Programmation dynamique et traitement d'images sur machines parallèles à mémoire distribuée

Serge Miguet

THESE

Présentée par Serge MIGUET

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 23 novembre 1988)

(Spécialité : **Informatique**)

PROGRAMMATION DYNAMIQUE ET TRAITEMENT D'IMAGES SUR MACHINES PARALLELES A MEMOIRE DISTRIBUEE

Date de soutenance : 17 Décembre 1990

Composition du Jury :

Président :	J.P. Verjus
Rapporteurs :	P. Cinquin M. Cosnard
Examinateurs :	J.L. Basille J.M. Chassery Y. Robert

Thèse préparée au sein des laboratoires LMC-Imag et LIP-Imag (ENS-Lyon)

A Anne-Marie, à Clémentine et à sa sœur attendue en février...

INSTITUT NATIONAL POLYTECHNIQUE
GRENOBLE

46 avenue F. Viallet - 38031 GRENOBLE Cedex -
Tél : 76.57.45.00

ANNEE UNIVERSITAIRE 1989

Président de l'Institut
Monsieur Georges LESPINARD

PROFESSEURS DES UNIVERSITES

ENSERG	BARIBAUD	Michel
ENSIEG	BARRAUD	Alain
ENSPG	BAUDELET	Bernard
INPG	BEAUFILS	Jean-Pierre
ENSERG	BLIMAN	Samuel
ENSHMG	BOIS	Philippe
ENSEEG	BONNETAIN	Lucien
ENSPG	BONNET	Guy
ENSIEG	BRISSONNEAU	Pierre
IUFA	BRUNET	Yves
ENSHMG	CAILLERI E	Denis
ENSPG	CAVAIGNAC	Jean-François
ENSPG	CHARTIER	Germain
ENSERG	CHENEVIER	Pierre
UFR PGP	CHERADAME	Hervé
ENSIEG	CHERUY	Arlette
ENSERG	CHOVET	Alain
ENSERG	COHEN	Joseph
ENSEEG	COLINET	Catherine
ENSIEG	CORNUT	Bruno
ENSIEG	COULOMB	Jean-Louis
ENSERG	COUMES	André
ENSIMAG	CROWLEY	James
ENSHMG	DARVE	Félix
ENSIMAG	DELLA DORA	Jean-François
ENSERG	DEPEY	Maurice
ENSPG	DEPORTES	Jacques
ENSEEG	DEROO	Daniel
ENSEEG	DESRE	Pierre
ENSERG	DOLMAZON	Jean-Marc
ENSEEG	DURAND	Francis
ENSPG	DURAND	Jean-Louis
ENSHMG	FAUTRELLE	Yves
ENSIEG	FOGGIA	Albert
ENSIMAG	FONLUPT	Jean
ENSIEG	FOULARD	Claude
UFR PGP	GANDINI	Alessandro
ENSPG	GAUBERT	Claude
ENSERG	GENTIL	Pierre
ENSIEG	GENTIL	Sylviane
IUFA	GREVEN	Hélène
ENSIEG	GUEGUEN	Claude
ENSERG	GUERIN	Bernard
ENSEEG	GUYO T	Pierre
ENSIEG	IVANES	Marcel
ENSPG	JOST	Rémy
ENSPG	JOUBERT	Jean-Claud
ENSIEG	JOURDAIN	Geneviève
ENSIEG	LACOUME	Jean-Louis
ENSIEG	LADET	Pierre
ENSHMG	LESIEUR	Marcel
ENSHMG	LESPINARD	Georges
ENSPG	LONGEQUEUE	Jean-Pierre
ENSHMG	LORET	Benjamin
ENSEEG	LOUCHET	François
ENSEEG	LUCAZEAU	Guy
ENSIEG	MASSE	Philippe
ENSIEG	MASSELOT	Christian
ENSIMAG	MAZARE	Guy
ENSIMAG	MOHR	Roger
ENSHMG	MOREAU	René
ENSIEG	MORET	Roger
ENSIMAG	MOSSIERE	Jacques
ENSHMG	OBLED	Charles
ENSEEG	OZIL	Patrick
ENSEEG	PAULEAU	Yves
ENSIEG	PERRET	Robert
ENSHMG	PIAU	Jean-Miche
ENSERG	PIC	Etienne
ENSIMAG	PLATEAU	Brigitte
ENSERG	POUPOT	Christian
ENSEEG	RAMEAU	Jean-Jacqu
ENSPG	REINISCH	Raymond
UFR PGP	RENAUD	Maurice
UFR PGP	ROBERT	André
ENSIMAG	ROBERT	François
ENSIEG	SABONNADIERE	Jean-Claud
ENSIMAG	SAUCIER	Gabriele
ENSPG	SCHLENKER	Claire
ENSPG	SCHLENKER	Michel
ENSERG	SERMET	Pierre
UFR PGP	SILVY	Jacques
ENSHMG	SIRIEYS	Pierre
ENSEEG	SOHM	Jean-Clauc
ENSIMAG	SOLER	Jean-Louis
ENSEEG	SOUQUET	Jean-louis
ENSHMG	TROMPETTE	Philippe
ENSPG	VINCENT	Henri
ENSERG	ZADWORNY	François

PERSONNES AYANT OBTENU LE DIPLOME

d'habilitation à diriger des recherches

BECKER	M.	DANES	F.	GHIBAUDO	G.	MULLER	J.
BINDER	Z.	DEROO	D.	HAMAR	S.	NGUYEN TRONG	B.
CHASSERY	J.M.	DIARD	J.P.	HAMAR	R.	NIEZ	J.J.
CHOLLET	J.P.	DION	J.M.	LACHENAL	D.	PASTUREL	A.
COEY	J.	DUGARD	L.	LADET	P.	PLA	F.
COLINET	C.	DURAND	M.	LATOMBE	C.	ROGNON	J.P.
COMMAULT	C.	DURAND	R.	LE HUY	H.	ROUGER	J.
CORNUEJOLS	G.	GALERIE	A.	LE GORREC	B.	TCHUENTE	M.
COULOMB	J.L.	GAUTHIER	J.P.	MADAR	R.	VINCENT	H.
COURNIL	M.	GENTIL	S.	MEUNIER	G.	YAVARI	A.R.
DALARD	F.						

CHERCHEURS DU C.N.R.S.

1989/90

ALEMANY	Antoine
ALLIBERT	Colette
ALLIBERT	Michel
ANSARA	Ibrahim
ARMAND	Michel
AUDIER	Marc
BERNARD	Claude
BINDER	Gilbert
BONNET	Roland
BORNARD	Guy
CAILLET	Marcel
CALMET	Jacques
CARRE	René
CHATILLON	Christian
CLERMONT	Jean-Robert
COURTOIS	Bernard
DAVID	René
DION	Jean-Michel
DRIOLE	Jean
DURAND	Robert
ESCUDIER	Pierre
EUSTATHOPOULOS	Nicolas
FRUCHARD	Robert
GARNIER	Marcel
GLANGEAUD	François
GUELIN	Pierre
HOPFINGER	Emile
JORRAND	Philippe
JOUD	Jean-Charles
KAMARINOS	Georges
KLEITZ	Michel
KOFMAN	Walter
KRAKOWIAK	Sacha
LANDAU	Ioan
LEJEUNE	Gérard
LEPROVOST	Christian
MADAR	Roland
MERMET	Jean
MEUNIER	Jacques
MICHEL	Jean-Marie
NAYROLLES	Bernard
PEUZIN	Jean-Claude
PIAU	Monique
RENOUARD	Dominique
SENATEUR	Jean-Pierre
SIFAKIS	Joseph
SIMON	Jean-Paul
SUERY	Michel
TEODOSIU	Christian
VACHAUD	Georges
VAUCLIN	Michel
VENNEREAU	Pierre
VERJUS	Jean-Pierre
WACK	Bernard
YONNET	Jean-Paul

SITUATION PARTICULIERE

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J.Claude	Détachement.....	21/10/1989
ENSHMG	PIERRARD	J.Marie	Détachement.....	30/04/1989
ENSIMAG	VEILLON	Gérard	Détachement.....	30/09/1990
ENSIMAG	VERJUS	J.Pierre	Détachement.....	30/09/1989
ENSPG	BLOCH	Daniel	Récteur à c/.....	21/12/1988

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/1989
ENSHMG	BOUVARD	Maurice.....	30/09/1991
ENSEEG	PARIAUD	J.Charles	30/09/1991

.../...

PERSONNALITES AGREEES A TITRE PERMANENT A DIRIGER DES TRAVAUX DE RECHERCHE

(DECISION DU CONSEIL SCIENTIFIQUE)

ENSEEG	HAMMOU	Abdelkader
	MARTIN-GARIN	Régina
	SARRAZIN	Pierre
	SIMON	Jean-Paul
ENSERG	BOREL	Joseph
ENSIEG	DESCHIZEAUX	Pierre
	GLANGEAUD	François
	PERARD	Jacques
	REINISCH	Raymond
ENSHMG	ROWE	Alain
ENSIMAG	COURTIN	Jacques
C.E.N.G	CADET	Jean
	COEURE	Philippe
	DELHAYE	Jean-Marc
	DUPUY	Michel
	JOUVE	Hubert
	NICOLAU	Yvan
	NIFENECKER	Hervé
	PERROUD	Paul
	PEUZIN	Jean-Claude
	TAIEB	Maurice
	VINCENDON	Marc

Laboratoire extérieurs :

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

Remerciements

Un grand merci à Jean-Pierre Verjus pour l'honneur qu'il me fait en acceptant de présider mon jury de thèse.

De tout mon cœur, je remercie Yves Robert de m'avoir intégré à sa joyeuse équipe, et de l'aide qu'il m'a apportée au long de cette thèse. Il a toujours su allier efficacité et bonne humeur lors de nos travaux communs.

Je suis particulièrement reconnaissant à mes deux rapporteurs qui m'ont permis de passer en douceur de Grenoble à Lyon. Philippe Cinquin, qui reconnaîtra peut-être dans cette thèse des idées issues de nos recherches en informatique biomédicale. Michel Cosnard qui malgré son emploi du temps surchargé, a toujours été disponible pour m'aider à résoudre mes problèmes scientifiques ou administratifs.

Je tiens à remercier Jean-Luc Basille et Jean-Marc Chassery d'avoir accepté de faire partie de mon jury de thèse.

Enfin, mes remerciements vont aux autres chercheurs, techniciens, et administratifs, avec qui j'ai eu des rapports privilégiés. Pierre, toi qui depuis le début, as partagé avec moi les 18,20 m² de bureau qui nous ont été alloués, tu ne pouvais soutenir ta thèse que le même jour que moi. Jian-Jin, Ken, Stéphane, avec qui il a toujours été très agréable de travailler. Valérie, notre mère à tous, sans qui rien ne fonctionnerait plus. Giles, avec qui rien ne fonctionnerait plus non plus si Jean-Louis n'était pas là pour le surveiller ! A tous je vous dis Merci.

Introduction générale

Notre objectif est de contribuer à l'étude de la parallélisation d'algorithmes pour des machines à mémoire distribuée. Cette thèse est composée de trois parties.

La première partie est une introduction sommaire au monde du parallélisme. On y rappelle les principaux types d'architecture parallèle, en insistant sur les machines à mémoire distribuée. On y trouve quelques concepts d'algorithmique parallèle, et des notions d'analyse de performance.

Dans la deuxième partie de la thèse, nous passons en revue les différents problèmes pour lesquels nous proposons une implémentation parallèle. Il s'agit essentiellement d'algorithmes de programmation dynamique, de traitement ou de synthèse d'image, et d'algèbre linéaire. Nous expliquons leur intérêt, le lien qu'ils ont entre eux, et nous évoquons les difficultés essentielles posées par leur parallélisation.

Enfin, nos articles de recherche publiés dans des journaux spécialisés ou dans les actes de conférences internationales composent la troisième partie. Ils exposent plus en détail les problèmes liés à l'implémentation sur des machines parallèles à mémoire distribuée des algorithmes évoqués dans la deuxième partie. Ils présentent en outre des expérimentations qui corroborent nos modèles théoriques.

Table des matières :

Première partie : Algorithmique et architectures parallèles

Dans cette partie qui se veut être une brève introduction à l'informatique du parallélisme, nous par expliquons tout d'abord ce que sont les machines parallèles, leurs particularités, leurs avantages et leurs inconvénients. Nous décrivons ensuite la démarche générale que nous avons employée pour paralléliser un algorithme, et qui pourra servir de "recette" à un informaticien peu habitué à ce type d'ordinateurs. Enfin nous définissons des critères permettant de mesurer la qualité d'une parallélisation.

1.1. Les machines parallèles

Depuis l'aube de l'humanité, l'homme n'a cessé de découvrir, d'expérimenter, d'inventer, et de chercher à perfectionner ses inventions. Lors de la révolution industrielle déjà, le rythme des progrès successifs a été brutalement accéléré. Mais l'évolution a été encore plus spectaculaire dans le domaine de l'informatique : en moins de cinquante ans, on est passé des machines à calculer éléctro-mécaniques, effectuant péniblement les quatre opérations arithmétiques, à des ordinateurs capables d'exécuter des milliards d'opérations par secondes. Pour aller plus vite, on a commencé par inventer des composants électroniques, que l'on a miniaturisé au maximum pour les intégrer dans le silicium. En effet, plus un composant est petit, plus il réagit rapidement aux stimulations électriques qu'il subit. On atteint aujourd'hui des densités d'intégration qui permettent de disposer de plus d'un million de transistors sur le même circuit. Néanmoins, malgré les progrès qui restent a faire dans le domaine des semi-conducteurs, on sait que l'on est limité physiquement et

technologiquement dans la miniaturisation des circuits intégrés, donc dans la rapidité des composants électroniques.

De même que l'homme a voulu marcher sur la lune avant même de connaître le fond des océans de sa propre planète, il n'a pas attendu d'avoir atteint les limites technologiques de l'informatique classique avant de se tourner vers d'autres approches lui permettant de gagner des ordres de magnitude dans la puissance de ses machines. Il a fait appel pour cela à un principe qui remonte aux sources du règne animal, celui selon lequel l'union fait la force. Qui ne s'est jamais émerveillé devant l'ampleur du travail fourni par des abeilles ou des fourmis, en regard de la taille de ces insectes ? Leur puissance vient de leur nombre. Même si tous les individus n'agissent pas dans le même sens, même si le travail moyen de tous est bien inférieur à celui du plus besogneux, la tâche accomplie par l'ensemble du groupe reste inaccessible aux possibilités d'un seul.

L'avènement du parallélisme en informatique a été de multiplier les unités de traitement afin de pouvoir résoudre plus vite un problème donné, et de pouvoir appréhender des tâches plus importantes en un temps raisonnable. De nombreuses difficultés se posent lorsque l'on veut appliquer ce principe. Différentes solutions tentent de les résoudre, et donnent lieu à plusieurs classes de machines parallèles ayant chacune leurs spécificités. Mais rappelons tout d'abord la structure schématique d'un ordinateur traditionnel :

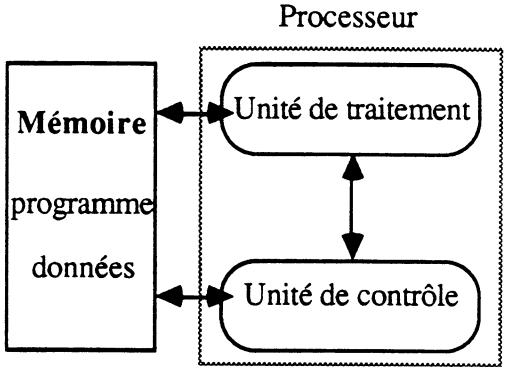


Schéma d'un ordinateur séquentiel

La mémoire contient à la fois le programme décrivant le traitement à effectuer, et les données à traiter. L'unité de contrôle est le chef d'orchestre qui lit les instructions du programme et commande en

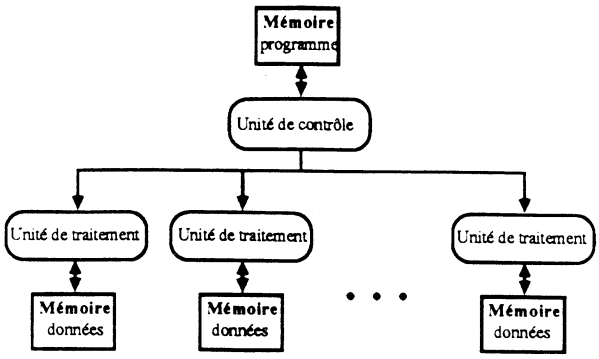
conséquence l'unité de traitement. Cette dernière est chargée de tous les calculs effectués par la machine. C'est elle qui lit, traite et stocke les données en mémoire.

Comme nous l'avons dit ci-dessus, il faut multiplier les unités de traitement pour disposer d'un calculateur parallèle. Mais qu'en est-il de la mémoire et de l'unité de contrôle ? Les principales classes d'ordinateurs parallèles se distinguent précisément par les réponses apportées à cette question.

Une première distinction se fait sur le type de mémoire utilisée. Quand la mémoire reste unique, les différentes unités de traitement y accèdent de manière concurrente, et on parle d'ordinateur à mémoire partagée. Quand chaque unité de traitement dispose de sa propre mémoire et est la seule à pouvoir y accéder, on parle d'ordinateur à mémoire distribuée.

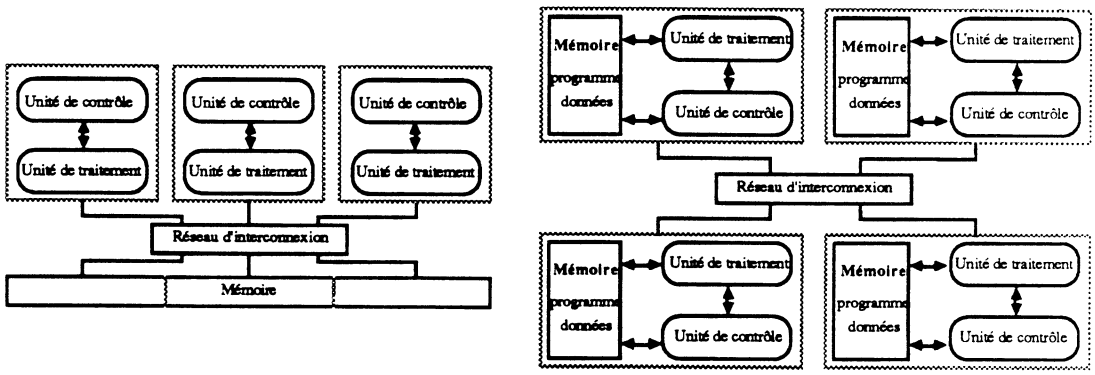
Une deuxième distinction, qui est à la base de la classification de Flynn [Fly], se fait sur le mode de contrôle de la machine.

Les ordinateurs qui conservent une seule unité de contrôle (on parle aussi de séquenceur), sont appelés SIMD, pour *Single Instruction, Multiple Data-stream*. La même instruction est envoyée par l'unité de contrôle unique à toutes les unités de traitement. Celles-ci effectuent donc le même traitement sur des données différentes. Seules certaines classes d'algorithmes se parallélisent de manière efficace sur ce type de machines, ce sont les algorithmes à "données parallèles". Notons que toutes les machines SIMD sont à mémoire distribuée. On remarque que l'on quitte le concept d'une mémoire contenant à la fois les données et le programme. Le séquenceur dispose de la mémoire programme, alors que les données sont dans les mémoires associées aux unités de traitement. Un exemple de machine SIMD dont parle beaucoup est la Connexion Machine, qui dans sa version complète dispose de 65536 processeurs.



Machine SIMD

Les machines au contraire, dans lesquelles on associe à chaque unité de traitement une unité de contrôle sont appelées MIMD, pour *Multiple Instruction, Multiple Data-stream.* A un instant donné, chaque processeur peut exécuter une instruction différente, sur des données différentes. Dans cette classe d'ordinateurs, certains sont à mémoire partagée, comme les Cray 2 ou les IBM 3090, et comportent au maximum quelques dizaines de processeurs, d'autres sont à mémoire distribuée, comme les hypercubes IPSC ou les machines à base de Transputers, et peuvent compter quelques centaines, voire des milliers de processeurs. Les algorithmes parallèles développés dans cette thèse sont destinés à ce dernier type de machines, et le paragraphe 1.2 leur est consacré de manière plus spécifique.



Machines MIMD

à mémoire partagée à mémoire distrubuée

On peut rajouter dans la classification de Flynn, que les ordinateurs traditionnels sont appelés SISD, pour *Single Instruction, Single Data-stream*, la classe MISD n'existant que dans la théorie.

Nous allons maintenant aborder les trois principaux problèmes qui se posent dans la mise en oeuvre d'un calculateur parallèle, et les manières dont ils sont résolus pour les différentes classes de machines.

1) Problème de **partitionnement** :

Les données doivent être distribuées entre les unités de mémoire pour pouvoir être utilisées par les unités de calcul en parallèle. Les machines à mémoire partagée divisent la mémoire en modules (ou bancs), et répartissent les données entre les différents modules. Dans les machines à mémoire distribuée, les données doivent être stockées ou acheminées dans la mémoire du processeur cible.

2) Problème d'**accès** :

Les unités de calcul doivent avoir des chemins d'accès parallèles aux données dont ils ont besoin au même instant. Pour les machines à mémoire partagée, les différents processeurs sont reliés aux différents bancs mémoire par un réseau d'interconnexion autorisant toutes les permutations. Toutes les données sont donc accessibles par tous les processeurs. Pour les machines à mémoire distribuée, il faut en revanche un réseau à très grande capacité interconnectant entre eux les différents processeurs.

3) Problème de **latence** :

Le délai dû aux accès simultanés ne doit pas diminuer la vitesse de traitement parallèle. On introduit des mémoires caches dans les processeurs des machines à mémoire partagée. Elles réduisent les requêtes d'accès à la mémoire, mais le maintien de leur cohérence peut être un facteur de ralentissement. Dans les machines à mémoire distribuée, les "producteurs" doivent envoyer les données assez tôt pour que les destinataires en disposent quand nécessaire.

Dans la section suivante, nous allons détailler les spécificités des machines auxquelles sont destinés les algorithmes parallèles développés dans cette thèse.

1.2. Machines MIMD à mémoire distribuée

Comme nous l'avons dit plus haut, la brique de base des ordinateurs MIMD à mémoire distribuée est un ordinateur séquentiel à part entière. Pour permettre à un ensemble de telles entités de coopérer, il faut les munir d'un système de communication. Nous allons passer en revue les différentes solutions qui sont adoptées, et qui permettent donc de

transformer un ensemble d'ordinateurs séquentiels en une machine parallèle.

1.2.1. Réseaux d'ordinateurs

Une première solution consiste à organiser de telles entités élémentaires autour d'un bus, ou d'un réseau local. C'est la solution en général adoptée pour relier entre elles des stations de travail. A plus grande échelle, les réseaux d'ordinateurs utilisent des lignes téléphoniques spécialisées. Certains problèmes très coûteux en temps de calcul, et ne nécessitant pas une coopération étroite entre les machines, peuvent être résolus sur ce type de réseaux. On peut citer par exemple les problèmes de factorisation de grands entiers, résolus en parallèle sur des milliers d'ordinateurs reliés en réseau. L'inconvénient majeur de cette classe de machines est la faible bande passante[1] disponible. Un seul transfert de donnée est autorisé à un moment précis pour les machines organisées en réseau local. Ce n'est pas le cas pour les réseaux à grande échelle, mais la vitesse de transmission est alors réduite. Les problèmes nécessitant des échanges intensifs entre processeurs ne peuvent donc pas être implémentés sur ce type de machines.

Les autres solutions établissent un réseau d'interconnexion point à point et à haut débit entre les processeurs. Nous allons en étudier quelques unes dans le paragraphe suivant.

1.2.2. Quelques topologies d'interconnexion

De nombreuses topologies d'interconnexion sont proposées et étudiées de manière théorique, sachant que les deux exigences contradictoires que l'on cherche à atteindre pour un graphe de communication sont :

1) un degré[2] Δ le plus faible possible, car on est limité physiquement par le nombre de pattes des circuits électroniques ;

2) un diamètre[3] D le plus petit possible, pour minimiser le temps de communication entre deux processeurs quelconques du réseau.

A cause de l'incompatibilité de ces deux exigences, on cherche des familles de graphes donnant des compromis intéressants. Nous allons

---

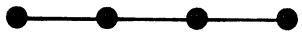
[1] La quantité d'information suceptible d'être échangée par unité de temps

[2] Le nombre maximal de voisins d'un noeud dans le graphe

[3] La distance maximale entre deux noeuds quelconques du graphe

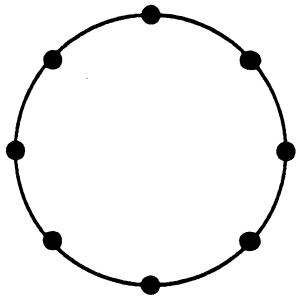
passer en revue quelques réseaux d'interconnexion classiques, en énonçant leurs avantages et leurs inconvénients. Nous donnons pour chacun d'eux, son degré Δ et son diamètre D en fonction de son nombre de sommets N.

Le réseau linéaire : $D = N-1$, $\Delta = 2$



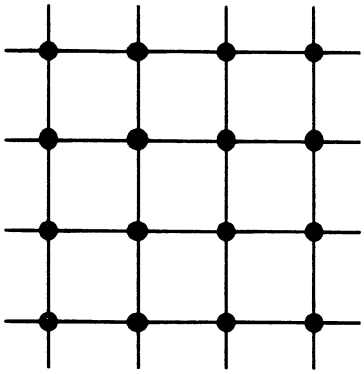
Son diamètre est grand. Il a pour intérêt d'être facile à fabriquer, et il suffit pour la mise en oeuvre de certains algorithmes ne comprenant que des communications locales.

L'anneau : $D = N/2$, $\Delta = 2$



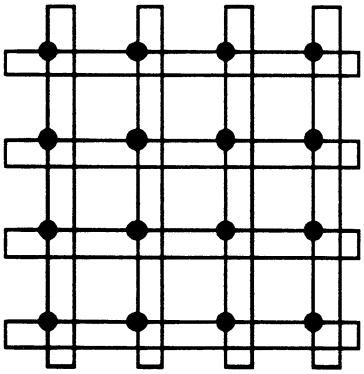
Il a les mêmes caractéristiques que le réseau linéaire, avec un diamètre plus faible. Par compte, si on souhaite ajouter un canal d'entrée-sortie avec le monde extérieur, son degré passe à 3.

La grille 2D : $D = 2(\sqrt{N}-1)$, $\Delta = 4$



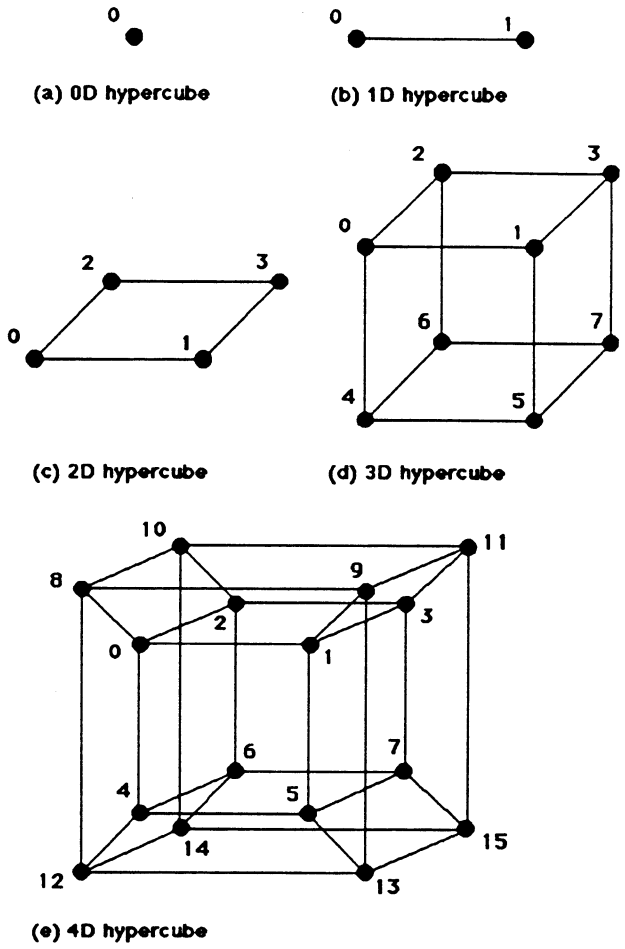
C'est une topologie facilement réalisable technologiquement. Son degré fixe permet de construire des réseaux de taille arbitraire, et son diamètre ne croît pas trop vite avec le nombre de processeurs. Elle est adaptée au traitement des problèmes réguliers comme l'imagerie par exemple.

Le tore 2D :$D = \sqrt{N}$, $\Delta = 4$



Il est à la grille ce que l'anneau est au réseau linéaire.

L'hypercube : $D = \log_2 N$, $\Delta = \log_2 N$

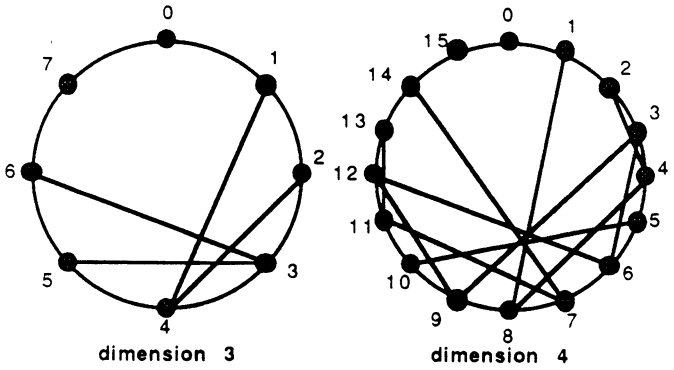


Définition :

- $N = 2^D$ sommets numérotés de 0 à N-1
- i relié à j si les écritures binaires de i et j diffèrent d'un bit.

C'est un graphe très intéressant pour plusieurs raisons : Son diamètre et son degré sont petits par rapport à son nombre de processeurs. On peut donc construire d'assez gros hypercubes (jusqu'à la dimension 10 environ) ayant des bonnes performances de communication. On dispose pour eux, d'algorithmes de routage de messages très performants. Ils admettent en outre de nombreuses sous-topologies comme les anneaux ou les tores, les arbres ou les hypercubes de dimension inférieures. Ils permettent donc d'exécuter facilement des algorithmes destinés à ces topologies. Son inconvénient majeur est que son degré n'est pas borné en fonction de son nombre de processeurs. Il est donc impossible technologiquement d'en réaliser d'arbitrairement gros.

L'anneau-mélange parfait : $D = 2\log_2 N$, $\Delta = 4$



Définition :

- $N = 2^D$ sommets numérotés de 0 à N-1
- anneau : i relié à (i+1) mod N
- mélange parfait : i relié à 2i mod (N-1)

Son principal avantage réside dans son degré borné. L'un de ses inconvénients par rapport à l'hypercube est que l'on ne sait pas y plonger un tore ou une grille.

Il existe d'autres familles de réseaux, qui font l'objet de nombreuses études en informatique comme en théorie des graphes. On peut par exemple citer les graphes de Kautz et de De Bruijn [BP].

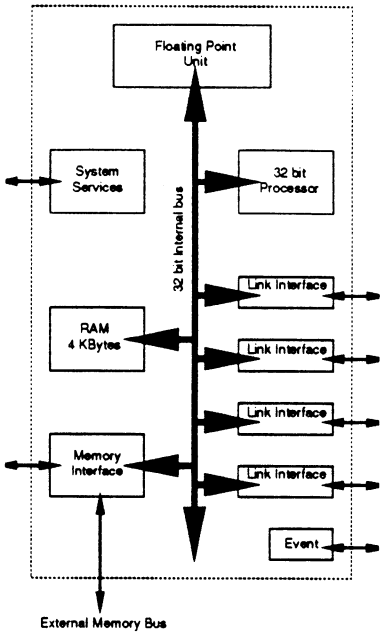
1.2.3. Le Transputer

Dans cette section, nous allons brièvement décrire l'un des processeurs les plus utilisés lors de ces dernières années (du moins en Europe) pour construire des machines à mémoire distribuée. Les algorithmes parallèles présentés dans cette thèse ont tous été implémentés sur réseau de Transputers. Le circuit est conçu pour gérer le parallélisme tant au niveau du processeur lui-même qu'au niveau des communications inter-processeurs. Il intègre par exemple la gestion de processus parallèles exécutés en temps partagé.

Nous commençons par une brève description de l'architecture du T800, la dernière version de Transputers, puis nous donnons un modèle des communications entre processeurs permettant d'évaluer analytiquement les performances d'un algorithme, en tenant compte du temps passé à effectuer des communications.

1.2.3.1. Architecture générale

Succinctement, le T800, est un micro-processeur RISC (Reduce Instruction Set Computer) regroupant sur le même composant plusieurs unités fonctionnelles (que l'on trouve d'habitude proposées en co-processeurs). Parmi les unités regroupées autour du bus interne, le T800 possède 4 liens série bidirectionnels avec DMA (Direct Memory Acces) pour les communications interprocesseurs, une unité rapide pour le calcul en notation flottante et 4Ko de RAM.



Architecture interne du T800

Il est très aisé de construire des machines parallèles à l'aide de Transputers. Deux fils suffisent en effet pour interconnecter deux processeurs de ce type. La seule limitation est que le degré du réseau que l'on peut ainsi former est limité à 4, à moins bien sûr de rajouter du matériel.

On peut par exemple multiplexer les liens pour accroître ce degré, c'est la solution employée dans les hypercubes FPS de la série T.

Certaines machines comme le T-Node, ont un réseau d'inerconnexion reconfigurable. Tout graphe de degré inférieur ou égal à 4 peut être réalisé entre les processeurs. Cette reconfiguration pour l'instant statique doit pouvoir être effectuée de manière dynamique, en cours d'application.

1.2.3.2. Modélisation des communications

Comme nous l'avons signalé ci-dessus, le T800 est équipé de quatre liens bi-directionnels gérés chacun par deux DMA (un pour chaque sens de communication) ce qui autorise le croisement de messages et l'envoi de messages en "parallèle" sur les quatre liens.

Pour modéliser la complexité des algorithmes parallèles, nous avons besoin de modèles réalistes et précis. Des expériences ont été effectuées et sont présentées dans [Des,Ube] pour déterminer ces modèles par interpolation polynomiale.

Pour l'envoi ou la réception de messages de taille L, sur m liens en parallèle nous avons le temps T :

$$T = \beta_m + L.\tau$$

La constante β_m correspond au temps d'initialisation de la communication sur m liens. C'est une fonction affine du nombre de liens utilisés simultanément. Quand on utilise un nombre fixe de liens, le temps de communication est une fonction affine de la longueur du message.

La constante τ, quant à elle, est le temps de transfert élémentaire d'un octet sur sur un lien. C'est l'inverse de ce qu'on appelle la bande passante, mesurée par exemple en Megaoctets par seconde.

1.3. Parallélisation d'algorithmes séquentiels

Nous donnons ici de manière informelle la démarche à employer pour paralléliser un algorithme séquentiel sur une machine à mémoire distribuée. Il ne s'agit que d'une recette, qui ne peut pas toujours s'appliquer de manière systématique et encore moins s'automatiser, mais elle pourra peut-être rendre service à une personne peu familière avec le domaine. Les questions qu'il faut se poser en abordant un problème sont essentiellement :

(1) quels sont les calculs qui peuvent être effectués en parallèle ?

(2) comment les données vont-elles être allouées aux processeurs ?

(3) quelle topologie d'interconnexion adopter entre les processeurs ?

(4) comment les communications vont-elles se dérouler ?

Dans un algorithme, certaines tâches font intervenir les résultats d'autres tâches. Elles doivent être effectuées séquentiellement. Lorsqu'au contraire, deux tâches font intervenir des données indépendantes, elle peuvent être effectuées simultanément par des processeurs distincts. On peut associer à l'algorithme, un graphe de dépendance, dont les sommets sont les tâches à effectuer, et comportant une arète entre les tâches T_1 et T_2 si T_2 utilise des données produites par T_1. Dès lors, deux tâches sont parallélisables, si elles sont incomparables pour l'ordre induit par le graphe de dépendance, c'est à dire s'il n'existe pas de chemin (orienté) les reliant. La réponse à la question (1) ne dépend donc que du problème traité, et non de la machine sur laquelle on veut l'implémenter. En revanche, les autres questions sont très liées à la machine cible, et ne peuvent pas être traitées indépendamment.

Il faut autant que possible allouer à des processeurs distincts, les tâches parallélisables détectées dans le graphe de dépendance.

Pour minimiser le volume de communications, il faut que des tâches adjacentes dans le graphe de dépendance, soient affectées au même processeur, ou à des processeurs peu distants dans le graphe d'interconnexion.

Lorsqu'un processeur a besoin d'une donnée produite par un autre processeur, il faut mettre en oeuvre une communication, entre le plus proche des processeurs connaissant la donnée et son "consommateur". Il se peut en effet que le "producteur" ait déjà transmis cette donnée à d'autres processeurs, plus proche de sa destination que lui-même.

Enfin, pour que la parallélisation soit efficace, il faut équilibrer la charge de calcul entre les différents processeurs, et ne pas laisser inactifs certains processeurs pendant que d'autres travaillent.

Bien sûr, ce ne sont que des principes, et il est souvent difficile de tous les mettre en application. De plus, on n'a pas toujours le choix du réseau d'interconnexion, qui est souvent figé dans une machine donnée. Néanmoins, on peut parfois se contenter de solutions algorithmiques indépendantes de la topologie, les communications interprocesseur étant prises en charge par le système d'exploitation.

Certains systèmes comme Helios ou Trollius, proposent par exemple des macro-instructions de communication plus évoluées que les communications point-à-point entre voisins. On peut citer le routage (envoi d'un message à un processeur distant), la diffusion (envoi du même message à un sous-ensemble ou à tous les processeurs du réseau) ou l'échange total (chaque processeur envoie un message à tous les autres, et donc reçoit un message de tous les autres). On peut écrire un programme parallèle en ne se servant que de ces macro-instructions. Il sera probablement moins performant que si on tient compte de la topologie du réseau. Par compte, il sera plus facilement portable d'une machine à une autre.

D'autres systèmes plus évolués encore, émulent un ordinateur à mémoire partagée. La mémoire locale est paginée, et en cas de faute de page, le système se charge de chercher la page manquante sur le réseau. Des difficultés identiques aux problèmes de cohérence des caches des machines à mémoire partagée apparaissent sur ce type de systèmes. Ils sont néanmoins très performants pour des applications particulières comme le lancer de rayon par exemple, où une grosse base de données est chargée de manière distribuée sur le réseau, mais n'est consultée qu'en lecture.

Pour conclure, nous dirons que certaines applications se contentent très bien de systèmes de haut niveau, indépendants de l'architecture du réseau, de même qu'en informatique classique, certains programmes peuvent être écrits en langage de haut niveau. En revanche, dès lors que l'on a affaire à des tâches très dépendantes entre elles, qui nécessitent un schéma de communications intensif, il faut absolument tenir compte de la topologie de sa machine cible, de même que l'on tient compte de l'architecture d'un circuit en écrivant des applications en langage machine lorsque le temps devient un facteur crucial.

1.4. L'efficacité, un critère de réussite

Le but essentiel de la parallélisation d'un algorithme est bien entendu de diminuer le temps nécessaire à son exécution. Nous avons donc besoin d'un critère permettant de mesurer la qualité d'une parallélisation. C'est pour cette raison que nous allons définir ici les notions de facteur d'accélération et d'efficacité d'un algorithme parallèle.

Soit un algorithme A_1 s'exécutant en le temps T_1 sur un processeur Π. Son implémentation parallèle A_p sur les processeurs $\Pi_1, \Pi_2, \ldots, \Pi_p$, identiques à Π, s'exécute en un temps T_p.

Sous certaines hypothèses, on a toujours $T_1 \leq p.T_p$.

L'argument permettant d'affirmer cela, est que l'on peut simuler les p processus parallèles s'executant sur $\Pi_1, \Pi_2, \ldots, \Pi_p$, de manière séquentielle sur le processeur Π, en temps partagé. Cela donne un algorithme A_1' qui s'exécute en un temps $T_1' \geq T_1$. (sinon, on aurait un algorithme séquentiel A_1' plus performant que l'algorithme A_1, et c'est celui-là que l'on aurait considéré). Chacun des processus de l'algorithme A_p s'exécute en un temps inférieur ou égal à T_p. Leur exécution en temps partagé prend donc un temps inférieur ou égal à $p.T_p$, c'est à dire : $T_1' \leq p.T_p$. On en déduit $T_1 \leq p.T_p$.

On a donc un paramètre assez naturel qui s'impose pour mesurer la qualité d'une parallélisation. Le facteur d'accélération (*speedup*), défini comme le rapport entre le temps séquentiel T_1 et le temps parallèle T_p, qui ne peut en principe excéder p. On peut le normaliser entre 0 et 1, en le divisant par p, on obtient le coefficient de parallélisation ou efficacité (*efficiency*) de l'algorithme parallèle.

Il y a quand même deux remarques importantes à faire :

- On a négligé le temps de contrôle de l'algorithme A_1'. Le changement de contexte entre les processus peut prendre par exemple un temps non négligeable. Les échanges de messages de l'algorithme parallèle doivent être simulés par exemple par des recopies en mémoire, que l'on n'a pas prises en compte. Nous n'avons décompté dans notre argumentation que le temps mis pour effectuer des calculs.

