levels in (II) are processed. The execution of the tasks of region (III) are distributed among processors of pool Q in such a way that all the p processors reach simultaneously curve (C), without any idle time.

Pool P for region (II)

When processor 1 (the first to be idle) has executed the last task of level n-3p+1, it joins pool Q and from this time executes tasks of region (III). After the processing the last task of level n-3p+3, processor 2 joins pool Q, and so on (processor k joins pool Q after the execution of the level n-3p+2k-1, $0 \leq k \leq p-1$). Let us give now some technical results:

(i) block k, $0 \leq k \leq p-1$, is processed within
$t_k = 2Ex(T_{n-3p+2k,n-3p+2k}) + 2Ex(T_{n-3p+2k+1,n-3p+2k+1}) = 12p-8k-2$ units of time

(ii) region (II) is processed in time equal to the length LP(II) of the longest path $\{T_{n-3p,n-3p+1}, T_{n-3p+1,n-3p+1}, T_{n-3p+1,n-3p+2}, \ldots, T_{n-p+1,n-p+1}\}$, i.e. $LP(II) = 8p^2+o(n^2)$.

(iii) during the total execution of region (II), processor j, j<k, has been active in pool Q during the processing of blocks j+1 to k (levels n-3p+2j to n-p+1), that is a time:

$$q(j) = t_{j+1} + t_{j+2} + \ldots + t_{p-1} - Ex(T_{n-3p+2j+1,n-3p+2j+1}) = (3p-2j)^2 - p^2 + o(n^2)$$

Pool Q for region (III)

We need now to explain the scheduling for processing region (III). Before explaining the final version of the algorithm, we derive a first (simplified) version.

(i) Simplified version

The simplest idea is to assign the processor j to the execution of the tasks of column n-p+j: processor 1 would execute the tasks of the first column of region (III), processor 2 the tasks of the second one, and so on.

Let's forget the precedence constraints for a while. If we want all processors to be synchronized at the end when reaching (C), we have to assign to processor j an amount of work equal to the time q(j) during which it has not been active in pool P. This leads to the curve (F1) in figure 3: each processor j works the same amount of time LP(II) for executing its tasks in pool P and the tasks in column n-p+j between (F1) and (C).

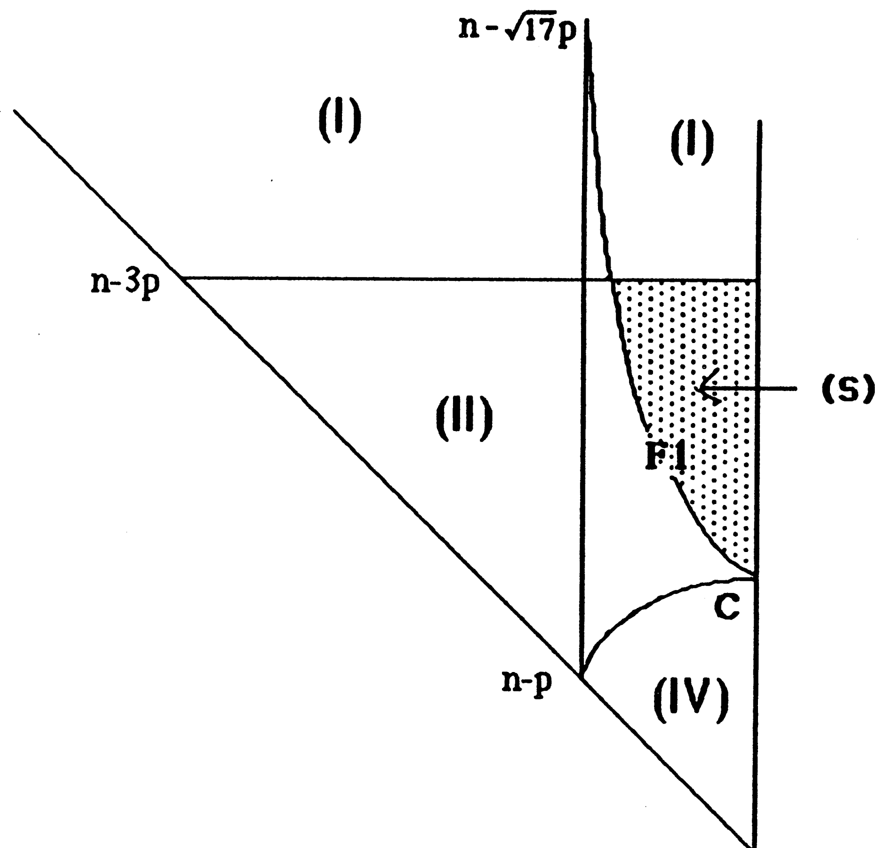


FIGURE 3
Building the curve (F1)

For example, processor 1 is active $q(1) = 8p^2 + o(n^2)$ units of time in pool Q. Hence it should start executing tasks in column 2p+1 from a level n-x such that the length of the segment of levels [n-x,n-p] is equal to q(1). We have to choose x such that

$$\Sigma_{n-p \le j \le n-x}\ n-j \;=\; x^2/2 - p^2/2 = \; q(1) = 8p^2 + o(n^2)$$

which gives $x = \sqrt{17}\, p$. This is the way we build up the curve (F1).

There remain some tasks above (F1) that have not been executed (surface (S)). Processor p is the one which has the maximum amount of work in its column: it has still to execute all the tasks in the last column between level n-3p to $n-\sqrt{2}p$, for which it will need a time $t = 7p^2/2 + o(n^2)$.

Hence we must assign to all the processors other tasks to execute in their columns, in such a way that they all work during an additional time equal to t. This corresponds to the equipotential (F2) to (F1) depicted in figure 4.

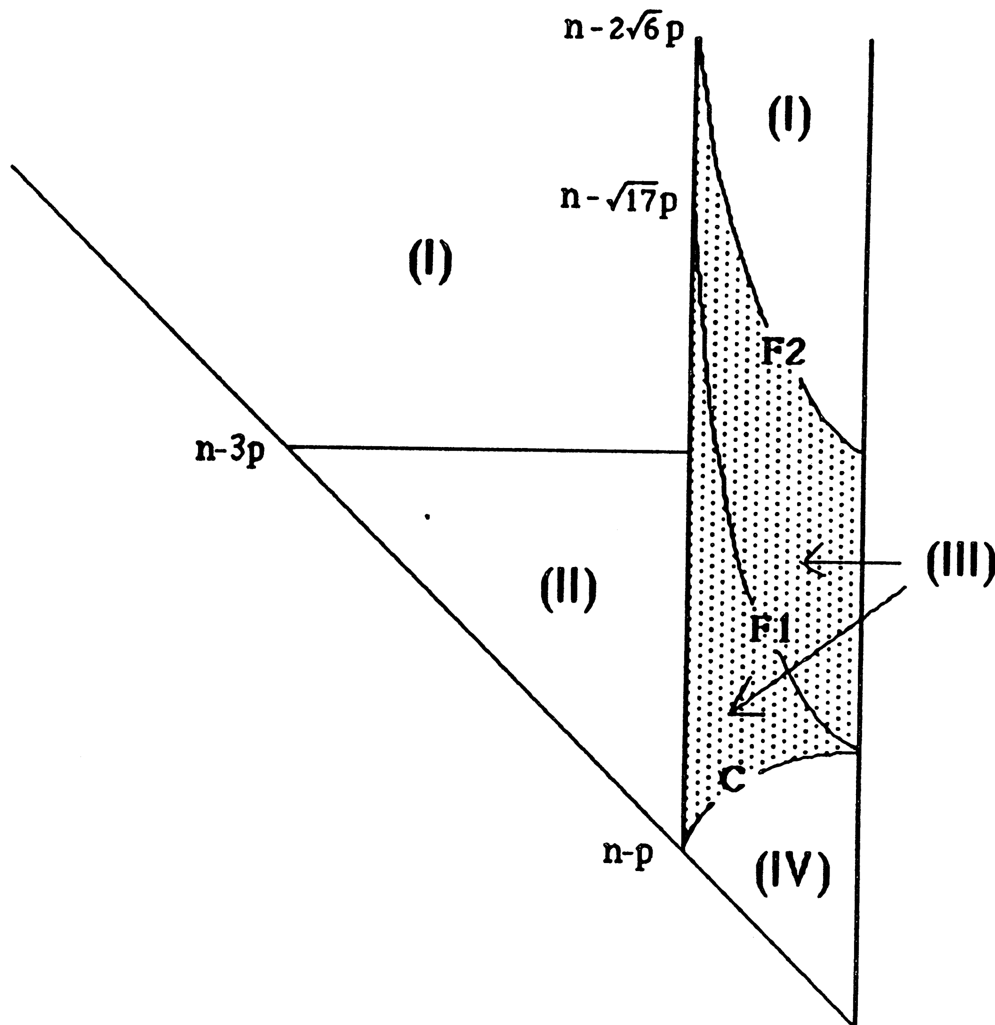


FIGURE 4
Second phase of the first algorithm, region (III)

Reversing things to their natural order, we identify region (III) as in figure 4: it is made of the tasks in the last p columns below (F2) and above (C). When it becomes active in pool Q, processor j is assigned to the execution of the tasks of column n-p+j in region (III). By construction, all processors work the same amount of time $LP(II) + t = 23p^2/2 + o(n^2)$ during phase 2 and reach simultaneously (within a linear factor) the curve (C).

Finishing the execution as explained below for phase 3 would lead to an optimal algorithm for all values of α less than $1/(2\sqrt{6}) \simeq 0.204$ (since the level $n-2\sqrt{6}p$, corresponding to the top of curve (F2), has to be positive).

We now detail a more elaborate solution which improves the bound on α up to $2/\sqrt{43} \simeq 0.305$.

(ii) Final version

Rather than assigning a column to each processor, we better balance the work by assigning a pair of columns to a pair of processors. Consider processors j and p-j, $j \le p/2$: they are responsible for the pair of columns 2p+j and 3p-j in region (III). Processor j has less work to do in region (II) than processor p-j, hence it can execute tasks both in column j and in column p-j. This will result in smoothing horizontally curve (F1).

More precisely, see figure 5: when processor j becomes active in pool Q, it executes two tasks per level, the one column j and the other column p-j, starting from level $h_2(j)$, until it reaches level $h_1(p-j)$. From this point on, it executes tasks only in column j, down to curve (C). When processor p-j becomes active in pool Q, it executes the tasks in column p-j, from level $h_1(p-j)$ down to curve (C). Note that when processor p-j becomes active, processor j is already further down in column j, at level $h_1(j)$.

Technically, let W(j,x,y) be the sum of the execution time of the tasks in column j from level x to level y, with $x \le y$. Let also W(j,x,C) denote the sum of the execution time of the tasks in column j from level x to curve (C).

The way we compute $h_1(j)$ and $h_1(p-j)$ is straigthforward: they are defined by the relation

$$W(j,h_1(j),C) = W(p-j,h_1(p-j),C) = q(p-j), \quad 1 \le j \le p/2$$

where q(p-j) is the time during which processor p-j is active in pool Q. We easily compute $h_1(j)$ and $h_1(p-j)$:

$$h_1(j) = n - (p^2 + 10\, pj + 7\, j^2)^{1/2} + o(n) \qquad 1 \le j \le p/2$$
$$h_1(p-j) = n - (2\, p^2 + 8\, pj + 7\, j^2)^{1/2} + o(n) \qquad 1 \le j \le p/2$$

To determine $h_2(j)$, we have the following relation:

$$W(j,h_2(j),C) + W(p-j,h_2(j),C) = q(j) + q(p-j) \quad 1 \le j \le p/2$$

We derive $h_2(j) = n - ((19/2)\, p^2 - 7\, pj + 7\, j^2)^{1/2} + o(n)$

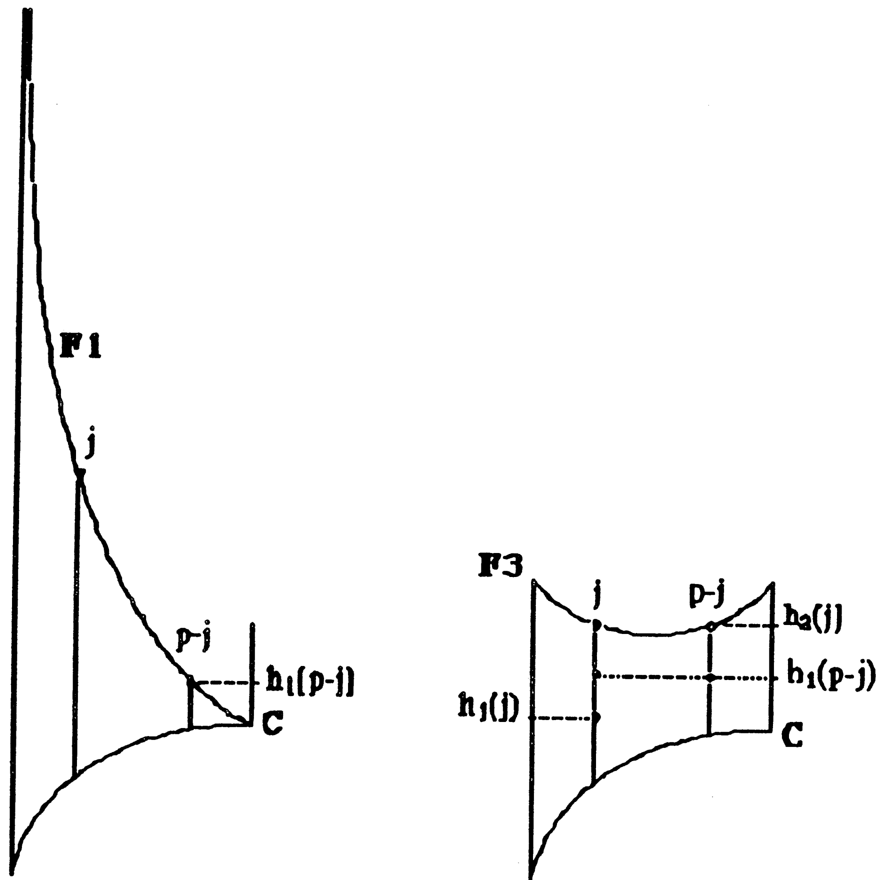


FIGURE 5
Balancing work among pairs of processors

Just as in the first version, some tasks in the last p columns have not been executed. As a consequence, we have to consider the curve (F), the equipotential to (F3), which intersects level n-3p in column n-p/2 (see figure 6). This fully explains now the partitioning of the task graph depicted above.

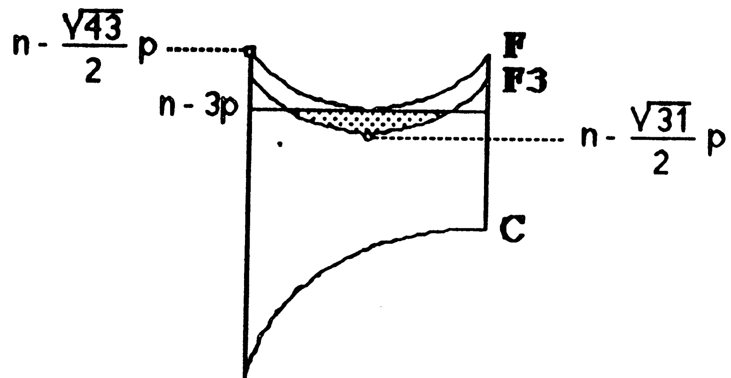


FIGURE 6
Definition of curve F

Provided that the precedence constraints are satisfied, the previous scheme will lead to a processing of region (II) and (III) with full parallelism. Just as for region (I), the execution time will be the sum of the surfaces of these two regions divided by p.

Lemma 1. The precedence constraints are satisfied by construction during phase 2 of the algorithm.

Phase 3

At the end of phase 2, processors are resynchronized, at the price of a linear factor $O(n)$ in the execution time. We affect a column to each processor: processor j executes tasks of column n-j, and at the end the diagonal task $T_{n-j,n-j}$.

To show briefly that precedence constraints (A) are satisfied, let the processors start the execution of region (IV) at time t=0. From the definition of C, we see that processor j reaches level n-j at time $t_{j,j} = cp(T_{n-p,n-p}) - (n-j)^2$. For all i>j, processor i reaches level n-j at time $t_{i,j} = cp\ (T_{n-p,n-p}) - (n-i)^2 - [(n-j)^2/2-(n-i)^2/2]$. Hence $t_{ij} \geq t_{jj}+Ex(T_{n-j,n-j})$, and the precedence constraints are fulfilled.
As a consequence, region (IV) is executed in time $cp(T_{n-p,n-p}) = p^2 + O(n)$.

4.2. Optimality of the algorithm

Lemma 2 : The asymptotic efficiency of the algorithm is $e_{\infty,\alpha} = 1/(1+\alpha^3)$

The sequential execution time is $T_1 = n^3/3 + O(n^2)$. The parallel execution time is equal to the sum Σ of the surfaces of regions (I), (II) and (III) divided by p (since these regions are processed with full parallelism) plus $cp(T_{n-p,n-p})$, the execution time of region (IV): $T_p = \Sigma / p + cp(T_{n-p,n-p})$. The surface of region (IV) is $\Sigma_4 = 2p^3/3$ (direct computation from the definition of C), and $cp(T_{n-p,n-p}) = p^2 + O(p)$. We have $\Sigma = T_1 - \Sigma_4$, hence $T_p = (n^3+p^3) / (3p)$, and the value of $e_{\infty,\alpha}$ follows.

Lemma 3 : The asymptotic efficiency of any parallel algorithm with $p = \alpha n$ processors is less than $1/(1+\alpha^3)$

Let CA be the computational area of the parallel algorithm, defined as the product of the number of processors and the execution time, less the area where not all the processors can be doing work [11]. We see from the task graph that:
- when $T_{n-1,n}$ is executed, only one processor can be doing work
- when $T_{n-1,n-1}$ and $T_{n-2,n-1}$ are executed, only two processors can be doing work
- ...

During each interval of length 2j, at most j processors are working. Thus, we have the following inequality:

$$CA \leq p\,T_p - \Sigma_{1 \leq j \leq p}\, 2j \cdot (p-j) = p\,T_p - p^3/3 + O(n^2)$$

The total amount of work is $TW = n^3/3 + O(n^2)$, it is also equal to the sequential execution time T_1. Since $CA \geq TW$, we have $p\,T_p \geq T_1 + p^3/3 + O(n^2)$, hence

$$e_p \leq 1/(1+p^3/n^3)$$

Letting $p = \alpha n$, the result follows. The theorem is a direct consequence of lemmas 2 and 3.

5. CONCLUDING REMARKS

We have presented a class of optimal scheduling algorithms for parallel Gaussian elimination on MIMD computers with a shared memory system. We have used a graph-theoretic approach first introduced in [9] [11] and further developed in [2] [12]. For a problem of size n, with $p = \alpha n$ processors, we have designed asymptotically

optimal algorithms for $\alpha \leq 2/\sqrt{43} \approx 0.305$. The corresponding efficiency is greater than 0.972, which evidences the high degree of parallelism that can be achieved.

For $\alpha = 2/\sqrt{43}$, the execution time of our algorithm is equal to $n^2 (1+\alpha^3)/3\alpha + O(n) \approx 1.167 n^2$. For α in the interval]0.293, 0.354[, the design (and performance) of an asymptotically optimal algorithm remains an open question.

REFERENCES

[1] E.G. COFFMAN, P.J. DENNING, Operating systems theory, Prentice Hall (1972)

[2] M. COSNARD, M. MARRAKCHI, Y. ROBERT, D. TRYSTRAM, Gauss elimination algorithms for MIMD computers, Proc. CONPAR 86, W. Händler et al. eds., Lecture Notes in Computer Science 237, Springer Verlag (1986), 247-254

[3] M. COSNARD, Y. ROBERT, Algorithmique parallèle: une étude de complexité, Techniques et Science Informatique 6, 2 (1987), 115-125

[4] M. COSNARD, D. TRYSTRAM, Communication complexity of Gaussian elimination on MIMD shared memory computers, Report Ecole Centrale Paris (1986)

[5] J.J. DONGARRA, F.G. GUSTAVSON, A. KARP, Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, SIAM Review 26, 1 (1984), 91-112

[6] G. H. GOLUB, C. F. VAN LOAN, Matrix computations, The Johns Hopkins University Press (1983)

[7] R.W. HOCKNEY, C.R. JESSHOPE, Parallel computers: architectures, programming and algorithms, Adam Hilger, Bristol (1981)

[8] K. HWANG, F. BRIGGS, Parallel processing and computer architecture, Mac Graw Hill (1984)

[9] S.P. KUMAR, Parallel algorithms for solving linear equations on MIMD computers, PhD. Thesis, Washington State University (1982)

[10] S.P. KUMAR, J.S. KOWALIK, Parallel factorization of a positive definite matrix on an MIMD computer, Proc. Int. Conf. ICCD 84 (1984), 410-416

[11] R.E. LORD, J.S. KOWALIK, S.P. KUMAR, Solving linear algebraic equations on an MIMD computer, J. ACM 30, 1 (1983), 103-117

[12] M. MARRAKCHI et Y. ROBERT, Un algorithme parallèle pour la méthode de Gauss. C.R. Acad. Sc. Paris 303, série I (1986), 425-429

[13] Y. SAAD, Communication complexity of the Gaussian elimination algorithm on multiprocessors, Report DCS/348, Yale University (1985)

[14] A. H. SAMEH, D.J. KUCK, On stable parallel linear system solvers, J. ACM 25, 1 (1978), 81-91

[15] M. VELDHORST, A note on Gaussian elimination with partial pivoting on an MIMD computer, Report RUU-CS-84-14, Univ. of Utrecht, The Netherlands (1984)

PARALLEL ALGORITHMS & ARCHITECTURES
M. Cosnard et al. (editors)

© Elsevier Science Publishers B. V. (North-Holland), 1986

COMMUNICATION COSTS VERSUS COMPUTATION COSTS IN PARALLEL GAUSSIAN ELIMINATION

M. COSNARD+, J.M. MULLER+, Y. ROBERT+ and D. TRYSTRAM#

+ C.N.R.S. - Laboratoire TIM3-IMAG
I.N.P.G. BP 68
38402 ST MARTIN D'HERES CEDEX, FRANCE

Ecole Centrale PARIS
Grande voie des vignes
92295 CHATENAY MALABRY CEDEX, FRANCE

This paper studies the influence of communication costs on the computational complexity of parallel Gaussian elimination algorithms. The target architecture is assumed to be a shared memory MIMD computer. The time for transfering a data from the central memory to the individual memory of a processor will be taken equal to the time for performing an arithmetic operation.
According to the classification introduced in [1] and in [2], we consider the form KJI-SAXPY modified of the generic Gaussian elimination algorithm. It is proved in [1] that this form is the most suitable for parallelization on an MIMD computer with zero communication cost. We show that the complexity of this algorithm is multiplied by two taking into account the communications.
Then we introduce the form KJI-SAXPY modified of the generic block Gaussian elimination algorithm. Neglecting the communication time, this algorithm is shown to be less efficient than the preceding one, but it is better otherwise.
Finally, we describe a new parallel version which combines the advantages of the pointwise KJI-algorithm and the block KJI-algorithm.

1. INTRODUCTION

The parallel Gaussian elimination method has been intensively studied on various parallel architectures. Six different versions for a vector pipeline machine have been considered in [2]. A parallel MIMD version of the algorithm with partial pivoting has been discussed in [3] and [4], and an implementation of the LDL^t decomposition algorithm in [5]. Together with MIMD versions of the [2] algorithms, the computational complexity of several Gaussian elimination like algorithms were analysed in [1]. Moreover systolic algorithms have been proposed in [6], [7] and [8]. Previous works are surveyed in [9], [10] and [11].

The purpose of this paper is to analyse the influence of communication costs on the computational complexity of parallel Gaussian elimination algorithms. In earlier publications on parallel algorithms, the communications costs were often not taken into account. However, in some cases, the time for transmitting one data item is not negligible relative to the time for performing a floating point operation. In [12], the communication complexity of the Gaussian elimination algorithm is studied on three parallel architectures: processors connected by a bus, a ring and a two dimensionnal grid array. It is shown that on the bus and the ring topologies the algorithm requires a communication time of at least $O(N^2)$ no matter what the number of processors is, where N is the size of the problem. On the grid, the total time including arithmetic can be reduced

---

This work has been supported by the Centre National de la Recherche Scientifique through the GRECO C3.

to $O(N^{5/3})$, and the communication time is at least $O(N^2/\sqrt{p})$ where p is the number of processors. In [13] a similar analysis has been presented on a ring consisting of a small number of processors relative to the size of the problem. In this case it was shown that the communication time is a low order term as compared with the computation time.

In this paper the target architecture is assumed to be a shared memory MIMD computer. The time for transfering a data from the central memory to the individual memory of a processor will be taken equal to the time for performing an arithmetic operation.

According to the classification introduced in [1] and in [2], we consider the form KJI-SAXPY modified of the generic Gaussian elimination algorithm (KJI-algorithm for short). Among the ten parallel versions analysed in [1] it is proved that this form is the most suitable for parallelization on an MIMD computer with zero communication cost.