- On ne dispose pas toujours d'assez de mémoire pour simuler les p processus parallèles sur un seul processeur. Il y a des problèmes que l'on peut ainsi résoudre sur une machine parallèle et qui sont impossibles ou très longs à résoudre sur une machine séquentielle. C'est un phénomène auquel nous sommes habitués dans la vie courante. Pour porter une poutre très lourde, par exemple, on peut mettre bien plus que deux fois plus de temps tout seul qu'à deux. On peut même ne pas y arriver du tout.

C'est essentiellement pour ces deux raisons que l'on voit présentés dans des articles de recherche des efficacités d'algorithmes parallèles supérieures à 1.

Deuxième partie : Présentation de la recherche

Dans cette partie, nous passons en revue les problèmes que nous avons abordés, et les raisons pour lesquelles nous nous y sommes intéressés. Il s'agit en général d'algorithmes séquentiels coûteux en temps de calcul et en place mémoire, pour lesquels il est donc très intéressant d'avoir une parallélisation efficace. Nous avons choisi des problèmes dont l'implémentation sur des machines à mémoire distribuée n'est pas aisée, essentiellement à cause de la grande interdépendance entre les différentes tâches composant l'algorithme.

Les deux premières classes de problèmes que nous étudions dans [1,2] sont des algorithmes de programmation dynamique. Ils font intervenir des relations de récurrence, par nature séquentielles, et donc difficiles à paralléliser.

Nous traitons ensuite dans [3] un problème d'algèbre linéaire, pour lequel la stratégie d'allocation des données nous a permis d'obtenir un algorithme parallèle optimal sous certaines hypothèses peu restrictives. Nos expérimentations donnent des facteurs d'accélération approchant de très près le nombre de processeurs utilisés.

La démonstration de l'optimalité du problème précédent passe par un résultat théorique que nous avons établi dans [4] : il s'agit de la complexité de l'opération de distribution d'un vecteur du processeur de tête d'un réseau linéaire à tous les autres. Un algorithme effectuant cette opération de manière efficace était connu depuis longtemps, mais son optimalité n'était pas prouvée.

Nous abordons ensuite dans [5] l'étude de l'algorithme du Z-Buffer, un algorithme de synthèse d'images. Nous l'implémentons sur un

ensemble de Transputers, à réseau d'interconnexion reconfigurable, et étudions l'influence de la topologie du réseau sur les performances de l'algorithme.

Les expérimentations liées au problème précédent nous ont amené à rechercher [6] de manière théorique, quel est le graphe le mieux adapté aux opérations de réduction. Nous utilisons pour cela un modèle de calcul et de communications pour des processeurs du type des Transputers, ayant un nombre de voisins maximal fixé.

2.1. Programmation dynamique

Chaque jour est le premier du reste de ta vie

La programmation dynamique est une technique particulière permettant de résoudre certains problèmes combinatoires. Le terme de programmation dynamique est peu parlant, et c'est probablement pour des raisons de mode que cette technique a été dénommée ainsi par Bellmann. En effet, ses travaux sont intervenus peu après l'époque où la découverte de l'algorithme du simplexe par G.B.Danzig en 1947 ait donné un nouvel élan à la programmation linéaire. Les travaux de Bellmann ont permis de formaliser un principe qui était déjà connu et utilisé à l'époque, qu'il a baptisé principe d'optimalité : lorsque l'on peut appliquer le principe d'optimalité à un problème, on peut résoudre ce problème à l'aide d'une méthode de programmation dynamique. Pour déterminer si un problème combinatoire est justiciable du principe d'optimalité, il faut exprimer sa solution comme une séquence de décisions. Dans une séquence optimale de décisions, quelle que soit la première décision prise les décisions subséquentes doivent former une sous-séquence optimale, compte tenu des résultats de la première décision [Sak]. La difficulté essentielle de la technique de programmation dynamique réside dans la formulation de la solution du problème en terme d'une séquence de décisions à laquelle on peut appliquer le principe d'optimalité.

De l'application du principe d'optimalité, on peut déduire des récurrences permettant de calculer la solution du problème en fonction des solutions de ses sous-problèmes. En résolvant tous les sous-problèmes en commençant par les plus simples, et en conservant dans une table les

résultats intermédiaires, on construit de proche en proche la solution de problèmes de plus en plus complexes, pour finir avec la solution du problème initial.

Dans l'article [1] nous nous intéressons à une classe particulière de problèmes dont la récurrence de programmation dynamique s'exprime de la même manière. Nous allons donner deux exemples de tels problèmes.

2.1.1. Produit chaîné de matrices.

Etant données n matrices M_0, M_1, ... , M_{n-1} dont on désire calculer le produit chaîné. La matrice M_i est de dimension $r_i \times r_{i+1}$ ($0 \leq i < n$) de telle sorte que le produit existe. Si on utilise l'algorithme classique pour effectuer le produit de deux matrices, le nombre de multiplications scalaires à effectuer pour calculer le produit d'une matrice à p lignes et q colonnes par une matrice à q lignes et r colonnes est p.q.r.

Puisque le produit de matrices est associatif, on peut parenthéser le produit chaîné d'un grand nombre de manières différentes. A chacune des parenthésations correspond un nombre de multiplications scalaires à effectuer. Le problème peut se formuler de la manière suivante : quel est le nombre minimal de multiplications scalaires à effectuer pour calculer le produit chaîné, et quelle est la parenthésation optimale associée ?

Le principe d'optimalité s'applique à ce problème, quand on remarque que dans la parenthésation optimale d'une chaîne de matrices, les sous-chaînes sont elles-même parenthésées de manière optimale. En effet, si on pouvait améliorer le coût d'une sous-chaîne, (c'est à dire le nombre de multiplications scalaires à effectuer pour en calculer le produit), le coût total en serait amélioré d'autant. De cette constatation, on peut déduire un algorithme de programmation dynamique qui consiste à calculer le coût optimal et la parenthésation associée de toutes les sous-chaînes de la chaîne initiale. On commence par celles de longueur 2, puis 3, et ainsi de suite jusqu'à celle de longueur n qui est la chaîne initiale.

Calculons le coût c(i,j) de la chaîne allant de la matrice i à la matrice j ($0 \leq i < j < n$). Si le dernier produit effectué dans la parenthésation optimale de la chaîne $M_i.M_{i+1}...M_j$ est la multiplication de la matrice $M_i.M_{i+1}...M_k$ (r_i lignes et r_{k+1} colonnes) par la matrice $M_{k+1}.M_{k+2}...M_j$ (r_{k+1} lignes et r_{j+1} colonnes), alors, c(i,j) est égal au coût optimal des deux sous-chaînes c(i,k) + c(k+1,j) plus le nombre de multiplications scalaires

nécessaires pour le dernier produit, c'est à dire $r_i \cdot r_{k+1} \cdot r_{j+1}$. On en déduit la récurrence de programmation dynamique en minimisant cette expression pour k compris entre i et j :

$$c(i,i) = 0$$

$$c(i,j) = \min_{i \le k < j} \{c(i,k) + c(k+1,j) + r_i \cdot r_{k+1} \cdot r_{j+1}\} \qquad \text{si } 0 \le i < j < n$$

Le coût du produit optimal est donné par c(0,n-1). Pour obtenir la parenthésation optimale, il faut stocker en plus du coût c(i,j), l'indice k(i,j) qui réalise le minimum sur k lors du calcul de c(i,j). Une fois la première phase de calcul terminée, on suit la récurrence en sens inverse pour obtenir la parenthésation optimale en un temps linéaire.

2.1.2. Arbres binaires de recherche

Etant données des clés de recherche $K_0 \le K_1 \le ... \le K_{n-1}$ et leur fréquence d'apparition estimée $f_0, f_1, ..., f_{n-1}$, on désire construire un arbre binaire de recherche minimisant le temps moyen d'accès à une clé. Les noeuds de l'arbre sont étiquetés avec les clés, et les clés du sous-arbre gauche d'un noeud sont inférieures à la clé de ce noeud alors que les clés du sous-arbre droit lui sont supérieures. Le temps d'accès a une clé dans l'arbre est proportionnel à sa profondeur à partir de la racine, et le temps moyen d'accès à une clé est égal à la somme des temps d'accès pondérés par les fréquences d'apparition.

Ce problème est lui aussi justiciable du principe d'optimalité puisque tout sous-arbre d'un arbre optimal est lui-même optimal : si on pouvait diminuer le coût d'un sous-arbre (son temps d'accès moyen), le coût de l'arbre complet serait diminué.

La récurrence formalisant ce problème est identique à celle du produit chaîne de matrices : soit c(i,j) le coût de l'arbre optimal faisant intervenir les clés K_i à K_{j-1}. c(i,j) peut être calculé à l'aide des coûts des arbres plus petits : On suppose que l'on place la clé K_k à la racine ($i \le k < j$), avec pour sous-arbre gauche l'arbre optimal faisant intervenir les clés K_i à K_{k-1} qui est de coût c(i,k) et pour sous-arbre droit l'arbre optimal faisant intervenir les clés K_{k+1} à K_{j-1} qui est de coût c(k+1,j). Le coût du nouvel arbre est égal au coût des deux sous-arbres plus la somme des fréquences estimées des clés i à j-1.

$$c(i,i) = 0$$

$$c(i,j) = \min_{i \le k < j} \left\{ c(i,k) + c(k+1,j) + \sum_{m=i}^{j-1} f_m \right\} \text{, avec } 0 \le i < j \le n$$

Le coût de l'arbre optimal est donné par c(0,n). Pour obtenir l'arbre lui-même, il faut à nouveau stocker en plus du coût c(i,j), l'indice k(i,j) qui réalise le minimum sur k lors du calcul de c(i,j). L'arbre s'obtient lui aussi en suivant la récurrence à l'envers, dans une deuxième phase.

2.1.3. Parallélisation sur un anneau de processeurs

Nous nous sommes intéressés dans [1], à la parallélisation pour une machine à mémoire distribuée du calcul de la récurrence suivante :

$$c(i,i) = 0$$

$$c(i,j) = \min_{i \le k < j} \{ c(i,k) + c(k+1,j) + f(i,j,k) \} \text{ , avec } 0 \le i < j < n$$

C'est une récurrence générique qui pourra donc servir à la résolution des deux problèmes cités ci-dessus. Nous n'avons mis en oeuvre que la résolution de la récurrence (c'est à dire le calcul de c(0,n-1)) et non la deuxième phase qui permettrait de déterminer la solution optimale en suivant "à l'envers" la relation de récurrence.

L'algorithme séquentiel consiste en un calcul des coûts c(i,j), diagonale par diagonale :

```
algorithme prog_dyn ;
   pour d de 1 à n-1
     pour i de 0 à n-d
       j := i + d ;
       inf := + infini ;
       pour k de i à j-1
          inf := min(inf,c(i,k)+c(k+1,j)+f(i,j,k));
       fpour ;
       c(i,j) := inf ;
     fpour ;
   fpour ;
```

La remarque essentielle que l'on peut faire en examinant la récurrence, est que les calculs des coûts d'une même diagonale sont indépendants entre eux. Autrement dit, si les d-1 premières diagonales

sont disponibles, les coûts de la $d^{ème}$ peuvent être calculés en parallèle. On devra donc autant que possible, affecter à des processeurs distincts, le calcul des coûts d'une même diagonale.

La deuxième remarque que l'on peut faire est la suivante : pour calculer le coût c(i,j), on a besoin de tous les coûts c(i,k) situés sur la même ligne que c(i,j) et de tous les coûts c(k+1,j) situés sur la même colonne que c(i,j) (avec $i \leq k < j$). Afin de faciliter le schéma de calcul, nous avons alloué tous les coûts d'une colonne donnée au même processeur. On appellera Alloc(j) le numéro du processeur (compris entre 0 et p-1), auquel on alloue la colonne j. On suppose ici que la taille n du problème à résoudre, est supérieure au nombre de processeurs p. Avec cette allocation par colonne, la moitié des données dont un processeur a besoin à chaque étape est présente dans sa mémoire puisqu'elle a été calculée par lui aux étapes précédentes. Pour l'autre moitié, c'est à dire pour les coûts c(i,k) situés sur la même ligne, on remarque que le processeur qui à l'étape précédente a calculé c(i,j-1), situé sur la colonne voisine, en avait besoin lui aussi. Si ce processeur est le même, c'est à dire si Alloc(j-1) = Alloc(j), il n'y a pas de communication à faire. Sinon, le processeur Alloc(j-1) doit transmettre au processeur Alloc(j) le coût c(i,j-1) qu'il vient de calculer, et les coûts c(i,k) $i \leq k < j-1$ dont il disposait pour calculer c(i,j-1). Il faudra donc autant que possible que les processeurs auxquels sont allouées des colonnes voisines soient des processeurs voisins dans le réseau, pour que cette transmission soit limitée à une seule communication. Deux solutions classiques pour résoudre cette contrainte de localité sont souvent envisagées [GH, MV,Saa] :

- configurer le réseau en anneau de p processeurs, et allouer les colonnes d'indice q modulo p au processeur P_q. C'est la répartition par colonnes entrelacées (wrap repartition).
- configurer le réseau en réseau linéaire de p processeurs, et allouer des blocs de n/p colonnes consécutives aux processeurs consécutifs. C'est la répartition par blocs complets (full-block repartition).

Dans notre cas, ni l'une ni l'autre de ces deux stratégies d'allocation n'est satisfaisante :

- l'allocation par colonnes entrelacées nécessite un volume de communication trop important. On a vu en effet qu'une communication est nécessaire pour le calcul du coût c(i,j) par le processeur Alloc(j), si la

colonne j-1 est allouée à un processeur différent. Avec cette stratégie d'allocation, c'est le cas pour chaque j ;

- l'allocation par blocs complets propose un schéma de communications plus réduit, puisque lorsque deux colonnes successives sont allouées au même processeur, on n'a pas besoin de communication. On n'a plus que p-1 "frontières" à franchir au lieu de n-1. En revanche, cette stratégie introduit un déséquilibre entre les processeurs : étant donnée la forme triangulaire de la matrice des coûts, le premier processeur du réseau se voit allouer bien moins de coûts que le dernier, et termine sa tâche avant les autres, ce qui est une perte de parallélisme.

Nous proposons donc une nouvelle répartition des données, plus générale que les deux précédentes et qui permet de trouver un compromis entre les deux exigences contradictoires : réduire au maximum le volume de communication pour ne pas pénaliser l'algorithme parallèle, et équilibrer le plus possible les calculs entre les processeurs, pour ne pas laisser inactifs des processeurs pendant que d'autres continuent à calculer. Sur un anneau de p processeurs, nous allouons des blocs de r colonnes consécutives aux processeurs consécutifs, et nous entrelaçons les blocs ainsi formés. Analytiquement, on a donc $Alloc(j) = \lfloor j/r \rfloor \bmod p$. La taille des blocs r est un paramètre de notre algorithme parallèle, dont nous aurons à déterminer la valeur optimale. Remarquons que les deux stratégies précédentes sont des cas particuliers de l'allocation par blocs de taille r, correspondant aux valeurs $r = 1$ et $r = n/p$ respectivement.

Nous évaluons analytiquement dans [1] le temps d'exécution de l'algorithme parallèle en fonction des différents paramètres :

- n, la taille du problème à résoudre ;
- p, le nombre de processeurs dont on dispose ;
- r, la taille des blocs pour l'allocation par blocs entrelacés ;
- ρ, le rapport τ_c / τ_a , avec :
  - τ_c, le temps de communication élémentaire, correspondant au transfert d'un coût entre deux processeurs voisins ;
  - τ_a, le temps arithmétique élémentaire, correspondant à une mise à jour dans la récurrence.

Nous obtenons le temps parallèle $T_{//}$ comme la somme d'un temps arithmétique T_A, et d'un temps de communication T_C. Nous montrons que

T_A est une fonction strictement décroissante de r : plus les blocs sont importants, moins l'arithmétique est équilibrée entre les processeurs. T_C est au contraire une fonction strictement croissante de r : plus les blocs sont gros, moins il y a de frontières entre blocs, et donc moins il y a de communications dans l'algorithme parallèle. En figure 1, nous représentons T_A, T_C et $T_{//}$ en fonction de r, avec n et p fixés.

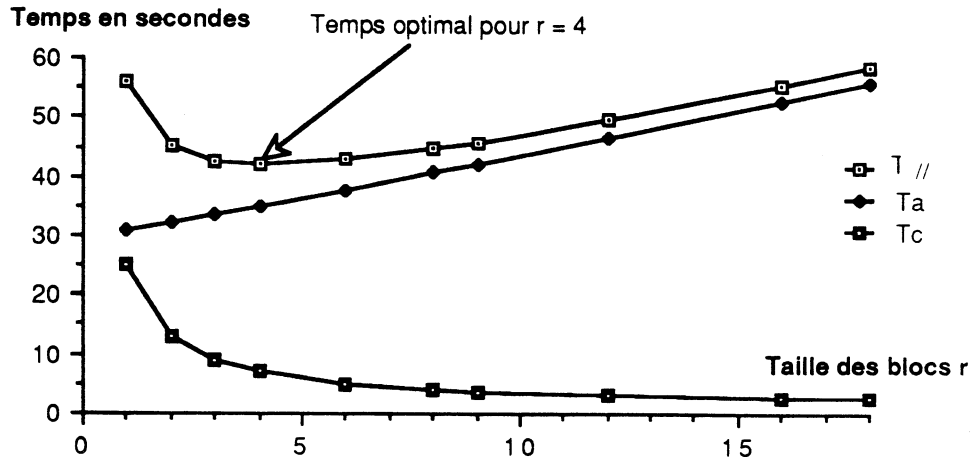


Figure 1 : temps parallèle en fonction de la taille des blocs
n = 576 ; p = 16 ;

On voit qu'il existe comme prévu une valeur optimale pour la taille des blocs, qui ne correspond ni à l'allocation par colonnes entrelacées, ni à l'allocation par blocs complets. Cette valeur de r peut être calculée en fonction de la taille du problème et des paramètre propres à la machine utilisée. Une formule approchée valable pour des problèmes de grande taille est très simple à exprimer :

$$r_{opt} = \sqrt{\frac{2\,\rho}{3\,\alpha}}$$

où ρ est le rapport entre le temps de communication élémentaire τ_c et le temps arithmétique élémentaire τ_a, et α est le rapport entre le nombre de processeurs p et la taille du problème n. Ce résultat est essentiel, dans la mesure où il permet de prévoir la répartition optimale pour cette classe d'algorithmes parallèles, dès que l'on connaît le rapport communication/arithmétique de la machine parallèle utilisée.

2.2. Algorithmes de balayage d'image

Dans le chapitre précédent, nous avons étudié la parallélisation de l'une des récurrences les plus classiques de programmation dynamique. Mais il en existe bien d'autres, qui ont des applications en robotique ou en intelligence artificielle. Nous nous sommes intéressés dans l'article [2] à l'une d'entre elles, ayant de nombreuses applications en traitement d'images. La version séquentielle est basée sur des balayages successifs d'une image, au cours desquels chaque pixel est mis à jour selon un masque appliqué à son voisinage. Deux applications importantes sont le calcul de la transformée en distance et la détermination de la trajectoire optimale, que nous allons détailler maintenant*. Dans la suite, les huit voisins d'un pixels p seront dénotés par les points cardinaux : N,E,S et W pour les quatre plus proches, et NE, SE, SW et NW pour les quatre autres.

2.2.1. Transformée en distance

Soit P une image binaire composée d'une figure F = {∞}, entièrement entourée par son complémentaire P\F = {0}. La transformée en distance (*distance transform*, DT) de de l'image P est une copie de P, dans laquelle on associe à chaque pixel sa distance (selon une certaine métrique) à P\F. Les pixels de P\F gardent leur valeur 0 et les pixels de F prennent pour valeur la longueur d'un plus court chemin menant à P\F. Dans l'image DT, chaque pixel peut être associé à un disque centré en ce pixel : le rayon du disque dépend de la valeur du pixel, tandis que la forme du disque dépend de la métrique choisie. Un disque est dit maximal si aucun autre disque ne le recouvre totalement. Comme l'union des disques maximaux coïncide avec F, l'axe médian, défini comme le lieu des centres des disques maximaux (les points de DT dont la valeur est un maximal local), joue un rôle important pour diminuer l'occupation mémoire et pour divers algorithmes d'analyse et de description des formes.

Pour ce calcul de transformée en distance, un algorithme de programmation dynamique bien connu consiste à balayer l'image deux fois. On effectue d'abord une passe avant, en balayant l'image de haut en

---

* D'autres applications telles que le calcul d'images mosaïques à partir d'images SPOT de deux régions adjacentes avec recouvrement entre les deux images, font intervenir la même récurrence [DP].

bas et de gauche à droite et en utilisant un masque en chevron (le voisin de gauche et les trois voisins du haut) :

$p := \min(p, W + t_1, NW + t_2, N + t_1, NE + t_2)$

Le principe de mise à jour est très simple : la distance de p au contour est le minimum de sa valeur actuelle et des valeurs des voisins du masque en chevron, incrémentées de leur distance à p. Les coefficients t_1 et t_2 sont les poids choisis pour les voisins horizontaux/verticaux et pour les voisins diagonaux respectivement. Pour $t_1 = 1$ et $t_2 = \infty$ on a la distance de Manhattan d4. Pour $t_1 = t_2 = 1$ on retrouve la distance de l'échiquier d_8. Pour $t_1 = 3$ et $t_2 = 4$ on a la distance $d^* = 2d_8 + d_4$, qui est une bonne approximation de la distance Euclidienne.

Le balayage arrière est similaire au précédant. On balaye l'image de bas en haut et de droite à gauche, à l'aide du masque en chevron symétrique, et en en appliquant la récurrence arrière :

$p := \min(p, E + t_1, SE + t_2, S + t_1, SW + t_2)$

Après les deux passes, chaque point est remplacé par sa distance à P\F. Plusieurs applications du calcul de la transformée en distance sont décrits dans [YBR]. Elle permet entre autres de déterminer des propriétés géométriques comme la surface, le contour et le périmètre. Montanvert [Mon] montre comment traiter l'ensemble des maxima locaux pour déterminer la ligne médiane (qui est une extension connexe de l'axe médian). La figure 2 ci-dessous illustre les deux passes de traitement sur une lune, avec la distance de l'échiquier d_8.

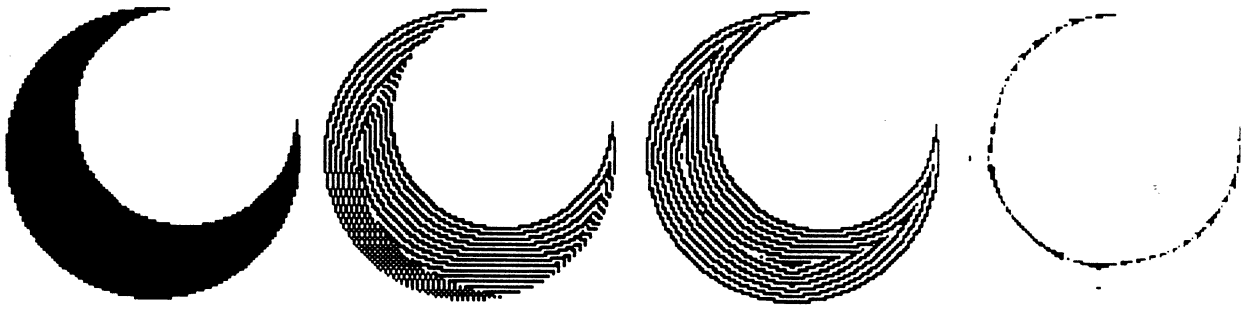


Figure 2: Détermination de l'axe médian d'une lune par calcul de la transformée en distance

2.2.2. Calcul de la trajectoire optimale

Le calcul de la trajectoire optimale s'effectue également à l'aide de passes successives sur une carte dont les points sont caractérisés par un coût de traversabilité [BK]. Il s'agit de déterminer un chemin de coût minimal d'un point donné de la carte (la source) à tous les autres points.

Sur la carte P de taille n x n, on associe à chaque point p un réel positif ou nul tc(p) correspondant au coût de la traversabilité en p. Etant donné un point p et un voisin q de p, l'arète (p,q) est pondérée par le coût :

$$c(p,q) = \frac{tc(p)+tc(q)}{2} \qquad \text{si } q \in \{N,S,W,E\}$$

$$c(p,q) = \frac{tc(p)+tc(q)}{\sqrt{2}} \qquad \text{si } q \in \{NW,NE,SW,SE\}$$

Le coefficient $\sqrt{2}$ reflète la distance supplémentaire due à la connexion diagonale. Etant donné un point particulier appelé la source, on veut calculer le plus court chemin (ou chemin de poids minimal) de la source à tous les autres points de la carte.

Bitz et Kung [BK] proposent l'algorithme suivant : initialement, le meilleur coût connu f(p) pour tout point p est initialisé à la valeur 0 à la source et $+\infty$ en tous les autres points. L'algorithme effectue une succession de passes avant et arrière, en utilisant le même masque en chevron que pour la transformée en distances. Pour la valeur courante du point p, on met à jour le meilleur coût connu f(p) s'il existe un meilleur chemin passant par l'un des voisins sélectionnés par le masque de balayage. Par exemple lors d'une passe avant, si le meilleur coût connu f(W) du voisin Ouest de p plus le coût c(W,p) de l'arète qui relie W à p est inférieur à f(p), alors on met à jour f(p), i.e. f(p) := f(p) + c(W,p). Dans le cas général, les formules de mise à jour sont les suivantes :

pour la passe avant,

f(p) := min(f(p), f(W) + c(W,p), f(NW)+ c(NW,p),
f(N) + c(N,p), f(NE)+ c(NE,p))

et pour la passe arrière,

f(p) := min(f(p), f(E) + c(E,p), f(SE)+ c(SE,p),
f(S) + c(S,p), f(SW)+ c(SW,p))

Etant données les valeurs initiales spécifiées plus haut, les passes avant et arrière sont effectuées successivement, jusqu'à ce qu'aucune valeur de f(p) ne change au cours d'une passe.

Nous avons construit un exemple (artificiel) dans la figure 3. L'altitude d'un point est proportionnelle à sa traversabilité. Ainsi la spirale est-elle de traversabilité élevée, tandis que le fond de vallée est de traversabilité faible. La source est au centre. On voit bien que la trajectoire optimale "hésite" entre emprunter les cols (chemin plus court mais coûteux) ou cheminer longuement dans le fond de vallée.

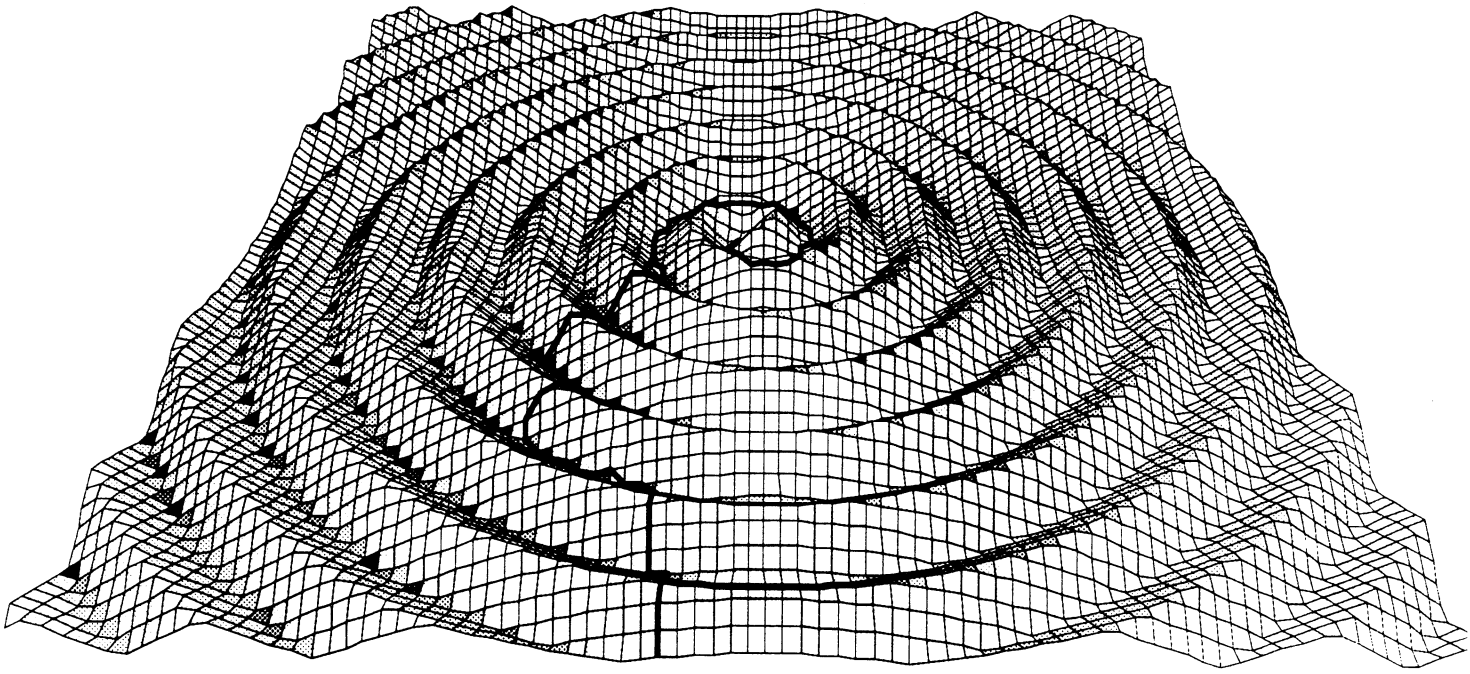


Figure 3 : Algorithme de trajectoire optimale.

2.2.3. Implémentation parallèle

Nous nous sommes intéressés dans [2], à la parallélisation sur un anneau de processeurs de l'algorithme générique de balayage d'images, permettant entre autres de résoudre les deux problèmes décrits ci-dessus. Nous nous limitons à l'étude de la passe avant, à cause de la symétrie des deux passes.

Dans ce qui suit, nous considérons un anneau de p processeurs numérotés de 0 à p-1. Chaque processeur accède à sa propre mémoire, et communique avec ses deux voisins par un protocole d'échange de messages : le processeur P_i échange des messages avec P_{i-1} et P_{i+1} (Les

numéros de processeurs sont pris modulo p). Nous allouons des lignes de l'image aux processeurs, tout le problème étant de trouver une distribution qui équilibre la charge de calculs entre les processeurs, sans dégrader les performances par des temps de communication trop importants.

L'algorithme parallèle proposé par [BK] pour le calcul de la trajectoire optimale, est destiné à s'exécuter sur la machine Warp. Ils utilisent un schéma de calcul "glouton" où les processeurs transmettent au plus tôt à leurs voisins les pixels calculés. Nous allons présenter cet algorithme qui minimise le temps de mise en en route et conduit très rapidement a des calculs bien équilibrés entre les processeurs. Puis nous expliquerons pourquoi cette implémentation n'est pas du tout adaptée à un anneau de processeurs usuels tels que les Transputers, à cause du coût prohibitif des communications par rapport aux calculs. Finalement, nous présentons une version modifiée, beaucoup plus performante dans notre cadre.

2.2.3.1. L'algorithme glouton

Supposons tout d'abord, que le nombre de processeurs p, est égal a la taille du problème n. Dans ce cas, la ligne i de l'image, est allouée au processeur i ($0 \leq i < n$). Pour le balayage avant, dès que le processeur i a calculé la valeur d'un pixel, il la transmet au processeur i+1, et calcule le prochain pixel de sa ligne. Notons bien que pour commencer une ligne, un processeur a besoin de deux valeurs de la ligne précédente, et ne peut donc commencer le calcul de sa ligne que deux étapes après son voisin.

A l'instant 2i+j, le processeur P_i effectue donc les opérations suivantes (partout où les indices ont un sens):

- Il recoit le pixel (i-1,j+1) de P_{i-1}
- Il calcule le pixel (i,j)
- Il transmet (i,j) à P_{i+1}

Quand p est inférieur à n, des techniques de partitionnement doivent être mises en œuvre. Supposons pour la clarté de l'exposé, et sans perte de généralité, que p divise n. Afin de permettre aux processeurs de débuter leurs calculs le plus tôt possible, une solution consiste à allouer les lignes de l'image aux processeurs de manière entrelacée : la ligne j est

allouée au processeur j mod p. Avec cette répartition, P_0 a besoin de valeurs calculées par P_{p-1}. Notons que P_0 reçoit le pixel (p-1,0) au temps 2p-1. Au temps 2p, P_0 reçoit le deuxième pixel (p-1,1) de la ligne p-1, et calcule le pixel (p,0). Donc, il ne faut pas que P_0 ait fini de calculer sa première ligne avant l'instant 2p, sinon, il serait inactif, en attente des valeurs transmises par P_{p-1}. Cela entraîne que $n \geq 2p$. Lorsque $n > 2p$, P_0 stockera simplement les pixels calculés par P_{p-1}, en finissant sa première ligne, puis utilisera les valeurs stockées pour calculer sa deuxième ligne.

Nous voyons que le délai entre deux processeurs voisins est petit (deux unités de temps). L'inconvénient majeur de l'algorithme est qu'il nécessite de nombreuses communications de petite taille. Pour les machines parallèles usuelles, le temps de transfert de L mots entre deux processeurs voisins, peut être modélisé par $\beta + L\,\tau$, où β est le temps d'initialisation de la communication, indépendant de sa longueur, et τ le temps de communication élémentaire. Il apparait que pour les machines actuelles, β est souvent plus important que τ, ce qui rend prohibitif le coût des messages courts.

Dans la suite, nous expliquons comment modifier l'algorithme glouton de manière à diminuer le coût des communications. Nous commençons par une description informelle du nouvel algorithme, et reportons l'analyse de sa complexité à la section suivante.

2.2.3.2. Augmenter la granularité de l'algorithme

La première technique utilisée pour diminuer le coût global des communications, est d'utiliser de plus longs messages. Nous adoptons la même allocation des données que précédemment, mais nous calculons k pixels consécutifs à chaque étape. Supposons qu'à une étape de l'algorithme, le processeur q calcule sur la ligne i, les pixels compris entre les colonnes j+1 et j+k. Après ce calcul le processeur q transmet ce segment à son successeur q+1 dans l'anneau qui peut calculer à l'étape suivante un segment sur la ligne i+1. Ce segment est nécessairement décalé d'un pixel vers la gauche par rapport au segment précédent. En effet, le processeur q+1 ne peut calculer sur la ligne i+1 que les pixels compris entre les colonnes j et j+k-1, puisqu'il a besoin du pixel (i,j+k+1) pour calculer le pixel (i+1,j+k). A chaque étape, sauf la première et éventuellement la dernière, chaque processeur calcule un segment de k

pixels consécutifs. Lorsque le segment déborde de la frontière d'une ligne, on termine la ligne courante et on débute la ligne suivante au cours de la même étape. A nouveau, le processeur 0 qui calcule la première ligne de l'image a un rôle particulier : au début, il est le seul à ne pas recevoir un segment avant d'en calculer un. Ensuite, il reçoit des segments du processeur p-1, mais les stocke en continuant les calculs sur sa première ligne avant de les utiliser.

Le nombre de pixels communiqués entre deux voisins est exactement le même que précédemment, mais plus k est grand, moins les communications sont coûteuses. D'un autre côté, plus k est grand, plus le délai d'attente entre deux processeurs adjacents est grand. Il nous faut donc trouver un compromis entre deux exigences contradictoires, à savoir un délai d'initialisation minimal (petit k), et des communications peu coûteuses (grand k).

2.2.3.3. D'autres stratégies d'allocation

Un autre moyen de diminuer le coût des communications est de diminuer leur volume total, en échangeant moins d'informations entre processeurs voisins. On considère des fonctions d'allocation plus générales que l'allocation entrelacée des lignes, en attribuant à un processeur, des blocs de r lignes consécutives, et en entrelaçant les blocs autour de l'anneau. C'est l'allocation par blocs de lignes entrelacés, similaire à celle que nous avons utilisée pour la programmation dynamique dans le premier chapitre. La figure 4 illustre une étape de l'algorithme parallèle. Le processeur q calcule maintenant r segments de k pixels à chaque étape, sans effectuer de communication. Les segments successifs sont à nouveau décalés entre eux d'un pixel vers la gauche. Après le calcul du "parallélogramme" ainsi formé, le processeur q transmet le dernier segment calculé à son successeur q+1, qui peut calculer un nouveau parallélogramme à l'étape suivante. A chaque étape, sauf la première et éventuellement la dernière, chaque processeur calcule un parallélogramme de r*k pixels. Lorsque celui-ci déborde de la frontière d'un bloc de lignes, on termine le bloc courant et on débute le bloc suivant au cours de la même étape.

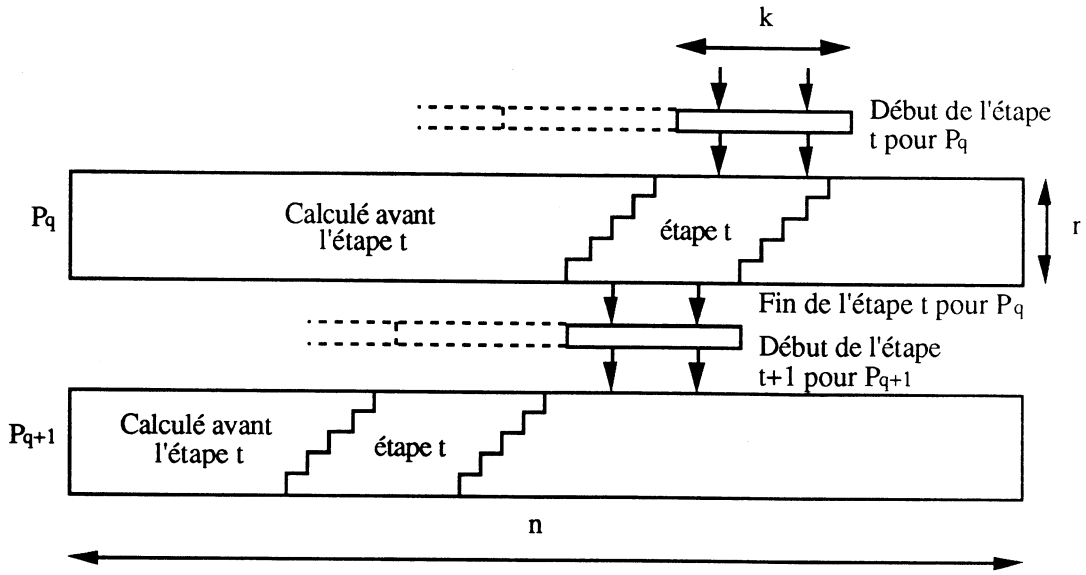


Figure 4 : Communications entre deux processeurs voisins.

Avec cette répartition des données, le volume total des communications entre deux processeurs voisins est r fois plus petit qu'avec la stratégie précédente, car les processeurs n'ont besoin d'échanger que les informations relatives aux lignes frontières entre blocs. Les segments appartenant aux lignes internes d'un bloc, ne requièrent pas de communications inter-processeurs.

Le prix à payer pour cette importante réduction du volume de communication, est encore l'augmentation du délai d'initialisation entre deux processeurs voisins. A nouveau, la meilleure valeur de r résultera d'un compromis, tout comme la meilleure valeur de k.

Dans l'article [2], nous analysons la complexité de l'algorithme parallèle décrit ci-dessus. Nous déterminons de manière analytique en fonction de n et p, les valeurs de k et de r qui minimisent le temps d'exécution. Cette évaluation se fait de la manière suivante :

- calcul de la durée d'une étape $T_{step}(r,k)$
- calcul du nombre d'étapes $N_{step}(r,k)$
- détermination des valeurs admissibles pour les paramètres k et r
- minimisation du temps parallèle $T_{//}(r,k) = T_{step}(r,k) * N_{step}(r,k)$ pour k et r appartenant au domaine admissible.

La comparaison des résultats expérimentaux et théoriques est très satisfaisante et valide donc notre modèle d'évaluation. Nous obtenons des

facteurs d'accélération allant jusqu'à 26 sur un anneau de 32 processeurs, ce qui représente une efficacité de 81% de l'algorithme parallèle.

Pour comparaison, nous donnons le temps d'exécution d'une passe de l'algorithme sur deux machines séquentielles et sur le FPS T40, un hypercube de 32 processeurs, dont nous utilisons la sous-topologie anneau. La taille du problème diffère d'une machine à l'autre, puisqu'il est impossible de résoudre des problèmes de très grande taille sur des machines conventionnelles. C'est pourquoi nous normalisons le temps de calcul par le nombre de pixels de l'image, afin d'obtenir un temps moyen de mise-à-jour d'un pixel.

Machine	n	t (secs)	t / n^2(ms)
Macintosh SE	170	6.40	221
Sun 3/260	512	11.45	44
FPS-T40	1920	9.05	2.4

Table 1: Temps d'exécution normalisés sur différentes machines

Finalement, la figure 5 ci-dessous représente les courbes de niveau de l'efficacité en fonction des deux paramètres r et k, à n et p fixés. On voit que les valeurs optimales des paramètres ne correspondent pas à des valeurs triviales. Nous n'en avons pas d'expression simple comme pour la programmation dynamique. On peut néanmoins en avoir les valeurs numériques par simulation, en minimisant l'expression de $T_{//}(r,k)$.

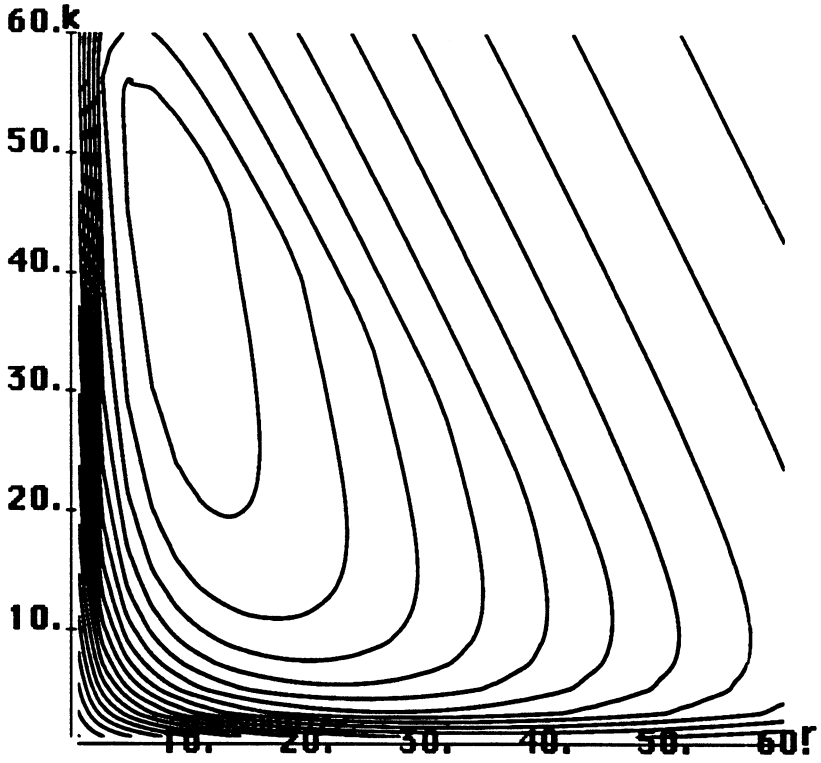


Figure 5 : Courbes de niveau de l'efficacité e(r,k)

2.3. Produit d'une matrice symétrique par un vecteur

Nous étudions dans l'article [3], l'implémentation d'un problème d'algèbre linéaire sur une machine parallèle à mémoire distribuée. Il s'agit du produit d'une matrice que l'on sait symétrique par un vecteur. Ce problème a des applications en robotique où une matrice de variance-covariance est utilisée pour mettre à jour une estimation de paramètres. La matrice en question a une dimension pouvant aller jusqu'à 4000 et son produit par un vecteur est au centre d'un algorithme itératif. Ce calcul doit absolument être effectué rapidement, et si possible en temps réel, pour pouvoir faire prendre au robot les décisions qui s'imposent connaissant sa position estimée. L'intérêt du parallélisme pour accélérer le processus prend tout son sens dans ce type d'applications.

2.3.1. L'algorithme standard

La première solution à laquelle on peut penser pour traiter ce problème est d'avoir recours à l'algorithme standard de multiplication d'une matrice A par un vecteur X sur une machine à mémoire distribuée : la matrice de dimension n est stockée par blocs complets de lignes, sur un anneau de p processeurs. Chaque processeur stocke dans sa mémoire un bloc de n/p lignes consécutives de A, et les n/p composantes correspondantes de X. Sans aucune communication, les processeurs peuvent effectuer des "bouts" de produit scalaire en multipliant les blocs diagonaux de A par les composantes de X dont ils disposent. Les processeurs font alors tourner X d'une position dans l'anneau. Ils envoient leur composantes du vecteur à leur successeur et recoivent de leur prédécesseur les composantes suivantes. Une nouvelle phase de calcul peut alors commencer. En p étapes de calcul et p-1 communications, le produit est calculé, et stocké de manière distribuée sur l'anneau.

Dans le cas d'une matrice symétrique, cet algorithme a cependant deux inconvénients majeurs :

1) Il double presque la place mémoire nécessaire au stockage de la matrice. En effet, l'élément $a_{i,j}$ étant égal à l'élément $a_{j,i}$, il est inutile de le stocker deux fois.

2) Dans le cadre de notre problème de robotique, la matrice est mise à jour à chaque itération. Le fait que l'information soit dupliquée pose alors un nouveau problème :

- soit le calcul de la nouvelle matrice est fait sur tous les éléments stockés. Les erreurs d'arrondi même en double précision font qu'au bout d'un certain nombre d'itérations, la matrice perd sa symétrie ;
- soit on ne fait la mise à jour que sur une moitié de la matrice, par exemple la partie triangulaire inférieure, mais il faut alors communiquer les nouveaux éléments aux processeurs sensés contenir la partie triangulaire supérieure de la matrice. C'est une perte de temps inutile.

2.3.2. Allocation par réflexion

Nous proposons dans [3] une version de l'algorithme où seule la partie triangulaire inférieure de la matrice est stockée. Nous nous affranchissons donc des deux problèmes mentionnés ci-dessus, et obtenons un temps de calcul identique à celui de l'algorithme standard. L'idée de base de la

méthode repose dans l'allocation des données aux processeurs. La matrice étant triangulaire, il est judicieux pour équilibrer l'occupation mémoire et la charge de calculs, d'allouer au même processeur, la première et la dernière ligne de la matrice, la deuxième et l'avant-dernière, et ainsi de suite. Plus précisément, on suppose n divisible par 2p et on pose r=n/(2p). On alloue au processeur q ($0 \leq q < p$), le $q^{ème}$ et le $(2p-q-1)^{ème}$ bloc de r lignes consécutives de la partie triangulaire inférieure de A, et les composantes correspondantes de X. Le $i^{ème}$ bloc est composé des lignes r.i, r.i+1, ... r.i+r-1. Cette répartition des données est connue sous le nom d'allocation par réflexion (*reflexion mapping*).

Pour le déroulement des calculs, il faut remarquer que les processeurs ont dans leur mémoire, des morceaux des lignes de la matrice. Ils peuvent donc comme dans l'algorithme initial, calculer des "bouts" de produit scalaire, sachant que les composantes de X vont tourner le long de l'anneau de la même manière que précédemment. Par compte, les éléments de A qui leur manquent pour calculer leurs composantes du produit, se trouvent répartis sur les autres processeurs de l'anneau. Réciproquement, ils détiennent des éléments de A qui doivent servir au calcul d'autres composantes du produit que les leurs.

Le principe schématique de l'algorithme vu par un processeur est le suivant : je commence par travailler pour le processeur le plus éloigné de moi. Je fais ensuite tourner le résultat de ce calcul, en même temps que mes composantes de X. Le résultat "se rapproche" donc de sa destination. A la deuxième étape, je travaille pour le processeur qui précède le plus lointain, et je combine le résultat avec celui que je viens de recevoir, qui est destiné au même processeur, puisque c'est le plus lointain pour mon prédécesseur... En procédant de cette manière, je travaille p-1 étapes pour les autres processeurs, et à chacune de ces p-1 étapes, l'un des processeur travaille pour moi. Un bel exemple de solidarité !

2.3.3. Résultats d'optimalité

Dans l'article [3], nous établissons que cet algorithme est optimal dans plusieurs cas :

- avec une machine pour laquelle les communications ne peuvent pas avoir lieu en même temps que les calculs, il est optimal parmi la classe des algorithmes parallèles sur un anneau orienté de processeurs, qui allouent le même nombre de lignes de la matrice aux différents processeurs. La contrainte de l'allocation des données est bien naturelle et

nous voyons mal comment les performances de notre méthode pourraient être améliorées par un algorithme ne la respectant pas. Nous supposons donc que nos hypothèses sont trop fortes, et que l'algorithme est optimal pour une allocation des données quelconque. Mais nous n'avons pas pu établir un tel résultat ;

- avec une machine pour laquelle communications et calculs peuvent être effectués en parallèle, nous avons un algorithme optimal pour toute topologie d'interconnexion et pour toute allocation de données, dès que la dimension de la matrice est suffisament grande. En effet, notre algorithme atteint l'efficacité maximale théorique de 1 dès que $n \geq p \frac{\tau_c + \sqrt{\tau_c^2 + 2 \beta \tau_a}}{\tau_a}$ (en reprenant les notations de la section 2.1).

Cela est aisément réalisable car à p fixé les calculs sont en $O(n^2)$ alors que les communications sont en O(n). Celles-ci sont donc complètement masquées par les calculs, dès que n est suffisament grand.

La figure 6 ci-dessous, représente pour un nombre de processeurs variable, le facteur d'accélération de l'algorithme parallèle en fonction de la taille de la matrice. On voit qu'il tend rapidement vers le nombre de processeurs utilisés.

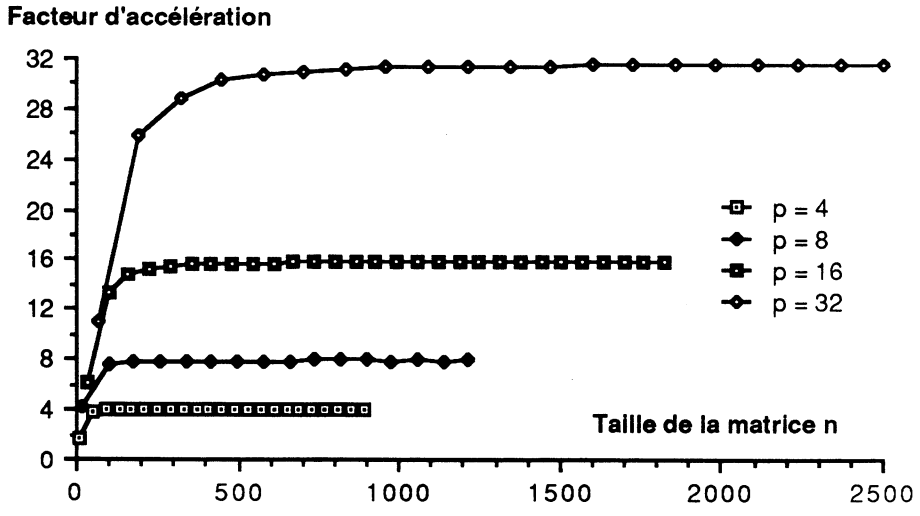


Figure 6 : facteur d'accéleration en fonction de n

On remarque aussi que les premières courbes "s'arrêtent" plus tôt que les dernières. Dans cette série d'expérimentations, on a utilisé les plus

grandes matrices possibles pour chaque nombre de processeurs. Puisque la mémoire est associée à chaque processeur, on dispose de quatre fois plus de mémoire avec 32 processeurs qu'avec 8, et on peut résoudre des problèmes deux fois plus importants (l'occupation mémoire est en $0(n^2)$ pour ce problème). Les machines à mémoire distribuée permettent non seulement de résoudre plus vite certains problèmes, elles permettent aussi d'en résoudre de plus grande taille.

2.4. Complexité de la distribution sur un réseau linéaire

La démonstration de l'optimalité du problème précédent passe par un résultat théorique que nous avons établi dans l'article [4] : il s'agit de la complexité de l'opération de distribution d'un vecteur du processeur de tête d'un réseau linéaire à tous les autres. Un algorithme effectuant cette opération de manière efficace était connu depuis longtemps, mais son optimalité n'était pas prouvée.

2.4.1. Le problème

Commençons par poser le problème : nous disposons d'un réseau linéaire de p processeurs numérotés de 0 à p-1, le processeur P_i étant relié aux processeurs P_{i-1} et P_{i+1} (là où ces indices ont un sens). Remarquons que ce qui suit est aussi valable pour un anneau orienté, où le processeur P_{p-1} serait en plus relié au processeur P_0, car on n'utilisera jamais ce lien de communication. Soit $X = (x_0, x_1, \ldots x_{N-1})$, un vecteur résident dans la mémoire locale du processeur P_0. Nous voulons le distribuer au reste du réseau. En d'autres termes, il faut qu'à la fin de l'exécution de l'algorithme, le processeur P_i ($0 \leq i \leq p-1$) connaisse le sous-vecteur X_i composé des composantes x_k de X avec $L * i \leq k < L * (i+1)$. Nous supposons donc que p divise N, et que les sous-vecteurs sont de taille identique L égale à N/p.

2.4.2. L'algorithme de distribution

L'algorithme classique est le suivant : à la première étape, P_0 envoie X_{p-1} à P_1 ; à la deuxième étape, il envoie X_{p-2} à P_1 tandis que P_1 transmet X_{p-1} à P_2, et ainsi de suite, jusqu'à la $(p-1)^{ème}$ étape où chaque sous-vecteur atteint son processeur destination. On donne ci-dessous le programme du processeur P_i :

```
pour b de p-1 à_l'envers i+1 faire
    { On fait circuler Xb, pour p-1 ≥ b ≥ i+1 }
  si i ≠ 0 alors recevoir (Xb) ;
  envoyer (Xb) ;
fpour ;
si i ≠ 0 alors recevoir (Xi) ;
```

Le temps d'exéction de l'algorithme est $(p-1)*(\beta + L * \tau)$, puisqu'il fait intervenir p-1 communications durant chacune $\beta + L * \tau$ (avec les notations de la section 2.1). L'article [4] montre que ce temps est optimal.

2.4.3. Preuve de l'optimalité

La preuve se fait en cinq étapes :

(1) il existe un algorithme optimal ;

(2) il existe un algorithme optimal dans lequel P_i envoie à P_{i+1}, les composantes x_N, x_{N-1}, ... , $x_{L(i+1)+1}$, $x_{L(i+1)}$ dans cet ordre, en les groupant éventuellement en plusieurs messages ;

(3) dans un algorithme optimal P_0 effectue au plus p-1 communications ;

(4) dans un algorithme optimal, l'un des messages de P_0 à P_1 contient entièrement l'un des sous-vecteur X_k ;

(5) par récurrence sur p, la borne inférieure au temps de distribution est $(p-1)*(\beta + L * \tau)$

Ce résultat est d'un intérêt théorique certain, car on dispose de peu de résultats de complexité de ce type pour les primitives de communication dans les machines à mémoire distribuée. Dans le cas de l'hypercube par exemple, on connait des algorithmes de communication très performants et qui approchent de près les limites théoriques. Mais on ne sait ni s'il en existe de meilleur, ni si les limites en question ne sont pas trop optimistes, et pourraient encore être raffinées.

2.5. Z-buffer sur une machine à base de Transputers

Dans l'article [5], nous nous sommes intéressés à l'implémentation d'un algorithme de synthèse d'image sur une machine à base de

Transputers. L'algorithme étudié, le Z-buffer est l'une des techniques séquentielles les plus rapides permettant d'obtenir une représentation réaliste d'une scène tridimensionnelle. Le parallélisme accélérant encore ses performances, peut permettre de calculer des images 3D en temps réel. Cette technique a de nombreuses applications, notamment en imagerie médicale, pour donner une représentation tridimensionnelle de structures vivantes à partir d'une série de coupes scanner [BM]. La machine cible pour laquelle nous étudions la parallélisation est le T-Node, une machine à base de Transputers, à réseau d'interconnexion reconfigurable. Elle permet de réaliser tout graphe d'interconnexion de degré inférieur ou égal à quatre. Nous nous sommes efforcés d'écrire un algorithme parallèle indépendant de la topologie du réseau, pour pouvoir l'exécuter sur toute une classe de graphes, et ainsi étudier l'influence de la topologie sur les performances de l'algorithme.

2.5.1. L'algorithme séquentiel

L'algorithme séquentiel est bien connu : la scène tridimensionnelle est composée d'objets définis par leur forme et leurs coordonnées dans un système de référence. Nous nous limitons dans cette étude à des facettes triangulaires modélisant la surface des objets. Elles sont définies par leurs trois sommets et la normale en ces sommets à la surface de l'objet extrapolé. Le but de l'algorithme est de produire une représentation bidimensionnelle (une image) de la scène, correspondant à ce qu'un observateur pourrait voir à travers une fenêtre. Cette fenêtre d'observation est assimilée à l'écran sur lequel les objets vont se projeter.

La première étape consiste à exprimer les coordonnées des objets dans un repère lié à l'observateur. Il s'agit essentiellement de transformations affines suivies par une projection perspective.

Pour une facette se projetant dans l'écran on balaye ensuite les pixels intérieurs au triangle en leur affectant une couleur. Cette couleur dépend de la couleur propre de l'objet en ce point, de la couleur et de l'orientation des sources lumineuses, de la direction d'observation et de la normale à la surface. Pour l'élimination des parties cachées, on ne dessine un point à un endroit de l'image que si sa distance à l'observateur (sa coordonnée Z dans le nouveau repère) est plus petite que celle du point précédemment dessiné au même endroit. Pour effectuer ce test, on conserve dans une autre image, (appelée image des profondeurs, ou *Z-buffer*) la coordonnée Z de chaque point dessiné.

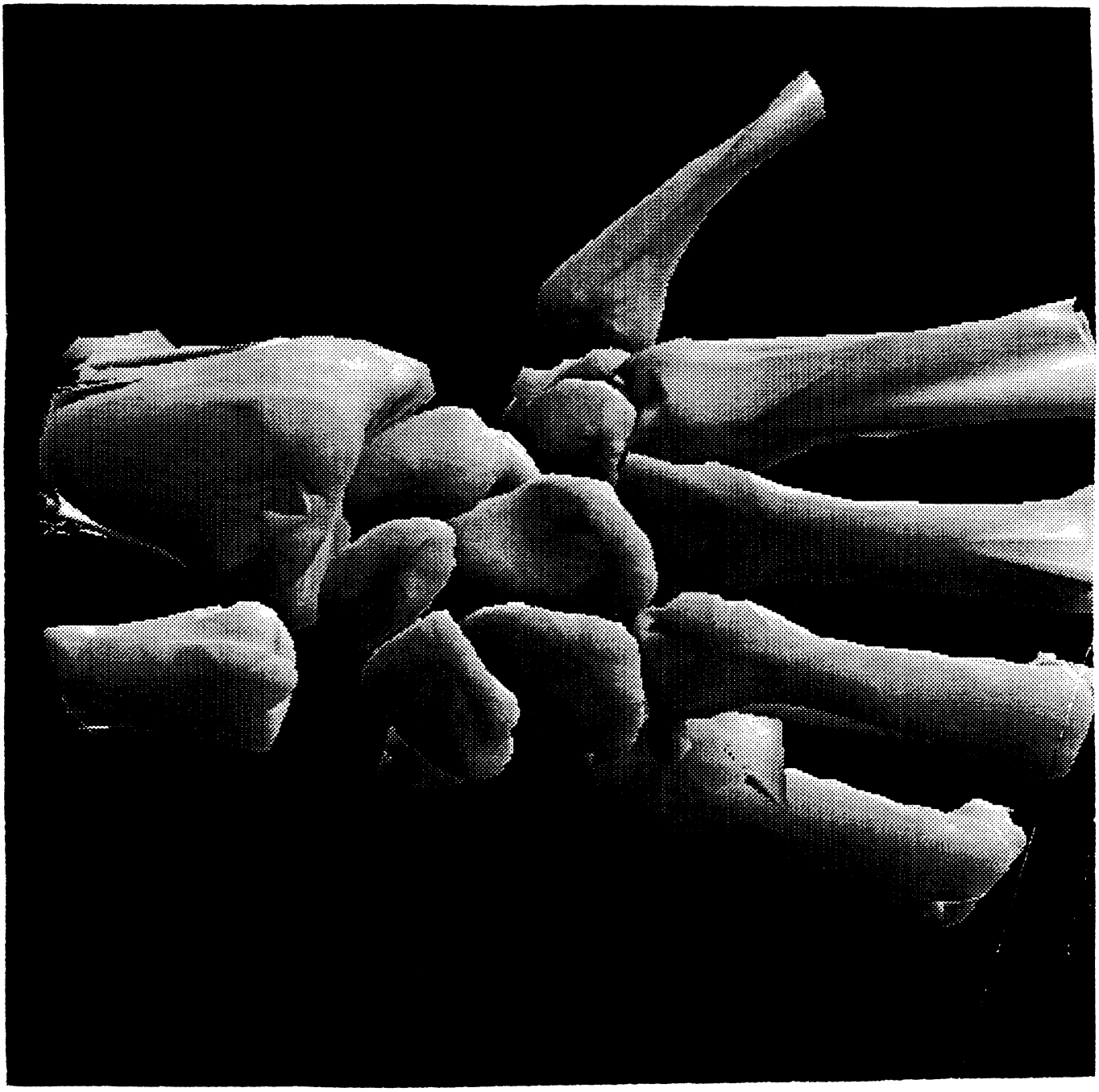


Figure 7 : Une application du Z-Buffer
à l'imagerie médicale (source : [BM])

Notre étude a donné lieu à deux implémentations parallèles différentes et complémentaires. La première est déstinée à un arbre de processeurs, et produit une image en mode pipeliné. La deuxième s'effectue sur un anneau, elle est basée sur une redistribution dynamique de la scène aux processeurs.

2.5.2. Parallélisation sur un arbre de processeurs

Notre première mise en oeuvre parallèle est basée sur la technique du *divide-and-conquer*, c'est à dire diviser pour régner. On distribue la description de la scène à l'ensemble des processeurs, chacun d'entre eux effectuant l'algorithme séquentiel avec les facettes qu'il a reçues. Une phase de fusion des images locales doit ensuite être mise en oeuvre pour reconstituer l'image définitive.

La première étape est indépendante de la topologie d'interconnexion. L'étape de fusion en revanche se fait suivant un arbre dont la racine est reliée au monde extérieur pour permettre le stockage ou la visualisation de l'image. La fusion part des feuilles de l'arbre et progresse par niveau pour terminer à la racine avec l'image définitive. Pour fusionner deux images source et leurs Z-Buffers associés, on les balaye simultanément, en stockant en position correspondante dans une image destination (et dans son Z-Buffer) la couleur et la profondeur du pixel ayant la plus petite profondeur dans les images source. Notons que nous avons toute liberté sur le degré, la forme et la hauteur de l'arbre. Nous pouvons donc étudier l'influence de ces différents paramètres sur les performances de l'algorithme.

Nous montrons dans l'article [5] diverses améliorations apportées à cette technique initiale. En particulier, nous expliquons comment diminuer la place mémoire requise par l'algorithme, en divisant l'image en bandes horizontales, et en calculant les bandes les unes après les autres. Cette division en bandes permet aussi de visualiser les parties déjà calculées de l'image pendant que les autres continuent à être évaluées, par un mécanisme de pipe-line dans l'arbre. Nous montrons aussi comment diminuer la complexité de l'opération de fusion, en faisant intervenir les "boites englobantes" des images partielles.

Dans certains cas, la parallélisation sur un arbre n'est pas efficace. En particulier, lorsque l'on veut produire de très grandes images, beaucoup de temps est perdu dans le processus de fusion des image locales. Ce n'est pas gênant lorsque la scène traitée est très volumineuse (composée de centaines de milliers de polygônes), car c'est alors le temps de projection et de numérisation des facettes qui prédomine. En revanche, le facteur d'accélération est très dégradé lorsque l'on traite une scène réduite. Nous étudions donc dans [5] une autre approche qui donne les résultats complémentaires.

2.5.3. Parallélisation sur un anneau de processeurs

Notre deuxième mise en œuvre de l'algorithme du Z-Buffer est basée sur une approche duale. Au lieu de permettre à chaque processeur d'intervenir dans le calcul de tous les points de l'image finale, nous partitionnons l'ensemble des pixels, en allouant des lignes consecutives de l'image aux processeurs. Un nouveau problème se pose alors, celui de l'allocation des objets. En effet, l'endroit de l'image où se projette un objet dépend des paramètres de visualisation, et n'est donc pas connu à l'avance. L'idée de base de cette nouvelle version est donc la suivante : après une distribution initiale de la scène, les processeurs projettent en parallèle les facettes qui leur ont été allouées, et calculent leur(s) processeur(s) destination (une facette peut être "à cheval" sur plusieurs zônes, elle doit alors être connue par tous les processeurs concernés). On effectue ensuite une phase de communication intensive, où chaque processeur envoie des messages différent à tous les autres. Cette opération est connue sous le nom de "multi-distribution" (*multiscattering*). Nous l'effectuons de manière efficace sur un anneau bidirectionnel de processeurs. La troisième phase de l'algorithme est ensuite élémentaire, puisque chaque processeur effectue l'algorithme séquentiel sur sa partie d'image.

Comme prévu, cette deuxième version est complémentaire de la première. Elle n'est pas très performante pour traiter les scènes trop volumineuses, car le temps de redistribution devient prépondérant. En revanche, le fait de générer des images de taille importantes n'est pas un handicap pour l'algorithme parallèle, puisque la taille supplémentaire des images n'engendre pas de surcoût de traitement par rapport à l'algorithme séquentiel.

2.6. Opérations de réduction sur un réseau reconfigurable

Les expérimentations liées au problème précédent nous ont amené à rechercher de manière théorique, quel était le graphe le mieux adapté aux opérations de réduction.

2.6.1. Un problème théorique

Posons le problème plus précisément : nous disposons de p processeurs P_i contenant chacun une donnée X_i ($0 \leq i < p$). Nous désirons qu'à la fin

de l'algorithme, l'un des processeurs, P_0 par exemple, contienne $\bigotimes_{i=0}^{p-1} X_i$,

où $\Diamond$ est un opérateur associatif et commutatif. Nous voulons aussi traiter le cas où X_i est un vecteur, et où $\Diamond$ s'applique composante par composante. Notons que l'opération de fusion des images de l'algorithme précédent est un cas particulier de ce problème : la fusion de deux images avec leurs deux Z-buffers associés est une image de même taille dans laquelle chaque pixel est calculé en effectuant un minimum sur la profondeur des pixels correspondants, stockée dans les Z-buffers.

Nous cherchons un graphe permettant d'effectuer cette opération de réduction en un temps minimal. Nous utilisons pour cela un modèle de calcul et de communications inspiré par les processeurs du type Transputer : un nombre de voisins maximal fixé, et la possibilité de calculer et de communiquer sur tous les canaux en parallèle, en un temps indépendant du nombre de communications effectuées simultanément.

2.6.2. Simplification de l'énoncé

Nous ne traitons pas le problème dans toute sa généralité, et nous restreignons notre étude à des graphes sans cycle. De plus, pour pouvoir utiliser des techniques de pipeline dans le cas où les X_i sont des vecteurs, nous utilisons un flot de données unidirectionnel : dans la stratégie de calcul que nous étudions, un processeur de l'arbre n'envoie une donnée à son père que lorsque ses fils lui ont transmis les données correspondantes, et que le calcul les faisant intervenir a été effectué. Nous avons baptisé cette stratégie "calculer-puis-envoyer". Le programme d'un noeud, quand les X_i sont des scalaires est donné ci dessous :

```
{ programme du processeur Pq }
Yq := Xq ;
pour chacun de mes fils faire en parallèle
   recevoir (Xfils)
   début de section critique
     Yq := Yq ◊ Xfils
   fin de section critique
fpour
envoyer (Yq) à mon père
```

On peut donc résumer la stratégie "calculer-puis-envoyer" de la manière suivante :

- la réception peut se faire en parallèle sur chacun des fils ;

- dès que l'une des réceptions est achevée, le calcul correspondant peut débuter ;
- le calcul qui fait intervenir la donnée de l'un des fils peut se faire en même temps que la réception de la donnée d'un autre ;
- les différents calculs doivent être effectués en exclusion mutuelle ;
- on n'envoie une donnée à son père que lorsque tous les calculs provenant des fils sont achevés.

2.6.3. Des arbres particuliers

Dans ce cadre, nous pouvons étudier le comportement d'un arbre particulier, et calculer le temps qu'il met pour effectuer la réduction. Nous l'exprimons en fonction du temps de arithmétique élémentaire A d'une opération X◊Y, et du temps de communication élémentaire C d'un scalaire X entre deux processeurs. Nous donnons un algorithme de programmation dynamique permettant de calculer l'arbre optimal ayant un nombre de processeurs fixé. Il est fondé sur un lemme que nous prouvons dans l'article, et qui donne le temps d'execution d'un arbre en fonction de celui de ses sous-arbres.

Lorsque C est grand devant A (respectivement A grand devant C), nous proposons une famille d'arbres construits de manière régulière, et qui sont optimaux pour un nombre de processeurs inférieur à une fonction exponentielle du rapport C/A (respectivement A/C). Nous appelons cette famille d'arbres "arbres de communication", (respectivement "arbres de calcul"), parce qu'ils sont performants lorsque le temps de communication (respectivement de calcul) est élevé.

Grâce à ces deux familles d'arbre, nous savons donner des bornes sur le temps d'exécution d'une opération de réduction avec notre stratégie "calculer-puis-envoyer".

Nous traitons ensuite le cas où les données X_i sont des vecteurs. En utilisant la stratégie "calculer-puis-envoyer" sur les composantes des vecteurs, nous calculons le temps d'exécution de la réduction quand ces composantes sont traitées par blocs et pipelinées dans l'arbre. Nous calculons la taille optimale des blocs, minimisant le temps d'exécution de l'algorithme.

Quelques expériences numériques ont été effectuées sur le T-node et sont présentées en fin de l'article [6]. Elles sont en bonne accordance avec nos prévisions théoriques.

Troisième partie : Articles de recherche

Article [1]

Dynamic programming on a ring of processors

Hypercubes and Distributed Computers
F. André and J.P. Verjus (Editors)
Elseiver Science Publisher B.V. (North-Holland), 1989

DYNAMIC PROGRAMMING ON A RING OF PROCESSORS

Serge MIGUET and Yves ROBERT
Laboratoire de l'Informatique du Parallélisme LIP-IMAG
Ecole Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France
e-mail: yrobert@ensl.ens-lyon.fr (uucp)

Abstract

We discuss various data allocation strategies for the parallel implementation of some dynamic programming problems on a ring of processors. We consider column oriented schemes and allocate blocks of r consecutive columns of the cost matrix to the processors in a wraparound fashion. We show that the smaller r, the better balanced the arithmetic, but also the more costly the communications. We analytically determine the optimal value of r which minimizes the parallel execution time as a function of the ratio $\alpha=p/n$ of the number of processors p over the problem size n, and of the ratio $\rho=\tau_c/\tau_a$ of the elemental communication time τ_c over the elemental arithmetic time τ_a. These theoretical results are corroborated by numerical experiments on a ring of Transputers.

1. INTRODUCTION

Many problems in computer science can be solved by the use of dynamic programming techniques. These include shortest paths, optimal parenthesization, matching problems, matrix chain products, and optimal binary search trees among others [AHU, GR, Sed]. The generic instance of a dynamic programming problem of size n can be stated as the following recurrent computation:

$$c(i,j) = \min_{i \le k < j} \{ c(i,k) + c(k+1,j) + f(i,k,j) \} \text{ for } 0 \le i < j \le n-1 \qquad (1)$$

where the initial costs c(i,i), $0 \le i \le n-1$, and the values of f(i,k,j), $0 \le i \le k < j \le n-1$ are known in advance. We restrict ourselves to the cases where f(i,k,j) = 0 or $f(i,k,j) = g_1(i) \Delta g_2(k) \Delta g_3(j)$, where Δ is any arithmetic operation and g_1, g_2 and g_3 are given functions. These cases include most practical situations. For example in the matrix chain product, we suppose that n matrices M_0, M_1, ..., M_{n-1} are to be multiplied together. We let c(i,j) be the minimum cost of computing $M_i.M_{i+1}...M_j$ for $0 \le i < j \le n-1$. If the i-th matrix M_i has r_i rows and r_{i+1} columns, we have c(i,i) = 0 and $f(i,k,j) = r_i * r_{k+1} * r_{j+1}$ in the previous recurrences.

This work has been supported by the Research Project C3 of CNRS.

Dynamic programming is a computationally intensive procedure: solving an instance problem of size n requires $n^3/6 + O(n^2)$ elementary steps, if we choose one update c(i,j) := min { c(i,j), c(i,k) + c(k+1,j) + f(i,k,j) } in (1) as a unit of time. As a consequence, several authors have studied its parallelization. Concerning shared memory systems, theoretical work using the CREW PRAM model of computation has been carried out by Gibbons and Rytter [GR] and Rytter [Ryt], who design a parallel algorithm running in $\log^2 n$ using $n^6 / \log n$ processors. Concerning distributed memory systems, several systolic implementations have been developed. In their pionnering work, Guibas, Kung and Thompson [GKT] have designed a triangular systolic array of n(n+1)/2 elementary processors that solves a dynamic programming problem of size n within 2n time-steps. Note here that a time-step is defined as the time to perform two updates in (1). The result of [GKT] has been improved by Gachet, Joinnault and Quinton [GJQ] and by Louka and Tchuente [LT] who use systematic design methodologies to achieve the same number of steps with fewer cells: $n^2/4$ for [GJQ] and $n^2/8$ for [LK]. Recently, Bitz and Kung [BK] have reported on the implementation of a special instance of dynamic programming, namely path-planning, on the Warp computer [AAG]. They use a linear array of processing cells and discuss two data partitioning methods when the number of cells p is less than the problem size n. Best results are obtained when allocating the rows of the data matrix to the processors in a wraparound fashion: row i is allocated to processor k if i mod p = k.

In this paper, we consider the implementation of dynamic programming on a ring of processors. We study the impact of data allocation strategies on the performances of the parallel implementation. More precisely, we allocate blocks of columns to processors in a wraparound fashion, and we determine the optimal value of the block size as a function of the ratio $\alpha = p / n$ of the number of processors p over the problem size n, and of the ratio $\rho = \tau_c / \tau_a$ of the elemental communication time τ_c over the elemental arithmetic time τ_a. Note that the usual wrap repartition (allocate columns of same index modulo p to the same processor) and full block repartition (allocate n/p consecutive columns to each processor) are particular cases of our analysis. Note also that our problem is entirely symmetric, and row oriented schemes can be handled similarly.

2. PARALLEL IMPLEMENTATION

When looking at the recurrences (1), we see that we can compute the costs c(i,j) in increasing value of d = j-i, ending with the computation of c(0,n-1). In other words, the computation can progress diagonal by diagonal, as depicted figure 1: if all the costs c(i,j) with j-i < d are available, then all the c(i,j) with j-i = d can be evaluated. Moreover, all these costs are independent, hence this scheme of computation is amenable to certain parallelism.

In what follows we consider a ring of p processors numbered from 0 to p-1. Each processor has a private memory and can communicate by a message passing protocol with its two neighbors: processor P_i exchanges messages with P_{i-1} and P_{i+1} (throughout, processor indexes are taken modulo p). The machine operates in an asynchronous MIMD mode. Communications are by rendez-vous, and there is no overlap between computations and communications.

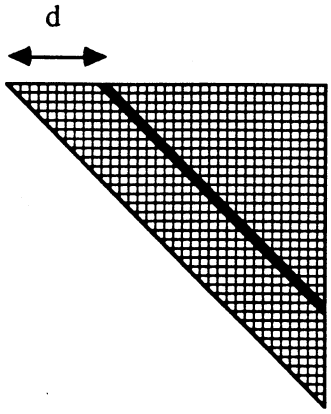


Figure 1 : Parallel evaluation of the costs along a same diagonal

For computing a dynamic programming problem of order n, we distribute the cost matrix to the processors. We have to choose a data allocation strategy so that the arithmetic workload is well balanced among the processors, and so that the comunication overhead is kept minimum. We choose a column oriented scheme, that is we distribute the columns of the cost matrix among the processors. We let *alloc* be the allocation function: processor q, $0 \leq q \leq p-1$, holds column j in its private memory if and only if alloc(j) = q.

2.1. Parallel algorithm with a wrap repartition

To explain our parallel implementation, we choose the wrap repartition for the allocation function, as it leads to the most easily understood algorithm. So we have alloc(j) = j mod p for all columns j, $0 \leq j \leq n-1$.

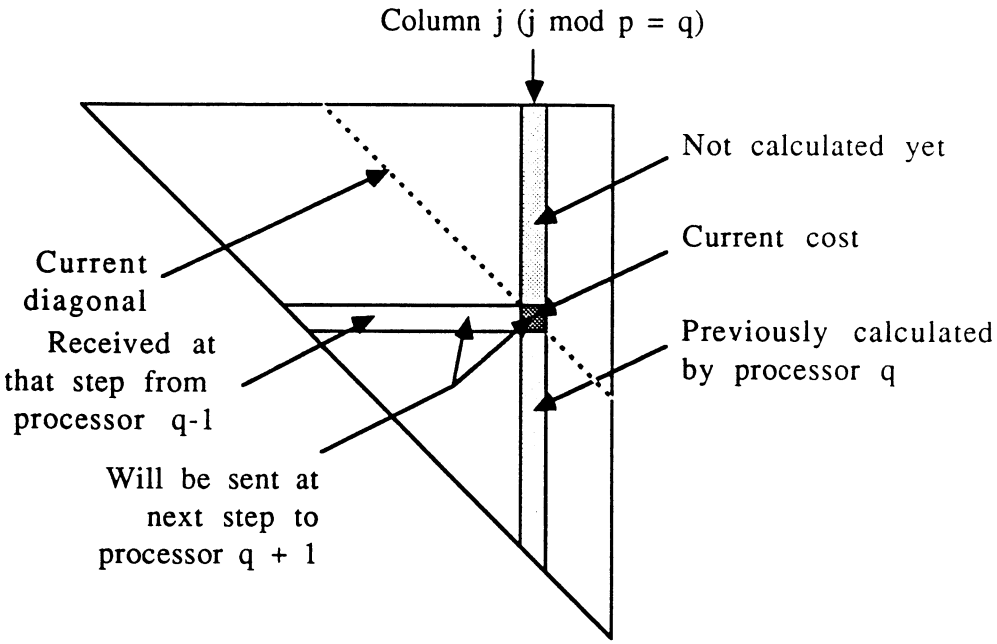


Figure 2 : Scheme of computation for the wrap allocation

The parallel algorithm proceeds in n-1 parallel steps. At step d, $1 \leq d \leq n-1$, the costs on the diagonal d, i.e. the costs $c(j - d,j)$, $d \leq j \leq n-1$, are computed. Let $i = j - d$ and let P_q be the processor holding column j, that is $alloc(j) = q$. To evaluate $c(i,j)$,we have the fomula

$$c(i,j) = \min_{i \leq k < j} \{ c(i,k) + c(k+1,j) + f(i,k,j) \}$$

Consider figure 2. All the costs $c(.,j)$ are resident in the local memory of P_q, and the information on the costs $c(i,k)$, $i \leq k < j$, will be received from its neighbor P_{q-1}. Note that at step d+1, these costs $c(i,k)$, $i \leq k < j$, together with the cost $c(i,j)$, will be needed by P_{q+1} to evaluate the cost $c(i,j+1)$, since $alloc(j+1) = q+1 \bmod p$. Note that in figure 2 we exhibit the update of a single cost at a given step by processor P_q, whereas several costs are updated by P_q at that step (see figure 3).

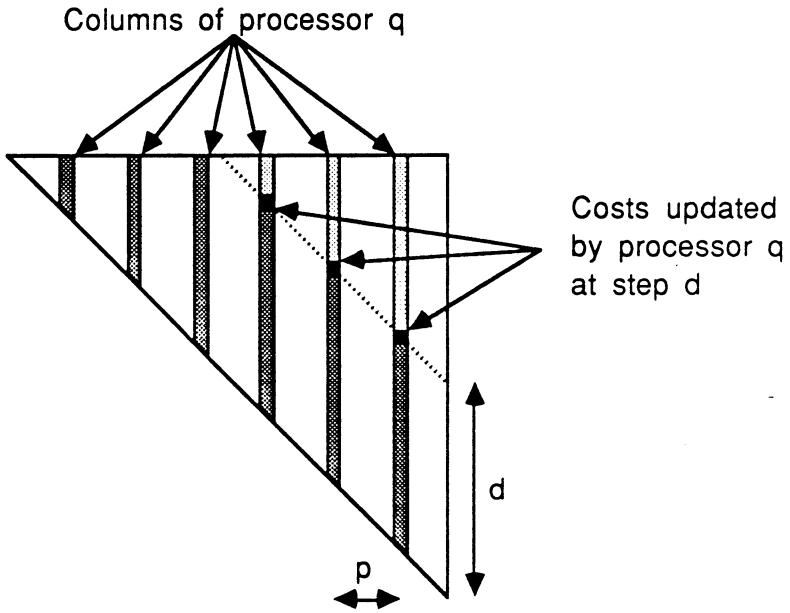


Figure 3 : Columns updated by processor q for the wrap allocation

The whole scheme is now outlined: at a given step d, each processor P_q updates the element of its internal columns which belong to diagonal d: the costs $c(i,j)$ such that $j-i = d$ and $alloc(j) = q$. For each updated $c(i,j)$, it receives the costs $c(i,.)$ which are already computed in row i from its left neighbor P_{q-1}. At step d+1, it will append $c(i,j)$ to those costs and transmit them to P_{q+1}.