In the second part of this paper we make precise the setting of our work by detailing the underlying architecture. Part 3 recall the form KJI-SAXPY modified of the generic Gaussian elimination algorithm and the computational complexity analysis of [1]. In the fourth part we show that the complexity is multiplied by two when we take into account the communication costs. The fifth part is devoted to the study of the form KJI-SAXPY modified of the generic block Gaussian elimination algorithm (block KJI-algorithm for short). Neglecting the communication time, this algorithm is shown to be less efficient than the preceding one, but it is better otherwise. Part six presents a new parallel version which combines the advantages of the pointwise KJI-algorithm and the block KJI-algorithm.

2. THE PARALLEL ARCHITECTURE

The underlying parallel architecture is composed of a set of p processors each with a local memory sharing a common centralized memory. The communication between this memory and the processors is done through a complete network so that in one time unit p data can be transferred simultaneously. Moreover we assume a system which is capable of supporting multiple instruction streams executing independently and in parallel on multiple data streams [5], [14], [15], [16] and [17]. The synchronization means are not presented but we assume that the system can handle the temporal precedence constraints which are imposed by the algorithms.

The cost analysis of the algorithms is based on the assumptions that each processor can perform any of the four arithmetic operations in an unit of time, that a data item can be read in the same unit of time and that there are no memory conflicts when accessing the shared memory. Moreover simultaneous loads of the same data is allowed whereas simultaneous stores in the same location are not. On a computer with a pipelined arithmetic unit, a communication co-processor and a memory divided in several banks these assumptions are fairly reasonable. For the sake of simplicity we assume that a processor can begin a task only when all its data are stored in its local memory, but the results of the computation can be stored in the central memory as soon as they are available. Hence the reads precede the processes but the writes overlap the processes. These are the only forms of hardware synchronization that we shall assume in the remaining.

Moreover, we shall assume in the following that we can access the elements of the matrix A by columns to be more realistic and close to a FORTRAN programming style environment. Then the two allowed transfer operations will be loading and storing a column. The duplication of a column will have a zero cost, i.e. we assume that it is possible to transfer simultaneously the same data to various processors.

Once the constraints are defined, the first step in the parallelization of a method is the definition of the elementary tasks and their precedence graph. The tasks are then assigned to the available processors according to the precedence graph. Throughout the paper, the relation $T << T'$ denotes the precedence constraint and means that task T is to be completed before task T' can start its execution [3], [4]. All other notations of graph theory will be consistent with [4] and [1].

To be more realistic, we add the following hypothesis when triangularizing a dense n by n matrix: an elementary task could not be of constant length (independent of n), contrarily to [5] and the number of processors is limited to O(n).

Most often, pivoting of rows or columns (or both) is used for stability reasons. However for the sake of simplicity and since our main interest is in algorithmic considerations, we shall not consider the overhead due to pivoting.

In order to present parallel versions of the algorithms we shall use standard fortran extended with a few new keywords for distributing the instances of a parallel loop among the processors: **doall** and **enddoall**, and for executing serial pieces of code on a single processor: **serbegin** and **serend**. The description of a full operating system can be found in [18].

3. THE KJI-ALGORITHM

The form KJI - SAXPY modified of the generic Gaussian elimination algorithm is introduced in [1]. The name of this algorithm comes from the fact that the three DO loops of Gaussian elimination are executed in the order k,j,i. It corresponds to the execution by columns of the classical form (k,i,j). The basic operation is SAXPY : update a vector by adding to it a scalar multiple of another vector. The form KJI - SAXPY has been introduced in [2] and modified in [1].

```
Do k = 1 to n-1
      Doall j = k+1 to n
            execute T_kj : < a_kj = a_kj/a_kk
                             Do i = k+1to n
                                a_ij = a_ij - a_ik*a_kj >
      enddoall
```

Each elementary task (T_{kj}) in KJI is a linear combination of two columns. The computation cost and communication cost of T_{kj} are respectively 2(n-k)+1 and 2(n-k)+2. The precedence constraints are $T_{k,k+1} << T_{k+1,j}$ for $1 \le k \le n-1$, $k+2 \le j \le n$, and $T_{kj} << T_{k+1,j}$ for $1 \le k \le n-1$, $k+2 \le j \le n$. The precedence graph is depicted in figure 1.

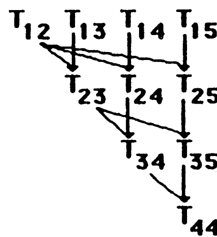


FIGURE 1
Precedence graph of Gaussian elimination
(form KJI modified) for a matrix of size 5.

The computational complexity of this algorithm has been analysed in [1]. An optimal way to assign tasks to processors according to the precedence graph is to use a greedy algorithm. Such a graph is also encountered when solving linear systems by diagonalization methods [19] and [20].

```
{ General Form of the greedy algorithm }
For k <- 1 to n-1
    For j <- k+1 to n
        Execute Tkj
```

We assume that the elementary tasks T_{kj} (for $j>k$) can be processed in $a*t_k$ units where a is an integer. For a given value of k, there are n-k tasks to be executed. The precedence graph is figured below (see also figure 1):

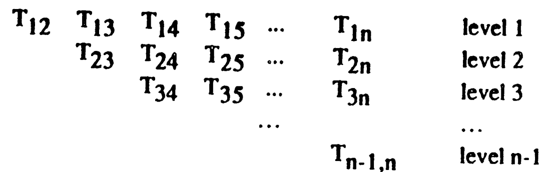


FIGURE 2
Greedy graph

The greedy algorithm G can be informally defined as follows: G executes the tasks from one level of the precedence graph to another, from left to right in each level, and it starts the maximum number of tasks at each time (hence the name of greedy). More precisely, the algorithm G executes the tasks in the following order:

$$T_{12} \leq\leq T_{13} \leq\leq \ldots \leq\leq T_{1n} \leq\leq T_{23} \leq\leq T_{24} \leq\leq T_{2n} \leq\leq T_{34} \leq\leq \ldots \leq\leq T_{n-1,n}$$

where $T \leq\leq T'$ means that the execution of T' cannot begin before that of T, but a simultaneous execution is allowed.

At time $t=1$, the algorithm G starts the execution of the tasks $T_{12}, T_{13}, \ldots, T_{1p}$. For any $t \geq 1$, if $q \leq p$ processors are idle, the algorithm G assigns these q processors to the execution of the following q tasks, provided that constraint (B) makes it possible (otherwise, the algorithm executes the maximum number of tasks).

We can now state the following result whose proof is presented in [1].

Theorem 1 :

The greedy algorithm G is asymptotically optimal. Its execution time is equal to

$$T_G = (a/p) \cdot \sum_{k=1}^{n-p} (n-k).t_k + a \cdot \sum_{k=n-p+1}^{n-1} t_k$$

where at_k is the execution time of T_{kj} (for $j>k$).

The execution time of the greedy algorithm is the sum of two terms. While the number of tasks per level is greater than the number of processors, the algorithm can be executed with efficiency one, leading to the first term. When the number of tasks per level is less than p the precedence constraints implies that a task of a level cannot be processed as long as the tasks of the preceding level are not completed. The second term is the sum of the execution time of each level.

4. COMPUTATIONAL COMPLEXITY OF THE KJI-ALGORITHM

We assume first that the communications between the processors and the shared memory have zero cost. Hence the execution of T_{ki} takes $t_k=2(n-k)+1$ time units. The elementary task is then of length $O(n-k)$ so that it could be assumed that synchronization and data communication do not overcome arithmetic operations. Moreover the number p of processors can be bounded by n since the number of tasks per level is less than n. Recall that it is shown in [12] that processor communication costs dominate arithmetic when $O(n^2)$ processors are used. Remark also that if p can be neglected with respect to n, the greedy algorithm can be executed with full efficiency which makes our analysis useless. More precisely, we set $p=\alpha n$, with $\alpha<1$. Our analysis will be asymptotic (with respect to n). Hence only the highest order of the execution times is given.

Proposition 2 :

Assuming zero communication costs, the KJI-algorithm can be executed in T_G time units:

$$T_G = 2n^3/(3p) + p^2/3$$

With $p = \alpha$ n processors, $T_G = (n^2/3)\ [\ (2/\alpha + \alpha^2)\ ]$

We assume now that the communication costs cannot be neglected. Recall that each processor can perform any of the four arithmetic operations in a unit of time, that a data item can be read in the same unit of time and that there are no memory conflicts. Moreover we assume that a processor can begin a task only when all its data are stored in its local memory, but the results of the computation can be stored in the central memory as soon as they are available. Clearly these assumptions are too simple for real parallel computers. However introducing a more complex communication model will not change the qualitative results but could lead to tedious computations and hide the significant part of the analysis. Hence the execution of T_{ki} takes $t_k=4(n-k)+3$ time units.

Proposition 3 :

Including communication costs, the KJI-algorithm can be executed in T'_G time units:

$$T'_G = 2T_G = 4n^3/(3p) + 2p^2/3$$

With $p = \alpha$ n processors, $T'_G = (2n^2/3)\ [\ (2/\alpha + \alpha^2)\ ]$

Hence taking into account the communication costs, the complexity is multiplied by two. It is worth noting that this result is not as pessimistic as those in [12] but are less optimistic than those in [13]: communication complexity is of the same order as computation complexity. This is due to the fact that we limit the number of processors to $O(n)$, that our model architecture has good communications capabilities and that the granularity of the elementary task is not too small ($O(n)$).

However, communications costs can no longer be neglected on such a model. In order that arithmetic could dominate communication, we must increase the granularity of the elementary task. This is done by considering block methods.

5. COMPUTATIONAL COMPLEXITY OF THE BLOCK KJI-ALGORITHM

Let $n=qr$. The matrix A is decomposed in q^2 square blocks of size r. The well known sequential block Gauss algorithm is:

```
Do k = 1  to q
       Do  j = k+1 to q
              Do  i = k+1 to q
                     Aij = Aij - Aik*Akk^-1*Akj
```

Of course here, the inverse of the diagonal blocks A_{kk} are not directly computed. We solve r linear systems using pointwise Gaussian elimination.

The form KJI - SAXPY modified of the block generic Gaussian elimination algorithm is the following:

```
Do k = 1 to q
       Doall j = k+1 to q
              execute  TBkj  : < Akj = Akk^-1*Akj
                                 Do i = k+1 to q
                                        Aij = Aij - Aik*Akj >
       enddoall
```

In order to derive the computation cost and communication cost of TB_{kj} we must make some new assumptions. Since we are interested in asymptotic analysis and we want to increase the granularity of TB_{kj} we assume that r and q go to infinity with n (recall that rq=n).

The computation of $A_{kj} = A_{kk}^{-1} * A_{kj}$ is equivalent to solve r systems of size r with the same matrix. Neglecting low order terms it requires $8r^3/3$ arithmetic operations (see [21]). The computation of $A_{ij} = A_{ij} - A_{ik} * A_{kj}$ corresponds to (q-k) matrix products and matrix additions, hence leading to $2r^3(q-k)$ time units. Then performing TB_{kj} requires $2r^3[(q-k)+4/3]$ arithmetic operations. These computations require the knowledge of A_{kk}, A_{kj}, A_{ij} and A_{ik} for i=k+1 to q. Hence the communications costs are $2r^2[(q-k)+1]$. Comparing both costs we deduce that communication is an order less than computation and can be neglected leading to the following result:

Proposition 4 :

The comunications costs are negligible in the complexity of the block KJI-algorithm. If T_{BG} and T'_{BG} denote respectively this complexity without communication costs and including communication cost, then

$$T_{BG} = T_{BG}.$$

In order to compute T_{BG} we remark that the precedence constraints are the same as for the pointwise KJI-algorithm. Hence the preceding analysis applies and in particular theorem 1 can be stated in the same way, only taking into account the changes in the number and the costs of the tasks.

Theorem 5 :

The greedy algorithm G is asymptotically optimal. Its execution time is equal to

$$T_{BG} = (a/p) \cdot \sum_{k=1}^{q-p} (q-k).t_k \; + \; a \cdot \sum_{k=q-p+1}^{q-1} t_k$$

The number p of processors can be bounded by q since the number of tasks per level is less than q. Remark again that if p can be neglected with respect to q, the greedy algorithm can be executed with full efficiency which makes our analysis useless. More precisely, if we set $p=\alpha q$, with $\alpha<1$, we have :

Proposition 6 :

The complexity of the block KJI-algorithm is:

$$T_{BG} = 2n^3/(3p) + r^3p^2/3$$

With $p = \alpha q$ processors, $T_{BG} = (n^3/(3q)) [2/\alpha + \alpha^2]$

For fixed q, T_{BG} is minimum when $2/\alpha + \alpha^2$ is minimum, hence for $\alpha=1$. This is satisfactory: the computation time is minimal when the number of processors is equal to the size of the problem ($p=q$).

The asymptotic efficiency is equal to $E_{BG} = (n^3/3)/(pT_{BG}) = 1/(2+\alpha^3)$. Hence the efficiency increases from 1/2 to 1 when p decreases from q to 1. We now compare this result with the complexity of the pointwise KJI-algorithm.

Proposition 7 :

Let $p=\alpha q$ be given where α is a constant less than 1.

$$T_G = 2n^3/(3\alpha q)$$

$$T'_G = 2T_G = 4n^3/(3\alpha q)$$

$$T'_{BG} = T_{BG} = [n^3/(3q)] (2/\alpha + \alpha^2)$$

Hence:

$$T_G < T'_{BG} = T_{BG} < T'_G$$

For illustrating the preceding proposition, let us consider $r=q=p=n^{1/2}$. We obtain:

$$T_G = (2n^{5/2})/3 < T'_{BG} = T_{BG} = n^{5/2} < T'_G = (4n^{5/2})/3.$$

These results might appear a little surprising: without communication costs the pointwise algorithm is more efficient than the block algorithm and the reverse is true taking into account the communications. The explanation of the first inequality is the following. The greedy algorithm has full efficiency while the number of tasks per level is greater than the number of processors. Since the granularity of the pointwise algorithm is less than that of the block algorithm, the greedy algorithm keeps full efficiency longer. Conversely the bigger granularity of the block algorithm makes the communication costs negligible.

6. THE ADAPTATIVE KJI-ALGORITHM

In order to take full advantages of both algorithms, we must construct an algorithm which keeps efficiency one as long as the pointwise algorithm but preserves a level of granularity large enough to maintain low communications costs. This can be done by reducing gradually the blocks size, hence adapting the size of the blocks to the number of processors and the size of the remaining matrix. Let p and n be given and set $q=p+1$.

At the first step we set $s_1=n$ and $r_1=\lceil s_1/q \rceil$ where $\lceil . \rceil$ denotes the ceiling function. Then we apply the first step of the block-KJI algorithm to a matrix of size s_1 decomposed in q blocks: q-1 of size r_1 and the last of size $s_1-(q-1)r_1$. Remark that only one processor could have a task of length smaller than the others, which is negligible since p goes to infinity with n.

At the second step we set $s_2=s_1-r_1$ and $r_2=\lceil s_2/q \rceil$ and apply the second step of the block-KJI algorithm to a matrix of size s_2 decomposed in q blocks.

We repeat this process as long as the size s_k of the remaining matrix is greater than q. Once $s_k<q$, we use the pointwise KJI-algorithm. We obtain the adaptative KJI-algorithm, described as follows:

```
Serbegin processor = 1
        s = n ; r = ⌈s/q⌉ ; k = 1
Serend

While s ≥ q

    Serbegin processor = 1
            Decompose A in p blocks of size r and 1 block of size s-pr
    Serend

    Doall j = 2 to q
            execute TA_kj  : < A_kj = A_kk^-1*A_kj
                              Do i = 2 to q
                                  A_ij = A_ij - A_ik*A_kj >
    enddoall

    Serbegin processor = 1
            s = s - r ; r = ⌈s/q⌉ ; k = k + 1
            Reduce A to a matrix of size s by deleting
            the first rows and columns
    Serend

Do k = n-q-1 to n-1

    Doall j = k+1 to n
            execute T_kj  : < a_kj = a_kj/a_kk
                             Do i = k+1to n
                                 a_ij = a_ij - a_ik*a_kj >
    enddoall
```

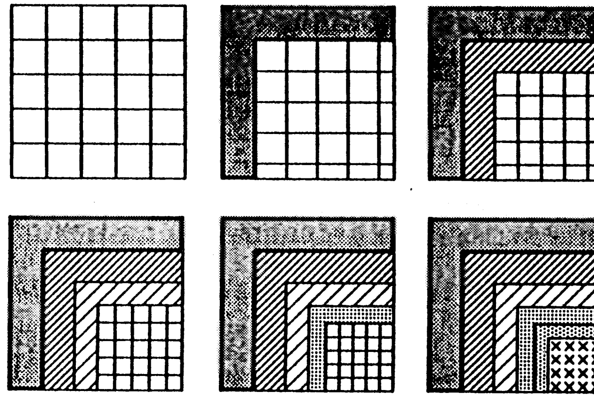


FIGURE 3
The first steps of the adaptative algorithm

7. EXPERIMENTS

We have computed the numbers of floating-point operations and memory accesses necessited by the following algorithms : KJI, bloc KJI and adaptative KJI, using the preceding formulaes.

The following tables show that if the communication costs are not neglected, the adaptative KJI algorithm is the best in the class studied in this paper. It has the efficiency of the pointwise algorithm, and it preserves a granularity big enough to maitain low communications costs, like the block-KJI algorithm.

7.1.Without communications.

p	r (for block KJI only)	Pointwise KJI	Block-KJI	Adaptative KJI
9	500	9.2	14.2	9.5
19	250	4.3	6.7	4.4
39	125	2.1	3.2	2.1
49	100	1.7	2.5	1.7
99	50	0.84	1.2	0.87
249	20	0.33	0.5	0.36
499	10	0.16	0.25	0.19

TABLE1
Execution times without communications of the three algorithms
for a matrix of size 5000x5000. The unit time is that of 10^9 floating point operations.

7.2. With communications

p	r (for block KJI only)	Pointwise KJI	Block-KJI	Adaptative KJI
9	500	18.5	14.2	9.6
19	250	8.7	6.8	4.5
39	125	4.2	3.3	2.2
49	100	3.4	2.6	1.8
99	50	1.6	1.3	0.89
249	20	0.67	0.52	0.38
499	10	0.33	0.27	0.22

TABLE 2
Execution times with communications of the three algorithms
for a matrix of size 5000x5000. The unit time is that of 10^9 floating point operations.

REFERENCES

[1] Cosnard, M., Marrakchi, M., Robert, Y., Trystram, D., Parallel Gaussian elimination on an MIMD computer, Report GEM-12, Ecole Centrale Paris, (1986)

[2] Dongarra, J.J., Gustavson, F.G., Karp, A., Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, SIAM Review 26, 1, (1984), pp. 91-112

[3] Kumar, S.P., Parallel algorithms for solving linear equations on MIMD computers, PhD. Thesis, Washington State University, (1982)

[4] Lord, R.E., Kowalik, J.S., Kumar, S.P., Solving linear algebraic equations on an MIMD computer, J. ACM 30, 1, (1983), pp. 103-117

[5] Kumar, S.P., Kowalik, J.S., Parallel factorization of a positive definite matrix on an MIMD computer, Proc. ICCD 84, pp. 410-416

[6] Gentleman, W.M., Kung, H.T., Matrix triangularisation by systolic arrays, Proc. SPIE, 298, Real time signal processing 4, (1981)

[7] Ahmed, H.M., Delosme, J.M., Morf, M., Highly concurrent computing structures for matrix arithmetic and signal processing, IEEE Computer 15,1, (1982), pp. 65-82

[8] Robert, Y., Block LU decomposition of a band matrix on a systolic array, Int. J. Computer Math. 17, 3, (1985), pp. 295-315

[9] Heller, D., A survey of parallel algorithms in numerical linear algebra, SIAM Review 20, (1978), pp. 740-777

[10] Sameh, A., An overview of parallel algorithms, Bulletin EDF, (1983), pp.129-134

[11] Ortega, J.M., Voigt, R.G., Solution of partial differential equations on vector and parallel computers, Siam Review 27, 2, (1985), pp. 149-240

[12] Saad, Y., Communication complexity of the Gaussian elimination algorithm on multiprocessors, Report DCS/348, Yale University, (1985), to appear in L.A.A. special issue on parallel computation.

[13] Ipsen, I.C.F., Saad, Y.,Schultz, M.H., Complexity of dense linear system solution on a multiprocessor ring, Report DCS-349 Yale University, (1985)

[14] Gajski,D.D., Peir, J.K., Essential issues in multiprocessors systems, IEEE Computer, (1985), pp. 9-27

[15] Hockney, R.W., C.R. Jesshope, C.R., Parallel computers: architectures, programming and algorithms, Adam Helger, Bristol, (1981)

[16] Hwang, K., Briggs, F., Parallel processing and computer architecture, MC Graw Hill, (1984)

[17] Schendel, U., Introduction to Numerical Methods for Parallel Computers, Ellis Horwood Series, J. Wiley & Sons, New York, 1984

[18] Stone, J.M., Norton, V.A., Norton, F.D., Rogers, F.D., Melton, E.A., Pfister, G.F., The VM/EPEX FORTRAN preprocessor reference, IBM Report, Yorktown Heights, New York (1985)

[19] Cosnard, M., Robert, Y., Trystram, D., Résolution parallèle de systèmes linéaires denses par diagonalisation, Bulletin EDF, to appear

[20] Cosnard, M., Robert, Y., Trystram, D., Comparaison des méthodes parallèles de diagonalisation pour la résolution de systèmes linéaires denses, C. R. Acad. Sc. Paris 301, I, 16, (1985), pp. 781-784

[21] Golub, G.H., Van Loan, C.F., Matrix computation, The Johns Hopkins University Press, (1983)

Computing 39, 187 – 199 (1987)

Computing
© by Springer-Verlag 1987

An Orthogonal Systolic Array for the Algebraic Path Problem

Y. Robert, Grenoble, and **D. Trystram**, Chatenay Malabry

Received August 2, 1985

Abstract — Zusammenfassung

An Orthogonal Systolic Array for the Algebraic Path Problem. This paper is devoted to the design of an orthogonal systolic array of $n(n+1)$ elementary processors which can solve any instance of the Algebraic Path Problem within only $5n-2$ time steps, and is compared with the $7n-2$ time steps of the hexagonal systolic array of Rote [8].

AMS Subject Classifications: 68A05 (05C35, 05C38, 16A78, 65F05, 68E10).

CR Categories and Subjet Descriptors: C.1.2 [processor architectures]: multiple data stream architectures (multiprocessors) – systolic arrays; G.1.0 [numerical analysis]: general-parallel algorithms; G.1.3 [numerical analysis]: numerical linear algebra-matrix inversion; G.2.2 [discrete mathematics]: graph theory-path problems; B.6.1 [logic design]: design styles-cellular arrays; B.7.1 [integrated circuits]: types and design styles-algorithms implemented in hardware; VLSI (very large scale integration).

General terms: Algorithms, design, performance.

Ein orthogonales systolisches Feld für das algebraische Wegproblem. Es wird ein orthogonales systolisches Feld (systolic array) mit $n(n+1)$ einfachen Prozessoren entworfen, das das algebraische Wegproblem in nur $5n-2$ Schritten lösen kann, im Vergleich zu $7n-2$ Schritten beim hexagonalen systolischen Feld von Rote [8].

1. Introduction

In a recent paper [8], Rote introduces the Gauss-Jordan elimination algorithm for the Algebraic Path Problem (APP for short), a general framework which unifies several algorithms arising from various fields of computer science, such as the determination of the inverse of a real matrix, the shortest distances in a weighted graph, and the transitive and reflexive closure of a binary relation. Rote [8] presents a hexagonally connected systolic array of $(n+1)^2$ processors which can solve any instance of size n of the APP in $7n-2$ time steps, each time being the time necessary to achieve a multiply-and-add in the underlying algebra.

The main result of this paper is the design of an orthogonally connected array of $n(n+1)$ processors which solves the same problem within only $5n-2$ time steps.

Section 2 briefly reviews the APP algorithm, as introduced by Rote [8]. The new systolic array is presented in Section 3, and its performance is compared with Rote's array and others from the literature in Section 4.

Using massive parallelism and pipelining, the systolic array concept allows a system implementor to design extremely efficient machines for specific computations. The reader not familiar with systolic arrays is referred to Kung [4]. All definitions from algebra or graph theory are consistent with Rote [8].

2. The Algebraic Path Problem

2.1. The Algorithm

The *Algebraic Path Problem* is defined as follows [10]: given a weighted graph $G=(V,E,w)$ where V is a finite vertex set, E an arc set, a function $w: E \to H$ with weights from a semiring $(H, \oplus, \otimes)$ with zero $\mathbb{O}$ and unity $\mathbb{1}$, find for all pairs of vertices (i,j) the quantities

$$d_{ij} = \bigoplus_{p \in M_{ij}} w(p),$$

where M_{ij} denotes the set of all paths from i to j.

Let us briefly recall that a semiring $(H, \oplus, \otimes)$ with zero $\mathbb{O}$ and unity $\mathbb{1}$ is an algebraic structure with two binary operations, fulfilling the following four axioms:

- $(H, \oplus)$ is a commutative semi-group with neutral element $\mathbb{O}$,
- $(H, \otimes)$ is a semi-group with neutral element $\mathbb{1}$,
- $\otimes$ is distributive over $\oplus$,
- $\mathbb{O}$ is absorptive with respect to $\otimes$.