Assume for the sake of simplicity that p divides n. Each processor is allocated n/p columns of the cost matrix in a wraparound fashion. At step 1, each processor P_q evaluates n/p costs $c(j-1,j)$ such that $alloc(j) = q$, that is $j \bmod p = q$ (except for P_0 which has only n/p-1 costs to compute). P_q receives a message from P_{q-1} containing the n/p costs $c(j-1,j-1)$ which it needs for its computation. At step 2, it will append to the previous message the n/p costs $c(j-1,j)$ calculated at step 1 and transmit them to P_{q+1}. We have the following kernel:

Program of processor P_q
message := initial costs c(j,j) such that alloc(j) = q
for d = 1 **to** n-1 **do**
communication: rotate(message)
arithmetic: update internal costs c(i,j) such that j-i = d and alloc(j) = q
append those costs to message

During the procedure *rotate* , each processor P_q sends the value stored in its internal buffer *message* to its right neighbor P_{q+1}. The processor P_e which contains the last column (this means that e = alloc(n-1)) has an additional work: it is responsible for deleting the values which will no longer be used in the rotated messages. For instance it sends only n/p-1 initial costs to P_{e+1} at step 1, which amounts to delete c(n-1,n-1) from its buffer *message*. At step 2, it deletes the information concerning row n-2, that is c(n-2,n-2) and c(n-2,n-1). More generally at step d, P_e deletes from its buffer message the information concerning row n-d.

In table 1, we outline the parallel execution for a problem of size n = 8 with p = 4 processors. We let ij denote the cost c(i,j). At each step, we exhibit the costs that each processor receives from its left neighbor.

We give now a brief explanation of the procedure *rotate*. For neighbor to neighbor communications we have the following primitives:

send/receive(sender, receiver, send_buffer, receive_buffer, length)

Since each processor has to transmit the content of its internal buffer *message* to its right neighbor, we need to duplicate this buffer, so as not to erase previous values during the communication. We could imagine the following procedure:

Procedure rotate (program of processor Pq)

send(P_q, P_{q+1}, send_buffer, receive_buffer, message_length)
receive(P_{q-1}, P_q, send_buffer, receive_buffer, message_length)

However, comunications are blocking, and the above procedure would lead to a deadlock, all the processors trying to send simultaneously. We are led to the following (exact) procedure:

Procedure rotate (program of processor P_q)

if q mod 2 = 0 **then**
send(P_q, P_{q+1}, send_buffer, receive_buffer, message_length)
receive(P_{q-1}, P_q, send_buffer, receive_buffer, message_length)
else
receive(P_{q-1}, P_q, send_buffer, receive_buffer, message_length)
send(P_q, P_{q+1}, send_buffer, receive_buffer, message_length)
endif

24

Messages received at step d by processor q

q	0		1		2		3	
d \ j	0	4	1	5	2	6	3	7
1	⊖	3 3	0 0	4 4	1 1	5 5	2 2	6 6
2		22 23	⊖	33 34	00 01	44 45	11 12	55 56
3		11 12 1 3		22 23 2 4	⊖	33 34 3 5	00 01 0 2	44 45 4 6
4		00 01 02 03		11 12 13 14		22 23 24 25	⊖	33 34 35 36
5		⊖		00 01 02 03 04		11 12 13 14 1 5		22 23 24 25 2 6
6				⊖		00 01 02 03 04 05		11 12 13 14 15 16
7						⊖		00 01 02 03 04 05 0 6

This symbol appears when the computation in a given column is completed ⊖

Table 1 : Exhibiting the messages for the wrap allocation (n=8, p=4)

2.2. General parallel algorithm

We now consider general allocation functions, and we assign blocks of r consecutive columns to the processors in a wraparound fashion. For instance with r = 2, n = 30 and p = 5 we have the following repartition:

	P_0	P_1	P_2	P_3	P_4
Columns	0,1	2,3	4,5	6,7	8, 9
	10,11	12,13	14,15	16,17	18,19
	20,21	22,23	24,25	26,27	28,29

The allocation function by blocks of size r is defined by

$$alloc(j) = floor(j/r) \bmod p,\ 0 \leq j \leq n-1$$

The whole principle of the algorithm is the same: instead of considering elementary costs, we now consider square blocks of costs of size r x r. We have only n/r - 1 macro-steps. At each macro-step, all the processors update square sub-matrices of r x r costs. Assume that p * r divides n. Messages are rotated just as before, and now consist of blocks of r partial rows. Initially, each processor updates n/(pr) triangular upper r x r matrices located on the diagonal. At the first macro-step, each processor transmits those triangular matrices to its right neighbor, except the last one which transmits one less. Then at each macro-step, messages consist of trapezoidal sub-matrices (see figure 4 for one such trapezoidal matrix). Note that when r = n/p, we retrieve the full block allocation function, where slices of n/p consecutive columns are assigned to the processors.

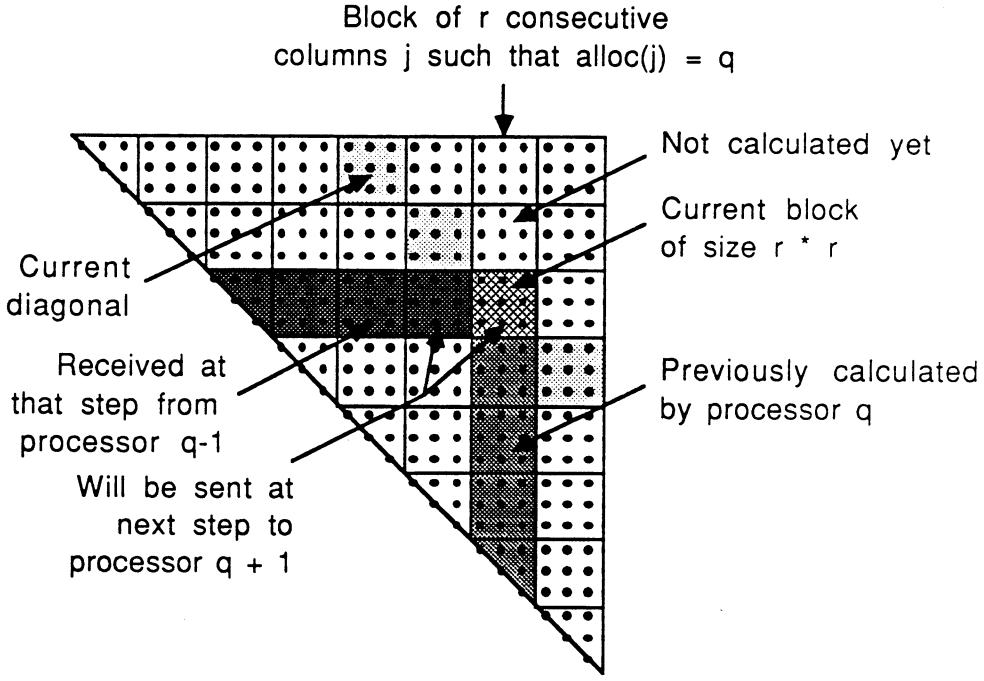


Figure 4 : Scheme of computation for the block-r allocation

As for the wrap allocation, we have only represented a single block update in figure 4. To give numbers, each processor q receives (n/(pr) - ceiling((d-q)/p)) trapezoidal submatrices of dr^2-r(r-1)/2 costs.

26

Intuitively, the larger r, the less balanced the computations, since the number of costs to update in column j is equal to j. In our example with r=2, n=20 and p=5, we see that processor P_4 has much more work to do than processor P_0, and that the difference would even increase for larger values of r. On the other hand, the larger r, the smaller the communication overhead: for example with the wrap repartition each processor receives c(0,0) n/p times whereas with the full block repartition it receives it only one time ! Table 2 below is the counterpart of table 1 for the full block repartition, and we clearly see that the volume of communications is dramatically decreased.

In table 2, we outline the parallel execution for a problem of size n = 8 with p = 4 and r = 2. We let ij denote the cost c(i,j). At each step, we exhibit the costs that each processor receives from its left neighbor.

q	0		1		2		3	
d \ j	0	1	2	3	4	5	6	7
1	⊖		00 01 11		22 23 33		44 45 55	
2			⊖		00 01 02 03 11 12 13		22 23 24 25 33 34 35	
3					⊖		00 01 02 03 04 05 11 12 13 14 15	

This symbol appears when the computation in a given column is completed ⊖

Table 2 : Exhibiting the messages for the full block allocation (n=8, p=4)

In the next section, we move to a quantitative analysis and analytically compute the arithmetic and communication times for general block-r allocations.

3. PERFORMANCE EVALUATION

In this section, we analyse the performances of the parallel algorithm described above. The problem is to determine the allocation function which minimizes the total execution time, defined as the sum of the arithmetic time T_a, and of the communication time T_c (which includes idle time). For the arithmetic, we let τ_a be the elemental time needed for an update in the generic recurrences. As noted in the introduction, the sequential time for a problem of size n is

$$T_{seq} = (n^3/6 + O(n^2)) \tau_a.$$

3.1. Memory requirement

The space requirement for the sequential algorithm is equal to the size of the cost matrix, that is $n^2/2 + O(n)$. Given a single processor with a memory of size M words, this implies that the maximal problem size that can be dealt with is $n_{max,1} \cong (2M)^{1/2}$. Consider now a ring of p processors. First of all, we have to determine the relationship between p and n. We have p memories of size M, so that we can solve in parallel a problem of size at most $n_{max,p} \cong (2pM)^{1/2}$. Note that we neglect here any additional storage required by the parallel implementation, such as the need for communication buffers. In fact, the value $n_{max,p}$ above is an upper bound. For the existing parallel architectures, we have $p \leq M$ (Intel iPSC, FPS T Series, Ncube) or at least p and M of the same order (note that for the largest configurations of the Connection Machine, or of the AMT DAP, we do have $p > M$ [Hwa]). As a consequence, in most practical situations we have $p \leq n$, and in the following we let $p = \alpha n$, where α is a given constant ranging from 0 to 1.

As stated before, we consider an allocation by blocks of consecutive columns of size r in a wraparound fashion, where $1 \leq r \leq n/p = 1/\alpha$. For the sake of simplicity (without loss of generality), we assume that p * r divides n, so that each processor holds the same number of columns in its local memory.

3.2. Parallel arithmetic time

As seen in section 2.2, processor q holds the columns j such that q = floor(j/r) mod p. In other words, we have floor(j / r) = u p + q, where $0 \leq u \leq n/(pr) - 1$, which can be written: j = u p r + q r + v, with $0 \leq v \leq r-1$. Obviously, the number of computations needed to update all the costs in a column j is Arith(j) = j (j+1) / 2, so that the arithmetic time for processor q is:

$$T_a(q) = \sum_{(j \,/\, alloc(j) = q)} Arith(j)\, \tau_a = \sum_{u,v} (upr+qr+v)\,(upr+qr+v+1)\,\tau_a / 2$$

The parallel arithmetic time is then $T_a = \max_{0 \leq q \leq p-1} Ta(q)$. We check easily that $T_a(q)$ is a non-decreasing function of q, so that $T_a = T_a(p-1)$, the arithmetic workload of the last processor. We obtain

$$T_a = [\, n / (12p) \,]\, [\, p^2r^2 + 3prn + 2n^2 + O(n) \,]\, \tau_a$$
$$= [\, 2/\alpha + 3r + \alpha r^2 \,]\; n^2\, \tau_a / 12 + O(n)$$

where r ranges from 1 (wrap) to $1/\alpha$ (full block). We see that given α, T_a is a non-decreasing function of r: in other words, the larger r, the less balanced the computations among the processors.

3.3. Parallel communication time

We let τ_c be the elemental time for rotating a message, so that rotating L words require $L\,\tau_c$ units of time. More accurately, we should choose an expression such as $\beta_c + L\,\tau_c$, where β_c is a start-up time. In the literature, the time to transfer L words between two adjacent processors is modelized by $\beta + L\,\tau$ [GH, KT, MV, Saa2] and basically, our rotate procedure amounts to twice that time, since each processor has to execute sequentially an emission and a reception. However, we can assume $\beta = 0$ without loss of generality, as in [ISS, Saa1]: each processor receives O(n) messages during

the procedure, and the number of cost transfers is $O(n^2)$, so that the contribution of the start-up times is negligible.

For a block-r allocation function, we have n/r-1 macro-steps. At step d, we rotate a message composed of (n/(pr) - floor(d/p)) trapezoidal blocks of $dr^2-r(r-1)/2$ coefficients. In other words, the length of the message to be rotated at step d is Rotat(d) = (n/(pr) - floor(d/p)) $(dr^2-r(r-1)/2)$. We let $d = u\ p + v$, $0 \le u \le n/(pr)-1$ and $0 \le v \le p-1$, and we compute the communication time T_c as

$$T_c = \sum_{1 \le d \le n/r-1} \text{Rotat}(d)\ \tau_c = \sum_{u,v} (n/(pr)-u)\ ((up+v)r^2-r(r-1)/2)\ )\ \tau_c$$
$$= [\ n / (12pr)\ ]\ [\ p^2r^2 + 3prn + 2n^2\ + O(n)\ ]\ \tau_c$$
$$= [\ 2/\alpha + 3r + \alpha r^2\ ]\ n^2\ \tau_c / (12r) + O(n)$$
$$= (\ T_a / r\ )\ (\ \tau_c / \tau_a\ ) + O(n)$$

Let $\rho = \tau_c / \tau_a$ be the ratio of the elemental times communication/arithmetic. The last expression of T_c shows that $T_c = \rho\ T_a / r$ if we ignore low order terms. We check easily that T_c is a non-increasing function of r (for $1 \le r \le 1/\alpha$) : the larger r, the more reduced the volume of the communications.

3.4. Parallel execution time

Given α, T_a is an increasing function of r whereas T_c is a decreasing one. In other words, T_a is minimum for the wrap repartition (r=1), while T_c is minimum for the full block repartition (r=n/p). The smaller r, the more balanced the computations, but the more expensive the communications. To determine the best allocation function, that is the value of r which minimizes the total execution time, we must be ready to find a compromise between the two contradictory exigences of balanced computations (small r) and inexpensive communications (large r). Collecting previous results, we have:

Proposition : Given a problem of size n and a ring of $p = \alpha\ n$ processors, the parallel execution time $T_{\alpha,r}$ for a block-r allocation, $1 \le r \le 1/\alpha$, is

$$T_{\alpha,r} = [\ 2/\alpha + 3r + \alpha r^2\ ]\ (\ \tau_a\ + \tau_c / r\ )\ n^2 / 12 + O(n)$$

Corollary : Given a problem of size n and a ring of $p = \alpha\ n$ processors, the efficiency $e_{\alpha,r}$ for a block-r allocation, $1 \le r \le 1/\alpha$, is

$$e_{\alpha,r} = 1 / [\ (1 + \rho/r)\ (1 + 3\alpha r/2 + \alpha^2 r^2/2)\ ] + o(1)$$

where $\rho = \tau_c / \tau_a$ is the ratio of the elemental times communication/arithmetic

Given α and ρ, it is easy to compute the value of r which maximizes the efficiency. We report numerical experiments in the next section. Note that for small α, the efficiency can be approximated as

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

(we drop the term in α^2). The efficiency is then maximum for $r_{opt} \approx [2\rho / (3\alpha)]^{1/2}$.

Corollary : Given a problem of size n and a ring of $p = \alpha n$ processors, where $\alpha << 1$, the optimal block size is $r_{opt} \approx [2\rho / (3\alpha)]^{1/2}$.

This is a key result for all situations where the number of processors is significantly less than the size of the problem we want to solve, a situation very likely to happen in practice.

4. NUMERICAL EXPERIMENTS

We report in this section numerical experiments on a ring of Inmos Transputers T414, using up to 16 processors. We use a FPS T20 hypercube [GHS], which we configure as a ring. First of all we have to determine τ_a and τ_c. For τ_a we choose an instance of dynamic programming which corresponds to the construction of an optimum binary search tree. Each update in (1) amounts to two additions, a comparison, the incrementation of two pointers plus some conditional logic. We find that τ_a = 15e-6 seconds for this problem.

For the communications, we know that on the FPS T Series with the C01 software release, the time to transfer L (integer) words between two adjacent processors is $\beta + L\tau$, with β = 0.8e-3 seconds and τ = 5.76e-6 seconds [KT]. We find experimentally that rotating a message of length L along the ring takes $\beta_c + L\tau_c$, where β_c = 1.6e-3 seconds and τ_c = 12e-6 seconds. The value of β_c is twice that of β, as expected from the procedure *rotate* . The value of τ_c is slightly greater than 2τ, because we take into account some additional time for copying and reindexing the messages at each step.

The first thing we check is that the parallel execution time obeys our formulas. For instance in figure 5, we choose the wrap distribution and plot the time over the cube of n, as a function of n for various values of p. As expected, we observe horizontal asymptotes in the curves. Note in figure 5, that the largest problem size that can be solved is a function of p.

In the next experiment, we fix n and p and let the block size r vary. We superimpose in figure 6 the experimental and theoretical curves (with the previous values of τ_c and τ_a) for the parallel time. Again, there is a very good adequation between the two curves. We find an optimal value of r_{opt} = 4. Note that execution time with r_{opt} is decreased by 30 % as compared to the wrap distribution.

30

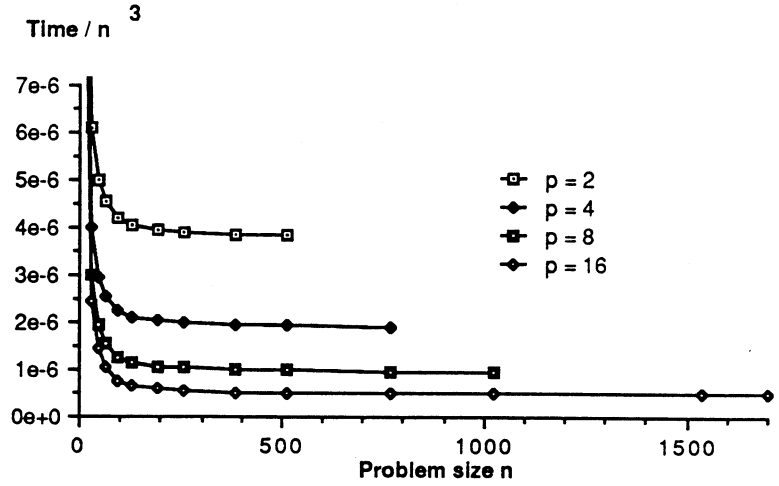


Figure 5 : t / n^3 as a function of n, where t = parallel time, n = problem size (wrap disribution, 16 processors)

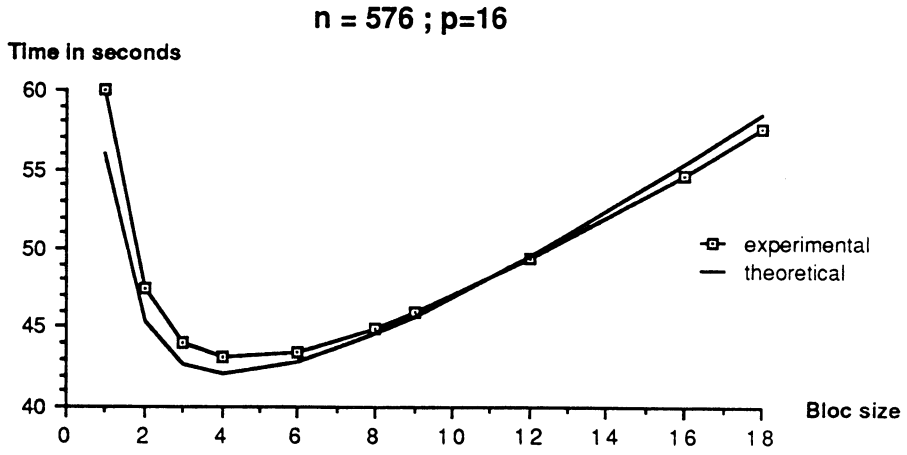


Figure 6 : Parallel time as a function of the block size

We have $\rho = \tau_c / \tau_a = 0.8$ in our numerical experiments. If we come back to the expression of the efficicency, we can numerically compute the optimal value of r as a function of α. Since we always use small values of α, we want to check whether the expression $r_{opt} \approx [\, 2\rho / (3\alpha)\,]^{1/2}$ is accurate (see section 3.3). In figure 7 we let p = 16 and we plot the time as a function of r for various values of n (hence various values of α). We give both theoretical and experimental values. The curves clearly show that r_{opt} is very well approximated by the expression $[\, 2\rho / (3\alpha)\,]^{1/2}$.

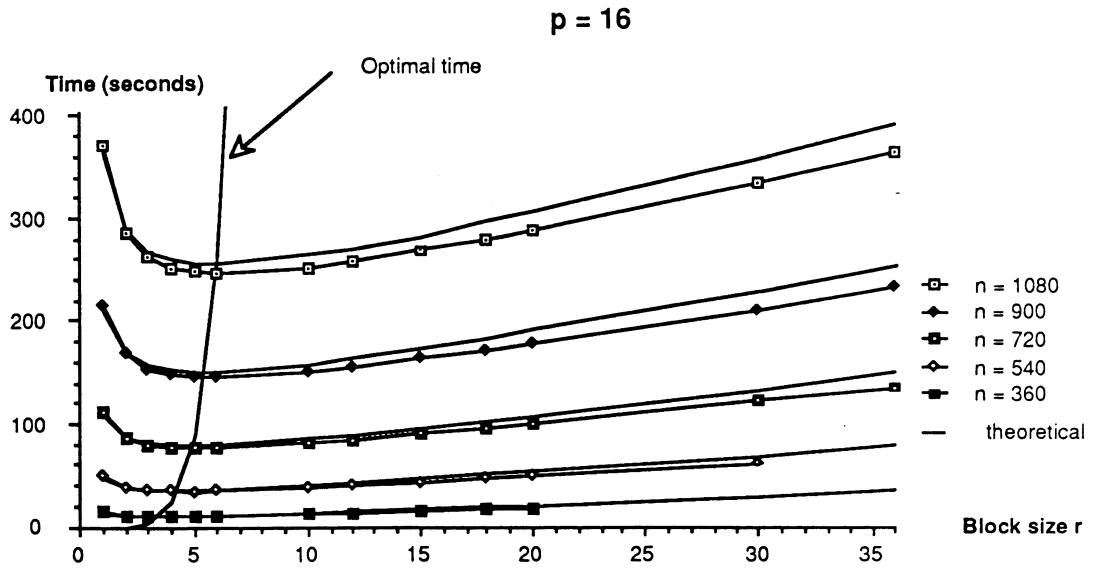


Figure 7 : Optimal block size (theoretical and experimental results)

In figure 8 and 9 we further investigate the validity of the formula $r_{opt} \approx [\ 2\rho/(3\alpha)]^{1/2}$. In both experiments we fix n and let p vary. We plot experimental results together with the theoretical optimal value of r_{opt}. In figure 8 we check than if we multiply the number of processors by a factor of 4, which amounts to divide α by 4 since n is fixed, then r_{opt} is doubled. On the other hand in figure 9 we evidence the influence of τ_a on r_{opt}. We redo the experiments with another instance of dynamic programming, namely the matrix chain product, for which an update in (1) takes τ_a = 25 µs. With p = 8 processors, we have r_{opt} = 5 in figure 8 and r_{opt} = 4 in figure 9, and the ratio 5/4 = 1.25 is in accordance with the ratio $(25/15)^{1/2} \approx 1.29$ of the values of ρ used in the two experiments.

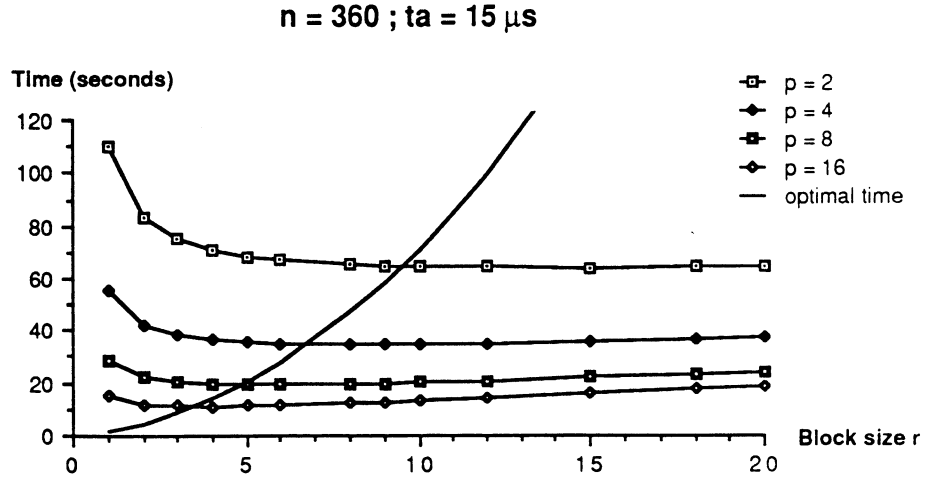


Figure 8 : r_{opt} as a function of α, τ_a = 15 µs

32

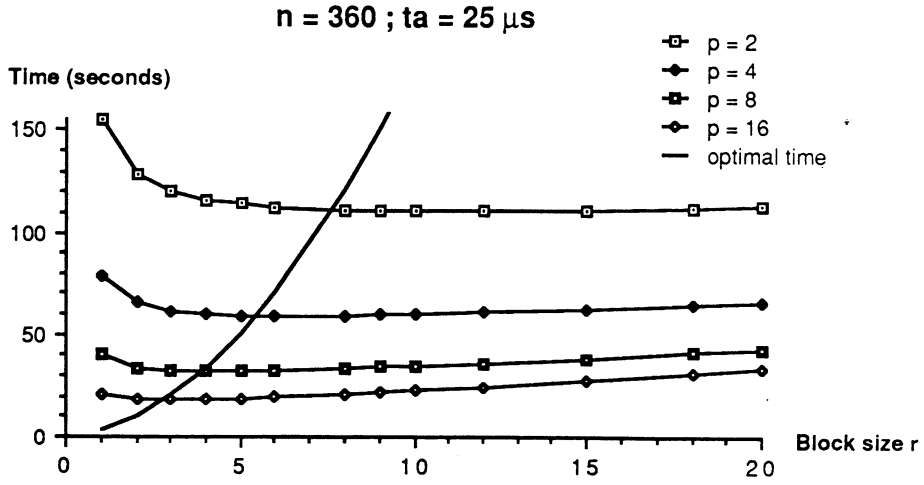


Figure 9 : r_{opt} as a function of α, τ_a = 25 µs

Finally in figure 10, we plot the speedups that we obtain with 16 processors when solving various sizes of problems, from n = 360 up to 1080. Note that these speedups are computed according Gustafson's recent proposal [Gus], in that they are normalized by the amount of arithmetic operations which they require (since it is impossible to solve such a large problem with a single processor). We report acceleration factors as high as 13. Note that for r = 1 (wrap) the speedup is theoretically bounded by 1 / (1+ρ) ≈ 8.9, which is in good adequation with the experimental results.

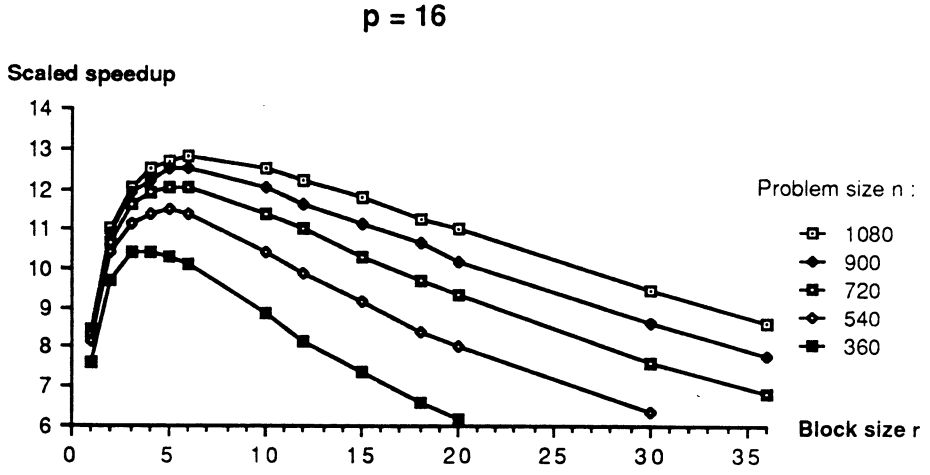


Figure 10 : Scaled speedup with 16 processors

5. CONCLUSION

We have studied the performances of various data repartitions for dynamic programming on a ring of processors. We have used column oriented schemes, and we have allocated consecutive blocks of the columns of the cost matrix to the processors in a wraparound fashion. We have derived the best blocksize to be used

as a function of the ratio α of the number of processors over the problem size and of the ratio ρ of the elemental costs communication/arithmetic. We point out the good adequation of these theoretical results with numerical experiments run on a ring of transputers.

6. REFERENCES

[AHU] A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN, Data structures and algorithms, chapter 10.2, Addison-Wesley (1983)

[BK] F. BITZ, H.T. KUNG, Path planning on the Warp computer: using a linear systolic array in dynamic programming, Intern. J. Computer Math. 25 (1988), 173-188

[GJQ] P. GACHET, B. JOINNAULT, P. QUINTON, Synthetizing systolic arrays using DIASTOL, in Systolic Arrays, W. Moore et al. eds., Adam Hilger (1986), 25-36

[GH] G.A.GEIST, M.T.HEATH, Matrix Factorization on a hypercube multiprocessor, Hypercube Multiprocessors 1986, M.T. Heath ed., SIA M (1986), 161-180

[GHS] J.L. GUSTAFSON, S. HAWKINSON, K. SCOTT, The architecture of a homogeneous vector supercomputer, in Proceedings of ICCP 86, IEEE Computer Science Press (1986), 649-652

[GKT] L.J. GUIBAS, H.T. KUNG, C.D. THOMPSON, Direct VLSI implementation of combinatorial algorithms, Caltech Conference on VLSI (1979), 509-525

[GR] A. GIBBONS, W. RITTER, Efficient parallel algorithms, chapter 3.6, Cambridge University Press (1988)

[Gus] J.L. GUSTAFSON, Reevaluating Amdahl's law, Communications of the A.C.M. 31, 5 (1988), 532-533

[Hwa] K. HWANG, Advanced parallel processing with supercomputer architectures, Proceedings of the IEEE 75, 10 (1987), 1348-1379

[ISS] I.C.F. IPSEN, Y. SAAD, M.H. SCHULTZ, Complexity of dense linear system solution on a multiprocessor ring, Lin. Alg. Appl. 77 (1986), 205-239

[KT] S. KUPPUSWAMI, B. TOURANCHEAU, Evaluating the performances of the FPS T Series hypercube computer, Research Report 708, IMAG Grenoble (1988)

[LT] B. LOUKA, M. TCHUENTE, Dynamic programming on two-dimensional systolic arrays, Information Processing Letters 29 (1988), 97-104

[MV] O.A. MAC BRYAN, E.F. VAN DE VELDE, Hypercube algorithms and implementations, SIAM J. Sci. Stat. Comput. 8, 2 (1987), s227-s287

[Ryt] W. RYTTER, On efficient parallel computations for some dynamic programming problems, Theoretical Computer Science 59 (1988), 297-307

[Saa1] Y. SAAD, Communication complexity of the Gaussian elimination algorithm on multiprocessors, Lin. Alg. Appl. 77 (1986), 315-340

[Saa2] Y. SAAD, Gaussian elimination on hypercubes, in Parallel Algorithms and Architectures, M. Cosnard et al. eds., North-Holland (1986), 5-18

Article [2]

Path planning on a ring of processors

Intern. J. Computer Math., Vol. 32, pp. 61–74
Reprints available directly from the publisher
Photocopying permitted by license only

PATH PLANNING ON A RING OF PROCESSORS*

SERGE MIGUET and YVES ROBERT

Laboratoire de l'Informatique du Parallélisme LIP-IMAG, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

(*Received 15 May 1989*)

In this paper, we discuss the implementation of Bitz and Kung's path planning algorithm on a ring of general-purpose processors. We show that Bitz and Kung's algorithm, originally designed for the Warp machine, is not efficient in this context, due to the intensive inter-processor communications that it requires. We design a modified version that is much more performing. The new version updates a segment of k positions within a step and allocates blocks of r consecutive rows of the map to the processors in a wraparound fashion. Bitz and Kung's algorithm corresponds to the situation $(k, r) = (1, 1)$. We analytically determine the optimal values of the parameters (k, r) which minimize the parallel execution time as a function of the problem size n and of the number of processors p. The theoretical results are nicely corroborated by numerical experiments on a ring of 32 transputers.

KEY WORDS: Path planning, image sweep, ring of processors, data allocation strategies, parallel algorithms, dynamic programming.

C.R. CATEGORIES: C.1.2, F.2.2, I.4.0.

1. INTRODUCTION

Given a map on which each position is associated with a traversability cost, the path planning problem is to find a minimum-cost path from a source position to every other position in the map. Bitz and Kung [2] have recently proposed a dynamic programming algorithm to solve the problem, and they have mapped this algorithm onto the linear systolic array in the Warp machine (Annaratone *et al.* [1]).

In this paper, we discuss the implementation of Bitz and Kung's path planning algorithm on a ring of general-purpose processors. We show that Bitz and Kung's algorithm is not efficient in this context, due to the intensive inter-processor communications that it requires. We design a modified version that is much more performing. The new version updates a segment of k positions within a step and allocates blocks of r consecutive rows of the map to the processors in a wraparound fashion. Bitz and Kung's algorithm corresponds to the situation $(k, r) = (1, 1)$. We analytically determine the optimal values of the parameters (k, r) which minimize the parallel execution time as a function of the problem size n and

*This work has been supported by the Research Project C3 of CNRS.

NW	N	NE
W	p	E
SW	S	SE

Figure 1 Labeling the eight neighbors of a position p.

of the number of processors p. We characterize the ring of processors by three constants: the elemental arithmetic time τ_a, the communication startup β, and the elemental communication time τ_c. The time to transfer L words between two adjacent processors is modelized by $\beta + L\tau$ (Geist and Heath [3], Miguet and Robert [7], Saad [9]). The theoretical results are corroborated by numerical experiments on a ring of 32 transputers.

2. PATH PLANNING ALGORITHM

A map M is an $n \times n$ grid of positions, for some positive integer n. The eight neighbors of a position p are indicated by the corresponding cardinal point in the compass (see Figure 1).

Each position p is associated with a non-negative real-number $tc(p)$ corresponding to the traversability cost of the position. Given a position p and a neighbor q of p, the edge (p, q) is weighted with a cost $c(p, q) = (tc(p) + tc(q))/2$ if $q \in \{N, S, W, E\}$ and $c(p, q) = (tc(p) + tc(q))\sqrt{2}/2$ otherwise: the $\sqrt{2}$ multiplier reflects the added travelling distance due to the diagonal connection. Given a position, called the source, we want to compute the shortest path (or minimum-cost path) from it to every position in the map.

Bitz and Kung [2] propose a dynamic programming algorithm to solve the problem. Initially, the best-known cost $f(p)$ for every position p in the map is assigned the value 0 at the source and ∞ at all other positions. The algorithm performs a succession of red and blue sweeps of the map.

2.1. Red Sweep

The red sweep is a forward scan of the map M in the row-major ordering. During the red sweep, each position p is updated according to the red mask depicted in Figure 2.

For the current position p of the sweeping, we update the best-known cost $f(p)$ if there exists a better path passing by one of the red neighbors of p. For instance if the best-known cost $f(W)$ of the west neighbor of p plus the cost $c(W, p)$ of the edge from W to p is smaller than $f(p)$, we update $f(p)$ into $f(p) := f(W) + c(W, p)$. In the general case, the update of $f(p)$ is defined as

$$f(p) := \min(f(p), f(W) + c(W, p), f(NW) + c(NW, p), f(N) + c(N, p), f(NE) + c(NE, p))$$

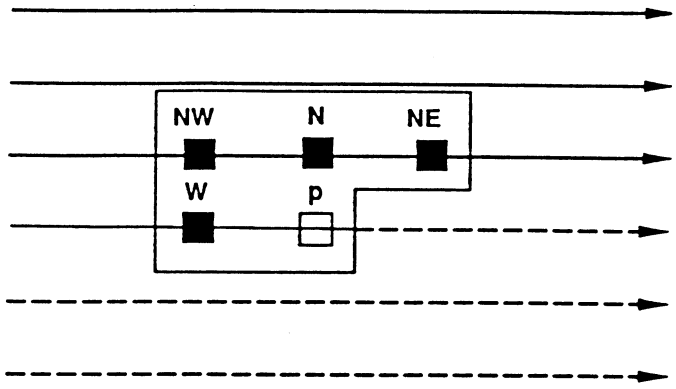


Figure 2 Red sweep and the associated mask.

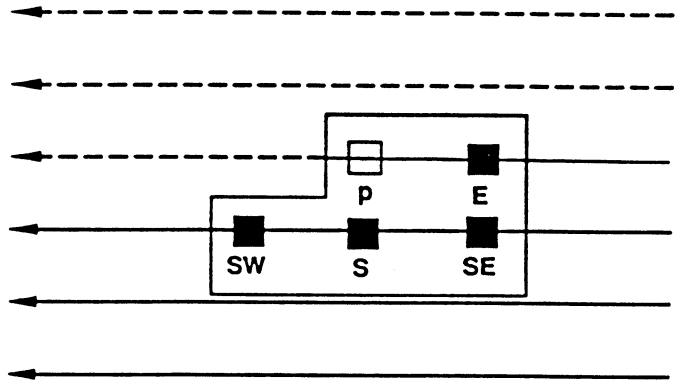


Figure 3 Blue sweep and the associated mask.

that is

$$f(p) := \min(f(p), f(W) + (tc(p) + tc(W))/2, f(NW) + (tc(p) + tc(NW))\sqrt{2}/2,$$

$$f(N) + (tc(p) + tc(N))/2, f(NE) + (tc(p) + tc(NE))\sqrt{2}/2). \quad \text{(RS)}$$

2.2. Blue Sweep

The blue sweep scans the map M in the reversed row-major ordering as shown in Figure 3. For the current position p of the sweeping, the update of $f(p)$ is defined similarly as for the blue sweep, but using the blue neighbors instead of the red ones:

$$f(p) := \min(f(p), f(E) + c(E, p), f(SE) + c(SE, p), f(S) + c(S, p), f(SW) + c(SW, p))$$

that is

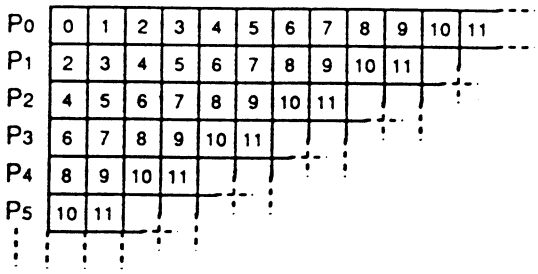


Figure 4 Time-steps for Bitz and Kung's parallel algorithm.

$$f(p) := \min(f(p), f(E)+(tc(p)+tc(E))/2, f(SE)+(tc(p)+tc(SE))\sqrt{2}/2,$$
$$f(S)+(tc(p)+tc(S))/2, f(SW)+(tc(p)+tc(SW))\sqrt{2}/2). \quad \text{(BS)}$$

2.3. Path Planning Algorithm

Given the initial values stated above, the red and blue sweeps are performed alternatively until no values are changed in one sweep. Let us color the edges of a path according to their directions: edges pointing to *W*, *NW*, *N* and *NE* directions are colored blue, whereas edges pointing to *E*, *SE*, *S* and *SW* directions are colored red. Then Bitz and Kung [2] show that the number of required sweeps before all positions receive their final values is C or $C+1$, where C is the maximum number of color changes in a shortest path from the source to any other position. Hence in the worst case, the number of required sweeps can be as large as $O(n^2)$. However in practical situations, it is expected to be much smaller than n (Bitz and Kung [2]).

In the following, we concentrate upon the parallel implementation of a single sweep (a red sweep) on a ring of processors.

3. PARALLEL IMPLEMENTATION

We briefly recall Bitz and Kung's solution for mapping the path planning algorithm onto the Warp. Such a solution is not suited to a ring of general-purpose processors, and we derive a modified version that is much more performing.

3.1. Bitz and Kung's Mapping Method

We consider a ring of processors numbered from 0 to $p-1$. Each row of the map is assigned to a processor. Assume first that the number of processors p is equal to the problem size n. In this case processor i gets row i, $0 \leqq i < n$. For the red sweep, immediately after processor i has computed the value of two positions, it will pass these values to processor $i+1$ to get it started. We summarize in Figure 4 the time-steps at which each position is updated.

At time $2i+j$, processor P_i operates as follows (wherever indices make sense):

- it receives position $(i-1, j+1)$ from P_{i-1}
- it updates position (i,j)
- it sends position (i,j) to P_{i+1}.

When p is smaller than n, partitioning techniques must be considered., Assume for the sake of simplicity that p divides n. Bitz and Kung propose to assign the rows of the map to the processors in a wraparound fashion: processor i gets rows j such that $j = i \bmod p$. The wrap mapping is a widely used technique to well balance the workload among the processors [3, 7, 8, 9]. Now P_0 needs to receive computed values from P_{p-1}. Note that P_0 receives the first value $(p-1, 0)$ from P_{p-1} at time $2p-1$. At time $2p$, P_0 receives the second value $(p-1, 1)$ and updates position $(p, 0)$. Hence we do not want P_0 to finish the updating of row 0 before time $2p$, otherwise it would stay idle for a while. This implies that $n-1 \geqq 2p$. If $n-1 > 2p$, P_0 will simply store the values it receives from P_{p-1} until it starts the updating of its second row.

We see that the latency between the startup times of two adjacent proccessors is small (two time-steps). The major drawback of the algorithm is that it involves many short communications between the processors. For current distributed memory machines, the time to transfer L words between two adjacent processors can be modelized by $\beta + L\tau$, and it turns out that β is significantly higher than τ ([3, 8, 9], see also the experiments reported in Section 5). This renders the cost of small messages prohibitive.

We explain below how to modify Bitz and Kung's algorithm in order to decrease the communication overhead. We describe the new algorithm informally, and postpone its complexity analysis up to next section.

3.2. Updating a Segment of Length k

The first way to decrease the communication overhead is to use longer messages. We use the same mapping strategy as before, but we update a segment of k consecutive positions at each step. The algorithm is illustrated in Figure 5. Note that k does not need to be a divisor of n. In Figure 5, we let l_0 be the number of positions updated by P_0 at time 0: we choose $l_0 = k-1$ in our implementation, just as if P_0 had received k fictitious values before beginning (but l_0 can be any number between 1 and $k-1$). Each processor always updates k positions, except they may be for the first and last updates: we start the update of the next row while finishinng the update of the current row (see Figure 5). The condition for P_0 not to finish its first before receiving data from P_{p-1} will be derived in the next section: we obtain the condition $(p+1)k \leqq n-p$.

The number of data items communicated between two neighbor processors is exactly the same as before, but the larger k, the more efficiently the communications are performed. On the other hand, the larger k, the greater the latency between the startup times of two adjacent processors. We must be ready to find a

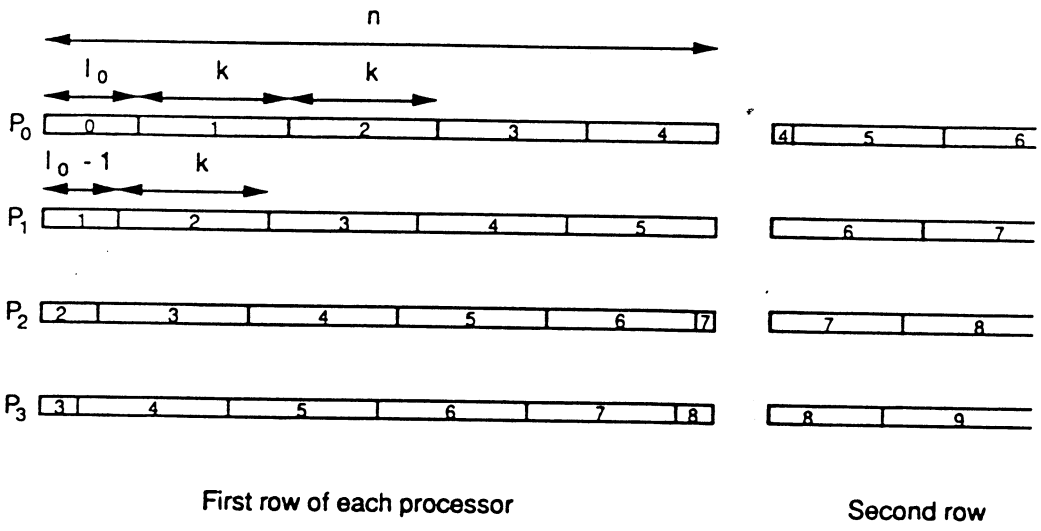


Figure 5 Time-steps when updating a segment of length k.

compromise between the two contradictory exigences of minimum startup delay (small k) and inexpensive communications (large k).

3.3. Moving to New Mapping Strategies

Another way to decrease the communication overhead is to communicate less data items between neighbor processors. We now consider more general allocation functions than the wrap mapping, and we assign blocks of r consecutive rows to the processors in a wraparound fashion. For instance with $r=3$, $n=36$ and $p=4$ we have the following repartition:

	P_0	P_1	P_2	P_3
Rows	0, 1, 2	3, 4, 5	6, 7, 8	9, 10, 11
	12, 13, 14	15, 16, 17	18, 19, 20,	21, 22, 23
	24, 25, 26	27, 28, 29	30, 31, 32	33, 34, 35

Such a repartition is illustrated in Figure 6, Analytically, processor i gets rows j such that $i=\text{floor}\,(j/r) \bmod p$, $0 \leqq j \leqq n-1$.

The time-steps are depicted in Figure 7. At each step except the first and last ones, all the processors update a parallelogram of $r*k$ positions. Just as before for $r=1$, we start the update of the next block while finishing the update of the current block (see Figure 7).

The condition that k and r must meet to keep all processors activated is the following: $(p+1)k \leqq n-pr$ (see next section). We show in Figure 8 an example where this condition is not met: we see that P_0 is idle at time 4, because it has not received in time from P_3 the first positions of row 11.

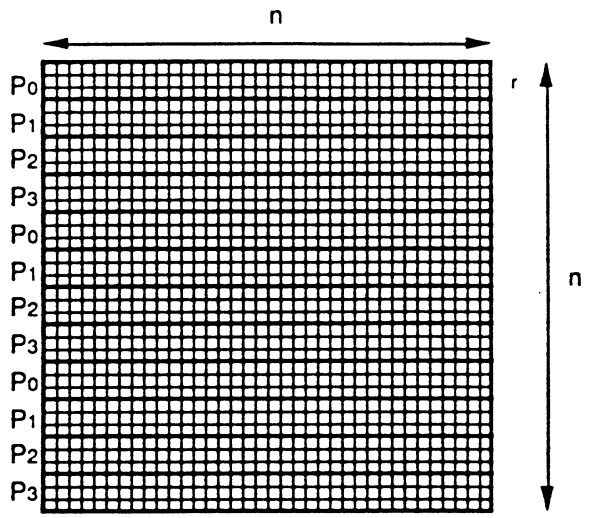


Figure 6 Block-r mapping, $n=36$, $p=4$, $r=3$.

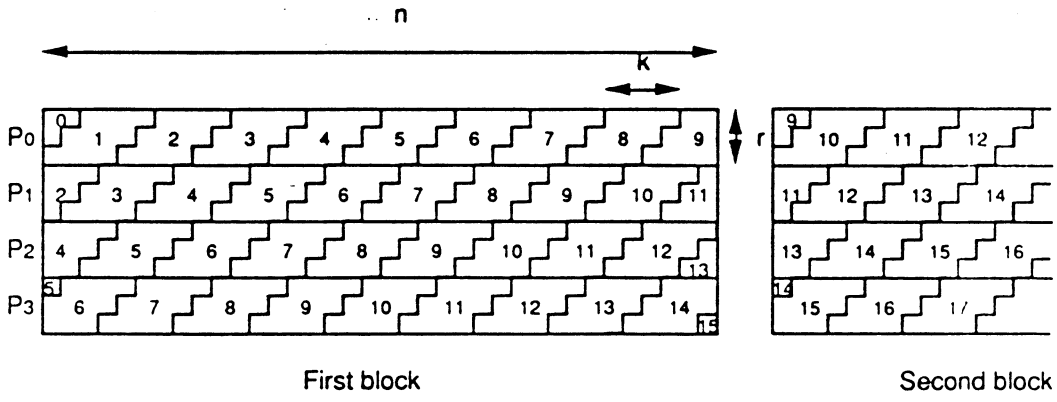


Figure 7 Parallel algorithm with $n=36$, $p=4$, $r=3$ and $k=4$.

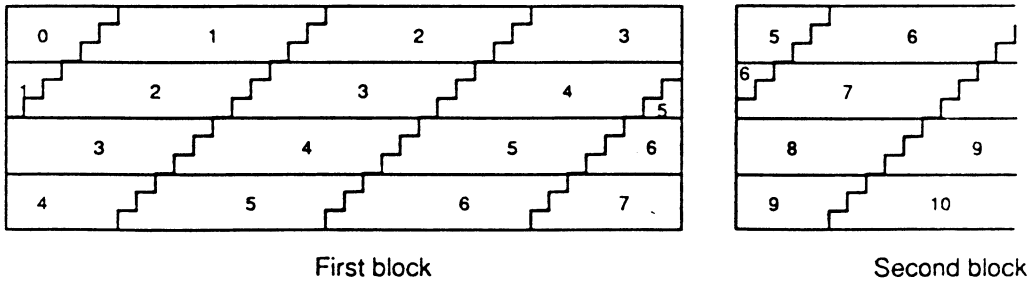


Figure 8 Parallel algorithm with $n=36$, $p=4$, $r=3$ and $k=11$.

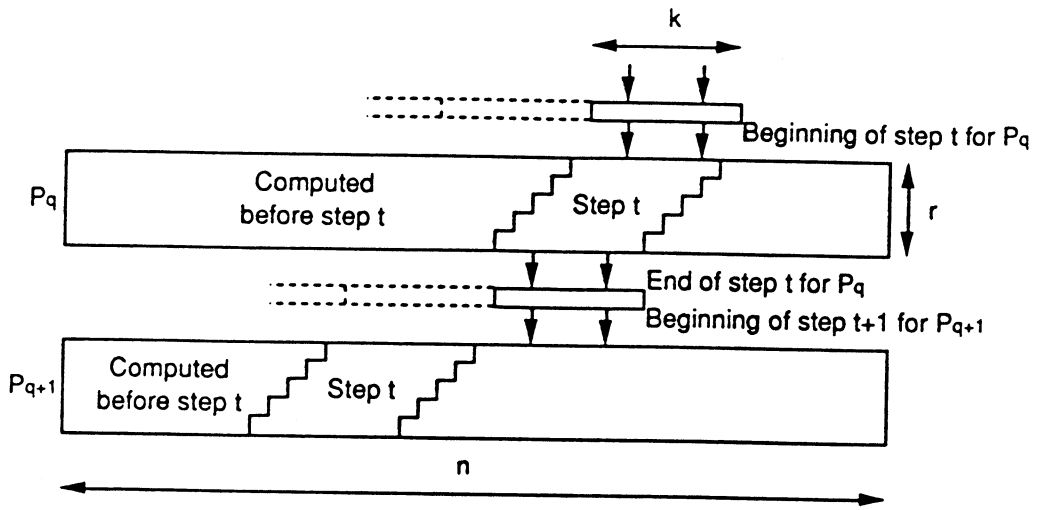


Figure 9 Communications between two neighbor processors.

Now, the number of data items communicated between two neighbor processors is r times smaller than in Bitz and Kung's implementation, because the processors only need to exchange informations relative to the boundary rows of each block. Segments belonging to an internal row of a block do not require any inter-processor communication. We illustrate the communications between two neighbor processors in Figure 9.

The price to pay for such a dramatic reduction of the communication volume is again an increase in the latency between the startup times of two adjacent processors. Hence the best value of r will result in a compromise, just as the best value of k.

In the next section, we perform a complexity analysis. Given n and p, we analytically determine the values of k and r that minimize the parallel execution time.

4. PERFORMANCE EVALUATION

In this section, we analyse the performances of the parallel algorithm described above. For the arithmetic, we let τ_a be the elemental time needed for updating a position during the sweep (formulae *RS* or *BS*). Since there are n^2 positions to update during a sweep, the sequential time for a problem of size n is $T_{seq} = n^2\tau_a$.

4.1. Memory Requirement

The space requirement for the sequential algorithm is proportional to the size of the map, that is n^2 positions. For each position, we need to store a word for its current value and 8 words for the traversability cost of the 8 adjacent edges. Let

us choose as a unit the memory requirements for a position. Given a single processor with a memory of size M, this implies that the maximal problem size that can be dealt with is $n_{\max,1}=M^{1/2}$. Consider now a ring of p processors. First of all, we have to determine the relationship between p and n. We have p memories of size M, so that we can solve in parallel a problem of size at most $n_{\max,p}\cong(pM)^{1/2}$. Note that we neglect here any additional storage required by the parallel implementation, such as the need for communication buffers. In fact, the value $n_{\max,p}$ above is an upper bound.

As stated before, we consider an allocation by blocks of consecutive rows of size r in a wraparound fashion, where $1\leqq r\leqq n/p$. For the sake of simplicity (without loss of generality), we assume that $p*r$ divides n, so that each processor holds the same number of rows in its local memory.

4.2. Parallel Execution Time

Even though the implementation is asynchronous, we can view the parallel algorithm as a succession of time-steps, where at each time-step each processor updates r segments of k positions. Within a time-step, processor P_i receives a message of length k from processor P_{i-1}, updates $r*k$ positions, and sends a message of length k to P_{i+1} (indices are taken mod p). Note that the emisson is non-blocking, whereas the reception is. P_i does not wait for its emission to be completed before moving to the next step. As a consequence, the communication within a time-step has a cost equal to $\beta+k\tau_c$. The total time needed to perform a time-step is $\tau_{\text{step}}=\beta+k\tau_c+rk\tau_a$.

To evaluate the total number of time-steps in the algorithm, we first compute the time-step at which a processor P_q, $0\leqq q\leqq p-1$, initiates its computation. Recall that P_0 updates l_0 positions in its first row at time $t_0=0$. We see that P_1 updates $l_1=(l_0-r) \bmod k$ positions in its first row at time $t_1=1+\text{ceiling}((r-l_0)/k)$, and more generally, that P_q updates l_q positions in its first row at time t_q, where

$$l_q=(l_0-q*r) \bmod k, \qquad t_q=q+\text{ceiling}((q*r-l_0)/k).$$

Now, we derive easily the total number of time-steps T_p, since P_{p-1} is the last processor to end its computation. After updating its first parallelogram, P_{p-1} has still

$$\text{ceiling}((n^2/(p*r)-l_q+r-1)/k)$$

parallelograms to update, so that

$$T_p=t_{p-1}+\text{ceiling}((n^2/(p*r)-l_q+r-1)/k).$$

The parallel execution time of the algorithm is then $T_{//}=\tau_{\text{step}}*T_p$.

This evaluation is valid only if the processors are not kept idle, waiting for some data they need from their predecessors. As explained in the previous section, this

condition is equivalent to ensuring that P_0 has not finished the updating of its first block before receiving from P_{p-1} the data that it needs for its second block. P_0 performs its first reception at time t_p. At that time it has already updated $l_0 + k * t_p$ positions in its first block. Their condition is that the sum of the remaining positions in its first block plus what it receives at time t_p is greater than or equal to k:

$$n - (l_0 + k * t_p) + l_p \geqq k.$$

After some algebra we get

$$k \leqq \frac{n - pr}{p+1}.$$

We retrieve the condition illustrated in Figures 7 and 8.

Neglecting low order terms and ceiling functions, we obtain the following analytical evaluation for the parallel execution time $T_{//}$.

PROPOSITION *Given a problem of size n and a ring of p processors, the parallel execution time $T_{//}$ for a block-r allocation, $1 \leqq r \leqq n/p$, using segments of length k, $1 \leqq k \leqq (n - p * r)/(p+1)$, is*

$$T_{//} = (\beta + k\tau_c + rk\tau_a)[(p-1)(1 + r/k) + n^2/(p * r * k)].$$

Given n, p and r it is easy to find the value $k_{opt}(r)$ of k that minimizes the execution time $T_{//}$. We obtain the value

$$k_{opt}(r) = \min(k_{max}(r), k_{//}(r))$$

where $k_{max}(r) = (n - pr)/(p+1)$ and $k_{//}$ is the optimal value obtained from the expression of $T_{//}$:

$$k_{//}(r) = \left[\frac{\beta}{\tau_c + r\tau_a} \left(\frac{n^2}{p(p-1)r} + r \right) \right]^{1/2}.$$

Given n, p and numerical values for the parameters β, τ_c, τ_a, it is easy to compute k_{opt} and to plug it into the expression of $T_{//}$ to determine the best value of r. We report numerical experiments in the next section.

5. NUMERICAL EXPERIMENTS

We report in this section numerical experiments on a ring of Inmos Transputers T414, using up to 32 processors. We use a FPS T40 hypercube [5], which we configure as a ring. First of all we have to determine τ_a and τ_c.

Each update in (*RS*) or (*BS*) amounts to four additions, four comparisons, plus

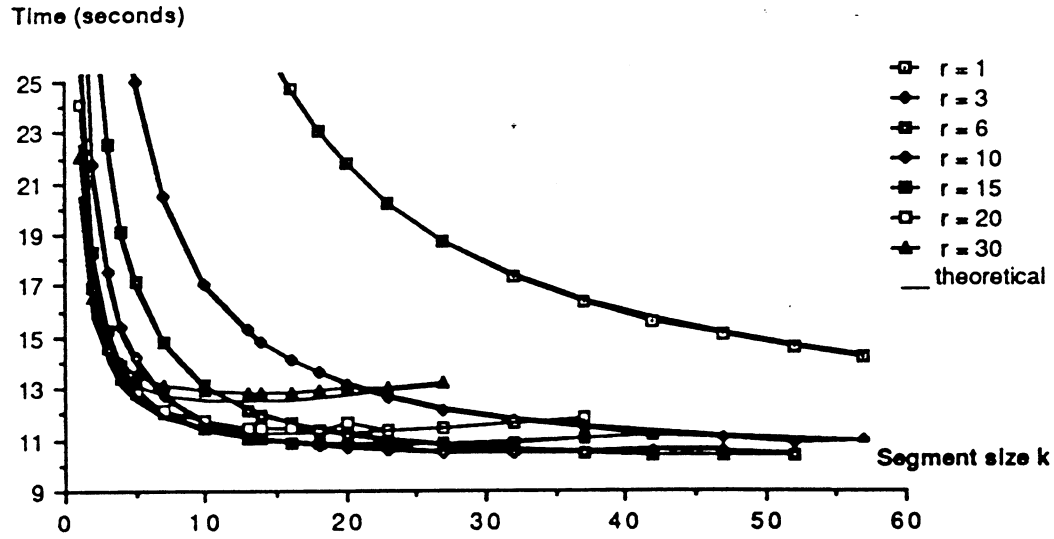


Figure 10 Parallel time as a function of the segment size k; problem size $n=1920$; number of processors $p=32$.

some conditional logic. We find that $\tau_a = 75e-6$ seconds. For the communications, we obtain experimentally that the time to transfer L words between two adjacent processors is $\beta + L\tau_c$, with $\beta = 2e-3$ seconds and $\tau_c = 12.5e-6$ seconds.

The first thing we check is that the parallel execution time obeys our formulas. We fix n and p and let the segment size k vary, with various values of the blocks size r. We superimpose in Figure 10 the experimental and theoretical curves (with the previous values of β, τ_c and τ_a) for the parallel execution time. There is a very good adequation between the curves.

We find experimentally the optimal values of r and k: $r_{opt}=6$ and $k_{opt}=52$. For these values we obtain $T_{//}=10.36$ seconds. These values are in good accordance with the theory: if we plug the values of $n=1920$ and $p=32$ in the formulas of the previous section, we obtain

$$k_{opt}(r) = k_{max}(r) \quad \text{for } r \leqq 5, \qquad k_{opt}(r) = k_{//}(r) \quad \text{for } r \geqq 6$$

and $T_{//}$ is minimum for $r=6$ and $k=k_{//}(r)=52$. We obtain $T_{//}=10.52$ seconds with the analytical expressions.

We point out that the execution time with (r_{opt}, k_{opt}) is divided by a factor of 23.8 as compared to Bitz and Kung's algorithm which corresponds to the values $(r, k)=(1, 1)$ and for which the execution time is as high as 247 seconds.

In Figure 11, we plot the speedups that we obtain with 32 processors when solving a problem of size $n=1920$. Note that these speedups are computed according to Gustafson's recent proposal [4], in that they are normalized by the amount of arithmetic operations which they require (since it is impossible to solve such a large problem with a single processor). Using 32 processors, we report acceleration factors as high as 26.

We finally show in Figure 12 a 3D-plot of the efficiency $e(r, k)$ of the algorithm

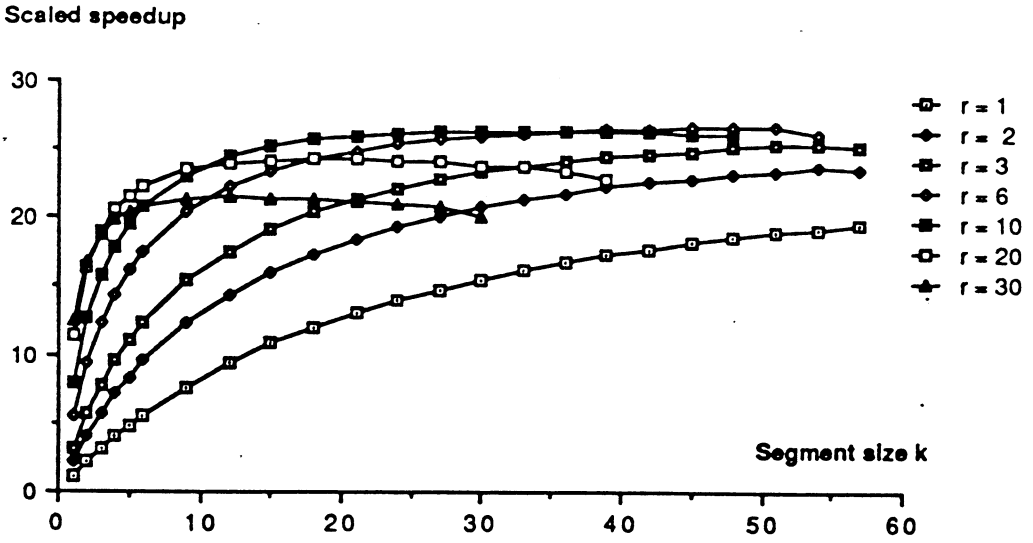


Figure 11 Scaled speedup as a function of the segment size k; problem size $n=1920$; number of processors $p=32$.

to better visualise the influence of the parameters on the execution time. The surface we show is the function $e(r,k)$ for the following values of r and k: $1 \leqq r \leqq n/(2p)$, $1 \leqq k \leqq k_{max}(r)$. The optimal efficiency $e=0.81$ is obtained for the highest point of this surface, with $r=6$ and $k=52$.

6. CONCLUSION

In this paper, we have discussed the implementation of Bitz and Kung's path planning algorithm on a ring of general-purpose processors. We have designed a modified version that updates a segment of k positions within a step and allocates blocks of r consecutive rows of the map to the processors in a wraparound fashion. We have analytically determined the optimal values of the parameters (k,r) which minimize the parallel execution time as a function of the number of processors p and of the problem size n. The theoretical results are nicely corroborated by numerical experiments on a ring of 32 transputers. We obtain a speedup of 23.8 over Bitz and Kung's algorithm.

Path planning on a ring of processors

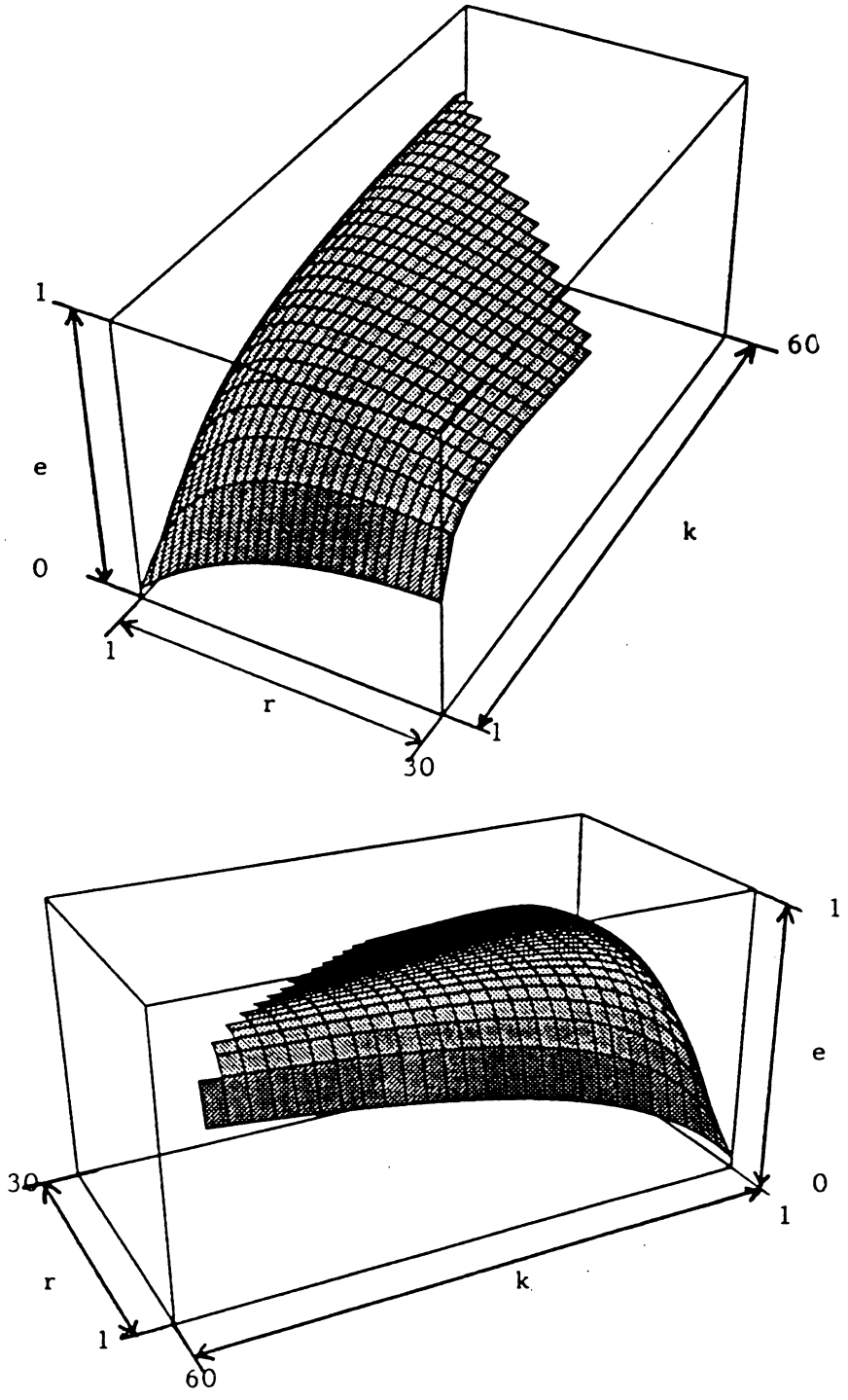


Figure 12 3D-plot of the efficiency $e(r, k)$, with $n = 1920$, $p = 32$.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu and J. A. Webb, The Warp computer: Architecture, implementation and performance, *IEEE Trans. Computers* **36** (1987). 1523–1538.

[2] F. Bitz and H. T. Kung, Path planning on the Warp computer: Using a linear systolic array in dynamic programming, *Intern. J. Computer Math.* **25** (1988), 173–188.

[3] G. A. Geist and M. T. Heath, Matrix factorization on a hypercube multiprocessor. In: *Hypercube Multiprocessors 1986*, M. T. Heath, ed., SIAM, 1986, pp. 161–180.

[4] J. L. Gustafson, Reevaluating Amdahl's law, *Communications of the A.C.M.* **31** (1988), 532–533.

[5] J. L. Gustafson, S. Hawkinson and K. Scott, The architecture of a homogeneous vector supercomputer. In: *Proceedings of ICCP 86*, IEEE Computer Science Press, 1986, pp. 649–652.

[6] K. Hwang, Advanced parallel processing with supercomputer architectures, *Proceedings of the IEEE 75* **10** (1987), 1348–1379.

[7] A. Miguet and Y. Robert, Dynamic programming on a ring of processors, *First European Workshop on Hypercube and Distributed Computers* (1989), to appear. Also available as *Technical Report LIP-IMAG 89-01.*

[8] O. A. Mac Bryan and E. F. Van de Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Stat. Comput.* **8** (1987), s227–s287.

[9] Y. Saad, Gaussian elimination on hypercubes. In: *Parallel Algorithms and Architectures*, M. Cosnard *et al.*, eds., North-Holland, 1986, pp. 5–18.

Article [3]

Symmetric Matrix Vector Product on a ring of processors

Information Processing Letters 35 (1990) 239–248
North-Holland

28 August 1990

SYMMETRIC MATRIX-VECTOR PRODUCT ON A RING OF PROCESSORS *

Ken GRIGG **, Serge MIGUET and Yves ROBERT

Laboratoire de l'Informatique du Parallélisme LIP-IMAG, École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

Communicated by L. Kott
Received 7 December 1989
Revised 27 March 1990

In this paper, we discuss the implementation of the product of a symmetric matrix by a vector on a unidirectional ring of general-purpose processors. This kernel arises in several applications: we discuss one of them issued from control systems. The difficulty comes from the fact that only one half of the matrix is stored – in a distributed way – in the processor memories. We derive an efficient algorithm, and we analytically determine its performance as a function of the problem size n and of the number of processors p. The algorithm is shown to be optimal, both with or without the assumption that arithmetic and communications can overlap. The theoretical results are nicely corroborated by numerical experiments on a ring of 32 transputers.

Keywords: Symmetric matrix-vector product, ring of processors, data allocation strategies, parallel algorithms, parameter estimation

1. Introduction

The matrix-vector product is an elementary linear algebra computational kernel that arises in numerous applications. As such, its parallelization has been extensively studied on shared memory systems, on systolic arrays and on distributed memory machines. In this paper, we address the problem of implementing the product of a *symmetric* matrix by a vector on a distributed memory machine.

Of course the problem reduces to a standard matrix-vector product if we assume that the entire matrix is stored in distributed memory. However, in many cases, storage limitations are critical and storing half the matrix would permit to double the maximum size of problems that can be solved. As pointed out by Gustafson [4], with parallel machines, we do not want only to solve the same problem faster, but we also want to solve larger problems.

These results have applications in the area of parameter estimation for fast adaptive control systems, which has been the subject of much research in the recent past due to increasing demands on control performance. Parameter estimation provides the means with which to identify physical system model parameters (constant or time-varying), and is a computationally intensive task. In attempting to derive parallel algorithms to be used in these adaptive control systems, the problem of the symmetric matrix-vector product arises.

The recursive least squares estimation method, for example, makes use of a symmetric matrix that tracks the covariance errors in the model parameters. This symmetric matrix is used recursively to

* This work has been supported by the research project C3 of CNRS.

** Permanent address: Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6. Partial funding for Ken Grigg was made available under the Operating Grants program of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Volume 35, Number 5 INFORMATION PROCESSING LETTERS 28 August 1990

update the parameter estimates, a task which involves a matrix-vector product. The matrix can be very large, depending upon the order of the model of the system and on the number of parameters estimated (e.g. up to 4000 for adaptive equalization applications) [8].

First (in Section 2) we briefly recall how the standard matrix-vector product has been dealt with in the literature. Then (in Section 3) we present our algorithm for the symmetric case. In Section 4 we move to an analytical model for determining the performance of the algorithm, and we show that it is optimal under a nonrestrictive hypothesis.

We characterize the ring of processors by the number of processors and three constants: the elemental arithmetic time τ_a, the elemental communication time τ_c, and the communication start-up time β. The time to transfer L words between two adjacent processors is therefore calculated as $T = \beta + L\tau_c$ [3,7,9]. The theoretical results are corroborated by numerical experiments on a ring of 32 transputers.

2. Standard matrix-vector product

Consider the product $y = Ax$ of an $n \times n$ matrix A by an n-vector x. The simplest solution to implement this product on a distributed machine is to configure the machine as an oriented ring

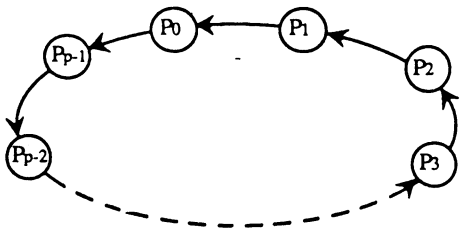


Fig. 1. An oriented ring of processors.

(see Fig. 1) and to equally distribute the rows of A to the processors. Each processor receives the components of x corresponding to the rows of A that it holds and is responsible for computing the same components of the product y.

This scheme requires inter-processor communications because each processor needs all the components of x to compute any component of y:

$$y_i = \sum_{j=0}^{n-1} a_{ij} * x_j.$$

We illustrate the scheme for a ring of p processors in Fig. 2. Processor P_q, $0 \leqslant q < p$, holds the block of $r = n/p$ consecutive rows of indices $q * r, q * r + 1, \ldots, q * r + (r-1)$, P_q also holds the vectors X_q and Y_q of the corresponding components of x and y. At step $s = 0$, P_q initiates the computation of Y_q using the diagonal subblock of A marked with 0's in the figure. At each step s,

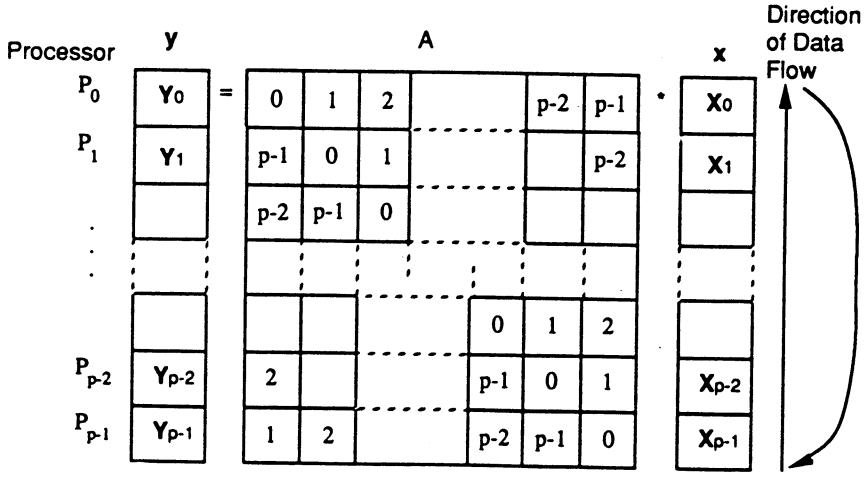


Fig. 2. A standard matrix-vector product. The elements of the A matrix shown here have been replaced with the time steps in which they are used to calculate the corresponding values of the vector y.

Volume 35, Number 5 INFORMATION PROCESSING LETTERS 28 August 1990

where $0 < s < p$, P_q receives a vector $X_{(s+q) \bmod p}$ of r components of x from $P_{(q+1) \bmod p}$ and updates Y_q using the block marked with the time step number in the figure. After $p-1$ time steps, processor P_q contains the resulting element subvector Y_q. Distribution of the results, if necessary, would then be achieved by another rotation of the ring.

A very similar solution has been proposed by MacBryan and van de Velde [6,7]. The idea is to distribute the columns of A (rather than the rows) to the processors, together with the corresponding components of x. The components of the result vector y are shifted along the ring and repeatedly updated by the processors, while the components of x remain in their original processors.

3. Symmetric matrix-vector product

We outline in this section the algorithm for computing the product of a symmetric matrix by a vector on a ring of processors. The lower triangular part of the matrix A is distributed among the processors. We number the processors in the ring from 0 to $p-1$, and we assume that the ring is oriented as before: processor P_q sends messages to $P_{(q-1) \bmod p}$. Also as before, we number the rows of A and the components of x and y from 0 to $n-1$.

For clarity here, we assume each processor P_q also receives a copy of the entire vector x before the algorithm begins (we will suppress this hypothesis later).

3.1. Case $n = p$

Consider first the case where the number of rows of A, n, is equal to the number of processors p. In this case, we assign row i of A to processor P_i. P_i is then responsible for computing the element y_i. We then form $y_i = y_i' + y_i''$ where:

$$y_i' = \sum_{j=0}^{i} a_{ij} * x_j, \qquad y_i'' = \sum_{j=i+1}^{n-1} a_{ij} * x_j.$$

Processor P_i can compute y_i' without any communication. Since $a_{ij} = a_{ji}$ for symmetric matrices, we have:

$$y_i'' = \sum_{j=i+1}^{n-1} a_{ji} * x_j.$$

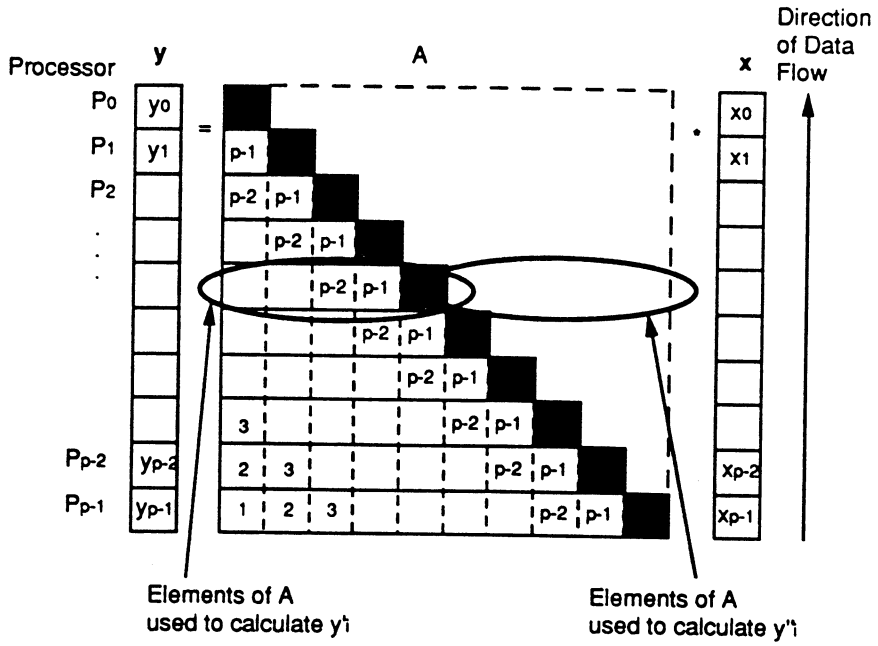


Fig. 3. A symmetric matrix-vector product ($n = p$). The elements of the A matrix shown here have been replaced with the time steps in which they are used to calculate the corresponding values of vector y.

Volume 35, Number 5 INFORMATION PROCESSING LETTERS 28 August 1990

Therefore y_i'' can be evaluated along the ring. Following Fig. 3, for example, at step 1, processor P_{p-1} sends $y_0'' = a_{n-1,0} * x_{n-1}$ to processor P_{p-2}, which updates this into $y_0'' = y_0'' + a_{n-2,0} * x_{n-2}$ at step 2, and so on until step $p-1$, when P_0 receives the final value of y_0''. More generally, several values of y_i'' are circulated along the ring, as described by the following algorithm:

Program for processor q
{computation of y_q''}
if $q = p-1$ **let** $u = 0$
  {P_{p-1} never receives u}
if $q \neq 0$
  for step $= p-q$ **to** $p-1$ **do**
    let $j = q + \text{step} - p$
      {j takes on 0, 1, 2, ..., $q-1$}
    compute $\gamma = a_{qj} * x_q$
    if $q \neq p-1$ **then receive** u
      {$u = \sum_{k=q+1}^{p-1} a_{kj} * x_k$}
    send $u + \gamma$
      {$u + \gamma = \sum_{k=q}^{p-1} a_{kj} * x_k$}
  endfor
endif
if $q \neq p-1$ **then receive** u
  {$u = y_q''$}
{computation of y_q' and finally y_q}
compute $y_q' = \sum_{j=0}^{q} a_{qj} * x_j$
compute $y_q = y_q' + y_q''$

As in Section 2 above, the resulting y_i elements are left in processors P_i and would then be distributed if necessary by another rotation of the ring.

3.2. General case: $p \leqslant n$

Now we move to the more general case where there are less processors than there are rows of A. We require that n be divisible by $2p$ and we let $r = n/(2p)$. We distribute two blocks of r rows to each processor to balance the arithmetic workload equally amongst the processors. If we index the blocks of r rows of A as block 0 to block $2p-1$, the ith block is then composed of rows $r * i$, $r * i + 1, \ldots, r * i + (r-1)$. We allocate blocks q and $(2p-1)-q$ to P_q (see Fig. 4).

Let us define some terminology. Given any block index i and any vector z of size n, we let Z_i be the subvector of the r components of z indexed by i. That is, components $r * i$, $r * i + 1$, $\ldots, r * i + (r-1)$. Similarly, A_{ij} denotes the square subblock of the matrix A made up of the rows and columns indexed by i and j respectively.

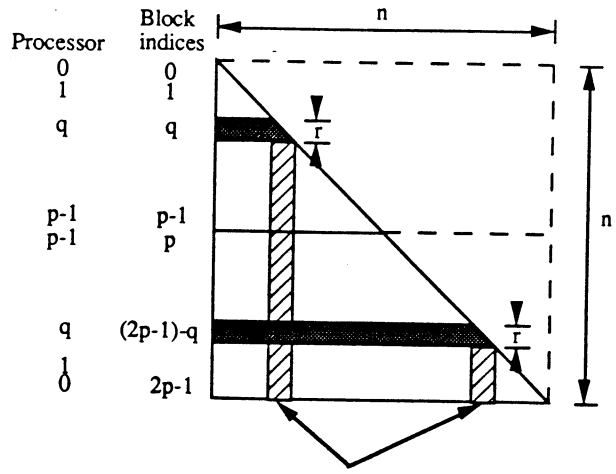


Fig. 4. Data allocation of A in the more general case ($r = n/(2p)$). The shadowed blocks of r rows are stored in processor q. The cross-hatched columns show where the elements for the complements of those rows are found.

Now, P_q is responsible for computing two segments of y, Y_q and $Y_{(2p-1)-q}$. As before, each component y_i in each of the two segments can be written as $y_i = y_i' + y_i''$. Processor P_q can compute the y_i' part without any communication. For the y_i'' part, again we let y_i'' be circulated along the ring and repeatedly updated. However, we transmit $2r$ values at each step. That is, we circulate two vectors, U_1 and U_2, of r components, along the ring for accumulating the values of Y_q'' and $Y_{(2p-1)-q}''$.