With the weighted graph (V,E,w) we associate as in [8] the n by n weight matrix $A=(a_{ij})$, where $a_{ij}=w(i,j)$ if $(i,j) \in E$ and $a_{ij}=0$ otherwise. We denote by $M_{ij}^{(k)}$ the set of all paths from i to j which contain only vertices x with $1 \le x \le k$ as intermediate vertices. In practice,

$$a_{ij}^{(k)} = \bigoplus_{p \in M_{ij}^{(k)}} w(p),$$

is equal to the successive values of a_{ij} which we want to compute, starting from the initial value $a_{ij}^{(0)} = a_{ij}$ up to $a_{ij}^{(n)} = d_{ij}$.

In the computational procedure we have to compute infinite sums of the following kind, for which we use the special notation introduced by Rote [8]:

$$c^* := \bigoplus_{l \ge 0} c^l = \mathbb{1} \oplus c \oplus (c \otimes c) \oplus (c \otimes c \otimes c) \oplus \ldots$$

Rote's algorithm [8] for solving any particular instance of the APP using only the semiring operations $\oplus$, $\otimes$ and $*$ is detailed below:

{phase 1: initialization of the network and
computation of the $a_{ij}^{(j)}$ if $i>j$ and $a_{ij}^{(i-1)}$ if $i \le j$}
for $i := 1$ **to** n
for $j := 1$ **to** n
begin
  for $k := 1$ **to** $\min(i,j)-1$
    $a_{ij}^{(k)} := a_{ij}^{(k-1)} \oplus a_{ik}^{(k)} \otimes a_{kj}^{(k-1)}$;

```
    if i=j then a_ii^(i) := (a_ii^(i-1))*;
    if i>j then a_ij^(j) := a_ij^(j-1) ⊗ a_jj^(j);
end;

{phase 2: computation of the a_ij^(k)
such that k=j if i≤j and k=i-1 if i>j}
for i:=1 to n
for j:=1 to n
begin
    if i<j then a_ij^(i) := a_ii^(i) ⊗ a_ij^(i-1);
    for k:=min(i,j)+1 to max(i,j)-1
            a_ij^(k) := a_ij^(k-1) ⊕ a_ik^(k) ⊗ a_kj^(k-1);
    if i<j then a_ij^(j) := a_ij^(j-1) ⊗ a_jj^(j);
end;

{phase 3: computation of the a_ij^(n)}
for i:=1 to n
for j:=1 to n
begin
    if i>j then a_ij^(i) := a_ii^(i) ⊗ a_ij^(i-1);
    for k:=max(i,j)+1 to n
            a_ij^(k) := a_ij^(k-1) ⊕ a_ik^(k) ⊗ a_kj^(k-1);
end;
```

Applications of the APP are obtained by specializing the operations ⊕, ⊗ and * in the appropriate semirings. Rote [8] describes three of them in detail: the determination of the inverse of a real matrix, the shortest distances in a weighted graph, and the transitive and reflexive closure of a binary relation:

(i) Determination of the inverse of a real matrix: A is a real matrix, ⊕ and ⊗ are the usual operations in $\mathbb{R}$, and the *-operation is defined by:

$$\textbf{if } c \neq 1 \textbf{ then } c^* := 1/(1-c).$$

In this case the algorithm computes $(I-A)^{-1}$. As pointed out in [8], straightforward modifications permit to compute directly A^{-1}.

(ii) Shortest distances in a weighted graph: the weights a_{ij} are taken in $H=\mathbb{R}\cup\{-\infty,+\infty\}$, ⊕ is the addition in $\mathbb{R}$ extended to H (with $-\infty \oplus +\infty = +\infty$), ⊗ is the minimum, and the *-operation is defined by

$$\textbf{if } c \geq 0 \textbf{ then } c^* := 0 \textbf{ else } c^* := -\infty.$$

(iii) Transitive and reflexive closure of a binary relation: the a_{ij} are boolean, ⊕ et ⊗ are respectively the "and" and "or" operations, and the *-operation is defined by

$$c^* := \text{true for all } c.$$

We point out that we do *not* assume the multiplication in the semiring to be commutative: this permits to easily implement blocks schemes (the APP is said to be decomposable).

We shall concentrate on the determination of the inverse of a real matrix, and we introduce some notations for the general APP problem which are derived from its interpretation in numerical linear algebra.

2.2. Matrix Inversion

We already pointed out that we have a very important application of the APP algorithm by letting $\oplus$ and $\otimes$ be the usual operations on real numbers, and defining the $*$-operation as follows:

$$\textbf{if } c \neq 1 \textbf{ then } c^* := 1/(1-c).$$

The APP algorithm will then compute the inverse $(I-A)^{-1}$ of the matrix $(I-A)$. In fact, the APP algorithm corresponds in this case to the well-known Gauss-Jordan elimination algorithm. Even more, we have a natural interpretation of the three phases of the algorithm: let us denote

- $L=(l_{ij})$ the strictly lower triangular matrix with $l_{ij}=a_{ij}^{(j)}$ if $i>j$,
- $U=(u_{ij})$ the strictly upper triangular matrix with $u_{ij}=a_{ij}^{(i-1)}$ if $i<j$,
- $D=(d_{ij})$ the diagonal matrix with $d_{ij}=a_{ii}^{(i)}$ if $i=j$.

We have the Gaussian decomposition

$$I-A=(I-L)\cdot(D^{-1}-U)=(I-L)\cdot D^{-1}\cdot(I-DU)$$

so that

$$(I-A)^{-1}=(D^{-1}-U)^{-1}\cdot(I-L)^{-1}=(I-DU)^{-1}\cdot D\cdot(I-L)^{-1}.$$

Phase 1 of the algorithm computes L, U and D, whereas Phase 2 computes the two inverses

$$(I-L)^{-1}=\begin{cases} a_{ij}^{(j-1)} & \text{if } i>j \\ 1 & \text{if } i=j \\ 0 & \text{if } i<j \end{cases} \tag{i}$$

and

$$(D^{-1}-U)^{-1}=(I-DU)^{-1}\cdot D=\begin{cases} a_{ij}^{(j)} & \text{if } i\leq j \\ 0 & \text{if } i>j \end{cases} \tag{ii}$$

Finally, Phase 3 computes the product

$$(D^{-1}-U)^{-1}\cdot(I-L)^{-1}=(I-A)^{-1}$$

Rote [8] shows how very simple cosmetic transformations permit to compute A^{-1} rather than $(I-A)^{-1}$. Also, these transformations could be made in a straightforward manner for the systolic array that we propose. An other way to proceed is to input directly the matrix $(I-A)$ to the array to get A^{-1} without modifications in the algorithm. We shall adopt this solution in our design, since it has the advantage to respect the general formalism of the APP.

Notation: Letting $\mathscr{L}=I-L$ and $\mathscr{V}=D-U$, we denote $\mathscr{L}^{-1}$ and $\mathscr{W}$ the matrices respectively defined by the relations (i) and (ii). We adopt the notation $\mathscr{L}^{-1}$ for the general instance of the APP, although it is only significant for the particular case of matrix inversion (where $(I-A)^{-1}=\mathscr{W}\cdot\mathscr{L}^{-1}$).

3. The Systolic Array

3.1. Formal Description

We use a two-dimensional array of orthogonally connected processors (see Fig. 1). The array is composed of n rows, each row k including $n+1$ processors numbered from left to right $P_{k1}, \ldots, P_{k,n+1}$.

The matrix A, followed by I, the identity matrix of order n, is fed into the array row by row. More specifically, row k of the n by $2n$ matrix (A, I) is input to processor P_{1k}, one new element each time-step, beginning at time $t=k$. This input format is depicted in the Fig. 1.

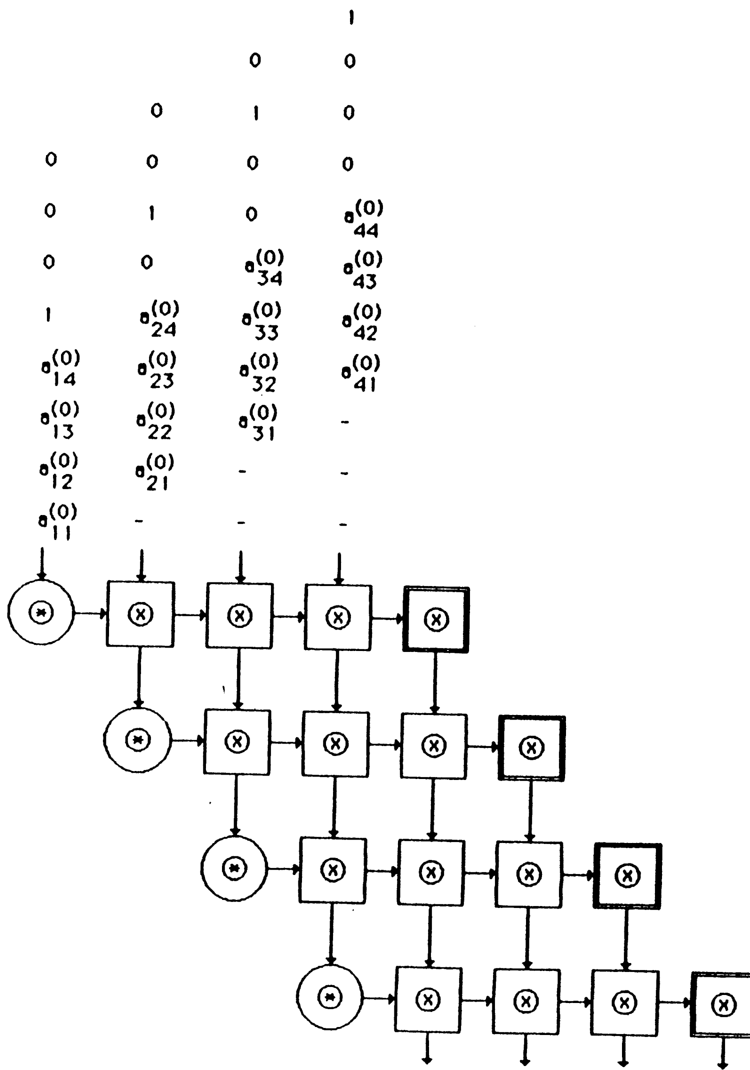
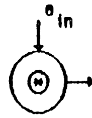
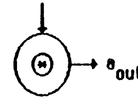


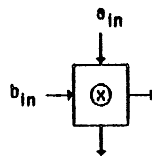
Fig. 1. Input of the systolic array

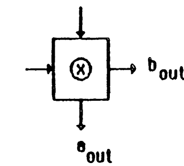




Step t   Step t+1

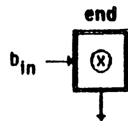
if init = true
then { perform *-operation }
    begin $a_{out} := a_{in}^*$;
        init := false ;
    end
else { transfer data }
    $a_{out} := a_{in}$

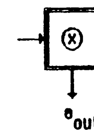




Step t   Step t+1

if init = true
then { initialize current register }
    begin $r := a_{in} \otimes b_{in}$;
        init := false ;
        $b_{out} := b_{in}$
        a_{out} := nil { no data is sent downwards }
    end
else { update a_{in} }
    begin $a_{out} := a_{in} \oplus r \otimes b_{in}$;
        $b_{out} := b_{in}$
    end





Step t   Step t+1

if init = true
then { initialize current register }
    begin $r := b_{in}$;
        init := false ;
        a_{out} := nil { no data is sent downwards }
    end
else { update b_{in} }
    $a_{out} := r \otimes b_{in}$

Fig. 2. Operation of the processors

The operation of each processor is detailed in the Fig. 2. There are three types of processors:

- Type 1: Circle processors perform the *-operation on their first input data. Afterwards they simply act as delay cells.
- Type 2: Square processors first initialize their current register by storing after modification their first input data; then they act as multiply-and-add cells (when $\oplus$ and $\otimes$ are the standard operations on real numbers, they become classical Inner-Product-Step cells [4], [6], [7]).
- Type 3: Double-square processors actually operate as square processors, with the exception that their current register is not initialized in the same way (remember that $\otimes$ is not commutative).

In the k-th row of the array, the leftmost processor $P_{k,1}$ is of Type 1 and the rightmost processor $P_{k,n+1}$ is of Type 3. All the other processors $P_{k,2}, \ldots, P_{k,n}$ are of Type 2.

The operation of a given processor in the array depends on whether it is the first data item it receives. As shown in the Fig. 2, there is a control bit named "init" (initialized to true) inside each processor which specifies the operation to be performed and the line along which the data is to be sent out.

We conceptually divide the array into two triangular sub-blocks, namely the left block and the right block, as illustrated in the Fig. 3.

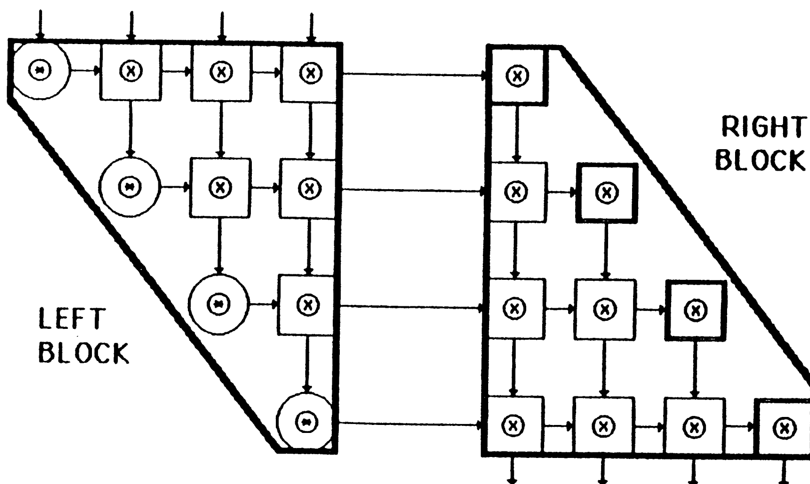


Fig. 3. Conceptual partition of the array into two sub-blocks

3.2. Operation of the Array

To explain the mapping of the APP algorithm on the systolic architecture, we describe sequentially the execution of each of the three phases of the algorithm, although they would be pipelined in an actual execution. This sequential description is used in the Fig. 4, which offers a synthetic view of the whole execution of the APP algorithm.

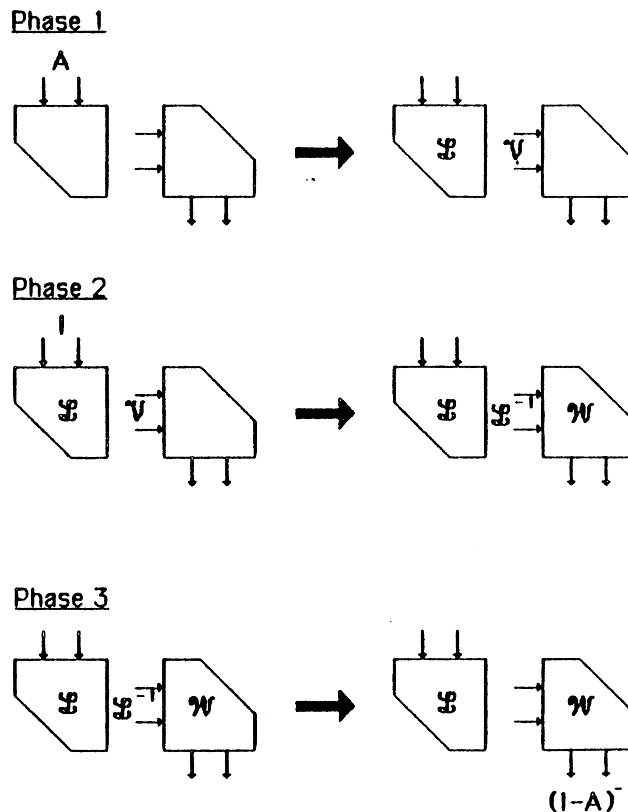


Fig. 4. Symbolic execution of the algorithm

Phase 1

We may consider here that the matrix A is input to the left triangular block of the array: we do not deal neither with the identity matrix that follows A, nor with the right part of the array. Moreover, the output of the left block after Phase 1 will be taken to be the input of the right block for Phase 2.

In Phase 1, the array acts essentially as the triangularization array of Ahmed et al. [1]. At Step 1, processor P_{11} computes $a_{11}^{(1)}$ and then operates as a one-step delay cell. At Step 2, P_{12} has his current register initialized with $a_{21}^{(1)}$ and then operates as a multiply-and-add cell: indeed at Step 3, it updates $a_{22}^{(0)}$ into

$$a_{22}^{(1)} := a_{22}^{(0)} \oplus a_{21}^{(1)} \otimes a_{12}^{(0)},$$

and so on with $a_{23}, a_{24}, \ldots$

The following table shows the operation of the first row of processors during Phase 1 (we choose $n=4$ in this example):

time-step	P_{11}	P_{12}	P_{13}	P_{14}
1	$a_{11}^{(1)}$	–	–	–
2	–	$a_{21}^{(1)}$	–	–
3	–	$a_{22}^{(1)}$	$a_{31}^{(1)}$	–
4	–	$a_{23}^{(1)}$	$a_{32}^{(1)}$	$a_{41}^{(1)}$
5	–	$a_{24}^{(1)}$	$a_{33}^{(1)}$	$a_{42}^{(1)}$

Fig. 5 shows the content of the registers of the left block processors as well as the output of the array after the execution of the first phase.

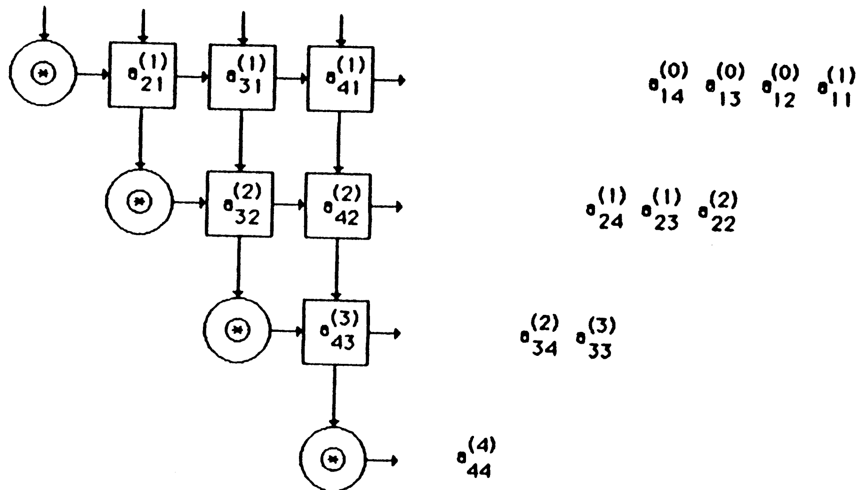


Fig. 5. Phase 1 of the algorithm

Phase 2

The left block processors are now initialized and operate as multiply-and-add cells, transforming the identity matrix into $\mathscr{L}^{-1}$. Indeed, the array realizes a linear transformation [2]. Since it transforms A into $\mathscr{V}$, it applies the transformation $\mathscr{L}^{-1}$ to any matrix. If the identity is fed in the initialized array, the output is $\mathscr{L}^{-1}$.

In parallel, the right block of the array operates in a similar way as the left one during the previous phase. Firstly the right block processors are initialized, then they operate as multiply-and-add cells. See the Fig. 6 for a global look of the array after the execution of Phase 2.

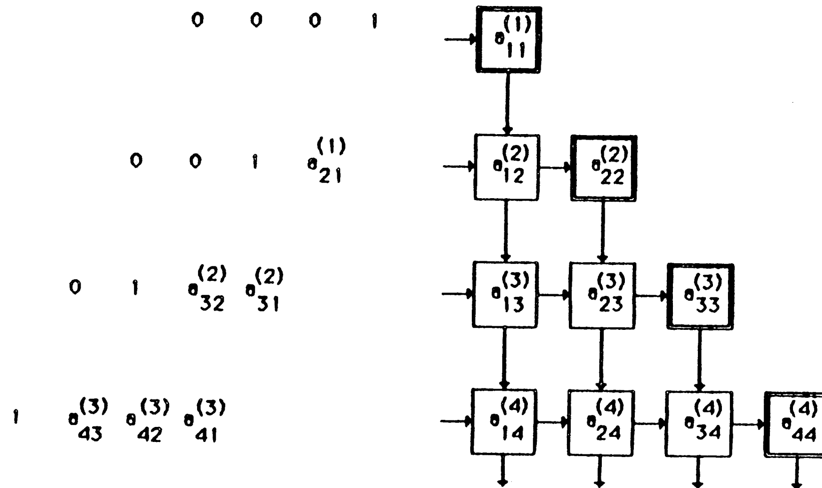


Fig. 6. Phase 2 of the algorithm

Phase 3

Left processors are idle. Right block processors continue to operate as multiply-and-add cells, and they compute the product $\mathscr{W} \cdot \mathscr{L}^{-1}$ whose last element is output at time $5n-2$.

In an actual execution, these three phases are pipelined, according to the following schedule:

- Phase 1 begins at time 1 and ends at time $3n-2$.
- Phase 2 begins at time $n+1$ and ends at time $4n-2$.
- Phase 3 begins at time $2n+1$ and ends at time $5n-2$.

Hence we can state the following theorem:

Theorem: *The orthogonal systolic array of $n(n+1)$ processors can solve any instance of the APP in $5n-2$ time-steps, each step being the time necessary to achieve a multiply-and-add in the underlying algebra.*

Sketch of proof: The formal proof of the validity of the array will result of the correctness of the data movement during each of the three execution phases described above. To demonstrate this correctness, we assume that the execution begins at time 1 (when processor P_{11} computes $a_{11}^{(1)}$) and we show analytically, using induction on k, that the following five assertions hold:

Phase 1 (see the Fig. 5):

(Assertion 1) The element $a_{k+j,k}^{(k)}$ is stored in processor $P_{k,j+1}$ at time $2k+j-1$, for $1 \le j \le n-k$ and $1 \le k \le n-1$.

(Assertion 2) For $1 \le k \le n$, the processor $P_{k,n-k+1}$ outputs successively

- $a_{kk}^{(k)}$ at time $2k+n-2$,
- $a_{k,k+j}^{(k-1)}$ at time $2k+n+j-2$, for $j=1,2,\ldots,n-k$ if $k \le n-1$.

Phase 2 (see the Fig. 6):

(Assertion 3) The element $a^{(k)}_{k+j-n,k}$ is stored in processor $P_{k,j+1}$ at time $2k+j-1$, for $n-k+1 \leq j \leq n$ and $1 \leq k \leq n$.

(Assertion 4) For $1 \leq k \leq n$, the processor $P_{k,n-k+1}$ outputs successively
- $a^{(k-1)}_{kj}$ at time $k+2n+j-2$, for $j=1,2,\ldots,k-1$, if $k \geq 2$,
- 1 at time $2k+2n-2$,
- $0 (n-k)$ times, starting at time $2k+2n-1$ up to time $k+3n-2$, if $k \leq n-1$.

Phase 3 (see the Fig. 7):

(Assertion 5) For $1 \leq k \leq n$, the processor $P_{n,k+1}$ outputs the element $a^{(n)}_{kj}$ at time $3n+k+j-2$, for $j=1,2,\ldots,n$.

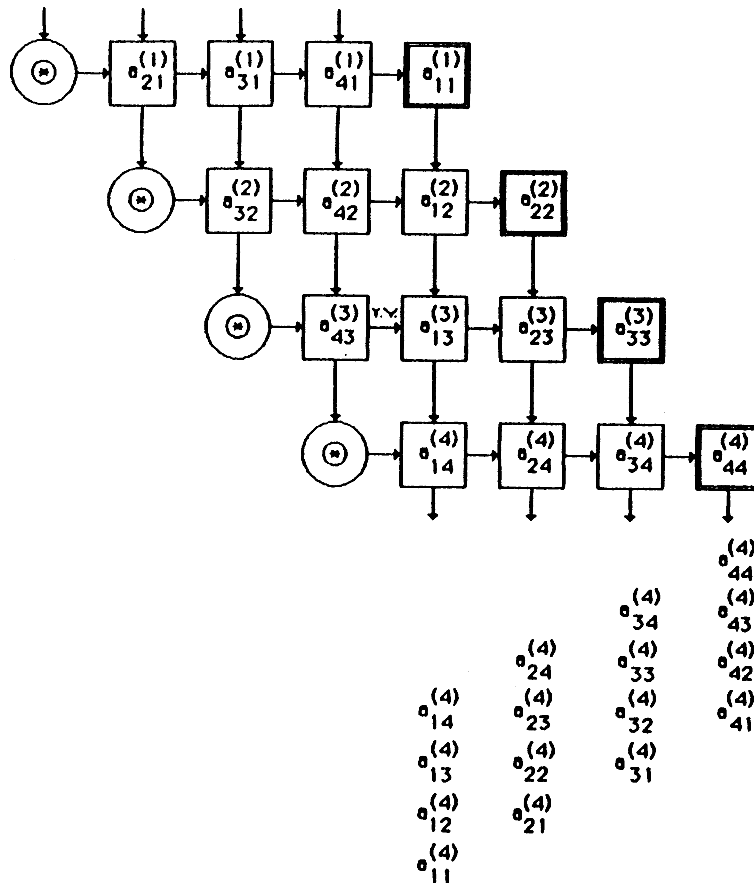


Fig. 7. Phase 3 of the algorithm

4. Conclusion

We have introduced a new systolic array involving the same number of processors as that of Rote [8] which solves the APP within $5n-2$ time-steps, instead of $7n-2$. As pointed out in [8], this value is optimal within one step, provided that each of the n^2 data elements is used only once in each step. Moreover, the solution of a new instance of the APP can begin every $2n$ steps. Also, in the case of matrix inversion, if we input the matrix A followed by a n by n matrix B instead of I, it can be seen that the array delivers $(I-A)^{-1}B$ (or $A^{-1}B$ with the straightforward modifications) within the same number of steps.