We still want the algorithm to complete within $p-1$ steps. This means that the computation of Y_q'' and $Y_{(2p-1)-q}''$ must be initiated at time step $s = 1$ in processor $P_{(q-1) \bmod p}$ to reach P_q in time for step $s = p-1$. When travelling along the ring, Y_q'' and $Y_{(2p-1)-q}''$ accumulate their final values. The situation is similar to the previous case $n = p$, but the difficulty arises from the fact that $Y_{(2p-1)-q}''$ needs less than $p-1$ updates, while Y_q'' requires more.

This situation is illustrated in Figs. 5 and 6. First (in phase 1 of the algorithm below) both Y_q'' and $Y_{(2p-1)-q}''$ are updated, while at the end (in phase 2) only Y_q'' is modified.

The algorithm follows. Note that the transpose of x, x^T, is seen because the computation of Y'' requires the use of $A^T x$, so instead we use the equality: $(x^T A)^T = A^T x$.

Program of processor q

{Processor P_q holds the blocks q and $(2p-1)-q$ (of r rows of A)}
{Computation of Y_q'' and $Y_{(2p-1)-q}''$}
{**PHASE 1**: both Y_q'' and $Y_{(2p-1)-q}''$ are updated}
let $w = (2p-1)-q$;
let $U_1 = [0,\ldots,0]$; **let** $U_2 = [0,\ldots,0]$;
if $q < p-1$
for step $= 1$ **to** $(p-1)-q$ **do**
let $j = q + \text{step}$; **let** $k = (2p-1)-j$;
{Use blocks $A_{w,j}$ and $A_{w,k}$}
compute $\Gamma_1 = (X_w^T * A_{w,j})^T$;
compute $\Gamma_2 = (X_w^T * A_{w,k})^T$;
send $(U_1 + \Gamma_1, U_2 + \Gamma_2)$
receive (U_1, U_2);
endfor
endif
{**PHASE 2**: only Y_q'' is updated}
if $q > 0$
for step $= p-q$ **to** $p-1$ **do**
let $j = q + \text{step} - p$;
{Use blocks $A_{q,j}$ and $A_{w,j}$}
compute $\Gamma_1 = (X_q^T * A_{q,j})^T$;
compute $\Gamma_2 = (X_w^T * A_{w,j})^T$;
send $(U_1 + \Gamma_1 + \Gamma_2, U_2)$
receive (U_1, U_2);
endfor
endif
compute $Y_q'' = U_1 + (X_w^T * A_{w,q})^T$;
{Add in opposite diagonal part}
let $Y_w'' = U_2$;
{Computation of Y_q' and Y_w'}
$Y_q' = \sum_{k=0}^{q-1} A_{q,k} * X_k + A_{q,q} * X_q$
{Only use lower part of $A_{q,q}$}
$Y_w' = \sum_{k=0}^{w-1} A_{w,k} * X_k + A_{w,w} * X_q$
{Only use lower part of $A_{w,w}$}
{Computation of Y_q and Y_w}
let $Y_q = Y_q' + Y_q''$;
let $Y_w = Y_w' + Y_w''$

Note that the blocks on the diagonal are not fully stored in the processors, so that the products $A_{q,q} * X_q$ and $A_{w,w} * X_w$ are computed using only the lower triangular halves of the blocks.

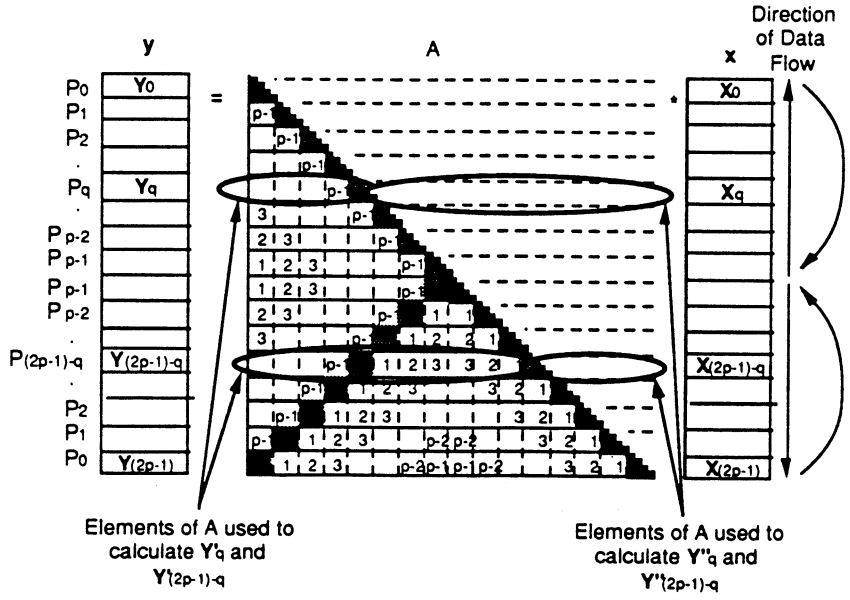


Fig. 5. Time-steps when $n = 2pr$.

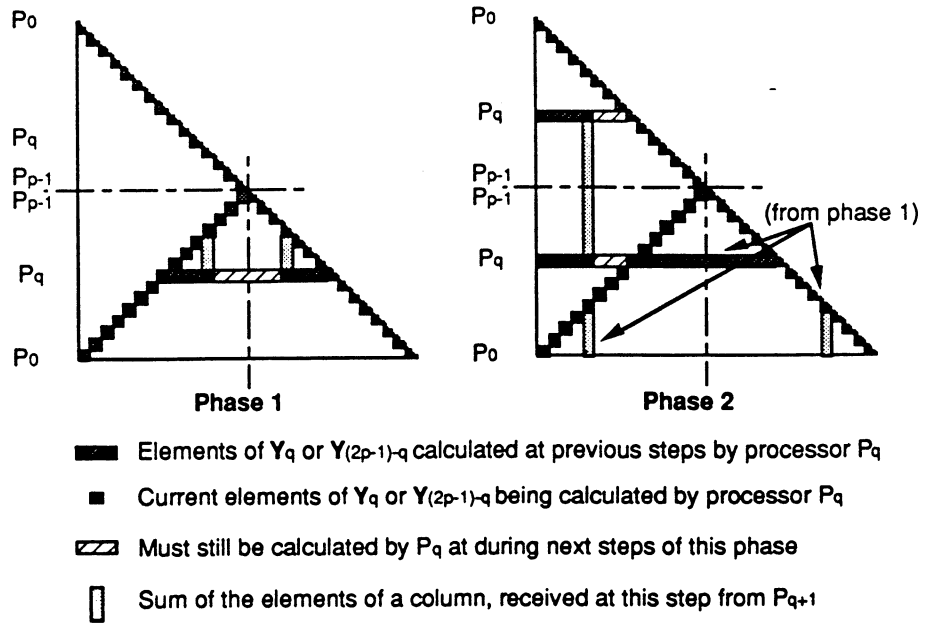


Fig. 6. Illustration of the two phases of the computation of Y''.

3.3. Distributing x

Finally we have to explain how to modify the algorithm when we assume that the vector x is distributed among the processors. A distributed x only affects the calculation of y', so the modification is straightforward:

- the computation of y' is interleaved with the updating of y'' along the ring;
- at step 1, each processor P_q initializes two vectors Y_1' and Y_2' as follows:
  let $Y_1' = A_{q,q} * X_q$
  let $Y_2' = A_{(2p-1)-q,(2p-1)-q} * X_{(2p-1)-q}$
  (as before, the diagonal block computations include only the lower triangular halves of the blocks);
- together with U_1 and U_2, P_q transmits two vectors $\Phi_1 = X_q$ and $\Phi_2 = X_{(2p-1)-q}$;
- at a given step $s \geq 2$, each P_q receives four segments U_1, U_2, Φ_1 and Φ_2.

U_1 and U_2 are updated as before, while Φ_1 and Φ_2 are propagated unchanged. However, they are used to update the local variables Y_1' and Y_2' when needed. Technically, there are two phases just as before:

(1) while $q + s < p$, P_q will update Y_2' using two blocks of the index $(2p-1)-q$; and
(2) afterwards P_q updates both Y_1' and Y_2'.

4. Performance evaluation

In this section, we analyze the performance of the parallel algorithm described above. Define τ_a to be the elemental time for a multiply-and-add. The sequential time for a matrix-vector product of size n is $T_{seq} = n^2\tau_a$. For the communication, we let the time to transfer L words between two adjacent processors be modeled by the expression $\beta + L\tau_c$, where β is the communication startup time, and τ_c is the elemental communication time [3,7,9].

4.1. Memory requirements

The space requirement for the sequential algorithm is proportional to the size of the lower half of the matrix, that is $n(n+1)/2$ words. Given a single processor with a memory of size M words, this implies that the maximum problem size that

can be dealt with is $n_{\max,1} \cong \sqrt{2M}$. Consider now a ring of p processors. We have p memories of size M, so that we can solve in parallel a problem of size at most $n_{\max,p} \cong \sqrt{2pM}$. Note that we neglect here any additional storage required by the parallel implementation, such as the need for communication buffers. In fact, the value $n_{\max,p}$ above is an upper bound.

As stated before, we distribute two blocks of r consecutive rows to each processor P_q, namely the rows q and $(2p-1)-q$. The memory requirement for each processor is then equivalent to one block of r rows of n elements, or $r * n$ words. For the sake of simplicity (without loss of generality), we assume that $2 * p * r$ divides n, so that each processor holds the same number of rows in its local memory.

4.2. Parallel execution time

4.2.1. With no overlap between arithmetic and communication

Assume first that arithmetic and communication do not overlap. In this case, we can easily compute the total execution time as the sum of the arithmetic time T_a and the communication time T_c (which includes idle time).

The arithmetic workload is equally balanced among the processors, so that:

$$T_a = \frac{T_{seq}}{p} = \frac{n^2\tau_a}{p}.$$

For the communication time, assume first that a copy of the entire vector x is available to all processors at the beginning. We have $p-1$ communications of length $2r$ (we concatenate message vectors U_1 and U_2 to save a startup time), so that:

$$T_c = (p-1)(\beta + 2r\tau_c) = (p-1)\left(\beta + \frac{n}{p}\tau_c\right).$$

The corresponding efficiency is

$$e_p = \frac{T_{seq}}{p(T_a + T_c)} = \frac{1}{1 + \left(\frac{p-1}{n^2}\right)\cdot\left(\frac{p\beta + n\tau_c}{\tau_a}\right)}.$$

Note that e_p rapidly tends to 1 as p/n tends to 0.

Now assume that each processor holds only two blocks of r components of x in its private memory. The length of the messages is doubled, so that:

$$T_c = (p-1)(\beta + 4r\tau_c) = (p-1)\left(\beta + 2\frac{n}{p}\tau_c\right).$$

We obtain easily the new value of e_p.

Theorem. *Consider the product of an order-n symmetric matrix A by an n-vector on an oriented ring of p processors, and assume that only the lower triangular half of the matrix is distributed among the processors. Under the assumption that each processor holds exactly n/p rows of the (lower half) of A, and that communication and arithmetic do not overlap, the algorithm described above is optimal in time. Its execution time is:*

$$T = \frac{n^2\tau_a}{p} + (p-1)\left(\beta + \frac{n}{p}\tau_c\right).$$

The theorem is valid also if we assume that x is duplicated or distributed among the processors. In the latter case, we assume that each processor is given n/p components of x.

Proof. T_a is obviously minimal. To see that T_c is minimal, let P_q be the processor which holds the last row of A. Consider any other processor $P_{q'}$ on the ring, and let $i_1, i_2, \ldots, i_{n/p}$ be the indices of the rows which have been allocated to them.

Assume first that x is duplicated. The n/p elements $a_{n-1,i_1}, a_{n-1,i_2}, \ldots, a_{n-1,i_{n/p}}$ stored in P_q are needed by $P_{q'}$ for the computation of the n/p independent components $y_{i_1}, y_{i_2}, \ldots, y_{i_{n/p}}$ of y, which means P_q has to send (at least) n/p distinct values to $P_{q'}$. In other words, P_q has to send a message of length n/p to every other processor, and these $(p-1)$ messages are distinct. Such an operation is known as scattering [10], and will take at least a time of $(p-1)(\beta + (n/p)\tau_c)$ on the oriented ring (see the complexity result of [2]). This shows that T_c in the algorithm is minimal.

Assume now that x is distributed equally among the processors. In addition to the n/p previous values, P_q must also send a copy of its components of x to its predecessor. Concatenat-

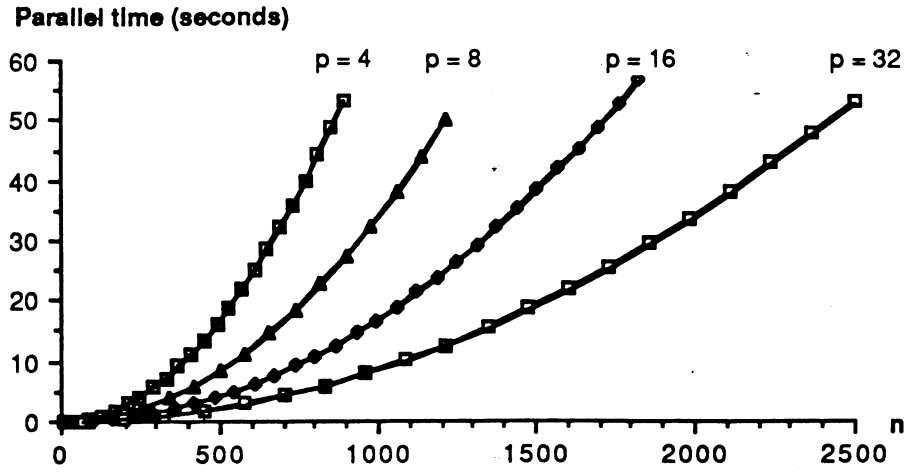


Fig. 7. Experimental and theoretical parallel execution time as a function of n and p.

ing all values into a single message is the best that can be achieved, and this leads to the optimal communication time of our algorithm. □

4.2.2. With overlap between arithmetic and communication

If the machine architecture allows for overlapping communications and computations, we can improve the performance of the algorithm. The idea is to have each processor update y' while communicating messages relative to y''. At each step, P_q can independently perform $2r^2$ arithmetic operations for updating its $2r$ components of y'. Assume that we have:

$$2r^2\tau_a \geqslant \beta + 2r\tau_c$$

(case where x is duplicated) (*)

or

$$2r^2\tau_a \geqslant \beta + 4r\tau_c$$

(case where x is distributed) (**)

In this case, at each step the communication can

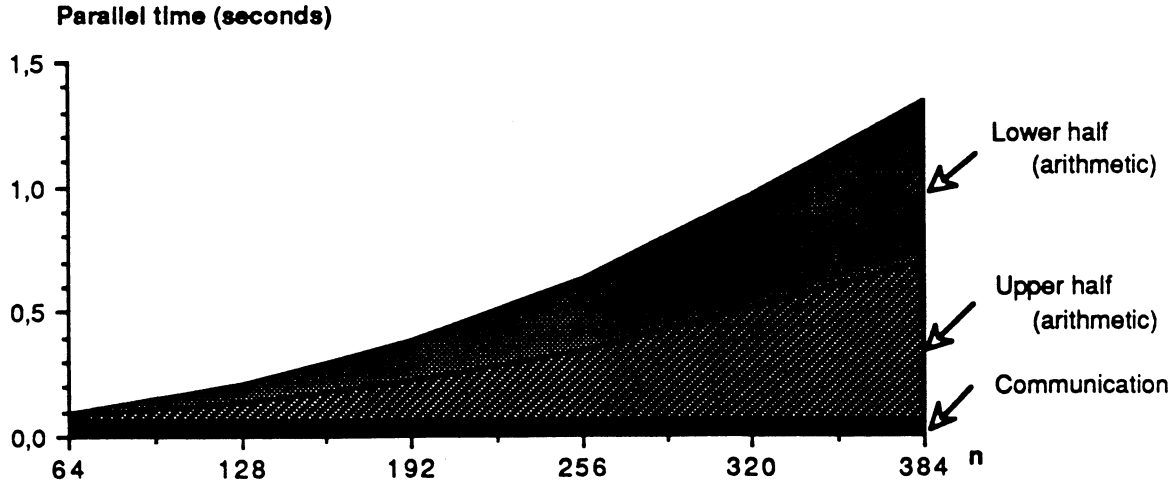


Fig. 8. Relative costs in the parallel execution time as a function of n ($p = 32$).

Volume 35, Number 5 INFORMATION PROCESSING LETTERS 28 August 1990

be performed within the computation time relative to y', so that the total execution time becomes simply T_{seq}/p, and the efficiency is 1.0! Note that the condition (*) or (**) is likely to be fulfilled using a fixed-size machine (fixed p) and when solving large problems (large n).

4.3. Comparison with standard matrix-vector product

The obvious advantage of our scheme over the standard algorithm is its dramatic saving in terms of memory. The price to pay for this saving is very low, as summarized by the following case analysis:

- if the machine architecture allows for overlapping communications and computations, then our scheme is just as efficient as the standard one (efficiency 1.0), and it should always be used.
- if arithmetic and communication cannot overlap:
  - if x is duplicated, then the standard scheme requires no communication, while the communication time for our scheme is $T_c = (p-1)(\beta + (n/p)\tau_c)$;
  - if x is distributed among the processors, then the standard scheme requires $T_c = (p-1)(\beta + (n/p)\tau_c)$, while for our scheme $T_c = (p-1)(\beta + 2(n/p)\tau_c)$.

In all cases, there is no overhead for the arithmetic, which remains perfectly balanced among the processors.

5. Numerical experiments

We report in this section numerical experiments on an FPS T40 hypercube [5], which we configure as a ring of Inmos Transputers T414, using up to 32 processors. The values of β, τ_c, and τ_a were first obtained experimentally as: $\beta = 1.9$, $\tau_c = 0.025$, and $\tau_a = 0.27$ (all in milliseconds).

The version of the algorithm that we deal with here corresponds to the case where x is duplicated in all processors. Very little changes when we distribute x, because T_c is small in comparison to T_a. Also there is no overlap between arithmetic and communication.

Firstly we checked that the parallel execution time obeyed our formulas. In Fig. 7 we superimpose the experimental and theoretical curves (using the previous values of β, τ_c and τ_a) for the parallel execution time. There is an almost perfect match between the curves (so close that each pair of curves appears as one).

In Fig. 8, we let $p = 32$ and show the relative costs of communication and arithmetic time for different values of n. We used small values of n, otherwise the contribution of T_c would have been insignificant. Note that T_c is a linear function of

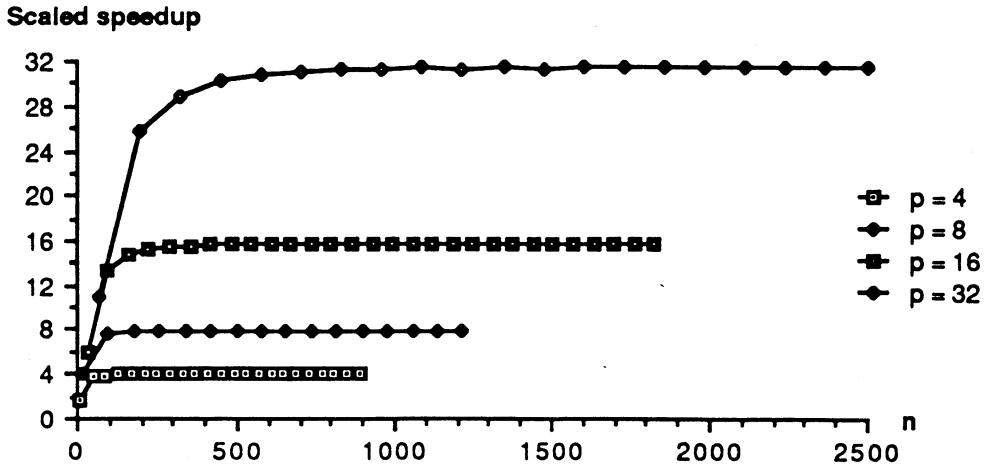


Fig. 9. Scaled speedup as a function of n and p.

n, whose slope $((p-1)/p)\tau_c$ is small compared to its value $(p-1)\beta$ for $n=0$.

We plot the speedups that we obtained in Fig. 9. Note that these speedups are computed according to Gustafson's recent proposal [4,1], in that they are normalized by the amount of arithmetic operations which they require [1] (since it is impossible to solve large problems with a single processor). Using 32 processors, we report acceleration factors close to 32.

6. Conclusions

We have presented an optimal algorithm for solving a symmetric matrix-vector product on a distributed memory machine where processors are arranged in an oriented ring. It achieves an overall performance of:

$$T = \frac{n^2\tau_a}{p} + (p-1)\left(\beta + \frac{n}{p}\tau_c\right),$$

where τ_a is the characteristic multiply-and-add time, τ_c is the communication time between neighbouring processors, β is the communication startup time, p is the number of processors, and n is the dimension of the symmetric matrix. These results have been corroborated on an FPS T40 Transputer network of up to 32 processors.

The single restriction on this algorithm is the requirement that the dimension of the symmetric matrix, n, be divisible evenly by twice the number of processors. Practically however, this is of little consequence since smali adjustments to the algorithm can be made for specific system optimization.

[1] Given a number p of processors, consider the largest problem size that can be solved using the available memory, and compute $\tau_{max}(p)$, the average time needed for one arithmetic operation, as the inverse of the Mflops performances for this largest problem size. This is a renormalization of the problem. The ratio between the values of $\tau_{max}(p)$ for different p gives the Gustafson's speedup.

This research continues in the application of the results presented herein towards folding them into a parallel algorithm for numerically stable recursive least squares parameter estimation. The problem under review is the efficient distribution amongst the processors of the intermediate results in a numerically stable algorithm such as the UDU^T covariance factorization method [11].

References

[1] M. Cosnard, Y. Robert and B. Tourancheau, Evaluating speedups on distributed memory architectures, *Parallel Comput.* **10** (1989) 247–253.

[2] P. Fraigniaud, S. Miguet and Y. Robert, Scattering on a ring of processors, Tech. Rept. LIP-IMAG 90-07 (1990); also: *Parallel Comput.*, to appear.

[3] G.A. Geist and M.T. Heath, Matrix factorization on a hypercube multiprocessor, in: M.T. Heath, ed., *Hypercube Multiprocessors 1986* (SIAM, Philadelphia, PA, 1986) 161–180.

[4] J.L. Gustafson, Reevaluating Amdahl's law, *Comm. ACM* **31** (5) (1988) 532–533.

[5] J.L. Gustafson, S. Hawkinson and K. Scott, The architecture of a homogeneous vector supercomputer, in: *Proc. ICCP 86* (IEEE Computer Soc. Press, Silver Spring, MD, 1986) 649–652.

[6] O.A. MacBryan and E.F. van de Velde, Hypercube programs for computational fluid dynamics, in: M.T. Heath, ed., *Hypercube Multiprocessors 1986* (SIAM, Philadelphia, PA, 1986) 221–243.

[7] O.A. MacBryan and E.F. van de Velde, Hypercube algorithms and implementation, *SIAM J. Sci. Statist. Comput.* **8** (2) (1987) 227–287.

[8] C. Mohtadi, Numerical algorithms in self-tuning control, in: K. Warwick, ed., *Implementation of Self-Tuning Controllers* (Peregrinus, London, 1988) 67–95.

[9] Y. Saad, Gaussian elimination on hypercubes, in: M. Cosnard et al., eds., *Parallel Algorithms and Architectures* (North-Holland, Amsterdam, 1986) 5–18.

[10] Y. Saad and M.H. Schultz, Data communication in parallel architectures, *Parallel Comput.* **11** (1989) 131–150.

[11] C.L. Thornton and G.J. Bierman, Filtering and error analysis via the UDU^T covariance factorization, *IEEE Trans. Automat. Control* **23** (1978) 901–907.

Article [4]

Scattering on a ring of processors.

Parallel Computing 13 (1990) 377–383
North-Holland

Short Communication

Scattering on a ring of processors *

Pierre FRAIGNIAUD, Serge MIGUET and Yves ROBERT

Laboratoire de l'Informatique du Parallélisme LIP-IMAG, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

Received October 1989

Abstract. In this note, we prove that the complexity of scattering in an oriented ring of p processors is $(p-1)(\beta+L\tau)$ where L is the length of the messages, β the communication startup, and τ the elemental propagation time.

Keywords. Local memory multiprocessor, communication, scattering, processor ring.

1. Scattering

In a recent paper, Saad and Schultz [4] study various basic communication kernels in parallel architectures. They point out that interprocessor communication is often one of the main obstacles to increasing performance of parallel algorithms for multiprocessors. They consider the following data exchange operations:

(1) one-to-one: moving data from one processor to another,

(2) broadcast: moving the same data packet from one processor to all others,

(3) total exchange: moving a data packet from each processor to every other processor. This is in effect a broadcast operation (2) from each node,

(4) scattering: a node sends a packet to every other processor. These packets, although different, are ideally all of the same size.

The difference between the broadcast (2) and the scatter (4) is that in the scatter operation a different data is sent to every processor. Note that the dual operation of the scatter is the gather: the node receives a packet of (ideally) equal size from each of the other nodes.

We focus in this note on the scattering operation (4). We derive the complexity of scattering on the simplest model of architecture studied by Saad and Shultz [4], namely the ring architecture. More precisely, we show that the standard intuitive algorithm which consists in pipelining the packets along the ring is optimal.

We outline that very few complexity results are available for the communication kernels (1) to (4). The asymptotic complexity of broadcasting on a ring has been given by Saad and Schultz [4]. Note that we give here the exact complexity for the scattering on a ring. No exact complexity results are known on the hypercube both for the broadcast and the scattering, even

* This work is, in part supported by the Research Project C3 of CNRS, and by the Direction des Recherches et Etudes Techniques (DGA).

though very tuned algorithms and lower bounds have been proposed by Johnsson and Ho [2] and Stout and Wager [5].

2. Statement of the problem

Our model of architecture is an oriented ring of p processors numbered from 0 to p-1 (Fig. 1). Each processor P_i can simultaneously write to its successor P_{i+1} and read from its predecessor P_{i-1} (processor indices are taken modulo p). The time to read/write a packet of L words from to nearest neighbors is of the form $\beta + L\,\tau$ [1–4].

The hypothesis that the ring is oriented is by no means a restriction for broadcast or scattering operations. In fact, if the ring is not oriented, the processor which initiates the broadcast or scatter operation simultaneously sends data to its both neighbors, and two independent communication schemes progress along the two halves of the ring: the scheme amounts to two independent broadcasts/scatters in an oriented ring of half size.

The scattering problem is to distribute a vector $X = (x_0, x_1, \ldots, x_{N-1})$, resident in the local memory of the processor P_0 to all the other processors. In other words, we want that at the end of the algorithm, processor P_i $(0 \leqslant i \leqslant p-1)$ holds the subvector X_i of X composed of components x_k with $Li \leqslant k < L(i+1)$. We assume therefore that p divides N, and that all the subvectors are of same length L. The classical algorithm is the following: at the first step, P_0 sends X_{p-1} to P_1, then at the second step it sends X_{p-2} to P_1 while P_1 transmits X_{p-1} to P_2, and so on until each block reaches its destination processor at the last $(p-1)$th step. The program of processor P_i can be written as follows:

```
Program of processor P_i:
for b := p − 1 downto i + 1 do      {Circulate subvector X_b, for p − 1 ≥ b ≥ i + 1}
   begin
      if i ≠ 0 then receive (X_b);
      send (X_b);
   end;
if i ≠ 0 then receive (X_i).
```

The execution time of the algorithm is $(p-1)(\beta + L\tau)$, since there are $p-1$ communications during each $\beta + L\tau$. The purpose of this paper is to show that this time is optimal.

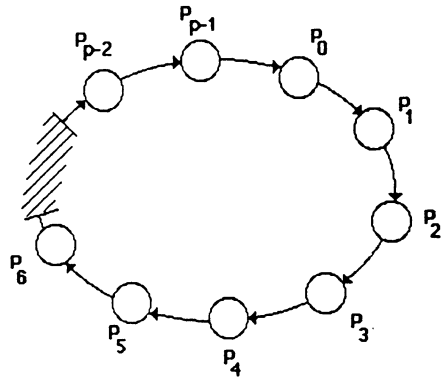


Fig. 1. Oriented ring of p processors.

3. The complexity of scattering

For us, a scattering algorithm is a succession of communications and we only consider the state of the advancement at the beginning and at the end of the communications. For the pending communications, we consider that the transmitted data are still in the sender's memory. We will call a step of an algorithm a transition between two successive states. Less formally, P_0 sends a number of messages m_1, $m_2, \ldots, m_q$ to P_1. P_1 receives them and in turn sends messages to P_2, and so on. Of course P_1 has the possibility of modifying the number, the length and the content of the messages that it has received. However, it cannot send a data x_i before having completed the reception of the whole message which contains x_i.

Proposition 1. *There exists an optimal algorithm.*

Proof. Let us define the following distance d, which somehow evaluates how far the algorithm is from its completion:

$$d = \sum_{i=0}^{i<N} \left| \mathrm{PE}(i) - \left[\frac{i}{L} \right] \right|$$

where $\mathrm{PE}(i)$ is the processor which currently holds the data x_i. Note that $\mathrm{PE}(i)$ is well defined since there is no advantage to duplicate the data. Note also that $[i/L]$ is the destination of element x_i. The algorithm ends when $d = 0$ at state **end**; at the beginning (state **init**), we have

$$d = L\frac{p(p-1)}{2}$$

since each subvector X_i of L components is at distance i from its destination.

In the following, we will only consider algorithms for which the distance d is strictly decreasing during the execution. Indeed suppose that an algorithm uses a communication which does not decrease d. Such a communication moves a component away from its destination processor or transmits useless data. It is easy to transform this algorithm into another algorithm at least as efficient, and for which d is strictly decreasing.

Let us prove that there is a finite number of such algorithms. Consider the finite set V of the functions $f: \{0, \ldots, N-1\} \to \{0, \ldots, p-1\}$. An element of V is a state of an algorithm, it associates to each component x_i of X the processor $\mathrm{PE}(i)$ which holds it. Note that it is possible to associate a value $d(f)$ of d to each element f of V. Let $G(V, E)$ be the directed graph whose vertices are the elements of V. There will exist an edge of E between two vertices f and g if and only if there exists a step of an algorithm which leads to this transition with $d(f) > d(g)$. Any algorithm will be described by a path from **init** to **end**. Conversely, given a path, it is possible to deduce the execution of a unique algorithm. Therefore there is a bijection between the set of the algorithms an the set of the paths from **init** to **end** of graph G. The set of such paths is finite since the distance d is strictly decreasing along each path. In this finite set of algorithms, one of them is optimal. □

Proposition 2. *There exists an optimal algorithm such that each processor P_i sends to P_{i+1} the data $x_{L(i+1)}, \ldots, x_{N-1}$ in reverse order, eventually grouping them in many messages.*

Lemma 3. *There exists an optimal algorithm such that P_0 terminates its emissions by the data destinated to P_1.*

Proof. Let us consider an optimal algorithm (Proposition 1). Let y be the first component of X_1 sent by P_0, and z be the last component of $X \setminus X_1$ sent by P_0 (see Fig. 2). If z is sent before

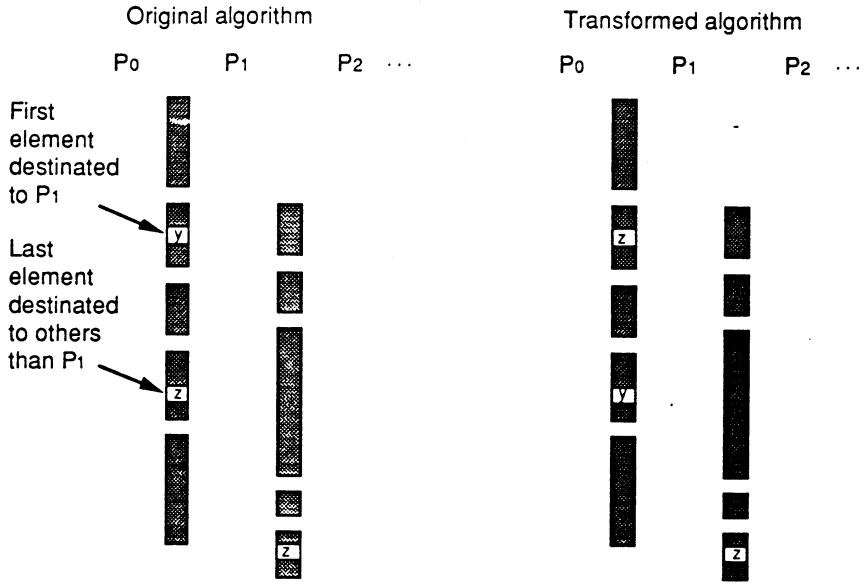


Fig. 2. Transforming an algorithm into another of same execution time, finishing by X_1.

y, this algorithm satisfies the condition of Lemma 3. Otherwise, we modify the original algorithm by exchanging y and z in the emissions of P_0. The new algorithm has the same execution time, since we do not modify the emission times of P_1. The reception of z has been advanced, therefore z can be stored if needed, and then sent in due time. The reception of y has been delayed but this does not affect the remainder of the algorithm since the final destination of y is P_1. After a finite number of such interchanges, the algorithm will satisfy the condition that P_0 terminates its emissions by the data for P_1. □

Lemma 4. *There exists an optimal algorithm such that P_0 terminates its emissions by the data for P_2 followed by the data for P_1, and such that P_1 terminates its emissions by the data for P_2.*

Proof. Let us consider an optimal algorithm satisfying the condition of Lemma 3. First we reorder the emissions of P_1 such that it terminates by the data for P_2. Assume that there exists $y \in X_2$, $z \in X \setminus (X_1 \cup X_2)$ such that P_1 sends y before z. We cannot simply interchange these two emissions, because we are not sure that P_1 would have received z in due time to send it in the place of y. But we can interchange the emissions of y and z from P_0 and let P_1 send as before. After a finite number of such interchanges, the algorithm will satisfy the condition that P_1 terminates its emissions by the data for P_2.

We already know that P_0 terminates its emissions by the data for P_1 (we have not modified the emission of any element of X_1 above). Assume that there exists $y \in X_2$, $z \in X \setminus (X_1 \cup X_2)$ such that P_0 sends y before z (see Fig. 3). We interchange the emission of y and z from P_0. Since we know that P_1 sends z before y, we are sure to meet the deadline for both the emissions of y and z by P_1. After a finite number of such interchanges, the algorithm will satisfy the condition of Lemma 4. □

Proof of Proposition 2. We proceed by induction. The induction step is the following:

(I_k) There exists an optimal algorithm such that P_i, $0 \leqslant i < k$, terminates its emissions by the data for P_k then for P_{k-1} then for P_{k-2} and so on up to P_{i+1}.

(I_1) is proved by Lemma 3. The construction to move from (I_k) to (i_{k+1}) is exactly the same as the one we used in Lemma 4. □

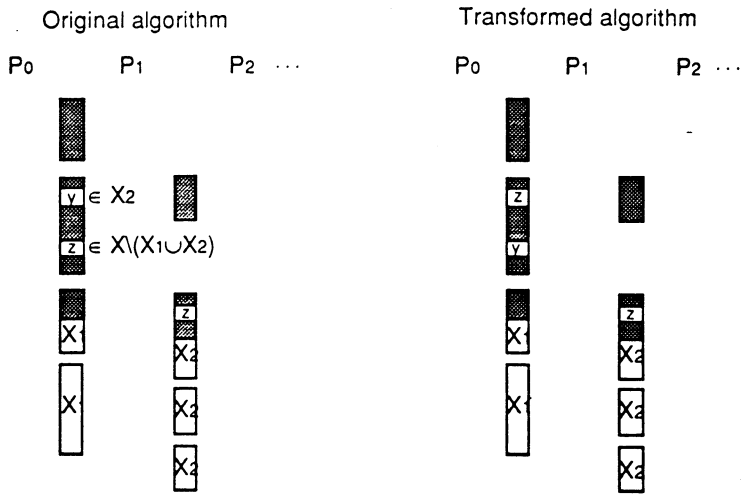


Fig. 3. Finishing with X_2 and X_1.

Proposition 5. *In an optimal algorithm, P_0 cannot perform more than $p-1$ communications.*

Proof. During the execution of any algorithm, P_0 must send $L(p-1)$ data to P_1. If it transfers these data in q communications, it will take a time of $q\beta + L(p-1)\tau$. The classical algorithm runs in $(p-1)\beta + L(p-1)\tau$. It comes that $q \leqslant p-1$ for any optimal algorithm. □

Proposition 6. *In an optimal algorithm, there exists a communication packet between P_0 and P_1 which includes a whole subvector X_k for some k, $1 \leqslant k \leqslant p-1$.*

Proof. The $p-1$ segments $[iL, (i+1)L[$, $1 \leqslant i \leqslant p-1$, have to be covered by q disjoint packets, with $q \leqslant p-1$ (see Fig. 4). Hence, one whole segment is necessarily included in a given packet. □

Theorem 7. *The lower bound for scattering on an oriented ring of p processors is*

$$(p-1)(\beta + L\tau)$$

where L is the length of the messages, β the startup of the communication, and τ the elemental propagation time. This bound is reached by the classical algorithm.

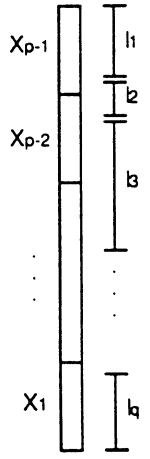


Fig. 4. Communications between P_0 and P_1.

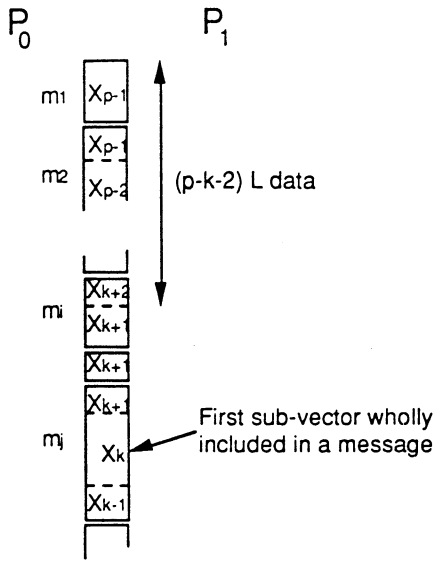


Fig. 5. Induction step of the proof.

Proof. We proceed by induction. The induction step is the following:

(I_p): The lower bound for scattering on an oriented ring of p processors is $(p-1)(\beta + L\tau)$.

(I_2) is clearly verified. Let us assume that (I_n) is true for $n < p$. P_0 successively sends messages $m_1, m_2, \ldots, m_q$ with $q \leqslant p-1$. Let X_k be the first subvector sent by P_0 which is wholly included in a single message m_j (see Proposition 6). First of all, let us prove that $j \geqslant p-k$. If $k = p-1$, then $j = 1$ since we know that the first components of X_{p-1} belong to the first message, and $j = p-k$.

If $k < p-1$, see Fig. 5. X_{k+1} cannot have been sent entirely in m_j since X_k is the first subvector satisfying the condition of Proposition 6. Let m_i be the message holding the first component of X_{k+1}, and let m_i' be the head of m_i corresponding to the components of X_{k+2}. The $(p-k-2)L$ last components of X have been sent within at most i messages m_1, $m_2, \ldots, m_{i-1}$, m_i' (m_i' can be empty). If we had the relation $i \leqslant p-k-2$, this would contradict the definition of X_k, because by the same argument as in Proposition 6, we would find another $X_{k'}$, $k' > k$, wholly contained in a single packet. Thus $i > p-k-2$. Since $j > i$, we obtain that $j \geqslant p-k$.

Let t be the time when P_1 can begin to send the first components of X_k. This corresponds to the time when P_1 completes the reception of some components of X_k. Since X_k is sent by P_0 in one message, t is equal to the time of the complete reception of X_k. Hence $t \geqslant j\beta + (p-k)L\tau$ since at least $(p-k)L$ data has been transferred in j messages. At time t, P_1 has not yet sent any components of X_h with $h \leqslant k$. Thus, P_1 must still realize a scattering involving processors 1 to k. By induction, this will take a time of at least $(k-1)(\beta + L\tau)$. Let T be the total execution time of the algorithm. We have

$$\begin{aligned} T &\geqslant t + (k-1)(\beta + L\tau) \\ &\geqslant j\beta + (p-k)L\tau + (k-1)(\beta + L\tau) \\ &\geqslant (p-k)\beta + (p-k)L\tau + (k-1)(\beta + L\tau) = (p-1)(\beta + L\tau). \end{aligned}$$

Hence, (I_n) is true for $n \leqslant p$. □

Corollary 8. *The complexity of a total exchange operation on an oriented ring of p processors is* $(p-1)(\beta + L\tau)$, *where L is the length of the messages,* β *the startup of the communication, and* τ *the elemental propagation time.*

Proof. In a total-exchange algorithm, processor P_0 gathers the messages from the other processors, and we know that gathering and scattering have the same complexity. Thus the optimal time for a total-exchange is greater than or equal to the optimal time for scattering. The standard total exchange algorithm which consists in shifting $(p-1)$ times packets of size L along the ring executes in time $(p-1)(\beta + L\tau)$, and is therefore optimal. □

References

[1] G.A. Geist and M.T. Heath, Matrix factorization on a hypercube multiprocessor, in: M.T. Heath, ed., *Hypercube Multiprocessors 1986* (SIAM, Philadelphia, 1986) 161–180

[2] S.L. Johnsson and C.T. Ho, Optimum broadcasting and personalized communication in hypercubes, *IEEE Trans. Comput.* **38** (9) (1989) 1249–1268.