Rote [8] mentions a unidirectional version (called GJE-UNI in [9]), which solves the APP in $5n-2$ steps too. However, this array requires $5n^2$ processors (instead of n^2), and its drawbacks are that the data flow is not as regular, and processors actions are not as uniform. On the contrary, the data moves in a regular and systematic way in our array. The importance of designing arrays without feedback cycles is emphasized in [5]: acyclic implementations usually exhibit more favorable characteristics with respect to fault-tolerance, two-level pipelining, and problem decomposition in general.

Finally, we point out that the array can be micro-programmed to solve various instances of tne APP by selecting the appropriate semiring operations, contrarily to the array of Guibas et al. [3] which can only compute the APP in semirings with idempotent addition.

Acknowledgements

We thank the two referees for their comments and suggestions which greatly improve the presentation of this paper.

References

[1] Ahmed, H. M., Delosme, J. M., Morf, M.: Highly concurrent computing structures for matrix arithmetic and signal processing. Computer *15/1*, 65–82 (1982).

[2] Delosme, J. M.: Algorithms for finite shift-rank processes. Ph. D. Thesis, Technical Report M 735-22, Sept. 1982, Stanford Electronics Laboratories.

[3] Guibas, L. J., Kung, H. T., Thompson, C. D.: Direct VLSI implementation of combinatorial algorithms, pp. 509–525. Proc. Caltech Conf. on VLSI: Architecture, design, fabrication, Calfornia Institute of technology, Pasadena (1979).

[4] Kung, H. T.: Why systolic architectures. Computer *15/1*, 37–46 (1982).

[5] Kung, H. T., Lam, M. S.: Fault-tolerance and two-level pipelining in VLSI systolic arrays. Journal of Parallel and Distributed Computing *1/1*, 32–63 (1984).

[6] Kung, H. T., Leiserson, C. E.: Systolic Arrays (for VLSI). In: Proc. of the Symposium on Sparse Matrices Computations, pp. 256–282 (Duff, I. S., Stewart, G. W., eds.). Knoxville, Tenn. (1978).

[7] Robert, Y.: Block LU decomposition of a band matrix on a systolic array. Int. J. Computer Math. *17*, 295–315 (1985).

[8] Rote, G.: A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). Computing *34*, 191–219 (1985).

[9] Rote, G.: A systolic array for the algebraic path problem (which includes the inverse of a matrix and the shortest distances in a graph). Rechenzentrum Graz, Austria, Bericht RZG-101 (1984).
[10] Zimmermann, U.: Linear and combinatorial optimization in ordered algebraic structures. Ann. Discrete Math. *10*, 1 – 380 (1981).

Yves Robert
CNRS, Laboratoire TIM 3
Grenoble Université, BP 68
F-38402 Saint Martin d'Hères, Cedex
France

Denis Trystram
Ecole Centrale Paris
F-92295 Chatenay Malabry, Cedex
France

Printed in Austria
Druck: Paul Gerin, A-1021 Wien

Architecture systolique de systèmes adaptatifs

Systolic implementation of adaptive systems

Pierre COMON

Laboratoire CEPHAG : Centre d'Études des Phénomènes Aléatoires et Géophysique de Grenoble, ENSIEG, BP n° 46, 38402 SAINT-MARTIN D'HERES Cédex.
Laboratory Information Systems, Department Electrical Engineering, Stanford University, STANFORD, CA 94305, USA.

Pierre Comon est né à Paris en 1959. Il a reçu le diplôme d'ingénieur de l'ENSIEG en 1982 et soutenu une Thèse de Doctorat en 1985. De 1982 à 1985, il a travaillé dans le département Aérospatial de la société Crouzet (Valence). Depuis 1981, il appartient au Laboratoire CEPHAG. Il est actuellement détaché au Information Systems Laboratory de l'université de Stanford, USA. Ces centres d'intérêt regroupent le filtrage adaptatif multivariable, l'élimination de bruit, l'analyse haute résolution et les réseaux systoliques.

Yves ROBERT

CNRS, Laboratoire TIM3 : Techniques pour l'Informatique, les Mathématiques, la Microélectronique et la Microscopie quantitative, Institut National Polytechnique de Grenoble, 38031 GRENOBLE Cédex.

Yves Robert, ancien élève de l'École Normale Supérieure de l'Enseignement Technique, agrégé de Mathématiques (1980), a obtenu une Thèse de troisième cycle en Mathématiques Appliquées à l'Université de Grenoble. Il a soutenu en 1986, une Thèse d'État sur l'algorithmique parallèle pour les réseaux d'automates, les architectures systoliques et les machines SIMD et MIMD. Ses recherches actuelles portent sur les algorithmes et architectures parallèles, systoliques et VLSI pour l'algèbre numérique linéaire et le traitement du signal.

Denis TRYSTRAM

Laboratoire CREDI (Création Réalisation Étude et Développement Informatiques), École Centrale de Paris, Grande-Voie-des-Vignes, 92295 CHÂTENAY-MALABRY Cédex.

Denis Trystram est ingénieur de l'ENSIMAG, il a soutenu une thèse de Docteur-Ingénieur en 1984 au laboratoire IMAG, sur l'algorithmique numérique des grands systèmes linéaires. Il est actuellement Assistant en Informatique à l'École Centrale de Paris où il dirige un groupe de travail sur le calcul parallèle.

RÉSUMÉ

Nous nous intéressons à la mise en œuvre sur des architectures VLSI hautement parallèles (réseaux systoliques) du calcul des projections intervenant dans le traitement des signaux numériques, et plus généralement dans les systèmes adaptatifs. Après avoir montré comment se pose ce genre de problème en traitement adaptatif du signal nous passons en revue diverses architectures systoliques (connues ou originales) qui permettent une résolution en temps réel.

MOTS CLÉS

Filtrage adaptatif, estimation de paramètres, projections, algorithmique numérique, réseaux systoliques, calcul parallèle.

SUMMARY

We are interested in the systolic computation of projection operators entering digital signal processing, or more generally adaptive systems. We first show how such computations arise when dealing with adaptive signal processing. Then we propose an overview of various systolic arrays (including original ones) achieving real time computations.

KEY WORDS

Adaptive filtering, parameter estimation, projections, numerical algorithms, systolic arrays, parallel computing.

Introduction

Le modèle systolique a été introduit en 1978 par Kung et Leiserson [20] pour répondre à la demande sans cesse croissante d'une plus grande puissance de calcul à moindre frais. Il s'est révélé être un outil très efficace pour la conception de processeurs intégrés spécialisés (citons pour référence [14, 19, 21, 22] parmi d'autres). En un mot, une architecture systolique est agencée en forme de réseau composé d'un grand nombre de cellules élémentaires identiques et localement interconnectées. Chaque cellule reçoit des données provenant 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, en principe sous le contrôle d'une horloge globale : plusieurs calculs sont effectués simultanément sur le réseau et on peut résoudre plusieurs instances du même problème à la suite.

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

Nous nous intéressons dans cet article à la réalisation systolique du calcul des projections intervenant dans le traitement des signaux numériques. Comme nous allons le montrer dans la première section, le problème général peut être posé de la manière suivante : Calculer successivement $A_i^{-1} B_i$ pour $i=1, 2, \ldots,$ où les matrices A_i et B_i, respectivement de tailles $n \times n$ et $n \times p$, sont fonction de A_{i-1} et B_{i-1}.

Nous proposons tout d'abord, dans la section 2, une étude générale pour calculer $A^{-1}B$ par décomposition en sous-problèmes matriciels élémentaires. Nous faisons un tour d'horizon des meilleurs réseaux existants qui conduisent au produit $A^{-1}B$ par combinaisons des réseaux fondamentaux, après quoi nous proposons un réseau basé sur la décomposition de Gauss-Jordan qui permet de calculer directement $A^{-1}B$.

Enfin, nous donnons dans la section 3 la description de deux réseaux systoliques qui permettent de tenir compte des éléments de A_{i-1} et B_{i-1} dans le calcul de $A_i^{-1} B_i$. Le premier est basé sur la formule de Woodbury et le second est un réseau original qui inclut la mise à jour des matrices.

1. Projections en traitement du signal

Un système adaptatif peut être défini par un opérateur optimal, construit à partir d'un modèle d'observation et d'un critère d'optimisation (voir *fig.* 1). Un grand nombre d'opérateurs optimaux sont constitués de projections, notamment lorsque le critère est quadratique et le modèle linéaire. Dans le cadre de la modélisation et du traitement de signaux numériques observés sur une durée finie, les projections prennent une forme simple. Nous allons développer dans cette section quelques aspects de l'utilisation des projections dans le domaine du traitement adaptatif du signal.

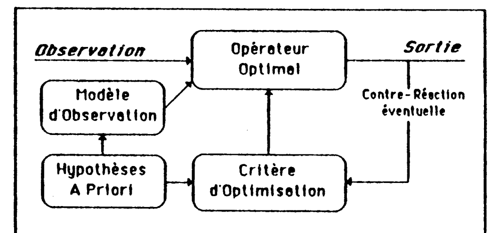


Fig. 1. — Schématisation d'un traitement adaptatif.

1.1. Projections

Considérons l'espace hilbertien L_p^2 des vecteurs déterministes de dimension p, muni du produit hermitien : $\langle x, y \rangle = x^t G^{-1} y$, où G est une matrice hermitienne $p \times p$ définie positive et où x^t désigne le vecteur transposé conjugué de x. Soit Y une matrice $p \times n$ formée de n vecteurs colonnes y_j. Cette matrice définit un sous-espace $L^2_{D(Y)}$ de dimension $D(Y) \leq p$ engendré par la famille $\{y_1, \ldots, y_n\}$. En particulier, si Y est de rang plein, $D(Y) = \inf(p, n)$. La projection d'un vecteur x de L_p^2 sur $L^2_{D(Y)}$ est caractérisée simplement par le vecteur de L_n^2 :

$$F_D(Y, x) = (Y^t G^{-1} Y)^{-1} Y^t G^{-1} x \tag{1}$$

où A^{-1} désigne l'inverse généralisée de A lorsque A est singulière [10]. En d'autres termes $F_D(Y, x)$ définit ici une application de L_p^2 dans L_n^2. Ce type d'expression est constamment rencontré; citons par exemple l'estimation par les moindres carrés généralisés de paramètres déterministes dans un modèle d'observation linéaire [28, 31] : coefficients de modèles AR [31, 3, 11], analyse spectrale paramétrique [12, 17], analyse haute résolution spatiale ou spectrale [12], ... Une

forme voisine est rencontrée si la projection est immergée dans L_p^2; le projecteur, que nous noterons $F_p(Y)$, est alors une application de L_p^2 dans lui-même :

$$(2) \quad F_p(Y)x = YF_D(Y, x) = Y(Y'G^{-1}Y)^{-1}Y'G^{-1}x$$

Remarquons qu'alors le projecteur sur l'espace orthogonal s'écrit :

$$(3) \quad F_p^0(Y) = Y[I-(Y'G^{-1}Y)^{-1}Y'G^{-1}]$$

Ces deux dernières expressions sont à la base de la théorie des filtres en treillis [17, 8].

Les projections prennent une forme similaire pour des signaux aléatoires. En effet, soit H_p^2 l'espace hilbertien des vecteurs aléatoires de dimension p, muni du produit hermitien : $\langle x, y \rangle = E\{x'G^{-1}y\}$. La projection d'un vecteur x de H_p^2 sur le sous-espace engendré par la matrice aléatoire Y, revêt la forme suivante :

$$(4) \quad F_p(Y)x = YF_D(Y, x) = YE\{Y'G^{-1}Y\}^{-1}E\{Y'G^{-1}x\}$$

$F_D(Y, x)$ est ici un vecteur déterministe de L_n^2 représentant les coefficients de x dans sa décomposition sur la famille génératrice $\{y_1, \ldots, y_n\}$. Par exemple, ce vecteur correspondra au filtre linéaire optimal en détection et estimation bayesiennes; citons pour mémoire la théorie du filtrage de Wiener et de Kalman [16, 32]. L'expression (4) n'est utilisable dans la pratique que si les deux moments d'ordre deux y intervenant sont connus, ce qui limite fortement ses possibilités d'application. Ainsi, les matrices de covariance intervenant dans (4) sont en pratique remplacées par des matrices estimées à partir d'un ensemble d'observations $\{Y^\mu, x^\mu, 1 \leq \mu \leq M\}$ possédant les mêmes caractéristiques au second ordre, si bien que le projecteur à calculer prend la forme adaptative :

$$(5) \quad F_D(Y, x) = \left[\sum_{\mu=1}^{M} a_\mu Y^{\mu\prime}G^{-1}Y^\mu\right]^{-1} \times \left[\sum_{\mu=1}^{M} b_\mu Y^{\mu\prime}G^{-1}x^\mu\right]$$

Nous constatons donc que le calcul de produits tels que $A^{-1}B$, où A et B sont des matrices $p \times p$ et $p \times n$ respectivement, est omniprésent. Suivant le problème abordé, ces deux matrices peuvent avoir diverses formes particulières imposées : hermitiennes, symétriques réelles, Toeplitz, proches de Toeplitz, circulantes, Hankel, etc., ces structures pouvant être agencées par blocs dans le cas de signaux multivariables.

1.2. Structure des matrices

Limitons notre champ d'investigation à l'observation de processus localement stationnaires. Dans la pratique, les observations sont toujours de durée finie, et sont modélisées afin de pouvoir utiliser l'hypothèse de stationnarité au second ordre (au moins localement).

Cette modélisation, parfois implicite, a un impact important sur la structure des matrices de covariance figurant dans la formulation des projecteurs. En réalité, de manière non exhaustive, on peut grossièrement distinguer trois classes de modélisation, conduisant à trois types de structure de matrices de covariance :

1. les observations sont des réalisations infinies tronquées par une fenêtre d'observation; plusieurs types de fenêtres sont couramment utilisés à cet effet [17, 11, 8]. Ce type de modèle conduit à des matrices de covariance toeplitz symétriques par blocs dans le cas stationnaire;

2. les processus étudiés sont périodiques en moyenne quadratique, et les observations sont considérées comme étant des réalisations d'exactement une période de longueur. Les matrices de covariance de telles observations sont alors toeplitz circulantes par blocs. C'est cette modélisation qui est implicitement adoptée lorsqu'on exploite la Transformée de Fourier Discrète (TFD) des observations pour déterminer un traitement optimal [4];

3. Les observations sont directement délivrées dans le domaine fréquence ou vecteur d'onde. Les échantillons successifs sont par conséquent à valeurs complexes et théoriquement indépendants [4, 26]. Ceci permet, même si cette propriété n'est vérifiée dans la pratique qu'asymptotiquement, d'estimer facilement les matrices de covariance à partir d'un moyennage spectral ou spatial (périodogramme lissé) et d'en connaître les propriétés statistiques [4]. Dans ce type de modèle d'observation, les matrices de covariance de chaque échantillon sont hermitiennes, et la matrice de covariance de l'ensemble de l'observation est théoriquement hermitienne bloc-diagonale.

1.3. Types de récurrence

Afin de compléter cette caractérisation de l'acquisition des données, il faut définir le mode de succession des échantillons, dont tiendra nécessairement compte un traitement adaptatif en aval. Comme cela a été déjà souligné dans la littérature [8], il existe deux grandes familles de traitements adaptatifs :

1. les traitements dits « par blocs », dans lesquels les données arrivent par groupes disjoints qui sont traités séparément. L'estimation escomptée est alors calculée lors de chaque acquisition d'un nouveau bloc. En général, les observations spectrales sont délivrées de cette manière. Ces blocs peuvent éventuellement s'intersecter (recouvrement) [33] et être apodisés de fenêtres [4];

2. parallèlement, dans les traitements « récursifs », la sortie est recalculée à chaque acquisition d'un nouvel échantillon en fonction de la sortie calculée à l'instant précédent. Bien que ce soit malgré tout possible, les traitements récursifs ne fonctionnent habituellement pas conformément à une division par blocs.

Les classifications que nous venons de faire ci-dessus nous permettent maintenant de considérer les divers problèmes susceptibles d'être rencontrés :

1. Pour un traitement de type « par blocs », nous devrons calculer un nouveau projecteur à chaque acquisition d'un nouveau bloc de données. Si on se réfère à la pratique usuelle, ce type de traitement est souvent rencontré dans le domaine spectral. Des récurrences peuvent être construites en fonction du bloc précédent ou à l'intérieur même du bloc :

a. solution (bloc k) $= f\{$solution (bloc $k-1$)$\}$;

b. solution (bloc k, échantillon j)
$= f\{$solution (bloc k, échantillon $j-1$)$\}$.

Les matrices de covariance dans le premier cas (a) sont remises à jour par une modification de rang 1 :

$$\text{(6)} \qquad \begin{cases} B(k) = (1-\alpha_k) B(k-1) + \alpha_k u_k v_k^t; \\ 0 < \alpha_k < 1 \end{cases}$$

alors que dans le second (b), aucune relation particulière n'existe *a priori* entre les valeurs successives des covariances :

$$\text{(7)} \qquad B(k, j) = B(k, j-1) + \varepsilon(k, j)$$

En revanche, sous réserve de certaines propriétés de continuité de $B(k, j)$ en fonction de j, l'innovation $\varepsilon(k, j)$ pourra être considérée de petite norme. Sinon, les covariances sont recalculées à chaque pas k à partir de moyennes :

$$\text{(8)} \quad B(k, j) = \sum_{m=1}^{M} \sum_{q=-Q}^{Q} b_{k-m, j-q} u_{k-m, j} v_{k-m, j-q}^t$$

2. Pour les traitements « récursifs » sans division par blocs, le calcul des covariances se limite à :

$$B(j) = \sum_{q=1}^{Q} b_{j-q} u_{j-q} v_{j-q}^t$$

et la récurrence s'écrit :

$$\text{(9)} \quad B(j) = f\{B(j-1)\} = (1-\alpha_j) B(j-1) + \alpha_j u_j v_j^t; \quad 0 < \alpha_j < 1$$

On peut retrouver les estimateurs classiques des matrices de covariance en jouant sur le coefficient α_k :
Si $\alpha_k = 1/k$ et $B(0) = 0$ dans les récurrences (6) ou (9), l'estimateur de la covariance $B(k)$ n'est autre que celui du périodogramme (ou corrélogramme) moyenné :

$$\text{(10)} \qquad B(j) = \sum_{q=1}^{j} u_q v_q^t / j$$

En revanche, si $\alpha_k = \alpha$, le moyennage est de type exponentiel :

$$\text{(11)} \qquad B(j) = \alpha \sum_{q=1}^{j} \alpha^{j-q} u_q v_q^t + (1-\alpha)^j B(0)$$

1.4. Position du problème

Finalement, les calculs qui devront être réalisés seront ceux des projecteurs conformément aux expressions (1) à (3), dans lesquelles les matrices de covariance seront soit recalculées à chaque pas, soit calculées récursivement, à partir des transformations (6) à (9).
Nous avons déjà présenté dans [5] une implémentation systolique performante de projections dans le cas d'un calcul non récursif, et de matrices de covariance hermitiennes, ou symétriques réelles, sans structure imposée (le principe est rappelé en section 2.2). De plus, nous allons nous intéresser dans la section 3 aux calculs de projections régis par la récurrence exponentielle (9) et (11) :

$$\text{(12)} \qquad \begin{cases} \{F_p(Y, x)\}_j = A_j^{-1} B_j \\ A_j = (1-\alpha) A_{j-1} + \alpha u_j u_j^t; \\ B_j = (1-\alpha) B_{j-1} + \alpha u_j v_j^t; \\ 0 < \alpha < 1 \end{cases}$$

où $u_j = Y_j^t G^{-1/2}$ et $v_j = x_j^t G^{-1/2}$ sont donnés.
Dans un premier temps, aucune structure particulière ne sera imposée aux matrices de covariance si ce n'est l'hermicité (ou la symétrie) et la non-négativité. Elles ont une dimension dont l'ordre varie de 10 sur 10 à 100 sur 100. Dans la pratique, on peut envisager les réseaux permettant le calcul récursif des projections dans le cas des structures imposées (structure toeplitz symétrique par exemple). On ne gagne rien à introduire un réseau spécialisé sur un type particulier de structure, les réseaux proposés ici dans le cas général restent en effet d'une grande efficacité dans le cas des structures imposées. Il existe cependant des travaux traitant ces cas particuliers, nous nous contentons ici de renvoyer à la littérature correspondante [6, 7].

2. Réalisation systolique de $A^{-1}B$

2.1. Réseaux usuels de l'algèbre matricielle

Récemment, de nombreux réseaux systoliques ont été proposés pour résoudre les opérations fondamentales de l'algèbre linéaire (citons parmi les principaux les travaux de Gentleman et Kung [9], Kung et Leiserson [20], Hwang et Cheng [15], Kramer et Van Leeuwen [18], Li et Wah [23], Nash *et al* [25], Preparata et Vuillemin [27], Robert et Tchuente [29] etc.), le même réseau pouvant servir à différentes applications. Ils permettent de réaliser le produit de deux matrices, la triangularisation d'une matrice (décompositions LU et QR), l'inversion de matrices triangulaires, la multiplication de matrices et même la résolution de systèmes linéaires.
En ce qui concerne le problème particulier du calcul de $A^{-1}B$, on se ramène aux sous-problèmes élémentaires suivants : décomposition LU, inversion de matrices triangulaires et multiplication matricielle.

Plus précisément, on décompose la matrice A en LU, on inverse les matrices triangulaires L et U puis on les multiplie de façon à obtenir $U^{-1}L^{-1}$. Il ne reste alors plus qu'à postmultiplier par B pour obtenir le résultat.

Nous nous proposons dans cette section de faire un rapide tour d'horizon des réseaux systoliques élémentaires les plus performants pour chacun de ces problèmes élémentaires.

Multiplication matricielle

On présente ici un réseau systolique bidimensionnel qui réalise le produit de deux matrices denses A et B de taille n [27]. Comme il s'agit du premier réseau que nous décrivons, nous détaillons particulièrement cette présentation.

Le nombre de pas pour effectuer le calcul en séquentiel est n^3, et nous avons en vue un temps d'exécution linéaire (de l'ordre de n) sur un réseau de $O(n^2)$ cellules. Un produit de deux matrices de tailles n par n n'est jamais que le résultat de n^2 produits scalaires à n termes. Ainsi, pour calculer un élément du produit C, disons c_{ij}, on doit réaliser le produit scalaire de la ligne i de A et de la colonne j de B.

Une façon naturelle d'effectuer ce produit scalaire est illustrée sur les figures 2 et 3 (pour $n=3$). Ici, le réseau comprend une seule cellule, dont le registre interne est modifié par accumulations successives jusqu'à acquérir la valeur définitive du produit scalaire. On désigne par IPS (pour Inner Product Step), la cellule qui réalise 1 pas du produit scalaire (multiplication suivie d'une addition).

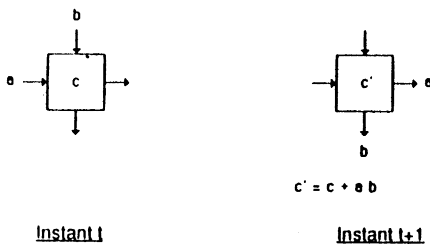


Fig. 2. – Cellule IPS.

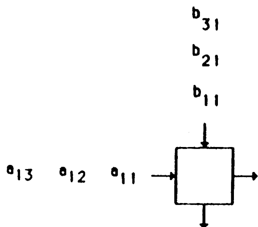


Fig. 3. – Calcul du produit scalaire.

Pour calculer l'élément c_{ij}, l'algorithme est donc le suivant :

$c_{ij} := 0;$
Pour $k := 1$ *à* n *faire*
$c_{ij} := c_{ij} + a_{ik} \cdot b_{kj};$

Pour calculer la matrice C complète, nous allons utiliser n^2 cellules, une par coefficient, comme le montre la figure 4 sur des matrices 3×3. Le réseau obtenu fonctionne en temps $3n-2$. On peut s'en convaincre facilement en suivant le parcours des derniers coefficients a_{nn} et b_{nn}.