[3] O.A. MacBryan and E.F. van de Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Statist. Comput.* **8** (2) (1987) s227–s287.

[4] Y. Saad and M.H. Schultz, Data communication in parallel architectures, *Parallel Comput.* **11** (1989) 131–150.

[5] Q.S. Stout and B. Wager, Intensive hypercube communication, Technical Report CRL-TR-9-87, Computing Research Laboratory, The University of Michigan, 1987.

Article [5]

Z-Buffer on a Transputer-Based machine

Z-Buffer on a transputer-based machine

Jian-Jin LI, Serge MIGUET[1]
Laboratoire de l'Informatique du Parallélisme LIP-IMAG
Ecole Normale Supérieure de Lyon
69364 Lyon Cedex 07, France
e-mail: miguet@ensl.ens-lyon.fr

Abstract : This paper describes the parallel implementation of the Z-Buffer algorithm on a distributed memory machine. The Z-Buffer is one of the most popular techniques used to generate a representation of a scene consisting of objects in a 3-dimensional world. We propose and compare two different parallel implementations on a network of Transputers. In the first approach, the description of the scene is distributed among the processors configured as a tree. The picture is processed in a pipelined fashion, in order to output parts of the image during the computation of the remainder. We show the influence of the degree and the heigth of the tree on the global performance of the algorithm. In a second approach, we have used a ring network to interconnect the processors. In this version, both the picture and the scene description are distrubuted to the processors. We have therefore to redistribute dynamically the tiles among the processors at the beginning of the computation. We show that the two approaches are complementary : for small pictures or large scenes, a tree-based algorithm performs better than a ring-based algorithm, but for large pictures or smaller scenes, it is the other way round. We obtain substantial speedups over the sequential implementation, with up to 32 processors.

1. Introduction

In recent years, many authors have studied and implemented 3D imaging on parallel architectures. The technique giving pictures with the best degree of reality is the ray tracing algorithm. It has the advantage of being highly parallel in nature, and calculation intensive enough to produce good speedup [CD 89] [BB 89]. But it cannot be used to compute pictures in real time. The Z-Buffer algorithm has not as many features as ray tracing but it is much faster in calculation time, and produces nevertheless pictures with a good degree of reality. Hence, its implementation on a parallel architecture has been widely studied in the last few years :

Dyer and Whitman [DW 87] have implemented a vectorized version on a Convex C-1 mini-supercomputer. The vectorization is accomplished at the scanline level and at the shading vector of the horizontal span. Such an approach cannot achieve the same speedup as a MIMD machine since only a small percentage of the code can be vectorized.

Robel [RO 88] has implemented a scanline Z-Buffer on the Intel iPSC hypercube, with a dynamical distribution of polygons among the processors. It has given a

[1] This work has been supported by the Research Projects C3 of CNRS and CMaP of IMAG

good solution with a small number of processors, but as the number of processors is scaled up, the region size is smaller and the overhead per region gets too important.

A another parallel implementation is that by Theoharis [TH 86] with Inmos Transputers. He uses a variation of Parke's spliter mechanism. Each Transputer handles a polygon and performs Viewing Transformation, Clipping, Perspective Transformation and Scan-conversion in a pipeline format. The algorithm makes good use of the Transputer's fast context switching and internode communication speed to allow pipelining. The drawbacks of this implementation are essentially due to the difference of processing time between the different stages of the pipeline : the Perspective Transformation and Scan-conversion are much slower than Viewing Transformation and Clipping, and produce a queue of polygons.

We discuss in this paper two different implementations of the Z-Buffer on a supernode of 32 Transputers [NI 88], with a reconfigurable interconnection network. The first problem to solve when parallelizing a problem is to choose an allocation strategy for the data. A trivial solution is to distribute the picture to the processors, and to duplicate the description of the scene in each of the local memories. The parallelisation is trivial, since each processor executes the sequential algorithm on a smaller image. This approach works well for scenes with a small number of objects, but cannot be used for large ones. The two implementations explained in this paper, show how to avoid this duplication of the scene description.
In our first solution, the processors are configured as a tree, and the description of the scene is statically distributed among the processors. Each processor has the possibility of participating in the computation associated with each pixel of the picture. In this implementation, we compute parts of the image in a pipeline fashion, so as to output them during the computation of the remainder of the image.
In our second implementation, each processor is responsible of the computation of a single part of the picture. The description of the scene is dynamically redistributed among the processors configured as a ring.

This paper is organized as follows : first of all, we briefly explain the sequential Z-Buffer algorithm. Then, we present our two parallel implementations on a T-Node of 32 transputers with a reconfigurable interconnection network, and we explain the choice of the interconnection topologies. Numerical experiments, allow us to compare the two approaches, which are indeed complementary.

2. The Z-Buffer algorithm

Given the description of elementary objects (shape, position, orientation, specular and diffuse properties, ...) and the position and orientation of the observer, the problem is to generate a 3-dimensional representation of the scene, as the observer can see it. For the sake of simplicity, we suppose that our objects are polyhedra composed of triangular tiles. These tiles are defined by 3 vertices, and for each vertex, its position, and the direction of the normal in the coordinate system of the scene. (see **Figure 1** below).

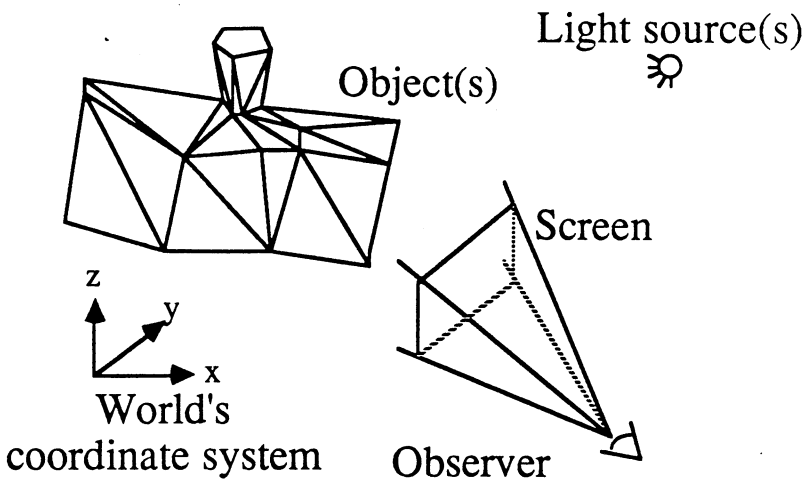


Figure 1 : 3-dimensionnal imaging

Figure 2 shows the traditional pipeline of the image production : after the vertices have been projected in the coordinate system of the screen, the tiles have to be clipped to the size of the picture. Then, we have to determine all the pixels of the image which are inside the projected triangles. This is done by a so called scan conversion algorithm, which scans a triangle row by row. These pixels are then colored according to a given shading model. We are using the Gouraud model [Go 71], where the color of a vertex is determined by the cosine between the normal to the surface and the direction of the light, and the colors of the pixels inside of the triangle result of a bilinear interpolation based on the color of the three vertices. The main problem is to eliminate the parts of the objects which are hidden by others. This is what the Z-Buffer is used for : each time a pixel has been drawn in a position of the picture, we note in the same position in another picture (the Z-Buffer), the Z coordinate of that point, that is, the distance between the point and the observer. Afterwards, a pixel is drawn in a given position, if and only if, its Z coordinate is smaller than the one previously stored in that position of the Z-Buffer.

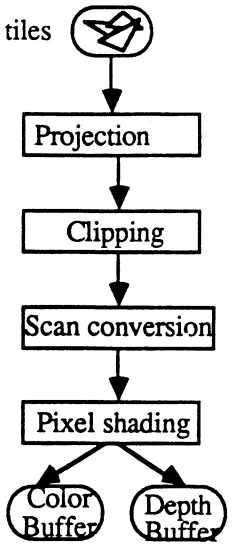


Figure 2 : The sequential algorithm

3. Parallel implementation on a tree

3.1. FIRST APPROACH

In a first version we suppose that each processor can hold a whole image in its memory. The parallelization of the algorithm is based on a divide-and-conquer approach: once the tiles have been distributed to the processors, each processor computes its contribution to the picture, by executing the sequential algorithm with the tiles that it has received. Then, in a second step, the processors merge their data to compute the final image. This merge step is easy to implement: given two images and the corresponding two Z-Buffers, we can compute a new image and a new Z-Buffer, by keeping for each position, the pixel with the smaller Z coordinate. The merge algorithm follows : (suppose images are of size `v_size` x `h_size`, and pixels have two fields : their color `C` and their depth `Z`.)

```
merge(im1,im2);        {computes the merge of        }
                       {images 1 and 2, result in 1  }
begin
  for i := 1 to v_size do
    for j := 1 to h_size do
      if im2[i,j].Z < im1[i,j].Z then
        im1[i,j].Z := im2[i,j].Z;
        im1[i,j].C := im2[i,j].C;
      endif;
    endfor;
  endfor;
end;
```

Algorithm 1: Merge of two images

Processors combine their images by merging them two by two, until only one image

remains. The problem is now to choose an interconnection topology to perform this step efficiently. Our target machine, the T-Node is a transputer-based machine, with a reconfigurable interconnection network. Each interconnection graph of degree smaller than or equal to 4 can be realized. The topology has no importance for the first phase of the algorithm, since processors do not communicate. For the second phase, a tree seems well adapted : the reduction starts from the leaves, and progresses up to the root. In a first approach, we have chosen to minimize the diameter of the tree, so that the total amount of communication is reduced, and therefore, we have to maximize the degree. The resulting tree is a ternary tree; processors are numbered from 0 to p-1, the sons of processor P_q are processors P_{3q+1},P_{3q+2} and P_{3q+3}, and its father is $P_{\lfloor (q-1)/3 \rfloor}$ (wherever the indices make sense). An example of such a tree with 11 processors is given in **Figure 3** Note that processor 0 has only 3 sons, since it needs a communication channel to the "outside world".

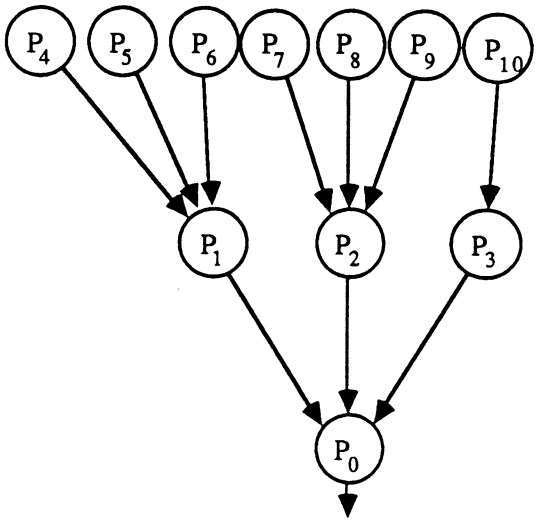


Figure 3 : A ternary tree with 11 processors.

We give the program of a node in **algorithm 2** below: Note that this algorithm can be executed on any kind of tree provided that each node knows the number of its sons NB_SONS. Section 3.5 will report numerical experiments conducted with various types of trees.

```
algorithm parallel_zbuf(im);
begin
  sequential_zbuf(im);
  for son := 1 to NB_SON do in parallel
    receive_from_son(im[son]);
  endfor;
  for son := 1 to NB_SON do
    merge(im,im[son]);
  endfor;
  send_to_father(im) ;
end;
```

Algorithm 2 : Naive distributed Z-Buffer

This algorithm has many weaknesses : first of all, it requires a lot of memory: each

processor has to be able to store a whole image and the associated Z-Buffer. Note for instance that a 500x500 picture with 24 bits per pixel for the color, plus say 16 bits per pixel for the Z-Buffer, will occupy more than 1 MByte of storage, which is the size of the memory of our nodes. Another drawback of the algorithm is that during the merge step, only the nodes of a single level in the tree are activated simultaneously, which is a loss of potential parallelism.

Now we explain how to decrease the memory requirement, and to improve the parallelism of the second phase. The idea is to split the image into bands (or strips), to pipeline the processing of these bands, and to overlap the communication of a band with the computation of the next one.

3.2. PIPELINING BANDS OF THE PICTURE

We split the picture into NB horizontal bands of pixels. All the processors successively apply **algorithm 2** to the different bands of the picture. If the transmission of a band is non-blocking, then after having computed band 1, the program of a processor can be viewed as NB-1 further steps: at step B ($2 \leq B \leq NB$), it performs in parallel the computation of its band B, the reception of the band B of its sons, and the transmission to its father of the band B-1 that it has computed at previous step. When the reception and computation of band B is completed, the merge process among the partial data for band B can begin. The transmission of the resulting band will be performed at next step. **Figure 4** below illustrates one step of the parallel algorithm.

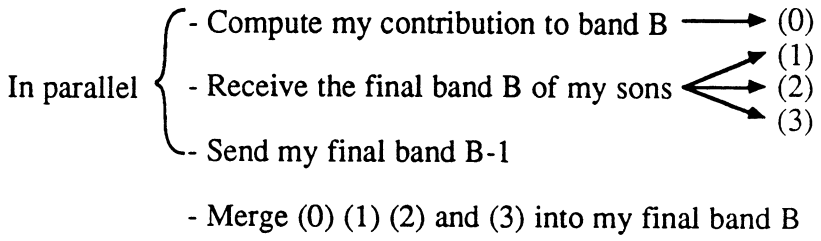


Figure 4: One step of the parallel algorithm

This scheme of computation and communication induces a pipeline in the tree. Even if the algorithm is asynchronous, it can be viewed as a succession of parallel steps : if the height of the tree is H, then after H steps executed on a leaf, all the processors will be active. A node of the tree computes the band B while its sons compute the band B+1, and its father computes the band B-1. Except during the initialization and the termination of the algorithm, all the processors are active at each step.

3.3. AVOIDING REDUNDANCY

In fact, the algorithm executed by the processors for each band of the picture is not exactly as described in **algorithm 2**. Otherwise, they would have to consider all the tiles at each step, to project them, and to clip them to the frame of the band. In order to improve the global performance, a preprocessing step is applied before the computation takes place. A processor projects its tiles in the coordinate system of the screen, to compute in which band(s) they will appear. When processing a band, it will only consider the tiles which may appear in that band. In that way, if a tile belongs to two or more bands, it won't have to be projected twice. This

preprocessing step is illustrated in **Figure 5**.

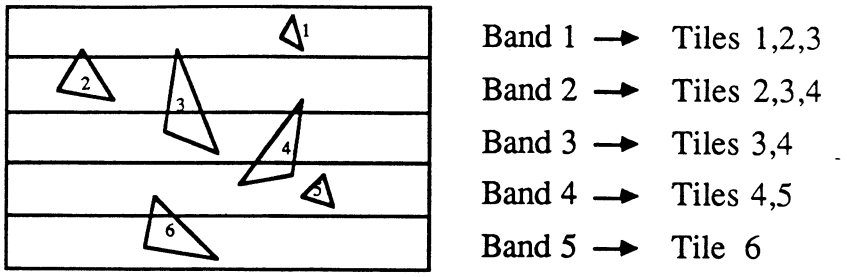


Figure 5 : Band allocation

3.4. USING BOUNDING BOXES

All the merge operations as well as the communication steps are tasks which do not belong to the sequential algorithm. In order to improve the speedup archived by the parallel algorithm, we must minimize the time spent in these tasks. A possible improvement is to communicate only the "bounding box" of the tiles that have been drawn in a band, instead of the whole band: while drawing into a band, we update the four coordinates defining the smallest upright rectangle containing all the non-zero pixels. The communication time decreases, since we only communicate a part of the band. The complexity of the merge operation is also reduced: The comparison of the depth of the pixels can be limited to the intersection of the bounding boxes. Outside of this intersection, the merge is reduced to a simple copy operation. **Figure 6** illustrates this situation.

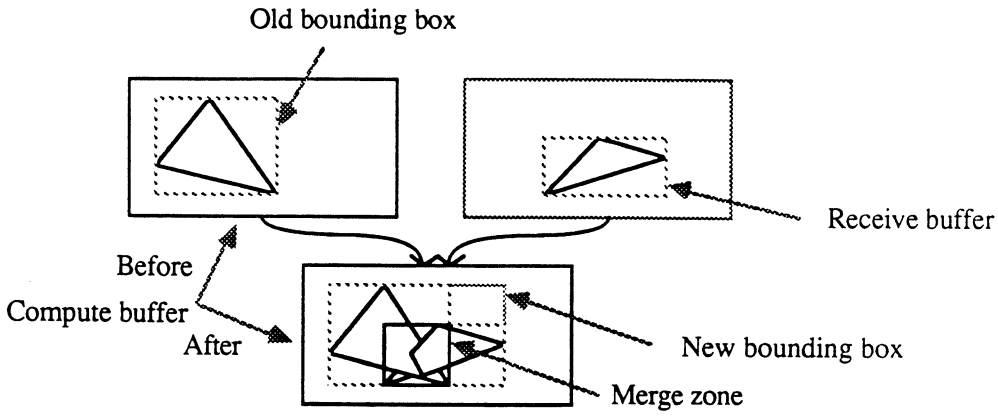


Figure 6 : Merge operation with bounding boxes.

3.5. EXPERIMENTS

In this section, we report experiments performed on a Supernode machine with up to 32 processors [Ni 88], with the famous University of Utah teapot consisting of 3752 triangles (see **Figure 7** below). It is a rather small scene, since it holds entirely in the memory of one processor.

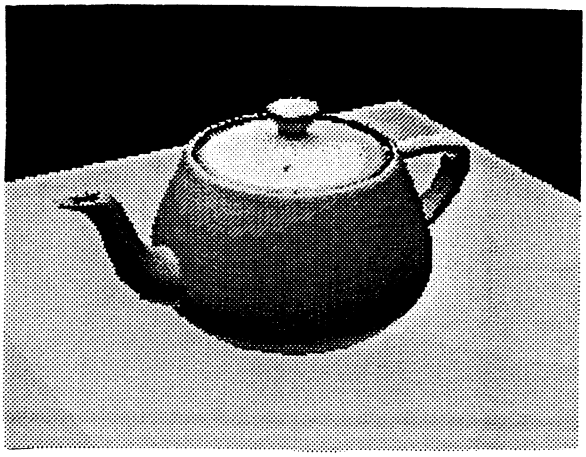


Figure 7 : The University of Utah teapot.

3.5.1 The Ternary tree

In our first experiment, we let the number of bands vary, with a fixed image size and number of processors (in **Figure 8**, we have taken `h_size=v_size=480`, and p=32). As we could foresee there is an optimal number of bands : when the bands are too large, the startup time of the pipeline is high. When they get too small and too numerous, the control cost increases, as well as the number of communications.

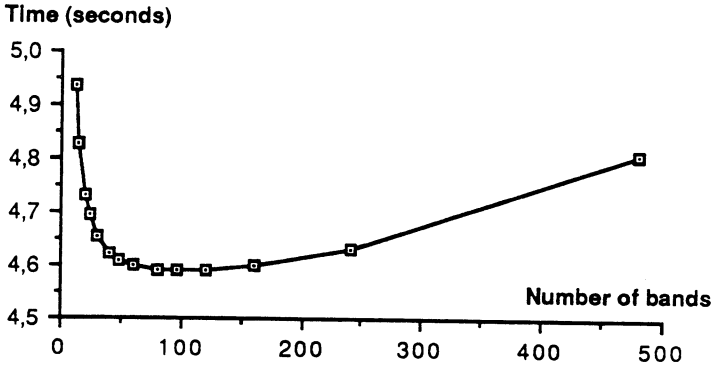


Figure 8 : Parallel time as a function of the number of bands

In our next experiment, we plot again the execution time as a function of the number of bands, but with various numbers of processors. It can be checked in **Figure 9a**, that for a small image, the time is approximately divided by two when the number of processors is doubled. This is no longer true when the image gets larger as in **Figure 9b**. This is due to the time spent in communications and merge operations which grows with the size of the picture faster than the scan conversion time. Furthermore, with a fixed scene, the projection time of the vertices remains constant. The speedup would be better for the same size of picture if we had a larger scene : there would be more computations, with the same communication

overhead.

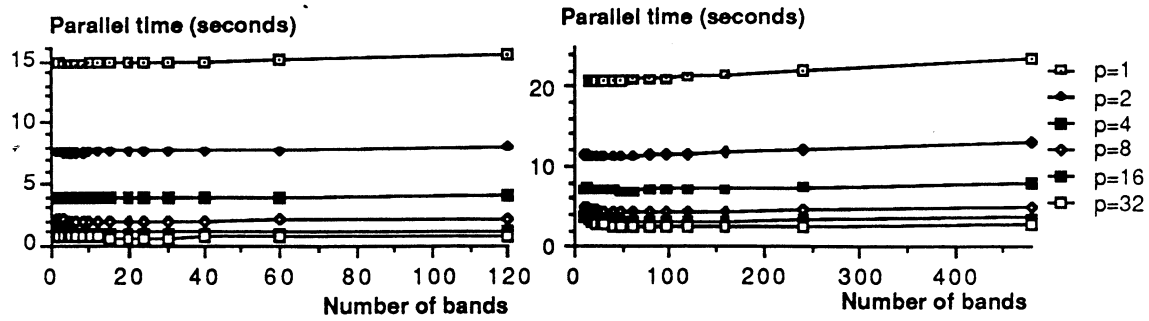


Figure 9a: Image size = 120 **Figure 9b**: Image size = 480

In our last experiment with the ternary tree, we plot the speedup achieved by the parallel algorithm as a function of the number of processors. You see that the algorithm is quite good for a small image size, whereas its performance decreases for larger images. But note that we work here with a fixed scene size. With a larger scene, consisting of many thousands of tiles, the speedup would be more interesting for large pictures.

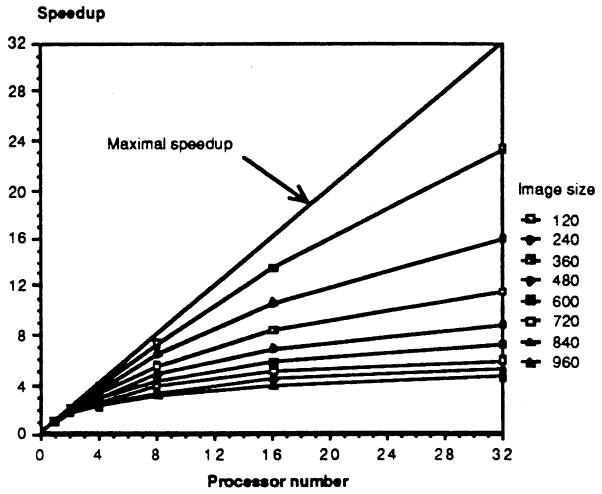
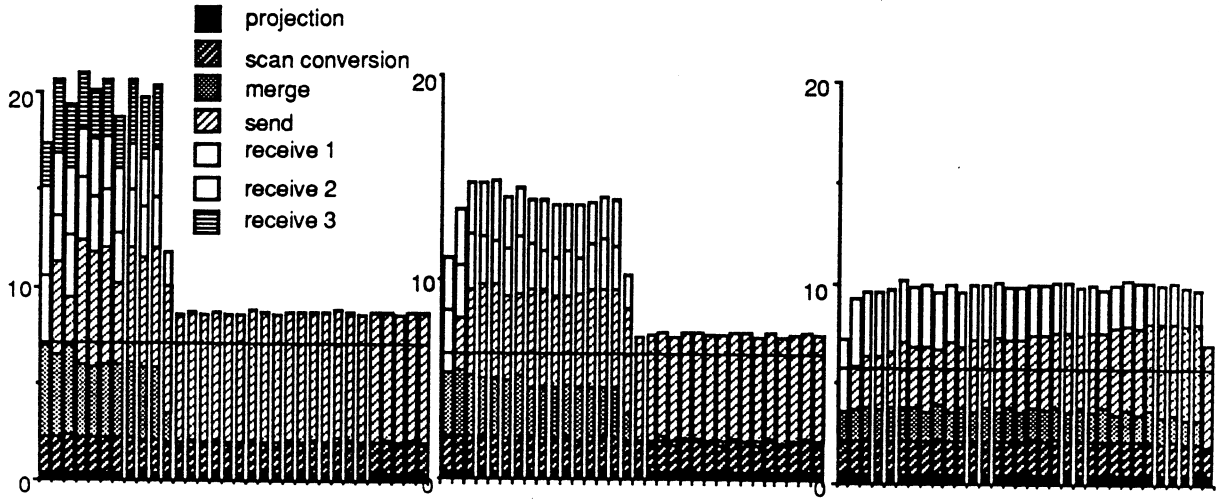


Figure 10 : Speedup of the ternary tree

3.5.2. Revisiting the topology

The ternary tree topology has been chosen in order to minimize the communication time, since the tree has a minimal diameter. But doing so, we have introduced a load imbalance for the merge operation which is only performed by the internal nodes of the tree. For such a small scene as our teapot, if we want to produce large pictures, we have thus to better balance the time spent in the merge operation. **Figure 11** below, shows the time spent in the different processes by the 32 processors when computing a large image of size 960 x 960. A ternary tree is used in **Figure 11-a**, a binary tree in **Figure 11-b**, and a unary tree (or linear network), in **Figure 11-c**. The ternary tree has only 11 internal (non-leaves) nodes whereas the binary has 15 and the linear network has 31. It can be seen in **Figure 11-a**, that the root processor (the leftmost in the histogram) spends two thirds of its

computation time in the merge process. Note also that the global execution time represented by the horizontal line is not (as one could gues, equal to the sum of the times spent in the different processes. This is due to the fact that communications are overlapped by computations. For this size of picture, the linear network achieves a better execution time than the two other trees. The difference would be even more important for a larger picture, which would require more communications, or for a smaller scene, where the relative importance of the projection and scan conversion time, would be smaller.



Ternary tree
Figure 11-a

Binary tree
Figure 11-b

Unary tree
Figure 11-c

In the last experiment, we plot a comparison between the three kinds of trees. We see in **Figure 12** that the ternary tree is better suited to small images, and that the unary tree is the best topology for large images.

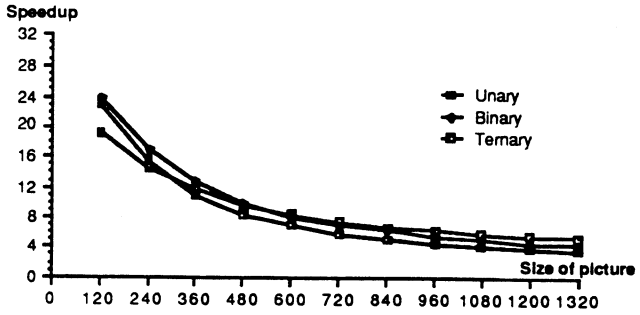


Figure 12 : Speedup for different trees (32 processors)

4. Parallel implementation on a ring

We have seen in the previous section, that the tree topology was not well suited for processing large pictures because of the prohibitive cost of the merge step. In order to avoid this step, we have chosen now to partition the image among the

processors. Now, a processor will be responsible of a part of the picture. The objects will have to be resnuffled between the processors, so that each of them knows the tiles that do project into their own part of image. After the initial distribution of the scene, each processor projects the tiles it has received, and computes their destination processor(s). In a first version, we allocate blocks of consecutive rows of the picture to the processors. The computation of the destination processors of a tile is similar to the band allocation of section 3.3. For instance, if a tile projects in the band 3 and 4 of the picture, it will have to be sent to the processors 3 and 4.
This operation is known as a "multi-scattering" problem in distributed memory systems : each processor has to send a different message to each other. Many efficient algorithm are known in the littereature to perform multi-scattering on hypercubes or other topologies [SS 89]. On a network of transputers, with a limited degree, a ring is well adapted.

4.1 MULTISCATTERING ON A RING OF PROCESSORS.

Let us first assume that the ring is oriented. Processor P_i can send messages to its successor P_{i+1}, and receive messages from its predecessor P_{i-1} (troughout the paper, processor and message indices are taken modulo NB_PROC). Let us suppose that processor P_i has to send the message $m_{i,j}$ to processor P_j. At the first step, processor P_i sends message $m_{i,i-1}$ to its successor P_{i+1}, and receives in parallel form its predecessor P_{i-1} the message $m_{i-1,i-2}$. At the next step, Processor P_i concatenates its own message $m_{i,i-2}$ to $m_{i-1,i-2}$ into a single message M_{i-2}. This mesage is then transmited in parallel with the reception of M_{i-3}. After NB_PROC-1 such communication steps, P_i receives in a single communicaton packet M_i, all the messages $m_{.,i}$ from the other processors. The algorithm is given below :

```
Algorithm multiscatter_unidir {Program of processor Pi};
  Mi-1:= ∅ ;                    {initiates buffer}
  for step:=1 to NB_PROC-1 do
     j:=(i-step) mod NB_PROC;{ Pj = destination processor}
     append mi,j to Mj
     do in parallel
        send( Mj );
        receive(Mj-1) ;
     enddo ;
  endfor ;
```

Algorithm 3 : multiscattering on a unidirectional ring

The transmission of a message of length L between two processors takes the time $\beta + L.\tau_c$ for current distributed memory machines [SS 89], where β is the initialisation time of the communication, and τ_c is the elemental communication time for one byte. If the messages $m_{i,j}$ are of equal length L, then the message M_i, sent at step t by processor i (with j = (i-t) mod NB_PROC) is of length L*t. The total execution time can therefore be expressed as :

$$T_1 = \sum_{t=1}^{NB\_PROC-1} (\beta + L.t.\tau_c) = (NB\_PROC-1)\,(\beta + L.\tau_c.NB\_PROC\ /\ 2)$$

In the case where the ring is bidirectional, messages can be circulated in the two

directions. Two messages M_i and N_i will start from the oposite processor of P_i (or its two opposite processors if NB_PROC is odd) and will "collect" messages $m_{*,i}$ destinated to P_i. M_i and N_i circulate in opposite directions. Only NB_PROC/2 communication steps are required. The algorithm is given below :

```
Algorithm multiscatter_bidir {Program of processor Pi};
  Mi+NB_PROC/2:= ∅ ;
  Ni-NB_PROC/2:= ∅ ;        {initiates buffers}
  for step:=1 to NB_PROC/2 do
     j1:=(i+NB_PROC/2-step+1) mod NB_PROC;
     j2:=(i-NB_PROC/2+step-1) mod NB_PROC;
                    { Pj1 and Pj2 = destination processors}
     append mi,j1 to Mj1 ;
     append mi,j2 to Nj2 ;
     do in parallel
        send Mj1 to Pi+1;
        receive Mj1-1 from Pi-1 ;
        send Nj2 to Pi-1;
        receive Nj2+1 from Pi+1 ;
     enddo ;
  endfor ;
```

Algorithm 4 : multiscattering on a bidirectional ring

Note that if NB_PROC is an even number, then the same message is sent in the two directions at the first step. This can be avoided by doing one step of unidirectional circulation followed by NB_PROC/2-1 steps of bidirectional circulation.

For transputer-based machines, it is slower to use a link in two directions than in one direction. Still, it is more efficient to do one bidirectional communication than two unidirectional ones. The bidirectional communication of messages of length L takes the time $\beta' + L.\tau'_c$ with $\beta < \beta' < 2.\beta$ and $\tau_c < \tau'_c < 2.\tau_c$. The values of these parameters (from [DT 90]) are given in the table 1 :

	Unidirectional	Bidirectional
Startup (μs)	$\beta = 25.8$	$\beta' = 36.7$
1/Bandwidth (μs / byte)	$\tau_c = 1.1$	$\tau'_c = 1.6$

Table 1 : communication times

At step t ($1 \le t \le$ NB_PROC/2), the messages are of length L.t. The total execution time is thus equal to :

$$T_2 = \sum_{t=1}^{NB\_PROC/2} (\beta' + L.t.\tau'_c) = NB\_PROC/2 \left(\beta' + \frac{L.\tau'_c.(NB\_PROC/2 + 1)}{2}\right)$$

When the size of the messages (or the number of processors) increases, the ratio T_1 / T_2 tends to $4.\tau_c/\tau'_c$, that is 2.75. The bidirectional multiscattering is 2.75 times faster than the unidimensionnal one.

4.2 BACK TO THE ALGORITHM

In our case, of course, messages are not of the same size. Nevertheless, we can compute the "ideal" time of the multiscatter operation by supposing that the tiles are randomly distributed to the processors, and that they project uniformly in the generated picture. This time gives us a good approximation of the complexity of the operation. If the whole scene is composed of F triangular tiles, then each processors receives approximately F / NB_PROC ones, and has to send F / NB_PROC^2 ones to every other processor. Our previous formulas can thus be applied with L = F. sizeof(facette) / NB_PROC^2.
In order to improve the initial algorithm and to have a more homogeneous repartition of the tiles in the picture, we can compute the extrema of the projected objects, and only distribute the parts of the picture included between these values. While a processor projects and clips its tiles, it maintains the smallest and the largest y coordinate of its projected objects. Then, all the processors compute a global minimum and a global maximum, by combining their local values. This can be done in NB_PROC-1 communications and minimum/maximum. We give the algorithm allowing to compute the global minimum ; the maximum is evaluated similarily :

```
algorithm minimum()
  y_min := local_y_min;
  for i:=1 to NB_PROC-1 do
    do in parallel
      receive(new_y_min) ;
      send(y_min) ;
    enddo ;
    y_min := min(new_y_min,y_min) ;
  endfor ;
  return(y_min) ;
end ;
```

Algorithm 5 : computing the extrema of the objects

Note that this minimum can be computed with twice as less communications if we use bidirectional links.

4.3. NUMERICAL EXPERIMENTS

In our first experiment (see **Figure 13**), we compare the unidirectional and the bidirectional circulation. We mesure the communication time as a function of the number of processors. As expected, we see that the bidirectional circulation is more efficient than the unidirectional one. Note that the ratio between the maximal values for unidirectional circulating $T_1 = 1.65$ and bidirectional circulating $T_2 = 0.6$ is equal to 2.75 which is in good accordance with the theoretical asymptotical ratio computed in section 4.1.

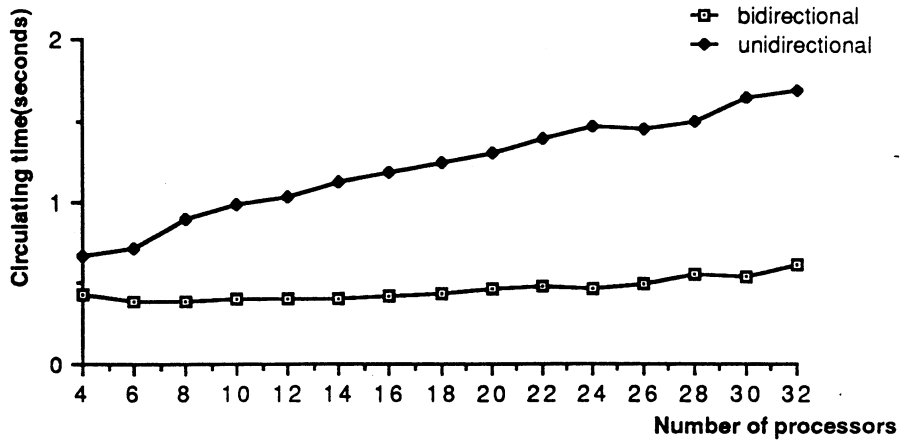


Figure 13

In the second experiment (see **Figure 14**) we show the composition of the processing time of each processor. Here, the size of picture is 1080 * 1080, and we use 32 processors. We see that the projection time is identical on all the processors. This is because they all have initially the same number of tiles. The communication time is also approximately uniform among the processors. But the scan convertion time is much less balanced : the tiles are not truly uniformly distributed after the shuffle. Even if they were, their still would be a load imbalance because the time to draw a triangle strongly depends of its shape. It is a really difficult problem to efficiently balance the scan conversion.

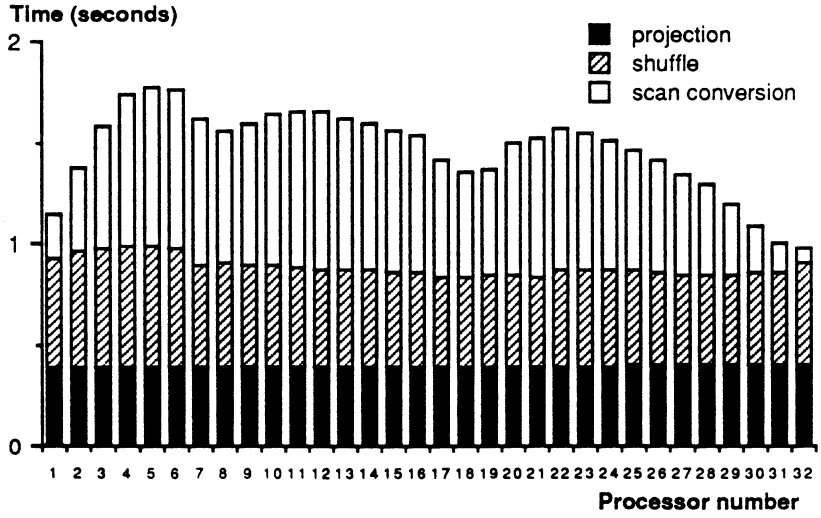


Figure 14

An interesting remark to do is that the communication time only depends on the scene and of the number of processors used, not of the size of the generated picture. If we augment the size of the picture, with a given scene and a fixed number of processors, only the scan conversion time will increase. This means that the speedup is an increasing function of the number of processors. See the experiments of next section to see an illustration of that fact.

5. Comparison of the two methods

In this chapter, we compare the two implementations we have proposed. As we explained it before, the tree is well adapted for the processing of large scenes, since the objects do not move away form their initial location. On the other hand, it doesn't perform well for generating large picture, because of the time spent in the merge of local images. The ring has a dual behavior : it cannot handle large scenes efficiently, because of the intensive communication scheme that it requires for the dynamic reallocation of the tiles. But it is very well suited to the synthesis of large pictures, since the communication overhead doesn't depend on the size of the picture.

Figure 15 below is a comparison of the two techiques, with a fixed scene and 32 processors. We plot the speedup as a function of the size of the picture. As expected, the speedup of the tree is a decreasing function of the size, whereas the speedup of the ring is an icreasing one. With our (small) scene, the intersection point occurs for pictures of size 260 x 260. It would be bigger for a larger scene, or for more processors. We conclude that above that intersection point, we have to use the tree algorithm, whereas the ring should be used below.

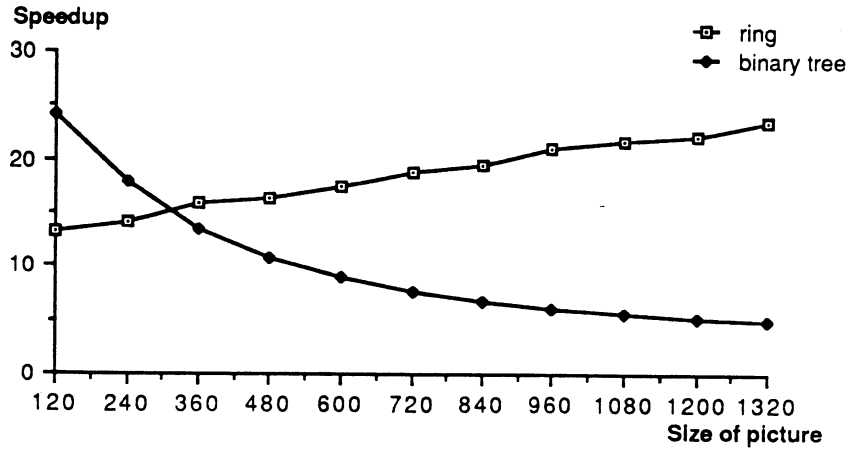


Figure 15

6. Conclusion

We have proposed two different parallel implementations of the Z-Buffer algorithm, for a distributed memory machine. The first method uses a tree-based algorithm, where the processor compute their contribution to the whole image, in a pipelined fashion. The second approach, uses a ring network to redistribute dynamically the tiles to the procesors, so that each processor is responsible of a single band of the picture.
For both techniques, we have tried to minimize the memory occupation, by avoiding the duplication of the scene in each of the local memories. This fact allows us to process large scenes which couldn't be handled by a single processor. The scalability of our implementations is handled fairly well. Our experiments show that

we can reach a good speedup for any picture size, and any number of processors, by using one of the two methods, which are in fact complementary : The tree-based approach is well adapted to the processing of large scenes and the synthesis of small pictures, whereas the ring approach performs better for handling smaller scenes and producing larger images.

7. References

[BB 89] Didier BADOUEL, François BODIN and Thierry PRIOL, "Lancer de rayon : Approches parallèles",Bigre N°61-62, Avril 1989, Languages et algorithmes du graphique, pp.213-225.

[BO 90] Christophe BONELLO, "Ténor: un configureur symbolique pour l'architecture T.node, Manuel de référence", Technical Report LIP-IMAG (01-90).

[CD 89] F.C.CROW, G.DEMOS, J.HARDY, J.Mc.LAUGHLIN, K.SIMS, "3D images synthesis on the Connection Machine", in Scientific Applications of the Connection Machine, H.D.Simon ed. World Scientific (1989), pp.260-281.