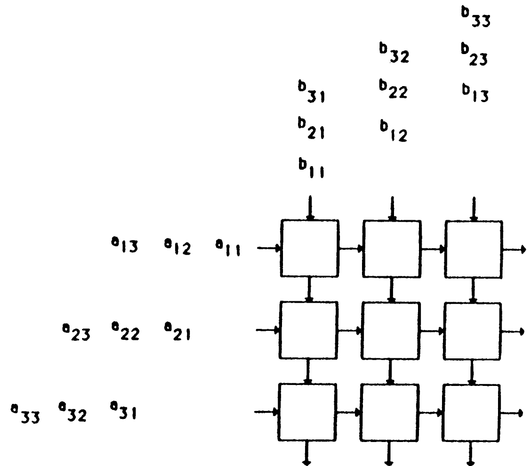


Fig. 4. – Produit de deux matrices denses.

Une difficulté cependant se pose. En effet, à la fin du calcul, les coefficients de la matrice C sont stockés dans le réseau. Il nous faut imaginer un mécanisme permettant de vider celui-ci. En effet, seules les cellules frontières peuvent communiquer avec l'hôte, il est donc exclu de lire directement et en parallèle (en un seul top d'horloge!) le contenu des registres internes des n^2 cellules. De même, l'instruction qui consiste à vider le réseau doit se propager de cellule en cellule, et ne peut être envoyée par un organe de contrôle central à toutes les cellules.

Plusieurs solutions peuvent être envisagées. Nous décrivons dans la suite un dispositif où le réseau se vide par la gauche, en utilisant le bus de circulation des coefficients a_{ij}. L'opération des cellules est maintenant commandée par un ou deux booléens de contrôle, qui circulent de la gauche vers la droite.

Informellement, le principe est le suivant : si la dernière accumulation dans une cellule K a lieu au temps t, au temps $t+1$ celle-ci envoie sur la droite le contenu c de son registre interne. Au temps $t+1$, sa voisine de droite K′ laisse passer inchangée la variable c en la transmettant à nouveau vers la cellule de droite K″. Au temps $t+2$, K′ envoie à son tour le contenu de son registre interne à K″. Le booléen contrôlant le vidage se propage donc à la vitesse 1/2, gagnant une nouvelle cellule tous les deux tops. Ce ralentissement est obtenu en insérant une bascule supplémentaire sur son parcours.

Nous pouvons maintenant décrire en détail ce que pourrait être le fonctionnement d'une cellule lors d'une mise en œuvre concrète. Tout d'abord, le contrôle des cellules doit coder les états suivants :

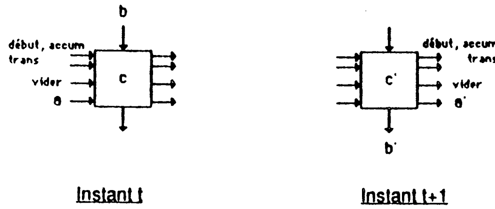
– *début :* commencer un nouveau calcul;

– *accum :* accumuler un pas de produit scalaire dans le registre interne (régime permanent);

– *trans :* transmettre inchangé un coefficient vers la droite (correspondant au registre interne d'une cellule plus à gauche);

– *vider :* transmettre le contenu du regitre interne vers la droite.

En fait, les trois variables *début*, *accum* et *trans* se propagent à la vitesse 1 et peuvent être codées sur deux bits, et un bit supplémentaire est nécessaire pour la variable *vider*, qui se propage à la vitesse 1/2. La figure suivante détaille le fonctionnement complet d'une cellule.



cas contrôle de
début : c' := a * b ; a' := a ; b' := b ;
accum : c' := c + a. b ; a' := a ; b' := b ;
vider : c' := nil ; a' := c ; b' := nil ;
trans : c' := c ; a' := a ; b' := nil ;

Fig. 5. – Contrôle des cellules du réseau.

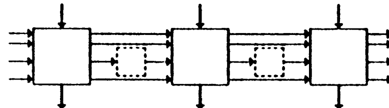


Fig. 6. – Circulation du booléen *vider*.

L'instruction *vider* doit arriver en même temps que l'instruction *trans*, avec priorité sur celle-ci, un top après le dernier coefficient significatif. Par exemple, la première cellule de la première ligne recevra cette instruction au top $n+1$.

Les figures 5 et 6 illustrent la communication entre deux cellules voisines d'une même ligne. On peut imaginer que la première cellule de chaque ligne transmette les instructions de contrôle qu'elle reçoit à sa voisine de dessous, puisque cette dernière effectue les mêmes opérations qu'elle avec un retard d'un top. Ceci permet de ne faire communiquer, en matière de contrôle, que la première cellule en haut à gauche avec l'extérieur, réduisant ainsi le nombre de ports d'entrée-sortie nécessaires.

Le premier coefficient c_{11} est délivré à la droite du réseau juste après a_{1n}, et un nouvel élément suit tous les tops. Cette phase de vidage requiert donc n tops supplémentaires, et le temps total d'exécution devient égal à $4n-2$.

Décomposition LU

De nombreux réseaux systoliques ont été proposés pour la triangularisation d'une matrice dense [1, 9]... Gentleman et Kung proposent un réseau orthogonal [9] qui permet d'effectuer la décomposition LU d'une matrice.

La décomposition d'une matrice en produit LU de matrices triangulaires est basée sur l'élimination de Gauss [10]. La décomposition LU est utilisée pour la résolution des systèmes linéaires $Ax=b$. Le second membre b subit les mêmes transformations que la matrice A, on posera à cet effet $A=(A, b)$ [matrice de taille $n\times(n+1)$]. On peut schématiser l'algorithme général de triangularisation de la manière suivante :

Pour $k := 1$ *à* $n-1$ *faire*
Pour $i := k+1$ *à* n *faire*

$$\begin{bmatrix} \text{ligne } k \\ \text{ligne } i \end{bmatrix} := M_{ik} \begin{bmatrix} \text{ligne } k \\ \text{ligne } i \end{bmatrix}$$

La matrice M_{ik} est choisie de manière à annuler le coefficient en position (i, k). L'algorithme procède en $n-1$ étapes. A l'étape k, la k-ième ligne sert de pivot. On la combine avec toutes les lignes inférieures pour annuler les éléments de la k-ième colonne situés sous la diagonale.

Le choix des matrices M_{ik} dépend des propriétés de la matrice du système. En effet, il est exclus d'introduire des techniques de pivotage, même partiel, car cela aurait pour effet de détruire la régularité du réseau. Aussi emploie-t-on la méthode de Gauss (sans pivotage) pour des matrices symétriques définies positives ou à diagonales dominantes [10]. Dans le cas général, il faut recourir à des matrices de factorisations orthogonales (matrices de Givens) pour des raisons de stabilité. L'algorithme est le même dans les deux cas, au choix de la matrice M_{ik} près. De notre point de vue, il importe seulement d'organiser les rencontres *ad hoc* entre les lignes de la matrice, indépendamment de la nature des combinaisons réalisées par les processeurs.

Gentleman et Kung [9] utilisent un réseau triangulaire de processeurs connectés orthogonalement, reproduit figure 7 pour $n=4$. Le nombre total de cellules est égal à $n[(n+1)/2+1]$. La matrice A entre dans le réseau colonne par colonne. Plus précisément, le processeur P_{1k} reçoit les coefficients de la colonne k, un nouvel élément à chaque top d'horloge, à partir du temps k. Ce format d'entrée est également représenté figure 7.

D'une façon informelle, disons que la k-ième ligne du réseau est destinée à effectuer l'étape k de l'algorithme. Pour cela, on procède en deux temps. D'abord, il faut stocker la k-ième ligne de la matrice dans le réseau, un coefficient dans le registre interne de chaque cellule. Les lignes suivantes traversent alors les processeurs $P_{k1}, \ldots, P_{k,n+2-k}$, et effectuent au passage une combinaison avec la ligne pivot k. A cet effet, les matrices $M_{k+1,k}, M_{k+2,k}, \ldots, M_{nk}$ sont générées par la cellule ronde et transmises à toutes les autres cellules de la ligne. Bien sûr, ces deux phases sont pipelinées, si bien que la cellule P_{k1} génère les premières matrices M_{ik} alors que les derniers coefficients de la ligne pivot ne sont pas encore stockés.

Examinons en détail le fonctionnement de la première ligne du réseau. Tout d'abord, la première ligne de la matrice A est stockée dans les registres internes des

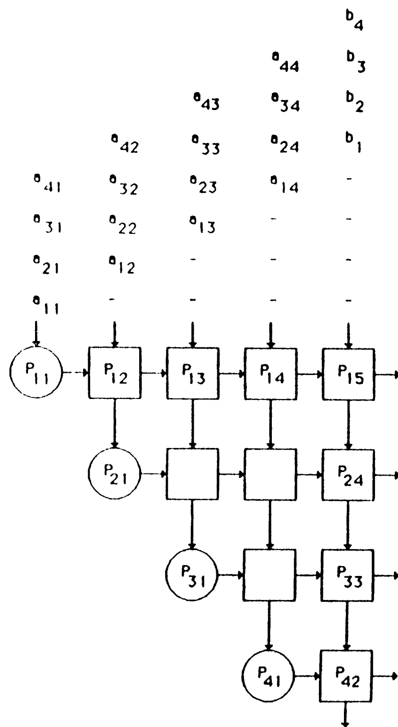


Fig. 7. — Triangularisation d'une matrice dense.

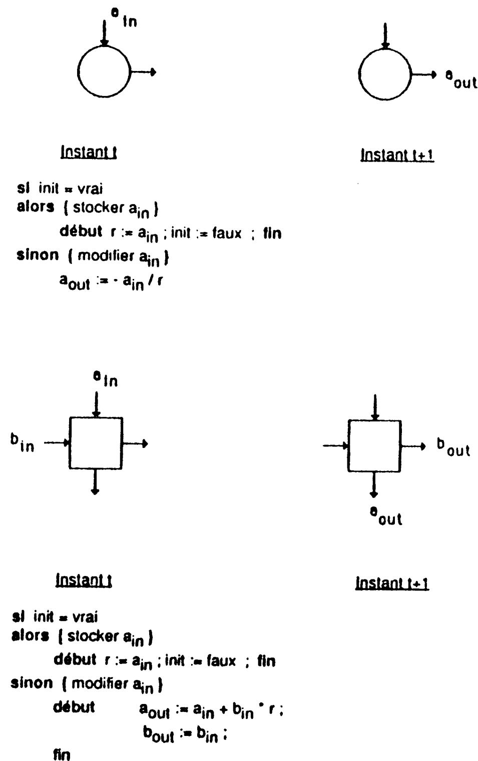


Fig. 8. — Programmation des cellules pour l'élimination de Gauss.

cellules : l'élément a_{1k} est stocké dans le processeur P_{1k} au temps k. Quand une ligne numéro $i\,(i \geqq 2)$ est lue par le réseau, elle est combinée avec la ligne 1 de manière à annuler l'élément a_{i1}. La matrice M_{i1} est générée par la cellule P_{11} au temps i, puis elle est envoyée aux autres cellules de la ligne. Plus précisément, la cellule $P_{1k}\,(k \leqq 2)$ effectue au temps $i+k-1$, la transformation suivante :

$$(a_{1k},\ a_{ik})^t := M_{i1} \cdot (a_{1k},\ a_{ik})^t$$

On procède de la même manière pour toutes les lignes k.

Il y a deux types de cellules dans le réseau de la figure 7 : les cellules représentées par des ronds et celles représentées par des carrés.

Plus précisément, la k-ième ligne du réseau est composée de :

– la cellule P_{k1} (cellule ronde), qui génère les matrices M_{ik} pour $i > k$;

– toutes les autres (carrées), qui appliquent les transformations M_{ik}.

La programmation détaillée des cellules est décrite ci-dessous dans la figure 8 dans le cas de l'élimination de Gauss pour une matrice 4×4.

Le programme des processeurs est l'unique chose à modifier pour traiter le cas d'une factorisation orthogonale avec des matrices de Givens. Le flot des données et l'organisation des rencontres entre celles-ci reste inchangé, quel que soit l'algorithme de triangularisation utilisé. Dans la pratique, il serait facile de combiner les deux approches Gauss et Givens pour obtenir des cellules programmables, le choix de l'algorithme dépendrait alors d'un bit de contrôle. Bien mieux, on peut imaginer des variantes mixant les deux stratégies selon la taille du pivot rencontré. Cependant, le synchronisme total limite l'intérêt de telles stratégies, puisque le réseau fonctionne au rythme du processeur le plus lent.

L'évolution des opérations de la première ligne du réseau est résumée dans les figures suivantes sur un exemple avec $n=4$. Dans la table de la figure 9, on note M_{i1} pour la génération de la matrice d'élimination. De même, on note a_{1j} le contenu des registres internes des cellules dans la figure 9. $a_{ij}^{(k)}$ est la valeur de a_{ij} après k étapes d'élimination.

A la fin des $n-1$ étapes de l'algorithme, la matrice triangulaire (U, b') est stockée dans le réseau de la manière suivante :

$$\begin{matrix} u_{11}u_{12}u_{13}u_{14}\,b'_1 \\ u_{22}u_{23}u_{24}\,b'_2 \\ u_{33}u_{34}\,b'_3 \\ u_{44}\,b'_4 \end{matrix}$$

Il reste à vider le réseau. La philosophie générale du modèle impose un vidage systolique, par propagation d'une variable de contrôle. De plus, on voudrait le vidage du réseau de la phase de calcul, de manière à éviter n tops supplémentaires. Nous adoptons un processus de vidage où chaque cellule envoie le contenu de son registre interne à sa voisine de droite quand elle a fini de calculer.

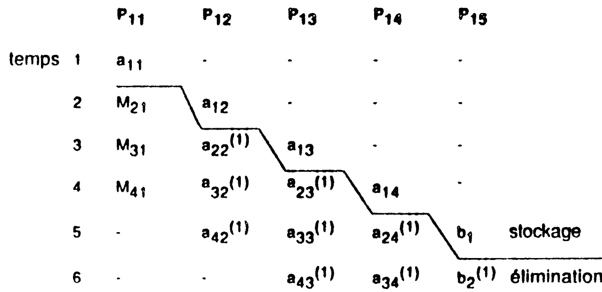


Fig. 9. — Opérations de la première ligne du réseau.

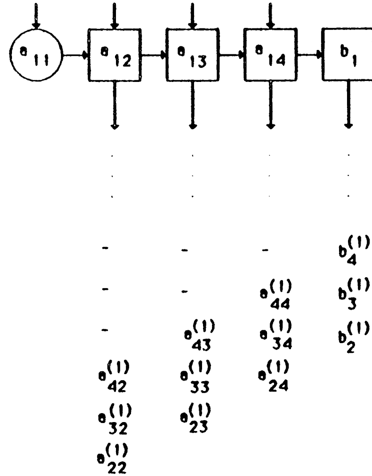


Fig. 10. — Visualisation de la première ligne du réseau.

Dans l'exemple précédent avec $n=4$, au temps 4 la cellule P_{11} travaille pour la dernière fois. P_{11} reste inactive durant un top puis envoie u_{11} vers la droite. P_{12} finit son travail au temps 5 et transmet son résultat au temps 6. Il transmet u_{11} vers la droite au temps 7, puis envoie le contenu de son propre registre u_{12} au temps 8. Le processus se poursuit, et u_{1j} est délivré en sortie par la cellule P_{15} au temps $9+i$ $(1 \leqq i \leqq n+1)$. De même, on commence à vider la deuxième ligne du réseau au temps 8. On voit que les premières lignes du réseau sont vidées alors que les dernières continuent à opérer: c'est là le «pipeline» que nous désirions.

Dans le cas général, le dernier calcul à lieu au temps $2i+3$ dans la cellule P_{i2}, et $u_{i,\,i+k}$ $(1 \leqq i \leqq n,$ $0 \leqq k \leqq n-i+1)$ est délivré en sortie du réseau au temps $2n+i+k+1$. Enfin, remarquons que l'on peut simplifier le vidage dans le cas de l'élimination de Gauss. Comme nous l'avons déjà remarqué, les $u_{i,\,i+k}$ ne seront plus modifiés une fois stockés, et peuvent donc être envoyés immédiatement vers le bas lors de leur stockage.

Le temps de calcul total pour effectuer la décomposition LU d'une matrice de taille n est donc $3n+2$ sur ce réseau. Dans le cas de l'élimination de Gauss, on obtient également la décomposition LU de A: les coefficients de la matrice L se déduisent directement (par changement de signe [10]) des matrices M_{ik}, qui sont délivrées en sortie à la droite du réseau lors du processus d'élimination.

Inversion de matrices triangulaires

Le réseau orthogonal précédent permet d'inverser une matrice triangulaire dans un temps $3n$. Li et Wah proposent un réseau de cellules connectées hexagonalement [23] qui calcule l'inverse en un temps asymptotique de $2n$ seulement. Cette performance (la meilleure connue) a été obtenue à l'aide d'une méthodologie de synthèse systématique des réseaux.

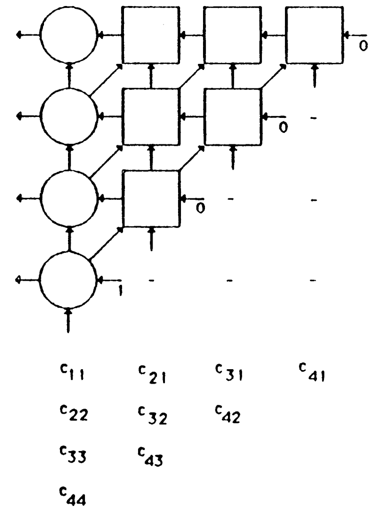


Fig. 11. — Inversion d'une matrice triangulaire.

Le réseau est représenté dans la figure 11 pour une matrice triangulaire inférieure C de taille 4×4. Comme le montre la figure, le flot des entrées se fait diagonale par diagonale au rythme de 1 coefficient tous les tops d'horloge. Le réseau est composé de deux types de cellules, dont l'action est décrite à la figure 12.

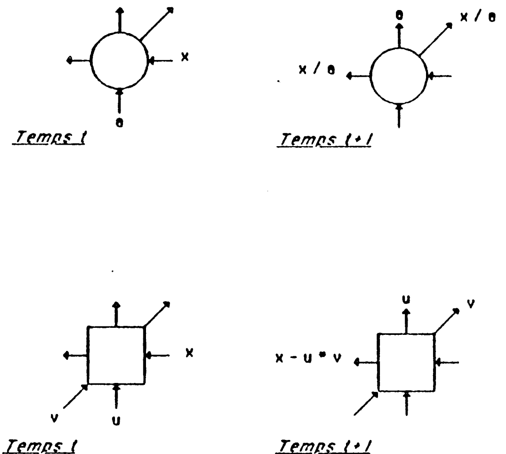


Fig. 12. — Opérations des cellules pour l'inversion.

Plutôt que de proposer une explication détaillée du fonctionnement de ce réseau, nous invitons le lecteur à suivre lui-même le processus d'élimination sur le réseau.

Bilan

On peut maintenant évaluer le temps de calcul global du produit $A^{-1}B$ à partir des réseaux élémentaires précédents pour une matrice A de taille $n \times n$. On obtient asymptotiquement la triangularisation en $3n$ unités de temps, l'inversion d'une matrice triangulaire en $2n$ et la multiplication en $4n$. Soit $11n$ unités de temps pour obtenir la matrice A^{-1}, donc $15n$ au total pour obtenir $A^{-1}B$. De plus, le flot des entrées n'est pas compatible d'une application à l'autre (entrée des matrices par colonnes ou par diagonales). Il faudrait donc en toute rigueur rajouter au temps de calcul, le temps de réordonnancement des données, et le coût d'un stockage intermédiaire en mémoire de l'architecture hôte.

Il est possible d'utiliser d'autres réseaux élémentaires pour effectuer le calcul de $A^{-1}B$, comme le réseau de Kramer et Van Leewen qui calcule l'inverse d'une matrice. Mais cette opération doit être suivie d'une multiplication sur un autre réseau, et la même remarque s'applique. A ce sujet, nous renvoyons à la discussion de Moraga [24] qui montre tout l'intérêt d'une approche basée sur un seul réseau, comme celui que nous décrivons maintenant.

2.2. Calcul direct de $A^{-1}B$

Le calcul direct de $A^{-1}B$ peut être vu comme la résolution de p systèmes linéaires de même matrice:

$$A x_i = b_i, \qquad 1 \leqq i \leqq p$$

On note B la matrice de taille $n \times p$ formée des p vecteurs seconds membres b_i et C la matrice (A, B) de taille $n \times (n+p)$.

Nous proposons ci-dessous un réseau orthogonal de $n(n+1)$ cellules qui permet le calcul de $A^{-1}B$ en temps $4n+p$. C'est le réseau transposé de celui décrit dans [5].

Le calcul de $A^{-1}B$ s'effectue à partir de la diagonalisation de Jordan. Comme dans le cas de la triangularisation, si la matrice A n'a pas les propriétés de stabilité requises, il faut mixer l'algorithme d'élimination avec un algorithme de factorisation orthogonale. Supposons pour l'instant que A est non singulière, et préoccupons nous exclusivement de l'implantation systolique de la méthode, sans souci numérique. L'algorithme vise à réduire la matrice $C=(A, B)$ en la matrice $(I_n, A^{-1}B)$ en la prémultipliant par n matrices élémentaires $J_1, J_2, \ldots, J_n$ (matrices de Jordan).

Posons $C_k = J_k . C_{k-1}$, la matrice obtenue après k étapes de l'algorithme, en partant de $C_0 = C$. Soit C_k^0 la matrice de taille $n \times (n+p-k)$ formée des $n+p-k$ dernières colonnes de C_k. On choisit J_k de telle sorte que les k premières colonnes de C_k soient celles de I_n (matrice identité d'ordre n). C_k a donc la structure suivante:

$$C_k = [\text{les } k \text{ premières colonnes de } I_n, C_k^0]$$

En particulier, C_n^0 est la matrice $A^{-1}B$ cherchée. La matrice J_k diffère seulement de l'identité par sa k-ième colonne, que l'on note:

$$[c_{1k}^{(k)} \ldots c_{nk}^{(k)}]^t$$

Les coefficients de la matrice C_k^0 sont notés $(c_{ij}^{(k)})$, $1 \leqq i \leqq n$, $k+1 \leqq j \leqq n+p$. La valeur des $c_{ij}^{(k)}$, $1 \leqq i \leqq n$, $k \leqq j \leqq n+p$, $1 \leqq k \leqq n$, est calculée récursivement par l'algorithme suivant, partant de $c_{ij}^{(0)} = c_{ij}$:

pour $k := 1$ *à* n *faire*
  début
    {calcul de J_k}
(A) $c_{kk}^{(k)} := 1/c_{kk}^{(k-1)}$;
    pour $i := 1$ *à* n, $i \neq k$, *faire*
(B) $c_{ik}^{(k)} := -c_{ik}^{(k-1)} * c_{kk}^{(k)}$;
    {calcul de C_k^0}
    pour $j := k+1$ *à* $n+p$ *faire*
      début
        pour $i := 1$ *à* n, $i \neq k$, *faire*
(C) $c_{ij}^{(k)} := c_{ij}^{(k-1)} + c_{ik}^{(k)} * c_{kj}^{(k-1)}$;
(D) $c_{kj}^{(k)} := c_{kk}^{(k)} * c_{kj}^{(k-1)}$;
      fin;
  fin;

Attention, ici $c_{ik}^{(k)}$ désigne un élément de la k-ième colonne de J_k, tandis que $c_{ij}^{(k)}$ avec $k<j$ se rapporte à la matrice C_{k2}^0.

Pour implanter cet algorithme sur un réseau systolique, nous utilisons un réseau orthogonal à $n(n+1)$

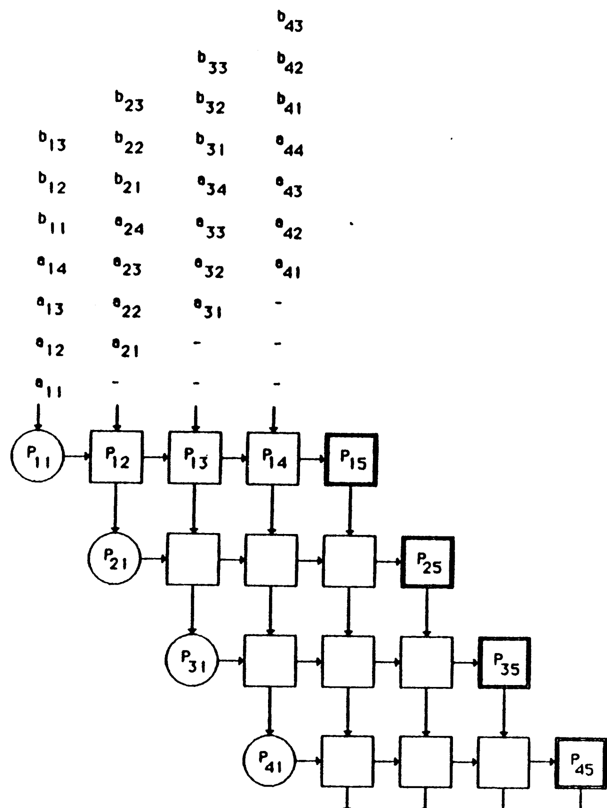


Fig. 13. — Calcul direct de $A^{-1}B$.

cellules. Le réseau se compose de n lignes, chaque ligne k comprenant $n+1$ processeurs numérotés de gauche à droite $P_{k,1}, \ldots, P_{k,n+1}$. Il est décrit dans la figure 13 sur un exemple de taille $n=4$ et $p=3$. La matrice A entre dans le réseau ligne par ligne, suivie de la matrice B. Plus précisément, la k-ième ligne de la matrice C est en entrée de la cellule P_{1k}, un nouvel élément tous les tops d'horloge, à partir du temps k. Ce format d'entrée est également décrit figure 13.

Avant de décrire en détail le fonctionnement de ce réseau, il convient de donner un mot d'explication informelle: dans le réseau de Gentleman et Kung, les lignes de la matrice sont stockées dans les processeurs, et les matrices M_{ik} circulent à travers le réseau. Ici, le fonctionnement est dual: les lignes circulent, et les matrices M_{ik} sont stockées dans les cellules. En effet, dans l'algorithme de Jordan, chaque ligne doit, quand elle devient la ligne pivot, rencontrer toutes les autres, et non plus seulement celles d'indices supérieurs. Plutôt que de faire repartir à travers le réseau une ligne déjà stockée, comme nous l'avons fait précédemment, il est préférable ici de faire circuler toutes les lignes de la matrice C. Ceci explique aussi que la matrice A entre dans le réseau sous forme transposée par rapport au réseau précédent de Gentleman et Kung.

Il y a n étapes dans l'algorithme de Jordan. Nous implantons chacune d'entre elles sur une ligne du réseau. Plus précisément, la k-ième instance de la boucle externe est réalisée par la k-ième ligne du réseau, qui reçoit la matrice C^0_{k-1} de la ligne $k-1$ et délivre la matrice C^0_k à la ligne $k+1$.

Chaque étape k comporte deux opérations distinctes: d'abord la génération de la matrice J_k, puis le calcul du produit $C^0_k = J_k C^0_{k-1}$. L'opération des cellules est définie en conséquence: les cellules de la ligne k calculent et stockent les coefficients $c^{(k)}_{ik}$, puis effectuent des calculs IPS.

Une fois les n matrices élémentaires J_k calculées et stockées, le réseau est initialisé. Il agit alors comme un opérateur linéaire qui transforme toute matrice B en $A^{-1}B$.

Les opérations des cellules sont décrites dans la figure 14. Il y a trois types de cellules:

– type 1: les cellules rondes calculent l'inverse de leur première entrée valide, suivant l'instruction (A) dans l'algorithme. Ensuite, elles agissent simplement comme des cellules de délai;

– type 2: les cellules carrées initialisent d'abord leur registre interne en stockant (après modification) leur première entrée valide [instruction (B)]. Puis elles agissent comme des cellules IPS [instruction (C)];

– type 3: les cellules à double carré opèrent en fait comme les cellules carrées, mais leur registre interne n'est pas initialisé de la même façon (voir *fig.* 14). Elles effectuent l'instruction (D).

Dans la k-ième ligne du réseau, le processeur de gauche $P_{k,1}$ est de type 1, et le processeur de droite $P_{k,n+1}$ est de type 3. Tous les autres processeurs sont de type 2. Comme pour le réseau de triangularisation, l'action des cellules est controlée par une variable

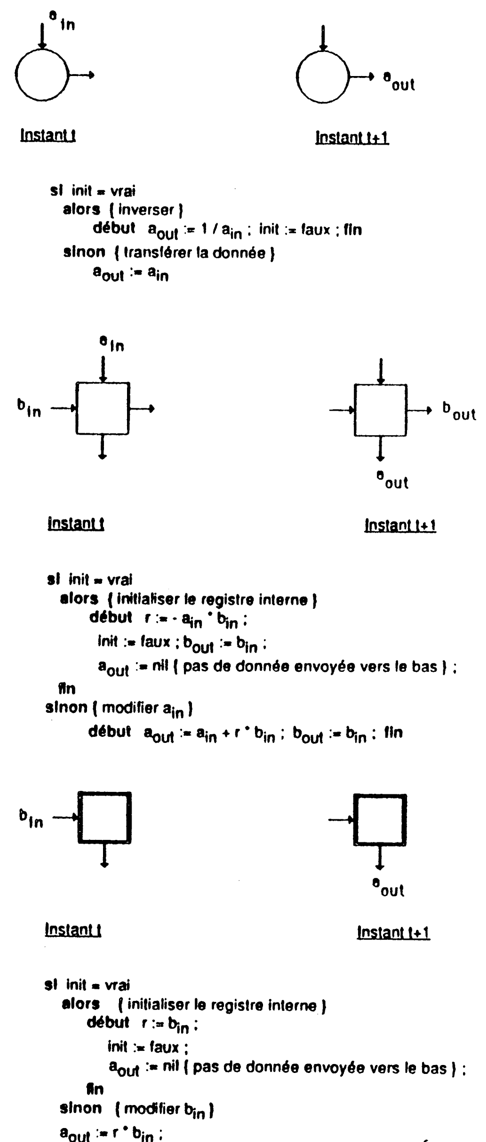


Fig. 14. – Opérations des cellules pour la diagonalisation de Jordan.

booléenne «init» qui repère si la première donnée que reçoit la cellule est valide. La même simplification a été adoptée: «init» (initialisé à la valeur «vrai») est supposé résider dans les cellules au début de l'exécution. Comme P_{kj} opère pour la première fois au temps $3k+j-3$, un signal en entrée au temps 1 de P_{11}, circulant à vitesse 1 vers la droite et 1/2 vers le bas, fait l'affaire.

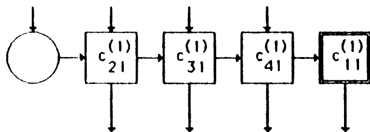
Décrivons en détail l'action de la première ligne du réseau. Au temps 1, le processeur P_{11} calcule $c^{(1)}_{11} := 1/c^{(0)}_{11}$ et agit ensuite comme une cellule délai. Au temps 2, P_{12} initialise son registre interne, avec le calcul $c^{(1)}_{21} := -c^{(0)}_{21} * c^{(1)}_{11}$. P_{12} agit alors comme une cellule IPS: au temps 3, il modifie $c^{(0)}_{22}$ en $c^{(1)}_{22} := c^{(0)}_{22} + c^{(1)}_{21} * c^{(0)}_{12}$, et de même avec $c_{23}, c_{24}, \ldots$ P_{13}, P_{14}, P_{1n} opèrent comme P_{12}, commençant respectivement au temps 3, 4, ..., n. Au temps $n+1$,

$c_{11}^{(1)}$ gagne le processeur $P_{1,n+1}$ et se stocke dans son registre interne. De $n+2$ jusqu'à la fin de l'exécution, $P_{1,n+1}$ évalue la première ligne de C_1^0 : à $n+2$, il calcule $c_{12}^{(1)}=c_{11}^{(1)} * c_{12}^{(0)}$, et de même avec c_{13}, c_{14}, ..., suivant l'instruction (D).

Les figures 15 et 16 suivantes résument, pour $n=4$ et $p=3$, l'action de la première ligne du réseau. On note que cette première ligne délivre la matrice C_1^0, de taille 4×6. Dans le cas général, C_1^0 est de taille $n \times (n+p-1)$, une colonne a été perdue en traversant la première ligne du réseau.

temps	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	
1	$c_{11}^{(1)}$	-	-	-	-	
2	·	$c_{21}^{(1)}$	-	-	-	
3		$c_{22}^{(1)}$	$c_{31}^{(1)}$	-	-	
4	-	$c_{23}^{(1)}$	$c_{32}^{(1)}$	$c_{41}^{(1)}$	-	
5	-	$c_{24}^{(1)}$	$c_{33}^{(1)}$	$c_{42}^{(1)}$	$c_{11}^{(1)}$	calcul de J_1
6	-	$c_{25}^{(1)}$	$c_{34}^{(1)}$	$c_{43}^{(1)}$	$c_{12}^{(1)}$	calcul de C_1

Fig. 15. — Opération de la première ligne du réseau.



			$c_{17}^{(1)}$
		$c_{47}^{(1)}$	$c_{16}^{(1)}$
	$c_{37}^{(1)}$	$c_{46}^{(1)}$	$c_{15}^{(1)}$
$c_{27}^{(1)}$	$c_{36}^{(1)}$	$c_{45}^{(1)}$	$c_{14}^{(1)}$
$c_{26}^{(1)}$	$c_{35}^{(1)}$	$c_{44}^{(1)}$	$c_{13}^{(1)}$
$c_{25}^{(1)}$	$c_{34}^{(1)}$	$c_{43}^{(1)}$	$c_{12}^{(1)}$
$c_{24}^{(1)}$	$c_{33}^{(1)}$	$c_{42}^{(1)}$	-
$c_{23}^{(1)}$	$c_{32}^{(1)}$	-	-
$c_{22}^{(1)}$	-	-	-

Fig. 16. — Visulaisation de la première ligne du réseau.

De façon similaire, la k-ième ligne du réseau effectue le calcul $C_k^0=J_k C_{k-1}$. La matrice en entrée C_{k-1}^0 est de taille $n \times (n+p-k+1)$, et la matrice en sortie C_k^0 est de taille $n \times (n+p-k)$. Ainsi, après avoir traversé tout le réseau, une ligne de $n+p$ coefficients devient une ligne contenant p coefficients seulement. $C_0=C_0^0=C$ est en entrée de la première ligne, C_1^0 en entrée de la seconde et C_{n-1}^0 en entrée de la n-ième. Finalement, le réseau délivre en sortie la matrice $C_n^0=A^{-1}B$.

Pour bien comprendre le fonctionnement de ce réseau, il est important de noter que les lignes de la matrice initiale sont permutées circulairement en traversant le réseau, de manière à ce que la ligne pivot à l'étape k puisse être combinée avec toutes les autres dans les cellules P_{k1}, ..., $P_{k,n+1}$.

Résumons les performances du réseau que nous venons de présenter : on calcule $A^{-1}B$ en $4n+p-2$ tops d'horloge sur un réseau de $n(n+1)$ cellules où A et B sont des matrices denses de tailles respectives $n \times n$ et $n \times p$. Ces chiffres sont à comparer au bilan de la section précédente.

3. Application au filtrage adaptatif

3.1. Calcul de l'inverse d'après Woodbury

La décomposition de Woodbury permet de calculer l'inverse d'une somme de matrices à partir des matrices elles-mêmes et de leurs inverses [10], p. 3 et donc en particulier de calculer l'inverse d'une matrice ayant subi une modification de rang k.

Plus précisément dans ce cas, si U et V désignent deux matrices de taille $n \times k$, alors sous certaines hypothèses de régularité [10], l'inverse de la matrice $M_{i+1}=(M_i+UV^t)$ peut être obtenu par la formule suivante :

$$M_{i+1}^{-1}=M_i^{-1}-M_i^{-1}U(I_k+V^t M_i^{-1}U)^{-1}V^t M_i^{-1}$$

Dans le cas d'une modification de rang 1, on obtient plus simplement :

$$M_{i+1}^{-1}=M_i^{-1}-(M_i^{-1}uv^t M_i^{-1})/(1+v^t M_i^{-1}u)$$

(u et v désignant des vecteurs colonnes).

Dans le problème qui nous occupe ici, on cherche à obtenir le produit $A_{i+1}^{-1}B_{i+1}$, en fonction du produit $A_i^{-1}B_i$ au pas précédent. A_{i+1} et B_{i+1} sont des matrices respectivement de tailles $n \times n$ et $n \times p$, elles sont obtenues par modifications de rang 1 de la manière suivante :

$$A_{i+1}=(1-\alpha)A_i+\alpha u^{(i)}u^{(i)t}$$
$$B_{i+1}=(1-\alpha)B_i+\alpha u^{(i)}v^{(i)t}$$

où α est un réel appartenant à $]0,1[$, $u^{(i)}$ et $v^{(i)}$ sont respectivement des vecteurs de tailles n et p.

On applique alors la formule de Woodbury pour calculer A_{i+1}^{-1} en fonction de A_i^{-1}. La formule complète s'écrit :

$$A_{i+1}^{-1}=[A_i^{-1}-\alpha A_i^{-1}u^{(i)}u^{(i)t}A_i^{-1}/(1-\alpha+\alpha u^{(i)t}A_i^{-1}u^{(i)})]/(1-\alpha)$$

Ce calcul complexe comporte quatre étapes indépendantes qui s'organisent de la manière suivante :

— une première étape où l'on calcule le produit $d^{(i)}=u^{(i)t}A_i^{-1}$;

– le calcul de la matrice $E_i = \alpha d^{(i)t} d^{(i)}$ mené simultanément avec le calcul du coefficient $f_i = 1 - \alpha + \alpha u^{(i)t} d^{(i)}$;
– division $G_i = E_i / f_i$;
– calcul de $(A_i^{-1} - G_i)/(1-\alpha)$.

Bien que les calculs ci-dessus soient simples et admettent bien une réalisation de type «pipeline», ils se prêtent mal à une implantation systolique efficace. En effet, on peut utiliser des réseaux connus pour réaliser les opérations élémentaires de chacune des quatre étapes, mais on rencontre alors les mêmes problèmes que précédemment – format d'entrées/sorties incompatibles, stockages intermédiaires – auxquels vient s'ajouter la nécessité de dupliquer certaines données.

3.2. Réseau adaptatif

Nous proposons dans cette section un réseau systolique qui permet de calculer directement le produit $A_{k+1}^{-1} B_{k+1}$, en fonction de A_k et B_k au pas précédent en tenant compte des modifications de rang 1. Partant du réseau décrit au chapitre 2.2, il suffit de lui adjoindre en entrée un réseau de préconditionnement.

Étudions en détail le passage de l'étape i à l'étape $i+1$. Tout d'abord, pour simplifier l'écriture, on note u_i l'élément courant du vecteur u $(i = 1, \ldots, n)$, en omettant l'exposant i de l'étape (de même pour v).

Le réseau de préconditionnement a pour fonction de modifier les valeurs courantes des coefficients a_{ij} et b_{ij} selon les formules:

$$a_{ij} := (1-\alpha)\, a_{ij} + \alpha\, u_i u_j$$
$$b_{ij} := (1-\alpha)\, b_{ij} + \alpha\, u_i v_j$$

On adopte le schéma de fonctionnement suivant (voir *fig.* 17): chaque ligne de la matrice courante (A, B) circule à travers un anneau composé de $n+p-1$ registres et d'une cellule du réseau de préconditionnement. Lors de son passage dans le réseau de préconditionnement, chaque coefficient est modifié selon les formules précédentes. La nouvelle valeur, qui retourne circuler dans l'anneau, est également envoyée en entrée du réseau de calcul de la section 2.2. Après une phase d'initialisation du réseau, qui correspond au premier calcul de $A_1^{-1} B_1$, une nouvelle matrice $A_k^{-1} B_k$ est délivrée tous les $n+p$ tops.

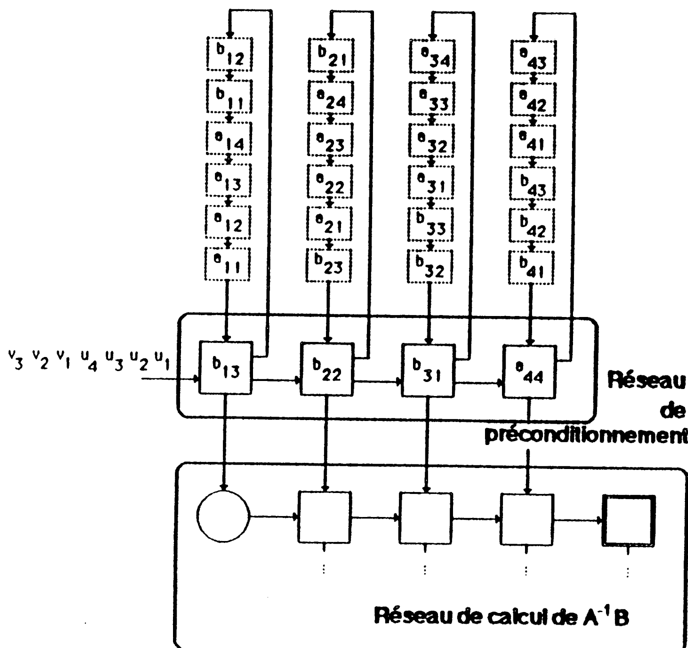
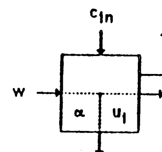
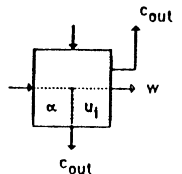


Fig. 17. – Fonctionnement du réseau de préconditionnement.

Le réseau de préconditionnement se compose de n cellules, une pour traiter chaque ligne de la matrice (A, B). Pour simplifier la description des cellules (*fig.* 18), on a supposé que l'élément u_i et le coefficient α résident dans la i-ième cellule du réseau. Pour s'affranchir de cette hypothèse, il suffit d'adjoindre un port vertical à chaque cellule i pour recevoir l'élément u_i (en même temps que a_{i1}) et de faire circuler α à travers le réseau, de la gauche vers la droite. Au passage, remarquons que cette solution permet de choisir une valeur différente de α à chaque étape, permettant ainsi de traiter le cas de durées de stationnarité variables.

Cellule i





Instant t   Instant t+1

$c_{out} := (1-\alpha)\, c_{in} + \alpha\, u_i\, w$

Fig. 18. – Opération des cellules du réseau de préconditionnement.

4. Conclusion

Deux caractéristiques dominantes du traitement du signal sont les suivantes: des débits d'entrées/sorties très rapides et une énorme masse de données en mémoire. Une puissance de calcul de l'ordre de la centaine de mégaflops est souvent nécessaire pour obtenir un résultat en temps-réel. Heureusement, les algorithmes utilisés possèdent pour la plupart des propriétés de régularité, répétitivité et localité, qui permettent d'envisager des solutions VLSI pour obtenir de telles performances à un coût raisonnable [21]. L'exemple du filtrage adaptatif que nous traitons dans cet article en est une bonne illustration.

Manuscrit reçu le 1er novembre 1986.

BIBLIOGRAPHIE

[1] H. M. Ahmed, J. M. Delosme et M. M. Morf, Highly concurrent computing structures for matrix arithmetic and signal processing, *IEEE Computer*, 15, n° 1, 1982, p. 65-82.

[2] M. Bellanger, *Traitement Numérique du Signal*, Masson, 1980.

[3] A. Blanc-Lapierre et B. Picinbono, *Fonctions aléatoires*, Masson, 1981.

[4] D. R. Brillinger, *Time series, data analysis, and theory*, Holden Day, 1981.

[5] P. Comon et Y. Robert, *A systolic array for computing* BA^{-1}, Rapport de Recherche TIM3/IMAG, Grenoble, n° 591, 1986.

[6] J. M. Delosme, Algorithms for finite shift-rank processes, *Ph. D. Thesis*, Technical Report M735-22, Sept. 1982, Stanford Electronics Laboratories.

[7] J. M. Delosme et I. C. F. Ipsen, An illustration of a methodology for the construction of efficient systolic architectures in VLSI, *Proc. Second Int. Symposium on VLSI technology, systems and applications*, Taipei, Taiwan, 1985, p. 268-273.

[8] B. Friedlander, Lattice filter for adaptive processing, *Proceedings of the IEEE*, 70, n° 8, 1982, p. 829-867.

[9] W. M. Gentleman et H. T. Kung, Matrix triangularisation by systolic arrays, *Proc. SPIE* 298, *Real-time Signal Processing IV*, San Diego, California, 1981, p. 19-26.

[10] G. H. Golub et C. F. Van Loan, *Matrix computations*, The Johns Hopkins University Press, 1983.

[11] C. Gueguen, An Introduction to Displacement Ranks. Les Houches, *Summer School, Session XLV "Signal Processing"*, Aug. 12-Sept. 6, 1985, Durani, Lacoume et Stora éd., North Holland (à paraître).

[12] S. Haykin éd., *Non Linear Methods for Spectral Analysis*, 1979, Springer Verlag.

[13] H. L. Honig et D. G. Messerschmitt. *Adaptive Filters; Structures, Algorithms and Applications*, 1984, Kluwer Academic Publishers.

[14] K. Hwang et F. Briggs, *Computer architecture and parallel processing*, Mac Graw Hill, 1984.

[15] K. Hwang et Y. H. Cheng, Partitioned matrix algorithm for VLSI arithmetic systems, *IEEE Trans. Computers*, 31, n° 12, 1982, p. 1215-12224.

[16] H. Jazwinski, *Stochastic processes and Filtering Theory*, 1970, Academic Press.

[17] T. Kailath éd., *Modern Signal Processing*, 1985, Springer Verlag.

[18] M. R. Kramer et J. Van Leeuwen Jr, *Systolic computation and VLSI, Foundations of Computer Science IV*, J. W. DeBakker et J. Van Leeuwen éd., 1983, p. 75-103.

[19] H. T. Kung, Why systolic architectures, *IEEE Computer*, 15, n° 1, 1982, p. 37-46.

[20] H. T. Kung et C. E. Leiserson, Systolic arrays for (VLSI), *Proc. of the Symposium on Sparse Matrices Computations*, I. S. Duff *et al.* éd., Knoxville, Tenn., 1978, p. 256-282.

[21] S. Y. Kung, VLSI array processors, *IEEE ASSP Magazine*, 2, n° 3, 1985, p. 4-22.

[22] S. Y. Kung, H. J. Whitehouse et T. Kailath éd., *VLSI and Modern Signal Processing*, Prentice Hall. Englewood Cliffs, NJ, 1985.

[23] G. H. Li et B. W. Wah, The design of optimal systolic arrays, *IEEE Trans. Computers*, 34, n° 1, 1985, p. 66-77.

[24] C. Moraga, On a case of symbiosis between systolic arrays, *Integration the VLSI Journal*, 2, 1984, p. 243-253.

[25] J. G. Nash, S. Hansen et G. R. Nudd, *VLSI processor arrays for matrix manipulation, VLSI Systems & Computations*, Rockville, MA, H. T. Kung *et al.* éd., Computer Science Press, 1981, p. 367-378.

[26] B. Picinbono, Signaux Déterministes et Aléatoires: Analyse et Modes Houches, *Summer School, Session XLV "Signal Processing"* Aug. 12-Sept. 6, 1985, Durani, Lacoume et Stora, éd., North Holland (à paraître).

[27] F. P. Preparata et J. Vuillemin, Area-time optimal VLSI networks for multiplying matrices, *Information Processing Letters*, 11, n° 2, 1980, p. 77-80.

[28] C. R. Rao, *Linear Statistical Inference and its Applications*, 1973, Wiley.

[29] Y. Robert et M. Tchuente, Résolution systolique de systèmes linéaires denses, *RAIRO Modélisation et Analyse Numérique*, 19, n° 2, 1985, p. 315-326.

[30] R. Schreiber et P. Kuekes, *Systolic linear algebra machines*, in [22].

[31] S. A. Tretter, *Introduction to Discrete-Time Signal Processing*, 1976, Wiley.

[32] H. L. Van Trees, *Detection, estimation and modulation theory*, 1, Wiley, 1968.

[33] P. D. Welch, The Use of FFT for the Estimation of Power Spectra, *IEEE Trans. Audio Electro.*, 15, n° 2, June 1967, p. 70-73.

PARALLEL IMPLEMENTATION OF THE ALGEBRAIC PATH PROBLEM

Yves ROBERT[+] & Denis TRYSTRAM[++]

[+] CNRS, Laboratoire TIM3, BP 68, 38402 St Martin d'Hères Cedex, France

[++] Ecole Centrale Paris, 92295 Chatenay Malabry Cedex, France

Abstract: The Algebraic Path Problem is a general framework which unifies several algorithms arising from various fields of computer science. Rote [11] introduces a general algorithm to solve any instance of the APP, as well as a hexagonal systolic array of $(n+1)^2$ elementary processors which can solve the problem in 7n-2 time steps. We propose a new algorithm to solve the APP, and demonstrate its equivalence with Rote's algorithm. The new algorithm is more suitable to parallelization: we propose an orthogonal systolic array of n(n+1) processors which solves the APP within only 5n-2 steps. Finally, we give some experiments on the implementation of our new algorithm in the parallel environment developped by IBM at ECSEC in Roma.

INTRODUCTION

In a recent paper [11], Rote introduces the Gauss-Jordan elimination algorithm for the Algebraic Path Problem (APP for short), a general framework which unifies several algorithms arising from various fields of computer science. The Algebraic Path Problem is defined as follows [11] [14]: given a weighted graph G = (V,E,w) where V is a finite vertex set, E an arc set, a function w : E -> H with weights from a semiring (H,(+),(x),*) with zero 0 and unity 1, find for all pairs of vertices (i,j) the quantities d_{ij} = (+) { w(p), $p \in M_{ij}$ } where M_{ij} denotes the set of all paths from i to j. With the weighted graph (V,E,w) we associate the n by n weigth matrix A = (a_{ij}), where a_{ij} = w(i,j) if (i,j)$\in$E and a_{ij}=0 otherwise. We denote $M_{ij}^{(k)}$ the set of all paths from i to j which contain only vertices x with $1 \le x \le k$ as intermediate vertices. In practice, $a_{ij}^{(k)}$ = (+) { w(p), $p \in M_{ij}^{(k)}$ } is equal to the successive values of a_{ij} which we want to compute, starting from the initial value $a_{ij}^{(0)} = a_{ij}$ up to $a_{ij}^{(n)} = d_{ij}$. Applications of the APP are obtained by specializing the operations (+), (x), and * in the appropriate semirings. We detail three of them:

- (i) determination of the inverse of a real matrix: A is a real matrix, (+) and (x) are the usual arithmetic operations in **R**, and the *-operation is the following: **if** $c \neq 1$ **then** $c^* := 1/(1-c)$). The algorithm computes $(I-A)^{-1}$. Of course, straightforward modifications permit to compute directly A^{-1}.

Support from the IBM European Center for Scientific and Engineering Computing in Roma and the Coordinated Research Program C3 of CNRS is gratefully acknowledged.

- (ii) shortest distances in a weighted graph: the weights a_{ij} are taken in $H=\mathbb{R}\cup\{-\infty,+\infty\}$ (we let $a_{ij} = +\infty$ if the arc (i,j) is missing), (+) is the minimum in H, (x) is the addition in $\mathbb{R}$ extended to H (with $-\infty$ (+) $+\infty = +\infty$) and * is defined by (**if** $c\geq 0$ **then** $c^* := 0$ else $c^* := -\infty$)
- (iii) transitive and reflexive closure of a binary relation: the a_{ij} are boolean, (+) et (x) are respectively the "and" and "or" operations, and * is defined by (c^* := true for all c).

We do not assume the multiplication in the semiring to be commutative: this permits to easily implement blocks schemes. Rote [11] presents a hexagonally connected systolic array of $(n+1)^2$ processors which can solve any instance of size n of the APP in 7n-2 time steps, each time being the time necessary to achieve a multiply-and-add in the underlying algebra. In section 2, we introduce a new algorithm to solve any instance of the APP, which is more suitable to parallelization.

A NEW ALGORITHM FOR THE ALGEBRAIC PATH PROBLEM

```
{ The new algorithm) }
for k := 1 to n
      akk(k) := (akk(k-1))*
      for i := 1 to n, i ≠ k
            aik(k) := aik(k-1) (x) akk(k)
      for j := 1 to n, j ≠ k
      begin
            for i := 1 to n, i ≠ k
                  aij(k) := aij(k-1) (+)  aik(k) (x) akj(k-1) ;
            akj(k) := akk(k) (x) akj(k-1) ;
      end ;
end ;
```

The following lemma (which can be easily proven using induction on k) demonstrates the equivalence of the new algorithm with Rote's one [11]:

Lemma : given a n by n matrix $A^0 = (a_{ij}^{(0)})$, the new algorithm computes the solution matrix $A^n = (a_{ij}^{(n)}) = D$ of the APP.

THE SYSTOLIC ARRAY

We use a two-dimensional array of orthogonally connected processors (see figure 1). The array is composed of n rows, each row k including n+1 processors numbered from left to right $P_{k,1}$, ... , $P_{k,n+1}$. The operation of each processor is detailed in the figure 2. There are three types of processors:

• type 1: circle processors perform the * operation on their first input data. Afterwards they simply act as delay cells.

• type 2: square processors first initialize their current register by storing after modification their first input data; then they act as multiply-and-add cells (when (+) and (x) are the standard operations on real numbers, they become classical IPS cells [5] [6] [8] [9]).

• type 3: double-square processors actually operate as square processors, with the exception that their current register is not initialized in the same way (remember that (x) is not commutative).

In the k-th row of the array, the leftmost processor $P_{k,1}$ is of type 1 and the rightmost processor $P_{k,n+1}$ is of type 3. All the other processors $P_{k,2}, ..., P_{k,n}$ are of type 2.

The matrix A, followed by I, the identity matrix of order n, is fed into the array row by row. More specifically, row k of the n by 2n matrix (A,I) is input to processor P_{1k}, one new element each time-step, beginning at time t=k. This input format is depicted in the figure 1. We can state the

Theorem : given a dense nonsingular n x n matrix $A^{(0)} = A$, the orthogonal systolic array of n(n+1) processors computes the solution $A^{(n)}$ of the APP within 5n-2 time-steps.

Rote's hexagonal array [11] involves $(n+1)^2$ processors and solves the APP in 7n-2 time-steps. As pointed out in [11], the value of 5n-2 time-steps achieved by our orthogonal array is optimal within one step, provided that each of the n^2 data elements is used only once in each step. We point out that the performances of our array overcome those of the array of [3] for computing the transitive closure (n^2 processors and 6n-2 time-steps) and those of the array of [8] for matrix inversion (n(3n+1)/2 processors and 5n-2 time-steps). Moreover, the solution of a new instance of the APP can begin every 2n steps. Finally, in the case of matrix inversion, if we input the matrix A followed by a n by n matrix B instead of I, it can be seen that the array delivers $(I-A)^{-1}B$ (or $A^{-1}B$ with straighforward modifications) in the same number of steps.

IMPLEMENTATION IN THE VM/EPEX ENVIRONMENT

In this section, we deal with the implementation of the new algorithm 2 using the VM/EPEX environment at ECSEC in IBM Roma. The Experimental Software for Parallel FORTRAN Programming VM/EPEX has been developped at the IBM T.J. Watson Research Center [13] and runs under the Virtual Machine/Conversational Monitor System (VM/CMS).

The general philosophy of the VM/EPEX software is to provide the user with facilities to run a program on several Virtual Machines (VMs for short) rather than one, hence allowing parallelism [1] [2] [13]. The problem of communication and synchronisation between the VMs where pieces of programs run (tasks with the definition of [7]) has been solved in two different ways [2]:

• shared memory : the user has the possibility to create a shared memory which can be accessed by all the VMs

• message passing : the VMs can exchange messages and data through communication subroutines (VMFACS).

The Algorithm 2 that we want to implement requires an amount of $O(n^3)$ arithmetic operations; it consists of 0(n) iteration steps, each step being composed of O(n) independent tasks, which are identical but operate on different data. Each task has an execution time of O(n). A full synchronisation is needed at the end of each step. Thus a shared-memory approach is better suited to such a synchronisation-intensive problem.

The programmer writes his program using an extension of standard IBM FORTRAN that specifically includes a few new keywords for multitasking handling [2] [13]:

- creating shared data : **@Shared** (statement equivalent to COMMONs in sequential FORTRAN programs)
- distributing the instances of a parallel loop among the VMs : **@Do** and **@Enddo** with the **Wait** and **Nowait** options. Successive values of the loop index will be randomly distributed among the VMs. The default option **Wait** specifies that a process must wait for the completion of all the iterations of the loop before proceeding, whereas the **Nowait** option permits the process to continue as soon as all the work of the loop has assigned, even though other processes may still be completing some of the loop iterations
- executing serial pieces of code on a single VM : **@Serial begin** and **@Serial end**, with the same options and the possibility of specifying the VM where the piece of code is to be executed (Process=)
- each VM is identified by its own number **@Mynum**. The total number of Vms is **@Numprocs**.

We do not detail further here the use of these extensions (see [13]). We concentrate now on the problem of the parallelization of our new algorithm. The algorithm will be executed by several VMs, which communicate through the shared memory. We have the following conceptual scheme:

```
      { Algorithm 2 }
      Do 1 k = 1, n
            execute task Tkk  { updating of column k }
            execute in parallel the tasks Tkj, j≠k, 1≤j≤n
1     Continue

      { Task Tkk }
      A(k,k) = Star(A(k,k))
      Do 11 i = 1,k-1
            A(i,k) = A(i,k) (x) A(k,k)
11    Continue
      Do 12 i = k+1,n
            A(i,k) = A(i,k) (x) A(k,k)
12    Continue

      { Task Tkj (j≠k) }
      Do 21 i = 1,k-1
            A(i,j) = A(i,j) (+) A(i,k) (x) A(k,j)
21    Continue
      Do 22 i = k+1,n
            A(i,j) = A(i,j) (+) A(i,k) (x) A(k,j)
22    Continue
      A(k,j) = A(k,k) (x) A(k,j)
```

Here Star(x) is the function which computes x^* for an input data x. Notice that task T_{kk} vanishes when

computing the transitive closure of a binary relation

The task T_{kk} must be executed by a single processor (say the first one) and must be completed before it is possible to start the execution of any task T_{kj}, $j \neq k$. This leads to the following implementation (we omit the lecture of the matrix A):

```
        { create shared memory }
        @Shared  /MATRIX/A(n,n)
        Do 1 k = 1, n
              { serial piece of code to be executed by Processor 1 }
              @Serial  begin  process=1
                      execute task Tkk
              @Serial  end
              { execute in parallel the tasks Tkj, j≠k }
              @Do j = 1, k-1
                      execute task Tkj
              @Endo  nowait
              @Do j = k+1, n
                      execute task Tkj
              @Endo
1       Continue
```

After the **@Serial end**, the default option is **Wait**, hence all the processors 2, ..., p are waiting at this synchronization point for processor 1 to complete the execution of T_{kk}. In the contrary, the **Nowait** option is used after the first **@Enddo** to ensure that some processors can start the execution of the tasks $T_{k,k+1}$, $T_{k,k+2}$, ... before all the tasks T_{kj} with $j<k$ are completed.

Now we have to measure the performances of our parallel algorithm. Considering an algorithm executed on a parallel computer with p (identical) processors in time T_p, and letting TSEQ be its sequential execution time (with 1 processor), we define the speed-up factor as the ratio S_p = TSEQ/T_p, and the efficiency is given by $e_p = S_p/p < 1$ [4] [12].

The first idea is to use the available Fortran procedure CPUT to compute the CPU time of each virtual machine. The interest is twofold (i) this will permit to check whether the tasks have been evenly distributed among the VMs, and (ii) we can compute the overhead due to the parallelization. However, this does not permit to evaluate PARTIME, the execution time of the parallel algorithm. We have to know whether the tasks have been well distributed among the VMs inside each instance of the outer sequential Do loop on k. A global measure of the task repartition is not sufficient. For instance, one can imagine an execution scheme where every instance on k is executed by a single processor (not always the same) which performs all the tasks T_{kj}, $j \neq k$, whereas the global repartition of the work is still good. Therefore, we declare a two-dimensional array in the shared memory:

@Shared/TIME_ARRAY/TIME(@Numprocs,#steps)

and we store at each step k the virtual CPU time of a given VM in TIME(**@Mynum**,k). We can now compute the efficiency

e_p = TSEQ / (**@Numprocs** * PARTIME)

The results of some experiments are given in the figure 3. For each of the three aformentioned instances of the APP, the efficiency was computed for various sizes of matrix, ranging from 128 to 640. The curves are presented for 2 and 8 processors. With 2 processors (□ on the curves), the efficiency varies beetween 0.5 and 0.8, whereas for 8 processors (+ on the curves), it varies between 0.15 and 0.3. In the figure 4, the efficiency obtained for the three instances are compared, with 2, 4, 6 and 8 processors. □ refers to shortest distances, + to matrix inversion, and o to the transitive closure. The sequential time for 512 x 512 matrices is 842 seconds for shortest distances, 648 seconds for matrix inversion, and 533 seconds for the transitive closure. This shows that the elementary tasks are more costly for shortest distances than for matrix inversion and, in turn, than for the transitive closure. Since the number of tasks is the same in each of the three instances of the APP, this explains why the best efficiency is obtained for shortest distances, and the worst for the transitive closure.

The relatively low efficiency obtained with 4 processors or more is due both to the synchronization-intensity of our algorithm, and to the fact that the granularity of our tasks is relatively small (a few milliseconds in average). Nevertheless, our approach demonstrates that the VM/EPEX environment with the shared-memory option offers full possibilities to analyze the performance of parallel programs.

REFERENCES

[1] E. CLEMENTI, Progress report on our experimentation with parallel supercomputers ICAP1 and ICAP2, Conf. "Le Calcul ... Demain", P. Chenin et al. eds, Masson 1985

[2] P. DI CHIO, V. ZECCA, IBM ECSEC facilities: user's guide, IBM ECSEC Report, Roma 1985

[3] L.J. GUIBAS, H.T. KUNG, C.D. THOMPSON, Direct VLSI implementation of combinatorial algorithms, Proc. Caltech Conf. on VLSI, California Inst. Technology, Pasadena 1979, 509-525

[4] K. HWANG et F. BRIGGS, Parallel processing and computer architecture, Mc Graw Hill, 1984

[5] H.T. KUNG, Why systolic architectures, Computer 15, 1 (1982), 37-46

[6] H.T. KUNG, C.E. LEISERSON, Systolic arrays for (VLSI), Proc. of the Symposium on Sparse Matrices Computations, I.S. Duff and G.W. Stewart eds, Knoxville, Tenn. (1978), 256-282

[7] 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

[8] J.G. NASH, S. HANSEN, G.R. NUDD, VLSI processor arrays for matrix manipulation, VLSI Systems & Computations, H.T.Kung et al. eds, Computer Science Press (1981), 367-378

[9] Y. ROBERT, Block LU decomposition of a band matrix on a systolic array, Int. J. Computer Math. 17 (1985), 295-315

[10] Y. ROBERT, D. TRYSTRAM, Un réseau systolique pour le problème du chemin algébrique, C.R.A.S. Paris 302, I, 6 (1986), 241-244

[11] G. ROTE, A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion), Computing 34 (1985), 191-219

[12] U. SCHENDEL, Introduction to numerical methods for parallel computers, E. Horwood 1984

[13] J.M. STONE, V.A. NORTON, F.D. ROGERS, E.A. MELTON, G.F. PFISTER, The VM/EPEX FORTRAN preprocessor reference, IBM Report, Yorktown Heights, NY, USA (1985)

[14] U. ZIMMERMANN, Linear and combinatorial optimization in ordered algebraic structures, Ann. Discrete Math. 10, 1 (1981), 1-380

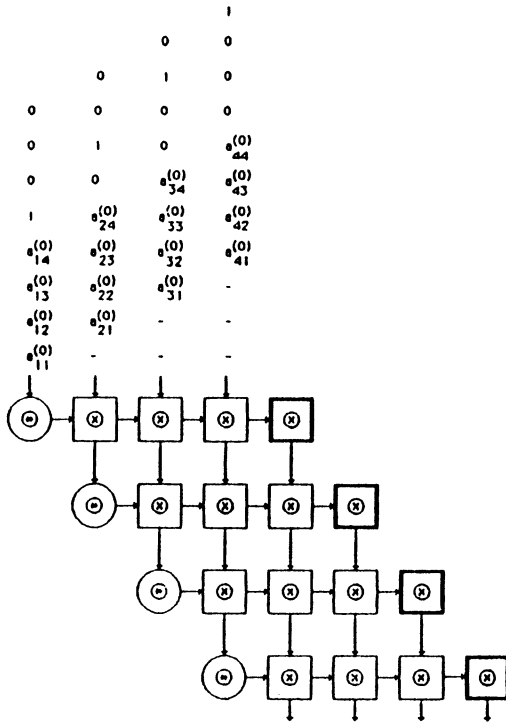


Figure 1
The systolic array

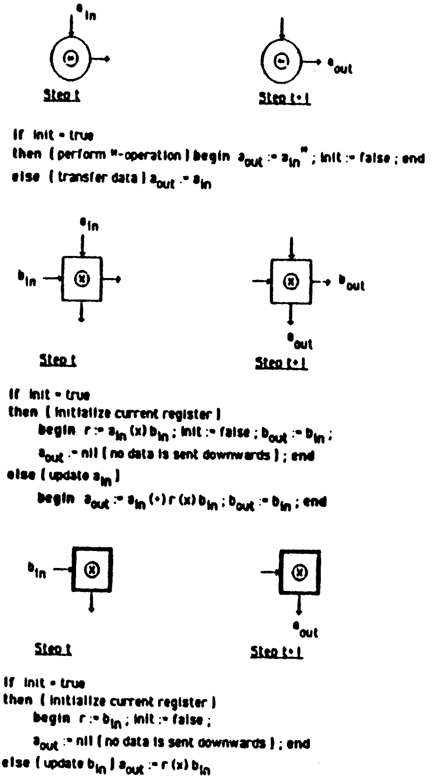


Figure 2
Operation of the processors

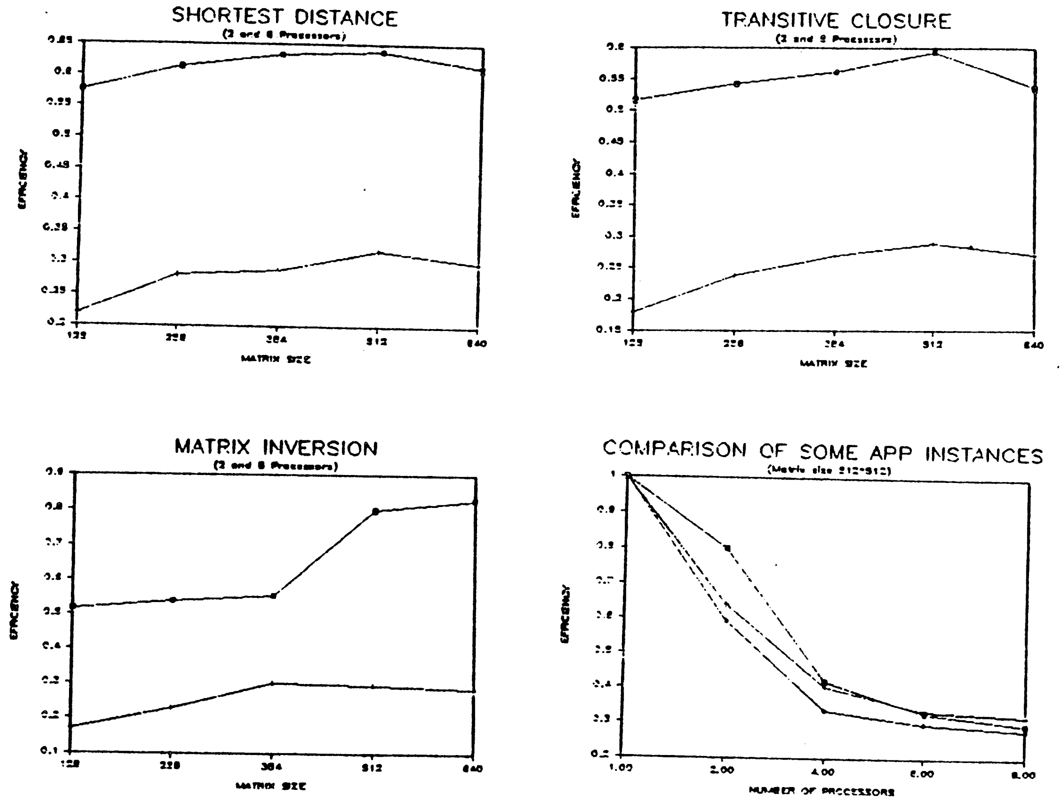


Figure 3
Efficiency for the three selected instances of the APP

Parallel Computing 00 (1988) PAC00201
North-Holland

Short Communication

Comments on scheduling parallel iterative methods on multiprocessor systems *

Yves ROBERT and Denis TRYSTRAM
CNRS, Laboratoire TIM3, INPG, 38031 Grenoble Cedex, France

Received ●●●

Abstract. In this note we improve results presented in the paper: N.M. Missirlis, Scheduling parallel iterative methods on multiprocessor systems, *Parallel Computing* 5 (1987) 295–302.

Keywords. Parallel numerical algorithms, parallel iterative methods, task graph

1. Introduction

In [7] parallel algorithms are presented for the classical point Gauss–Seidel (GS) iterative algorithm [1]. Using the same notations as in [7], the basic iteration of the algorithm consists in successively updating the components of the iteration vector $u^{(n+1)}$ as

$$Du^{(n+1)} = C_L u^{(n+1)} + C_U u^{(n)} + b$$

or in component form

$$u_i^{(n+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} u_j^{(n+1)} - \sum_{j=i+1}^{N} a_{ij} u_j^{(n)} \right) / a_{ii}.$$

In [7] Missirlis proposes a computation in two steps:
(i) compute the right-hand side $C_U u^{(n)} + b$;
(ii) solve the lower triangular system $(D - C_L) u^{(n+1)} = C_U u^{(n)} + b$.

We have the set of tasks $J = J_1 \cup J_2$, where $J_1 = \{T_{i,j}: 1 \leq j \leq i \leq N\}$ and $J_2 = \{T_{i,j}: 1 \leq i < j \leq N\}$. The tasks are defined as follows:
(i) for $i < j$, $T_{i,j}$: $\langle u_i^{(n+1)} := u_i^{(n+1)} - a_{ij} u_j^{(n)} \rangle$,
(ii) for $i > j$, $T_{i,j}$: $\langle u_i^{(n+1)} := u_i^{(n+1)} - a_{ij} u_j^{(n+1)} \rangle$,
(iii) for $i = j$, $T_{i,i}$: $\langle u_i^{(n+1)} := u_i^{(n+1)} / a_{ii} \rangle$,
where initially $u_i^{(n+1)} = b_i$ for all i.

* This work has been supported by the CNRS through the GRECO C3

0167-8191/88/$3.50 © 1988, Elsevier Scie

2. New results

Let $\lessdot$ denote the precedence constraint relation. Missirlis claims that the tasks of set J_2 are mutually non-interfering and can be executed in parallel. However, given i, all the tasks $T_{i,j}$, $j > i$, update the same component $u_i^{(n+1)}$ and must be executed serially. This leads to the task graph represented in Fig. 1 for $N = 8$.

In [7], the algorithm for executing the tasks belonging to J_2 is proposed as follows: when $u_k^{(n)}$ is available, all the tasks $T_{1,k}, T_{2,k}, \ldots, T_{k-1,k}$ can be executed in parallel. Due to the precedence constraints identified above, it is not obvious to design an algorithm of maximum efficiency $e = 1$. A solution is to reorganize the computation as shown in Fig. 2. For all values of $p \leqslant \frac{1}{2}N$, the algorithm which consists in executing the tasks in a greedy manner, row by row and from left to right, will be optimal. Note that all tasks in a given row update a different component of $u^{(n+1)}$, and that precedence constraints are met with this greedy scheme of execution.

For the set J_1, Missirlis [7] gives the relations: (a) $T_{i,i} \lessdot T_{i,k}$ for $i < k \leqslant N$ and (b) $T_{k,j} \lessdot T_{k+1,j}$ for $j < k \leqslant N$. As noted in [7], this is exactly the same task graph as for Gaussian elimination [5], except that the length of the tasks is a constant. The results of [5] have been improved in a collection of papers, including [2–4,6,8]. The same techniques described in these papers can be applied here. We summarize the results in the following theorem:

Theorem. *For a problem of size N, using $p = \alpha N$ processors, we can design parallel algorithms whose asymptotic efficiencies are*

(i) *for $\alpha \leqslant \frac{1}{4}$, $e_\alpha = 1/(1 + \alpha^2)$, and this value is optimal,*
(ii) *for $\frac{1}{4} < \alpha \leqslant \frac{1}{3}$, $e_\alpha = 1/(1 + \frac{3}{2}\alpha^2)$,*
(iii) *for $\frac{1}{3} < \alpha \leqslant \frac{1}{2}$, $e_\alpha = 1/(1 + 2\alpha^2)$.*

Proof. For (i) we refer to the algorithm presented in [4,8]. For (ii) we refer to the algorithm presented in [6]. For (iii) we refer to the algorithm presented in [2,3]. □

In most practical situations, the number p of available processors is likely to be much less than the size N of the problem. Note that for $\alpha \leqslant \frac{1}{4}$, we obtain *optimal* algorithms of efficiency $e_\alpha = 1/(1 + \alpha^2) \geqslant 0.94$. This evidences the high degree of parallelism that can be achieved.

In Fig. 3, we show the superiority of our results by plotting the efficiency versus the ratio $\alpha = p/N$ of the parallel algorithms obtained respectively in [7] and in this note.

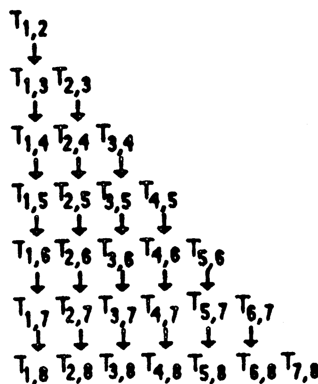


Fig. 1. Task graph for the set of tasks J_2.

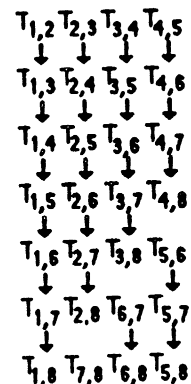


Fig. 2. Parallel scheme for J_2.

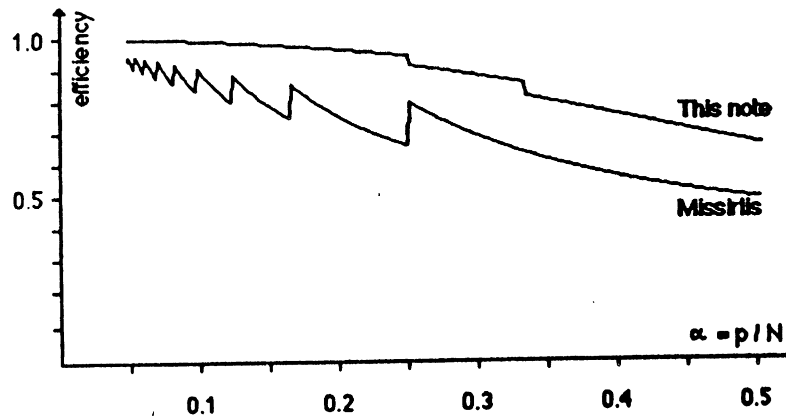


Fig. 3. Efficiency versus $\alpha = p/N$.

Finally, we point out that all our results extend to the block Gauss–Seidel algorithm, and that communication costs can be taken into account when defining the execution time of the tasks [8].

References

[1] P. Barlow and D.J. Evans, Parallel algorithms for the iterative solution of linear systems, *Comput. J.* **25** (1982) 58–60.

[2] M. Cosnard, M. Marrakchi, Y. Robert and D. Trystram, Gauss elimination algorithms for MIMD computers, *Proc. CONPAR 86*, Lecture Notes in Computer Science 237 (Springer, Berlin, 1986) 247–254.

[3] M. Cosnard, M. Marrakchi, Y. Robert and D. Trystram, Parallel Gaussian elimination on an MIMD computer, *Parallel Comput.* **6** (3) (1988) 275–296.

[4] J.C. Konig, Y. Robert and D. Trystram, Optimalité d'une classe d'algorithmes d'ordonnancement pour la méthode de Gauss and parallèle, *C.R. Acad. Sci. Paris Sér. I*, to appear.

[5] R.E. Lord, J.S. Kowalik and S.P. Kumar, Solving linear algebraic equations on an MIMD computer, *J. ACM* **30** (1) (1983) 103–117.

[6] M. Marrakchi and Y. Robert, Un algorithme parallèle pour la méthode de Gauss, *C.R. Acad. Sci. Paris Sér I* **303** (1986) 425–429.

[7] N.M. Missirlis, Scheduling parallel iterative methods on multiprocessor systems, *Parallel Comput* **5** (3) (1987) 295–302.

[8] Y. Robert and D. Trystram, Optimal scheduling algorithms for parallel Gaussian elimination, *Proc. International Symposium on High Performance Computer Systems*, (North-Holland, Amsterdam, 1988).

PARALLEL CONJUGATE GRADIENT ALGORITHM WITH LOCAL DECOMPOSITION

Pascal LAURENT-GENGOUX (1) and Denis TRYSTRAM (2)

(1) Ecole Centrale de Paris, grande voie des vignes, 92 295 Chatenay Malabry (France)
(2) laboratoire TIM3-INPG, 46 rue F. Viallet, 38 031 Grenoble Cedex (France)

ABSTRACT. *For the past 20 years, an increasing interest has been devoted to the Conjugate Gradient Method for solving large sparse linear systems (such as those arising from the Finite Element Method) and many other numerical problems. Moreover, the development of new parallel architectures seems to be one of the most efficient ways to accelerate the execution of the algorithms. We present in this paper an approach based on domain decomposition with local preconditioning of the conjugate gradient algorithm. It is especially adapted to elliptic problems which contain strong inhomogeneities, and leads to a natural parallelization. We discuss the implementation of this method on the IBM-3090 vector multiprocessor.*

RESUME. *La méthode du Gradient Conjugué a reçu récemment un regain d'intérêt considérable pour la résolution des grands systèmes linéaires creux tels que ceux qui proviennent de la discrétisation par Eléments Finis. Nous présentons dans ce rapport une approche basée sur un préconditionnement local du gradient conjugué, elle est spécialement adaptée aux problèmes elliptiques contenant de fortes inhomogénéités. Elle conduit à une parallélisation "naturelle", que nous expérimentons sur le multiprocesseur IBM 3090 VF.*

This work was supported by the IBM European Center of Scientific and Engineering Computing

1. INTRODUCTION

Recently, much effort has been devoted to find very efficient algorithms for solving linear systems. A good way to accelerate the execution time is to use preconditioned conjugate gradient algorithms. It seems today a very efficient method for solving large sparse linear systems which arise from the discretization of partial differential equations by finite element method [1], [5], [11]. It is currently used in an engineering environment for solving usual stress analysis in solid mechanics. However, some difficult problems remain not well solved now, because their modelization leads to ill-conditioned matrices. This is for instance the case in structural mechanics for the problems defined on an irregular mesh or for a structure composed with different materials.

The advantage of parallel architectures is that, if one can divide the computation into independent subproblems, one will be able to solve them simultaneously. Here, independent means that the subproblems do not access the same data, and at least do not modify the same items. The problems which are intrinsically partitioned into subdomains have this property of independence, and thus are very attractive for parallel computations. For this reason, the technique of domain decomposition has received a lot of interest recently [2], [8], [10].

The purpose of this paper is to present an efficient method based on domain decomposition with local preconditioning for solving problems of stress analysis, which contain strong inhomogeneities. This method can be implemented easily in parallel.

We will first recall the basic conjugate gradient algorithm, then we present the domain decomposition with local preconditioning and we compare it with the usual diagonal preconditioning. The parallelization is studied in the fourth section, before discussing its implementation for a practical problem in the parallel environment of the IBM-3090 VF.

2. PRESENTATION OF THE BASIC ALGORITHM

The Conjugate Gradient Method was introduced 30 years ago by Hestenes and Stiefel [7]. It is today often used for solving large sparse linear systems with preconditioning techniques [5], [6]. We give below the basic algorithm for solving Ax=b, where A is a n by n symmetric positive definite matrix. The preconditioning matrix M is chosen as an approximation of A, so that the extra system Mz=r which has to be solved at each step, does not require too much storage and is easy to solve. One can find in the literature many choices of M [5], [10], [11].

preconditioned conjugate gradient:

```
initialize (x,r,z,d)
    x given
    r = Ax-b
    solve Mz=r
    d=z
for k=1,n repeat
    (1) compute the product v=A*d
    (2) update (x,r)
         olderror = r^t.z
         rho = r^t.d / v^t.d
         x = x - rho*d
         r = r - rho*v
    (3) solve the system Mz=r
    (4) update (d)
         newerror = r^t.z
         beta = newerror / olderror
         d = z - beta*d
until sqrt(r^t.r) < given precision
```

We recall now some important properties of this algorithm:

1. In practice, it converges in less than n iterations for the bidimensional problems, still faster for the tridimensional problems and the faster as the size n increases.

2. For most problems, because of the fill-in of the matrix, the cost per iteration is of the order of one matrix-vector multiplication A*d and the resolution of the extra system Mz=r. The other operations (updating of x, r and d and the 3 inner products) can be neglected for a rough evaluation.

3. DOMAIN DECOMPOSITION WITH LOCAL PRECONDITIONING

Presentation

Let Ω be a domain defined by a solid. It is splitted into subdomains Ω each of them representing a coherent zone or a cell repeated by translatio in a composite material.

Matrices which arise from the modelization by finite element metho of stress analysis on such a domain are very ill-conditioned if th subdomains are composed with different materials (which introduc inhomogeneities in the matrix). For this reason, the usual methods ar difficult to employ. A good solution consists of solving subproblems o coherent subdomains, it means performing the matrix-vecto multiplication and the resolution of the extra system on each Ω_i. Thi allows users to have a very regular numerotation of the nodes. The sligh lack of independency between the subproblems is in the local boundaries We must take it into account while gathering the vectors after the loca operations.

We can notice that this resolution differs from a block decompositio of the matrix (where the numerotation is global and then the element of a given coherent zone are dispatched into the whole matrix A).

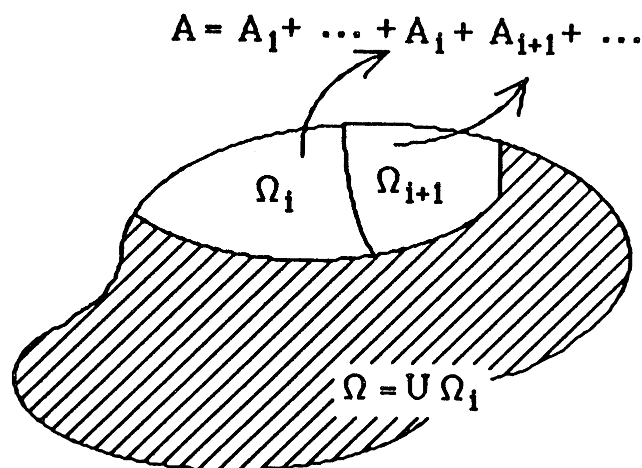


figure 1. Domain Decomposition

We give now the corresponding local decomposition algorithm:

initialize (x,r,z,d)
for k=1,n **repeat**
(1) **for** each subdomain Ω_i **do**
compute the local product $v_i=A_i{}^*d_i$
gather the v_i into v
(2) update (x,r)
(3) **for** each subdomain Ω_i **do**
solve the local system $M_iz_i=r_i$
gather the z_i into z
(4) update (d)
until sqrt(rt.r) < given precision

We will now discuss the choice of the preconditioning matrix. The basic choice of the local M_i matrices is the one corresponding to the local resolution on the subdomain, obtained by cancelling the outer nodes. This is exactly the matrix which is extracted from the stiffness matrix for the nodes on the considered subdomain, but it can be computed directly. This preconditioned subproblems $M_iz_i=r_i$ are solved exactly by a direct method.

Some remarks

It is noteworthy that this preconditioning is equivalent to the elimination of the nodes inside each subdomain, but it avoids the computation of the full matrix needed in that case.

The matrices M_i are band-stored, with minimum bandwidth, which comes from the coherence of the local numerotation. The band can be almost full for the problems with a few finite elements of high degree. If the zones are very large, it is possible to replace the above matrices by whatever approximation.

Efficiency of the local decomposition

We give now a comparison between this preconditioning and the usual diagonal preconditioning for two matrix sizes (2738 and 10658) with various values of the ratio of Young moduli for a bimaterial problem described further in section 5. The following curves represent the number of iterations versus this inhomogeneity ratio. For this problem, the local decomposition preconditioning costs a little more than twice times the diagonal one per iteration.

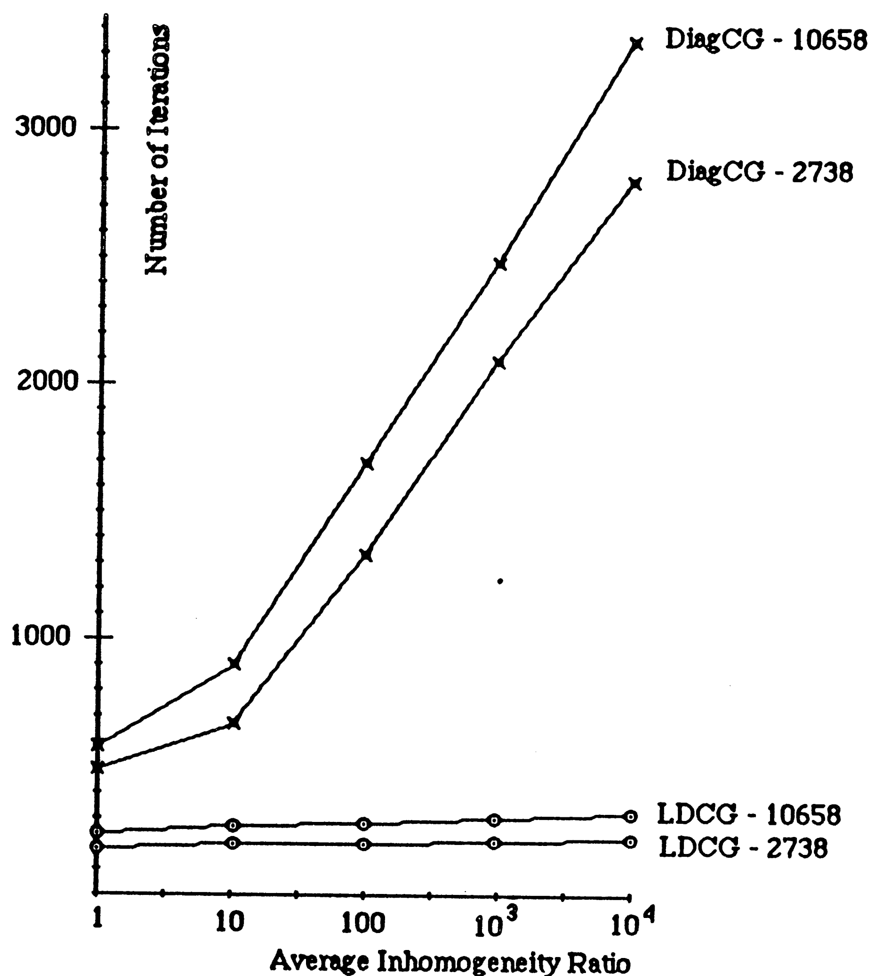


figure 2. Comparison between the preconditionings

It shows the very good efficiency of the local decomposition, even with a usual sequential computer. For instance, it runs more than five times faster than the diagonal preconditioning (39 seconds instead of 206 for the higher ratio in the case n=10658).

Moreover, this preconditioning is very efficient for the large matrices. The following figure depictes the small increasing of the number of iterations versus the matrix size (in this figure, the blocks size is constant).

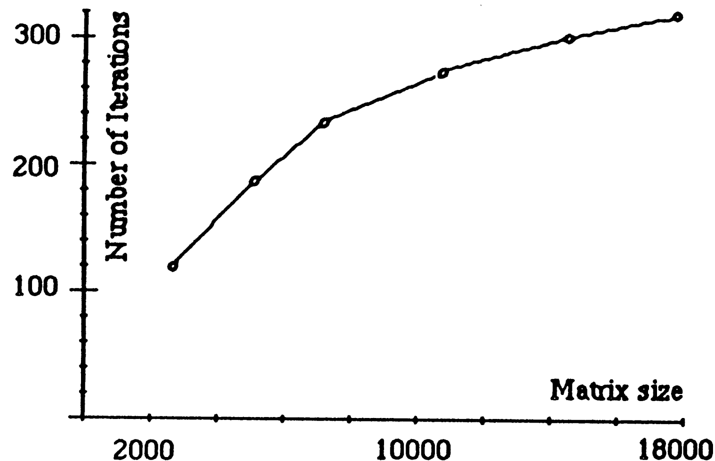


figure 3. variation of the number of iterations for LDCG

4. PARALLEL IMPLEMENTATION

How to parallelize conjugate gradient

There are many ways to parallelize the conjugate gradient algorithm. In the basic method, the only parts to parallelize are the matrix-vector multiplication and the resolution of the system inside the loop.

First, we can split the matrix into blocks. It is always possible and especially well adapted to approximatively block tridiagonal matrix. The simplest preconditioning consists in taking the block diagonal part of A. An improvement can be obtained by approximating A by a block incomplete cholesky factorization. We can also replace each diagonal blocks by an usual preconditioning of this block. It is possible to take overlapping blocks, that can be written as multisplitting [8], [10]. This was developed particularly for matrices which arise from finite element modelization.

Secondly, the domain can be splitted into subdomains. The preconditioning corresponds to a local problem (It differs only by the storage from the above overlapping approach). The storage takes into account this decomposition while writing the matrix A as the sum of the local contribution A_i of the each domain to the stiffness matrix. Thus, the parallelization follows a natural way by restricting the main operations to each subdomains.

Parallel Implementation

One can find the complete description of the IBM-3090 VF multiprocessor in [4]. We use the Multi Tasking Facility. It allows jobs to be organized so that a main task schedules complementary subtasks. The main task executes the part of the program that controls the overall execution. The subtasks execute the portions that run independently of the main task program and of each other. These portions are refered to as parallel subroutines. For a given problem, we define one macrozone per processor, each of them gathering some subdomains. The work is done on each macrozone independently.

For the parallelization of the local decomposition, we use in each domain an optimal numerotation and thus a band storage for the local matrix A_i and M_i. There are two conditions to satisfy for choosing the subdomains. The size of the subdomains have to be bounded in such a way as the matrices keep a good fill-in, but on the other hand the size must be large enough to obtain a good vectorization of the matrix operations (on each processor, the 8 vector registers of the 3090 use double precision vectors with a length of 128). This is the case for instance for high degree finite elements.

We point out that the code is partly vectorized. For instance, the matrix-vector multiplication is written diagonal by diagonal to get consecutive data in the vector registers. Only the phases (1) and (3) of the algorithm are parallelized. The other phases (2) and (4) are also intrinsically parallel, but taking into account their slight costs and the expensive implementation of the routines which dispatch the work on the processors, the improvement will not be significant (however, the vectorization of these operations is very efficient).

In order to give an insight to the parallel implementation, we give the code of the matrix-vector product (1) as follows:

```
define p macrozones (one per processor)
    (1) for each processor do in parallel
          call localproduct (macrozone)
          synchronize the processors
    gather the local product into the global vector v
```

5. NUMERICAL EXPERIMENTS

Description of the problem

We study here the analysis of the deformation in a composite material. We build a model problem in the following way: we suppose that the conditions of an usual plane stress analysis are respected, the domain is a cut of a solid represented by a rectangle with periodically disposed inclusions of another material (see below). We use rectangular elements with four nodes. Let denote m the total number of elements of the rectangular domain and q the total number of inclusions of size s.

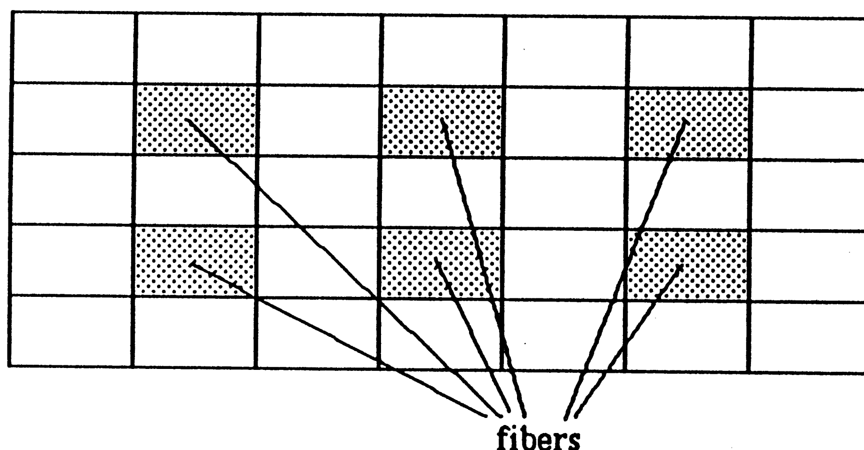


figure 4. Representation of the problem

Numerical results

Let take m=324 , s=50 and q=81 which lead to a matrix of size n=10658 with a bandwidth equal to A first set of results in the sequential case (with vector improvements) allows to compute the relative times of each phase of the local decomposition:

phase (1) matrix-vector multiplication: 33%
phase (3) resolution of the extra system: 62%
phases (2), (4) and a few other operations: 5%

The following table represents the variation of the blocks size on the convergence (number of iterations) for a given matrix of fixed size (n=2738 and n=10658). It shows the influence of the blocks size on the method. Less iterations, but more work per iterations (due to the resolution of the extra systems). Of course, the blocks size must not be

too large (we will then obtain the direct Cholesky resolution) neither not too small (in this case, the method becomes the simple Conjugate Gradient without preconditioning). The influence is not significant at all, we obtain very close execution times for both cases.

		matrix size	
		2758	10658
block size	50	116	261
	160	89	207

The last curve measures the efficiency of the algorithm. It is defined as the ratio of the sequential execution time on the product of the number of processor by the execution time with p processors. One can find in [3], the description of the theoritical tools for studying the complexity of parallel algorithms.

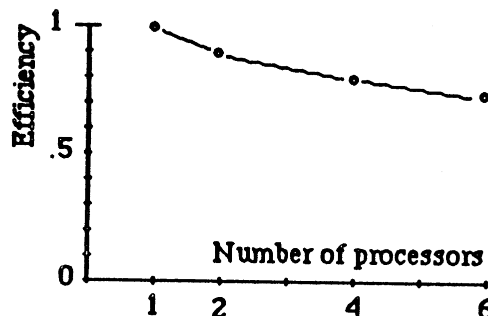


figure 5. Efficiency of the local decomposition

6. CONCLUDING REMARKS

We have presented a parallel implementation of a local decomposition used as preconditioning for the conjugate gradient algorithm. This method is particularly attractive for solving problems with local inhomogeneities. It leads to a very good efficiency when used on parallel computers (it has been tested on the IBM-3090 VF multiprocessor). These results are due to the good independency of the tasks (the data conflicts that come from the common boundaries are negligible), and the good rate of parallelism of the method (less than 5% of not parallelized work).

Acknoledgments

The authors are indebted to the staff of IBM-ECSEC for their help and their insight into the knowledge of the 3090 VF.

REFERENCES

[1] O. AXELSSON, a class of iterative methods for finite element equations, computer methods in applied mechanics and engineering 9, 123-137, 1976

[2] T. F. CHAN, analysis of preconditioners for domain decomposition, SIAM J. of Numerical Analysis, vol 2, N. 2, 1987

[3] M. COSNARD, M. MARRAKCHI, Y. ROBERT et D. TRYSTRAM, Parallel Gaussian Elimination on an MIMD Computer, Parallel Computing,1987

[4] VS FORTRAN Programming Guide, order N. SC26-4222-1, available through IBM branch offices, 1986

[5] G.H. GOLUB and G. MEURANT, résolution numérique des grands systèmes linéaires, Eyrolles, 1981

[6] G.H. GOLUB and C.F. VAN LOAN, matrix computations, Johns Hopkins Univ. Press, 1983

[7] M. HESTENES and E. STIEFEL, Methods of Conjugate Gradient for solving linear systems, J. Res. Nat. Bur. Stan. 49, 1952

[8] T.J. HUGHES, M. LEVIT and J. WINGET, element-by-element implicit solution algorithm for problems of structural and solid mechanics, Computer method in applied mechanics and engineering 36, 1983

[9] P. LAURENT-GENGOUX and D. TRYSTRAM, conjugate gradient algorithms for solving a large class of problems, Research Report ECP 1986

[10] D.P. O LEARY and E.B. WHITE, multi splittings of matrices and parallel solutions of linear systems, SIAM J. of Algebraic and Discrete Methods 6, 630-640, 1985

[11] D. TRYSTRAM, expérimentation d'algorithmes de préconditionnement pour des grands systèmes linéaires creux, Thesis University of Grenoble, 1984

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
. P. QUINTON, Directeur de recherche
. Y. SAAD, Professeur

Monsieur TRYSTRAM Denis

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme

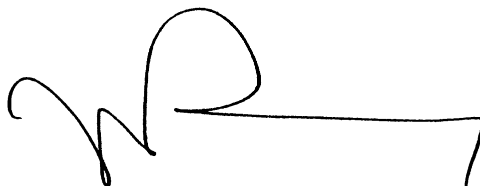
de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité

"Informatique".

Fait à Grenoble, le 1er avril 1988

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président.



Résumé.

L'objet de cette thèse est l'étude de la parallélisation d'algorithmes du calcul scientifique et leur implémentation sur des ordinateurs parallèles à mémoire partagée et sur des réseaux systoliques. Un accent particulier est mis sur l'obtention de résultats de complexité. La thèse est organisée autour d'articles et textes de conférences qui sont analysés et discutés dans une première partie de façon à permettre de replacer les problèmes traités dans leur contexte.

Dans le premier chapitre, nous présentons les principaux résultats théoriques concernant l'étude de complexité des algorithmes parallèles, ainsi qu'une description critique de l'architecture de référence, qui est une machine de type MIMD à mémoire partagée. Le chapitre suivant est dédié à l'ensemble des résultats de complexité concernant les algorithmes de diagonalisation et l'élimination de Gauss, il a pour but d'illustrer la méthodologie. Il existe en tout dix écritures possibles de la méthode de Gauss, qui conduisent principalement à deux grandes classes de graphes de précédence, conceptuellement différents : les graphes de type "glouton" et ceux du type "2-pas".

Ces types de graphes se rencontrent d'une manière plus générale dans d'autres problèmes d'algèbre linéaire et même dans certaines méthodes non numériques de la théorie des graphes. Nous développons les résultats de complexité concernant ces deux types de graphes sur les exemples les plus courant (versions kji et kij de Gauss en parallèle), puis nous montrons comment adapter l'étude en prenant en compte les temps de communication entre les processeurs, ce qui rend le modèle théorique plus réaliste.

Le chapitre 6 est consacré aux architectures systoliques. Le problème du chemin algébrique permet d'unifier plusieurs problèmes informatiques. Nous présentons un réseau résolvant ce problème en 5n-2 pas sur un réseau de taille n(n+1). De plus, quelques modifications permettent de calculer des projections en filtrage adaptatif en vu d'obtenir une solution en temps réel pour le traitement numérique des signaux.

Avant de conclure, nous présentons des résultats complémentaires de parallélisation effective sur d'autres types d'architectures : l'étude de l'algorithme du gradient conjugué sur des super calculateurs (CRAY-XMP et IBM 3090-VF).

Mots Clés.

Algorithmique Parallèle - Complexité - Graphe de Précédence - Ordonnancement - Méthode de Gauss - Réseaux Systoliques - Vectoriel