[DU 85] Tom DUFF, "Compositing 3D Rendered Images", proceddings of SIGGRAPH '85, (july 1985), pp.41-44.

[DT 90] Fréderic DESPREZ, Bernard TOURANCHEAU, Modélisation des preformances de communication sur le T-Node avec le Logical System Transputer Toolset. La Lettre du Transputer 7 (september 1990), pp65-72.

[DW 87] Scott DYER and Scott WHITMAN, "Parallel Peocessing Techniques for Hidden Surface Removal", Computer Graphics & Applications July, 1987, pp.34-44.

[GO 71] Henri GOURAUD, "Continuous Shading of Curved Surfaces", IEEE Transactions on computers, Vol.c-20, N°6, (June 1971).

[MV 87] O.A. MAC BRYAN, E.F. VAN DE VELDE, "Hypercube algorithms and implementations", SIAM J. Sci. Stat. Comput. 8, 2 (1987), s227-s287.

[NI 88] DENIS A. NICOLE, "Esprit Project 1085, Reconfigurable Transputer Processor Architecture", in CONPAR 88, C. R. Jesshope et al. eds., Cambridge University Press (1989), pp.81-89.

[PF 80] Frederic I. PARKE, "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems", Computer Graphics, vol.14, N°3, Proceeding of Siggraph, July, 1980, pp.48-56.

[RK 82] A.ROSENFELD, A.C.KAK, "Digital picture processing ", Academic press, New York (1982), vol.2.

[RO 88] Doug R. ROBEL, "A load Balanced Parallel Scanline Z-buffer Algorithm for the iPSC Hypercube", Proceeding of Pixim'88, 1988.

[SS 74] Ivan E. SUTHERLAND, Robert F. SPROULL and Robert A. SCHUMACKER, "A Characterization of Ten Hidden-Surface Algorithms", Computing Surveys, Vol.6, N°1, March 1974, pp.293-337.

[SS 89] Y. SAAD, M.H. SCHULTZ, Data communication in parallel architectures, Parallel Computing 11 (1989), 131-150

[TH 86] A.THEOHARIS, "Exploiting Parallelism in the Graphics Pipline", Oxford University Computing Laboratory. Technical Monograph PRG-54. June 1986.

[WI 78] Lance WILLIAMS, "Casting curved shadow on curved surfaces", Computer Graphic, vol.12,

N°3 (1978), pp.270-274.

[WH 80] T.WHITTED, "An improved illumination model for shaded display", Computer Graphics and Image processing, vol.23, N°6 (june 1980), pp.343-349.

Article [6]

Reduction Operations on a distributed memory machine with a reconfigurable interconection network

Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon
Institut des Sciences de la Matière de l'Université Claude Bernard de Lyon
Institut IMAG
Unité de Recherche Associée au CNRS n° 1398

Reduction operations on a distributed memory machine with a reconfigurable interconnection network

Serge MIGUET and Yves ROBERT[1]
Laboratoire LIP - IMAG
Ecole Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France
tel: + 33 72 72 83 89, fax: + 33 72 72 80 80
e-mail: yrobert@ensl.ens-lyon.fr, yrobert@frensl61.bitnet

Abstract :

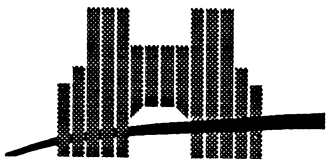
We are interested in performing reduction operations with distributed memory machines whose interconnection networks are reconfigurable. More precisely, we focus on machines whose interconnection graph can be configured as any graph of maximum degree d. We discuss the best way of interconnecting the p processors as a function of p, d, and some problem- and machine dependent-parameters that characterize the ratio communication/arithmetic for the reduction operation. Experiments on Transputer-based networks are in good accordance with the theoretical results.

Key-words :

reduction operation, communication kernels, communication/arithmetic ratio, reconfigurable interconnection networks, distributed memory machines, Transputer arrays, pipeline strategies

---

[1] This work has been supported by the Research Project C3 of CNRS, by the Research Project CMaP of IMAG, and by the ESPRIT Basic Research Action 3280 "NANA" of the European Economic Community



Ecole Normale Supérieure de Lyon,
46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 83 89 - Télécopieur : + 33 72 72 80 80
Adresses électroniques :
yrobert@frensl61.bitnet; yrobert@ensl.ens-lyon.fr (uucp).

1. Introduction

In a recent paper, Saad and Schultz [SS] study various basic communication kernels in parallel architectures. They point out that interprocessor communication is often one of the main obstacles to increasing performance of parallel algorithms for multiprocessors. They consider the following data exchange operations:
(1) One-to-one: moving data from one processor to another
(2) Broadcast: moving the same data packet from one processor to all the others
(3) Total exchange: moving a data packet from each processor to every other processor. This is in effect a broadcast operation (2) from each node
(4) Scattering: a node sends a packet to every other processor. These packets, although different, are ideally all of the same size.
The difference between the broadcast (2) and the scatter (4) is that in the scatter operation a different data is sent to every processor. Note that the dual operation of the scatter is the gather: the node receives a packet of (ideally) equal size from each of the other nodes.

Reduction operations are based upon the gathering primitive for communication, but a certain amount of computation is interleaved during the message-passing process. Assume that we have p processors P_0, ..., P_{p-1} interconnected by some network. Two typical reduction operations are the following:

❑ each processor P_i holds some data item x_i, and one of them, say P_0, needs to compute the maximum $x = \max_{0 \le i \le p-1} \{x_i\}$. Rather than gathering the p items and then compute sequentially their maximum, a more efficient idea is to perform some sub-computations in parallel, and to propagate the result of these sub-computations rather than the original items. For instance, if the interconnection network contains a binary tree rooted at P_0 as a sub-topology, the leaf processors send their x- item to their father. Each internal node computes the maximum of its x-item with the data received from its two sons, and propagates the result towards its father.

❑ each processor P_i holds some subblock of two large vectors X and Y that are stored in a distributed fashion among all the processors. Computing the dot product of X and Y is an operation that arises very frequently in parallel linear algebra, e.g. when solving triangular systems with a row-wise allocation of the matrix [HR, LC]. After computation of the local dot products within each processor, the assembling the final result is clearly a reduction operation.

Generally speaking, the implementation of divide-and-conquer algorithms on a distributed memory machine usually requires a reduction operation after the local processing (divide) to collect and process the partial results (conquer). Reduction operations are so important global operations that many vendors provide the programmer with such primitives (e.g. for the previous two examples the gdhigh and gdsum on the Intel iPSC2, or the scan_with_max and scan_with_+ on the Connection Machine).

In this paper, we are interested in performing reduction operations with distributed memory machines whose interconnection networks are reconfigurable. More precisely, we focus on machines whose interconnection

graph can be configured as any graph of maximum degree d. In other words, we assume to be possible (at boot time, or dynamically) to establish a direct link between any pair of nodes, provided that no node has more than d neighbors. Transputer-based networks are typical examples of reconfigurable networks with d=4: either they can be reconfigured at boot time, or they can be dynamically reconfigured after a synchronization point (e.g. the Supernode machine manufactured by Telmat and Parsys [Nic]).

The difficulty with reconfigurable networks (as opposed to the hypercube) is that we have the freedom to choose the topology before discussing efficient algorithm design, and of course the topology has a tremendous impact on the performance. Unfortunately, we will not succeed in deriving the complexity of performing the reduction operation on a reconfigurable network, which is an open problem. But note that the complexity of the reduction problem on a hypercube is not known. Even more, no exact complexity result is known on the hypercube for the broadcast and the gathering problems, which are simpler problems because no arithmetic is involved[1]. This short discussion clearly states the difficulty of the reduction problem in a distributed-memory environment.

In this paper, we focus on a strategy which is very natural for performing the reduction operation, and which we term the 'compute-then-send' strategy: we assume a tree-based interconnection network, and we suppose that each node sends the result of the reduction on its own subtree to its father. We analytically determine the best tree-based interconnection topology of maximal degree d as a function of the number of processors and some machine-dependent parameters such as the elemental computation and communication time. We also analyse pipeline strategies when the data items are vectors. We delay the precise (technical) statement of our results up to section 2, because we need to introduce several hypotheses and definitions beforehand.

[1] However, very tuned algorithms and lower bounds haved been proposed by Johnsson and Ho [JH] and Stout and Wager [SW].

2. Statement of the problem

2.1. Generalized reduction operation

We assume a distributed memory architecture with p processors numbered from 0 to p-1. Each processor P_i holds a data item X_i in its internal memory. The problem is that P_0 has to compute

$$X = X_0 \lozenge X_1 \lozenge X_2 \lozenge \ldots \lozenge X_{p-1}$$

where $\lozenge$ is an associative commutative operator.

We assume a reconfigurable interconnection network of maximum degree d. We also assume that there exists a physical link from P_0 to the external host, hence P_0 has exactly d-1 links available for inter-processor connection. See figure 1 for an example with d=4.

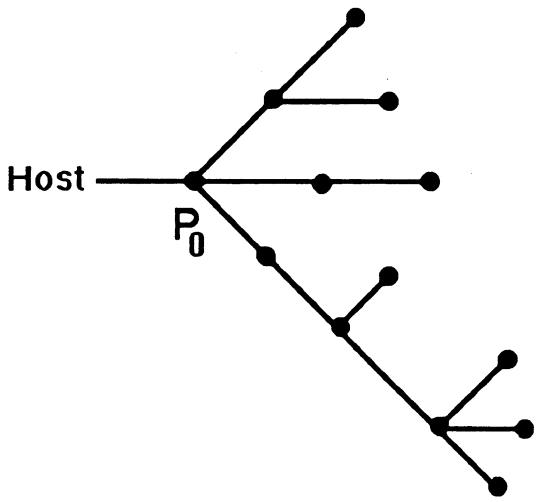


Figure 1 : Example of interconnection network with d=4

For the computation and communication protocols, we assume powerful state-of-the-art processors capable of simultaneously
- sending and receiving different messages on their different links
- performing some internal computation

In other words, computation and communication can be overlapped, and communication can occur in parallel on all the links. Such hypotheses are fulfilled by most current processors.

We let A be the time for computing an operation $X \lozenge Y$, and C be the time for sending/receiving an item X between two neighbor processors. If the data items X_i are vectors of length L, we assume that the $\lozenge$–operator is applied componentwise to the vectors, so that $A = L\,\tau_a$, where τ_a is the elemental time for a scalar $\lozenge$ operation. For the communication time, we have $C = \beta + L * \tau_c$, where β is a start-up time and τ_c is the elemental communication time for a scalar item [GH, HJ, MV, SS].

The start-up time β can depend upon the number of links that are activated in parallel, hence upon the maximum degree d of the network. If q links are activated in parallel, we can model this situation by $\beta = \beta_0 + (q-1)\,\beta_{link}$. For instance this is the case for Transputer-based machines, for which communication parameters are reported in Table 1. To ease the presentation, we simply write β in the following rather than $\beta(q)$.

1 / bandwith	τ_c = 1.1 µs / byte
startup	β_0 = 8.5 µs β_{link} = 9.1 µs

Table 1 : Experimental communication parameters for the Supernode machine

Note that if we let A = 0, we model a gather operation, whose complexity is not known. Again, this shows the intrinsic difficulty of the problem.

2.2. Examples

We give some examples in this section to help the reader approach the problem. For the sake of simplicity, we assume here that the start-up time for communications is independent of the number of activated links.

First consider the case where p=4 processors are available. Two possible interconnection networks are given in figure 2.

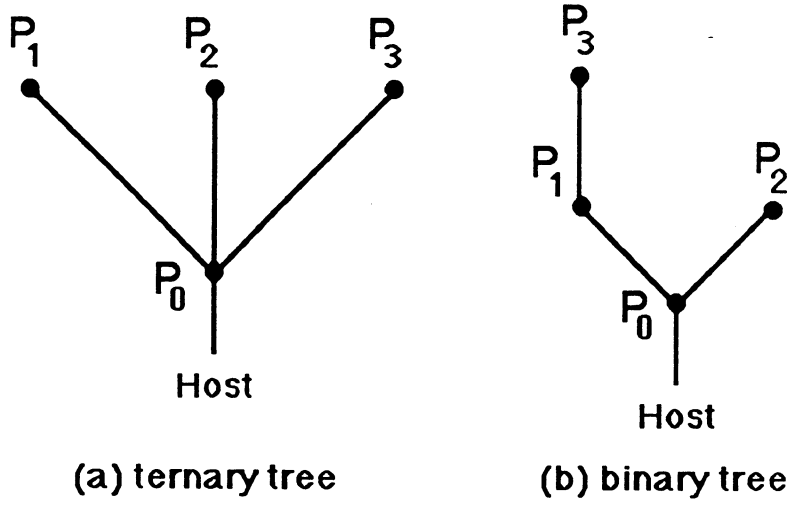


Figure 2 : Two networks with p = 4

For the ternary tree (d=4), see figure 2a. We can let P_1, P_2 and P_3 send their data item at time 0. P_0 receives them at time C and then sequentially computes 3 $\Diamond$-operations in any order, say $X = X_0 \Diamond (X_1 \Diamond (X_2 \Diamond X_3))$, which leads to a total execution time $T_1 = C + 3A$.

For the binary tree (d=3), see figure 2b. We can let P_2 and P_3 send their data item at time 0, to P_0 and P_1 respectively. P_0 and P_1 receive them at time C; P_0 computes $Y_0 = X_0 \lozenge X_2$ while P_1 computes $Y_1 = X_1 \lozenge X_3$. At time C+A, P_1 sends Y_1 to P_0, which receives it at time 2C+A. Finally P_0 computes $X = Y_0 \lozenge Y_1$, which leads to a total execution time $T_2 = 2C + 2A$. Clearly, none of the two networks is always superior to the other: the comparison depends upon the ratio C/A.

If the tree is not oriented (assume that links are bidirectional), we can obtain the execution time $T_2 = 2C + 2A$ with the ternary tree. At step 0, P_1 and P_2 send their data item to P_0, while P_0 sends X_0 to P_3. At time C+A, P_3 sends back to P_0 the value $Y_3 = X_0 \lozenge X_3$. Then P_0 computes $X = Y_0 \lozenge Y_3$ at time 2C+A, where $Y_0 = X_1 \lozenge X_2$, which leads to a total execution time equal to T_2.

Second, given a network of degree d, it is not easy to compute the time necessary for a reduction operation. For instance consider a ternary tree (d=4) with p=13 processors, as illustrated in figure 3.

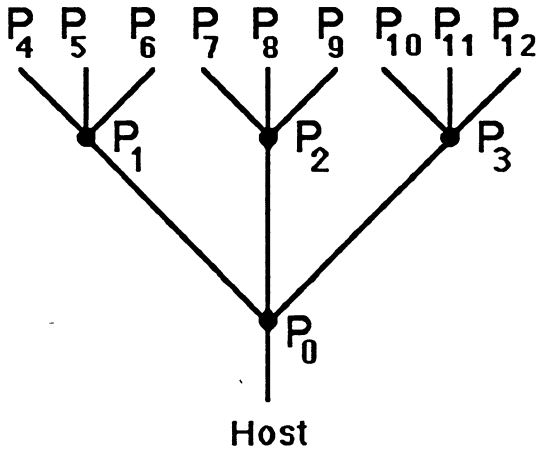


Figure 3 : A ternary tree with p = 13 processors

Generalizing the previous strategy for arbitrary ternary tree is easy: all the processors lying in the same level of the tree simultaneously receive the items from their sons, perform the reduction operation, and send the result to their father. This leads to a total time

$$T_p = n\,(C+3A),$$

where n is the number of levels in the tree. Since there is 1 processor at level 0, 3 processors at level 1, 9 processors at level 2, ..., the relationship between p and n for a complete ternary tree of n levels is

$$p = 1 + 3 + 3^2 + \ldots + 3^n = \frac{3^{n+1} - 1}{2}$$

However this strategy is not optimal. With 13 processors, we obtain $T = 2(C+3A)$. But we can let all processors but P_0 send their data item to their father. At time C, P_1, P_2 and P_3 respectively compute $Y_1 = X_4 \lozenge X_5 \lozenge X_6$, $Y_2 = X_7 \lozenge X_8 \lozenge X_9$ and $Y_3 = X_{10} \lozenge X_{11} \lozenge X_{12}$, while P_0 computes $Y_0 = X_1 \lozenge X_2 \lozenge X_3$. At time C+2A, P_1, P_2 and P_3 send Y_1, Y_2 and Y_3 to P_0, which starts computing $Z_0 = X_0 \lozenge Y_0$. At time 2C+2A, P_0 receives Y_1, Y_2 and Y_3. P_0 finishes to compute Z_0

at time C+3A. Hence P_0 starts computing $X = Z_0 \lozenge Y_1 \lozenge Y_2 \lozenge Y_3$ at time max(2C+2A, C+3A). The total execution time is then T = max(2C+2A, C+3A) + 3A = C + 5A + max(C,A), which is always better than the previous strategy.

Third, we have given examples with trees, but there is no reason for eliminating networks with cycles. For instance if the communication cost C is very low, data items should be reshuffled at each step for using the maximum parallelism available. Even though not very realistic, this situation could still be of theoretical interest.

In the following we restrict ourselves to oriented trees where each processor only communicates once and for all to its father, therefore sending the partial reduction corresponding to its whole subtree. The main reason for considering only such 'compute-then-send' strategies is that they are naturally suited to pipelining the resolution of many problem instances, because the flow of data from the leaves towards the root (and then the host) is unidirectional.

In section 3 we deal with scalar items and we show how to determine the best interconnection network, given p, the number of processors, d, the maxmum degree of the network, and C and A, the elemental communication and arithmetic times. In section 4 we move to the case where the data items are vectors, and we compare pipelined strategies on tree-based networks of different degrees. In section 5 we report numerical experiments using a 32-node Supernode machine, and we show that the theoretical results are nicely corroborated by the experiments.

3. The 'compute_then_send' strategy

In this section we deal with the analysis of the 'compute_then_send' strategy in the case of scalar items, and we show how to determine the best interconnection network, as a function of
• p, the number of processors,
• d, the maxmum degree of the network
• and C and A, the elemental communiction and arithmetic times.

Dealing with the compute_then_send strategy, we can assume without loss of generality that the interconnection network is an (oriented) tree of maximum degree d with p processors.

Then the program of the processors will look like this:

```
{ program of processor Pq }
Yq := Xq ;
for each of my son do in parallel
begin
      receive Xson
      begin critical section
            compute Yq := Yq ◊ Xson
      end critical section
end
send Yq to my father
```

The 'in parallel' statement should be understood as explained previously: P_q can receive the messages of its sons in parallel, computation and communication can be overlapped, with the only constraint that the reception of a given item should be completed before the processor can access it for computation. Finally, the father of the root processor P_0 is defined to be the host.

Intuitively, given p, the larger the degree of the tree, the smaller the number of levels in the tree, hence the less costly the communications. On the other hand, the larger the degree of the tree, the smaller the number of processors that perform arithmetic operations in parallel[1]. There is a trade-off to be found that depends upon the ration C/A. In this section, we do show how to recursively determine the best tree, given C and A. The key to the construction is based upon the following lemma (see figure 4):

[1] Leaf processors do not perform any arithmetic.

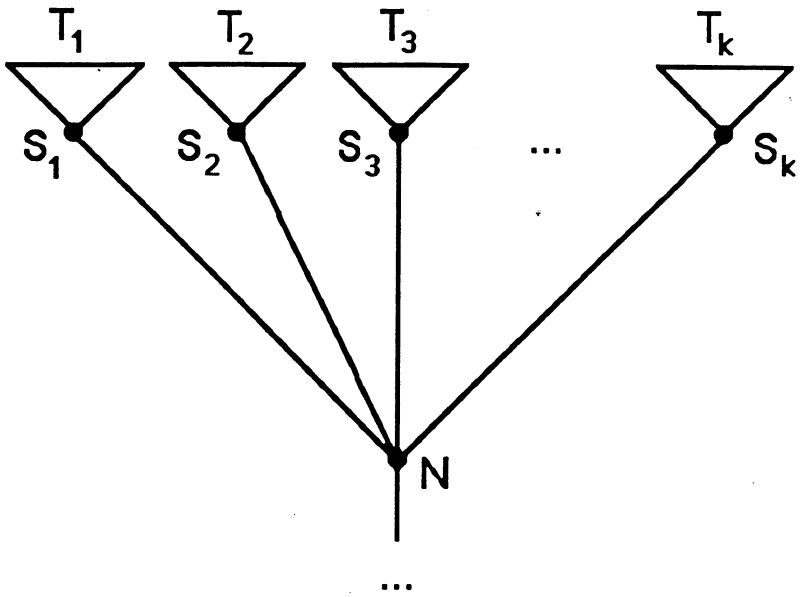


Figure 4: Compute_then_send strategy

Lemma 1 : Let N be a node with k sons S_1 , S_2, ..., S_k. Assume that S_i, $1 \le i \le k$, has terminated the reduction for its whole subtree T_i at time t_i, i.e. S_i has completed the computation $Y_i = \Diamond_{q \in T_i} \{X_q\}$. Assume without loss of generality that $t_1 \ge t_2 \ge ... \ge t_k$. Let T_N be the time at which node N has terminated its reduction , i.e. has computed $Y_N = X_N \Diamond Y_1 \Diamond Y_2 \Diamond ... \Diamond Y_k$. Then

$$T_N \ge \max(t_1+C+A, t_2+C+2A, t_3+C+3A, ..., t_k+C+kA)$$

Proof : Node N has completed the reception of the first message, from S_k, at time $t_k + C$. From that time on, it still has k $\Diamond$-operations to compute, hence it will not terminate its computation before time $t_k + C + kA$. More generally, node N has received the contribution of its son S_i, $1 \le i \le k$, at time t_i+C. From that time on, it remains (at least) i $\Diamond$-operations to compute, hence N will not terminate its computation before time $t_i + C + iA$. ■

3.1. Greedy strategy

The greedy strategy for executing a reduction operation on a tree rooted at P_0 is defined as follows:

- leaf processors send their data item to their father at time t=0
- internal processors update their data item as soon as possible after having received the contribution from their sons, in the order of the receptions. They send their result to their father as soon as the computation is completed.

Lemma 2 : The greedy strategy meets the inequality in Lemma 1: let N be a node with k sons S_1 , S_2, ..., S_k whose computations are terminated at time t_1, t_2, ..., t_k, with $t_1 \ge t_2 \ge ... \ge t_k$. Then using the greedy strategy, node N terminates its computation at time $T_N = \max(t_1+C+A, t_2+C+2A, t_3+C+3A, ..., t_k+C+kA)$

Proof : First take the example of a node N with two sons S_1 and S_2, and assume that $t_1 \geq t_2$. At time t_2+C, N starts computing $Y_N = X_N \Diamond Y_2$. Two cases can occur (see figure 5):

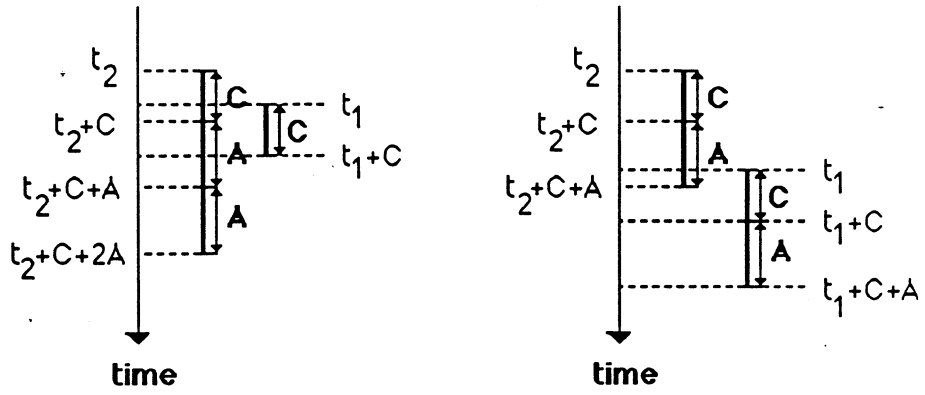


Figure 5 : Two cases with two sons

• If Y_1 has been received by the end of the computation of Y_N, i.e. if $t_1+C \leq t_2+C+A$, then N starts computing $Y_N := Y_N \Diamond X_2$ right after the first computation, at time t_2+C+A. Node N finishes its computation at time $T_N = t_2+C+2A = \max(t_1+C+A, t_2+C+2A)$.
• If Y_1 has not been received at time t_2+C+A, i.e. if $t_1+C > t_2+C+A$, then N remains idle in the interval $[t_2+C+A, t_1+C]$, waiting for receiving Y_2. Node N resumes computing at time t_1+C, hence it finishes its computation at time $T_N = t_1+C+A = \max(t_1+C+A, t_2+C+2A)$.

In the general case, node N receives its first message at time t_k+C and starts computing $Y_N = X_N \Diamond Y_k$. It computes then

• $Y_N := X_N \Diamond Y_{k-1}$ at time $\max(t_k+C+A, t_{k-1}+C)$
• $Y_N := X_N \Diamond Y_{k-2}$ at time $\max(t_k+C+2A, t_{k-1}+C+A, t_{k-2}+C)$
• ...

and finishes its computation at time $\max(t_1+C+A, t_2+C+2A, ..., t_k+C+kA)$. ■

In other words, the greedy strategy is optimal for the 'compute_then_send' strategy. Hence the best interconnection tree for p processors can be recursively constructed as the concatenation of the best interconnection networks for some smaller values of p: we use a dynamic programming algorithm. Given C, A and d, we let t(p) be the optimal time for performing a reduction operation with p processors, and we let H_p be a tree of maximal degree d for which the greedy strategy achieves the optimal time t(p) (such a tree is not necessarily unique).

To build a tree of degree d with p processors, we interconnect d-1 trees (some of them possibly empty) of cardinality $p_1, p_2, ..., p_{d-1}$ by letting P_0 be the father of all these sub-trees. Hence $p = 1 + p_1 + p_2 + ... + p_{d-1}$. Since t(p) is clearly a non-decreasing function of p, we can assume that $p_1 \geq p_2 \geq ... \geq p_{d-1}$. According to Lemma 2, the time to perform the reduction operation on the tree is equal to

$$\max_{1 \leq i \leq d-1} (t(p_i)+C+iA)$$

where $t(p_i) = 0$ if the i-th son does not exist. Searching among all possible splittings, we obtain the following dynamic programming recurrence:

Proposition 1 :

$$t(0) = 0$$

$$t(p) = \min \left\{ p_1 + p_2 + \ldots + p_{d-1} = p-1 \atop p_1 \geq p_2 \geq \ldots \geq p_{d-1} \geq 0 \right. \left(\max_{1 \leq i \leq d-1} (t(p_i)+C+iA) \right)$$

Let $(q_1, \ldots, q_{d-1})$ be a tuple for which the minimum is obtained. Then H_d can be obtained by interconnecting the trees $H_{q_1}, H_{q_2}, \ldots, H_{q_{d-1}}$ with P_0 as a new root. It is now easy to write a code[1] that given C and A computes t(p) and outputs H_p for all values $p = 1, 2, \ldots, P$.

3.2. Examples with d=4

With d=4 we write $p \leftarrow (p_1, p_2, p_3)$ if H_p is obtained by concatening the three subtrees H_{p_1}, H_{p_2}, and H_{q_3}, where $p = 1 + p_1 + p_2 + p_3$. We obtain the following results for p=32:

- if C = 1 and A = 10:

32	←	(13,11,7)
13	←	(6,4,2)
11	←	(5,3,2)
7	←	(3,2,1)
6	←	(2,2,1)
5	←	(2,1,1)
4	←	(2,1,0)
3	←	(1,1,0)
2	←	(1,0,0)

- if C = 10 and A = 1:

32	←	(11,10,10)
11	←	(4,3,3)
10	←	(3,3,3)
4	←	(1,1,1)
3	←	(1,1,0)

We obtain the trees depicted in figures 6 and 7 respectively.

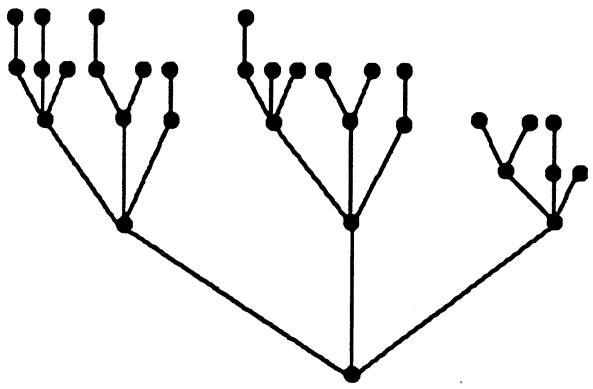


Figure 6 : Optimal tree with p=32, C=1, A=10 (d=4)

[1] The time complexity of such a code is at most $O(P^{d-1})$.

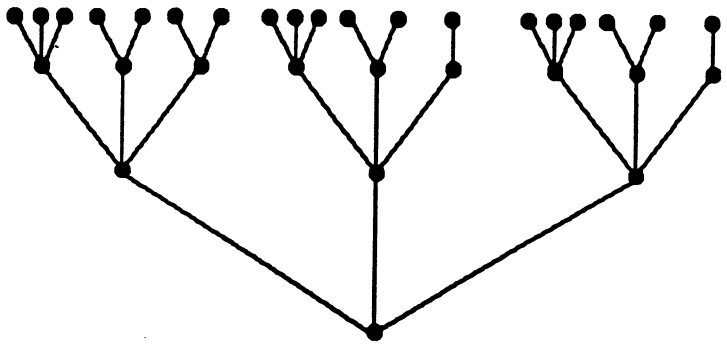


Figure 7 : Optimal tree with p=32, C=10, A=1 (d=4)

We can compute t(p) using the recurrence equation, as follows:

❑ if C = 1 and A = 10:

t(2)	=	C+A
t(3)	=	C+2A
t(4)	=	2C+2A
t(5)	=	C+3A
t(6)	=	2C+3A
t(7)	=	2C+3A
t(11)	=	2C+4A
t(13)	=	3C+4A
t(32)	=	3C+6A

❑ if C = 10 and A = 1:

t(3)	=	C+2A
t(4)	=	C+3A
t(10)	=	2C+4A
t(11)	=	2C+5A
t(32)	=	3C+7A

In the first case, we check that t(p) is not strictly increasing with p, as t(6) = t(7). Also, we point out that we obtain very different solutions for the two cases. When communication is cheap in front of the arithmetic (case 1, $\rho = C/A = 0.1$), the optimal tree is unbalanced, so that more arithmetic can be performed in parallel. On the other hand, when the ratio $\rho = C/A$ is high (case 2, $\rho = C/A = 10$), the tree is well balanced, so that its height is reduced.

We discuss in the following the performance of two special families of trees.

3.3. Two special families of trees

For the sake of clarity, let the ratio $\rho = \frac{C}{A}$ be irrational, so that when expressing t(p) as a function of C and A, namely $t(p) = \alpha A + \gamma C$, no ambiguity can occur. We have the following relation between α and γ, which directly comes from Lemmas 1 and 2:

Lemma 3: If $t(p) = \alpha A + \gamma C$, then $\alpha \leq \gamma \leq (d-1)\alpha$

3.3.1. Communication trees (their name intends that they perform well when the ratio communication/arithmetic is high) are defined for p belonging to the sequence $(u_n)_{n\geq 0}$ which we construct as follows:

- $u_0 = 1$
- $u_{n+1} = (d-1)\, u_n + 1$ for $n \geq 0$

In other words, we concatenate d-1 communication trees of height n to get the communication tree of height n+1. See figure 8 for examples with d=4.

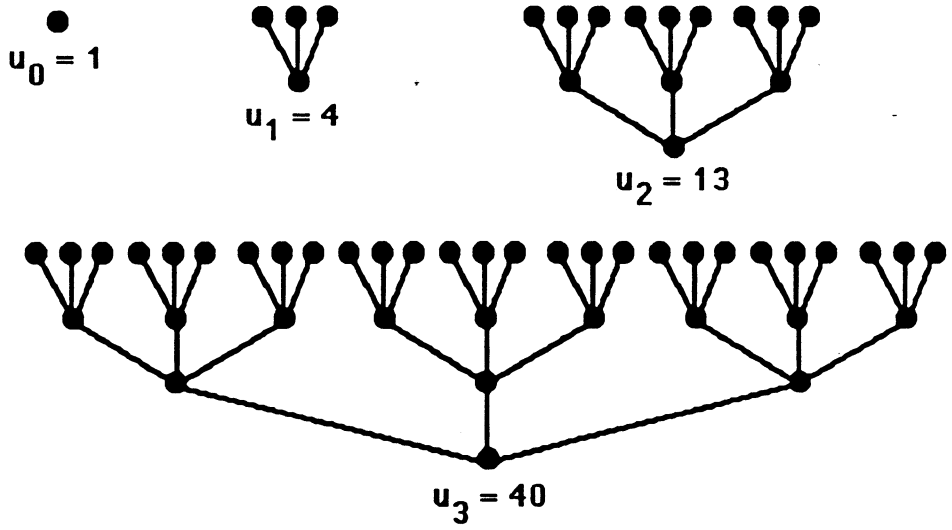


Figure 8 : First communication trees with d=4

Lemma 4 : Communication trees enable to perform the reduction operation in time $t_{comm}(u_n) = n\,(C + (d-1)A)$.[1]

The proof is by induction on n, using Lemma 2. With the notation of Lemma 3, we always have $\gamma = (d-1)\alpha$ for communication trees. We have:

- if $d \geq 3$, $u_n = \frac{(d-1)^{n+1} - 1}{d-2}$
- if $d = 2$, $u_n = n + 1$

3.3.2. Computation trees (their name intends that they perform well when the ratio arithmetic/communication is high) are defined for p belonging to the sequence $(v_n)_{n\geq 0}$ which we construct as follows:

- $v_{-1} = v_{-2} = \ldots = v_{1-d} = 0$
- $v_{n+d-1} = v_{n+d-2} + v_{n+d-3} + \ldots + v_n + 1$ for $n \geq 1-d$

In other words, the new tree is made up with the concatenation of the d-1 previous ones rooted at P_0. See figure 9 for examples with d=4.

[1] We write $t_{comm}(u_n)$ rather than $t(u_n)$ because $t_{comm}(u_n)$ is not the optimal time that can be achieved for a reduction operation with u_n processors.

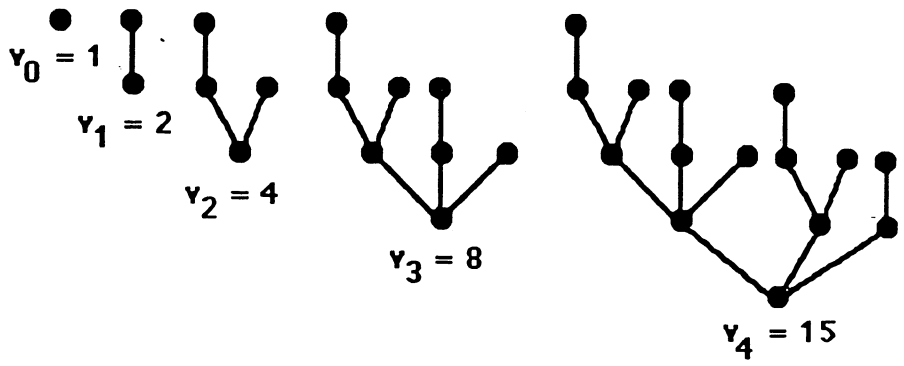


Figure 9 : First computation trees with d=4

Lemma 5 : Computation trees enable to perform the reduction operation in time $t_{comp}(v_n) = n\ (C+A)$.

Again, the proof is by induction on n, using Lemma 2. With the notation of Lemma 3, we always have $\gamma = \alpha$ for communication trees. We have:

- if $d \geq 3$, $v_n = \lambda_d\ K(d)^n + o(n)$, where λ_d is some constant, and K(d) is the largest real root of the polynomial $X^{d-1} - X^{d-2} - X^{d-3} - \ldots - X - 1$. Note that $K(d) \in]1,2[$. For instance with d=4, $\lambda_d \approx 0.76$ and $K(d) \approx 1.84$.
- if $d = 2$, $u_n = n + 1$

It is important to point out that u_n (resp: v_n) is not the maximal number of procesors p that can be arranged into a tree for performing the reduction in time $t(u_n)$ (resp: $t(v_n)$). However, the communication and computation trees are helpful for two reasons:

- they are simple to generate
- they are optimal for small n, if the ratio $\rho = C/A$ is large (communication trees) or small (computation trees)
- given p, they permit to bound the optimal time t(p), as shown in section 3.4.

3.4. Bounds

We conclude this section by deriving some bounds on the optimal time t(p) to perform a reduction operation with p processors. We use the two families of trees that we have introduced. Given p, we define comm(p) to be the smallest index in the sequence $(u_n)_{n\geq 0}$ such that p is smaller than the number of nodes of the associated communication tree, i.e.

$$u_{comm(p)-1} < p \leq u_{comm(p)}$$

Similarly, let comp(p) be the index such that $v_{comp(p)-1} < p \leq v_{comp(p)}$. For instance with d=4 and p=32:

$$u_2=13 < p=32 \leq u_3=40$$
$$v_5=28 < p=32 \leq v_6=52$$

hence comm(32)=3 and comp(32)=6.

Proposition 2 : We have the following inequalities:

$$comm(p).(C+A) \leq t(p) \leq \min \{comm(p).(C+(d-1)A),\ comp(p).(C+A)\}$$

Proof : The upper bound comes from the definition of comm(p) and comp(p):

- $p \leq u_{comm(p)}$ and $t(u_{comm(p)}) \leq t_{comm}(u_{comm(p)}) = comm(p).(C+(d-1)A)$

• $p \leq v_{comp(p)}$ and $t(v_{comp(p)}) \leq t_{comp}(v_{comp(p)}) = comp(p).(C+A)$
For the lower bound, we say that any tree of degree d with p processors has a height ≥ comm(p). Consider a leaf node N at a distance comm(p) from P_0 and follow the path $<N, F_1, F_2, ..., F_{comm(p)} = P_0>$:
• assume that N sends its data item to its father F_1 at time t=0
• the father F_1 will not send the data item to its own father F_2 before time C+A (Lemma 1)
• ...
• P_0 will not complete its computation before time comm(p).(C+A). ■

Note that

$$comm(p) \sim_{p \to +\infty} \frac{\log_2 p}{\log_2 d-1}$$

$$comp(p) \sim_{p \to +\infty} \frac{\log_2 p}{\log_2 K(d)}$$

so that we retrieve the fact that $t(p) = \Theta(\log_2 p)$. But we have derived a much more precise result here. We can use the asymptotic expressions for comm(p) and comp(p) for comparing the performances of the two families of trees for large p:

Proposition 3 : Let $\rho = \frac{C}{A}$ be the ratio communication/arithmetic. For large p, communication trees perform better than computation trees iff

$$\frac{\rho+(d-1)}{\log_2(d-1)} \leq \frac{\rho+1}{\log_2 K(d)}$$

Proof : The time to perform the reduction operation with p processors is the following:
• if the p processors are arranged into a communication tree:

$$t(p) = comm(p).(C+(d-1)A) = \frac{C+(d-1)A}{\log_2 (d-1)} \log_2 p + o(\log_2 p)$$

• if the p processors are arranged into a computation tree:

$$t(p) = comp(p).(C+A) = \frac{C+A}{\log_2 K(d)} \log_2 p + o(\log_2 p)$$

hence the result. ■

For instance with d=4, we find that communication trees perform asymptotically better than computation trees iff $\rho \geq 1.49$.

4. Pipelining reduction operations

In this section, we deal with the pipelining of several reduction operations. This problem arises when the data items are vectors, and when the reduction operation can be applied componentwise to these vectors.

So we assume that the data items X_i are vectors of length L. We pipeline the reduction operations which are performed componentwise. The time for computing a scalar $\Diamond$-operation is τ_a and the time for sending/receving a message of length M between two neighbor processors is $\beta + M\tau_c$. Leaf processors will send subvectors to their father in a pipeline fashion. More precisely, we assume that leaf processors send ν packets of length $\frac{L}{\nu}$, and we determine the optimal number of packets as a function of the number p of processors, the maximum degree d of the interconnection tree, the machine-dependent parameters τ_a, β and τ_c, and the total length L of the vectors.

Lemma 6 : The time to perform the reduction of ν packets of length $\frac{L}{\nu}$ on a perfect tree[1] of degree d and height n is equal to

$$t_{pipe}(\nu) = n(C+(d-1)A) + (\nu-1)\max(C,(d-1)A)$$

where $A = \frac{L}{\nu}\tau_a$ and $C = \beta + \frac{L}{\nu}\tau_c$

Proof : It takes $n(C+(d-1)A)$ units of time to process the first packet and the remaining ones follow at the rate $\max(C,(d-1)A)$. See figure 10 for an illustration with d=4: we consider a leaf processor F_0 that sends 3 messages to its father F_1, which in turn propagates them to its own father F_2 after reduction.

In the first case, we see that the steady rate of the pipeline is imposed by the computation, while it is imposed by the communication in the second case. ■

Given p, n, L and the machine-dependent parameters τ_a, β and τ_c, it is not difficult to compute the optimal number of packets ν_{opt}:

[1] All the levels of a perfect tree are full, except may be the last one which is filled from left to right.

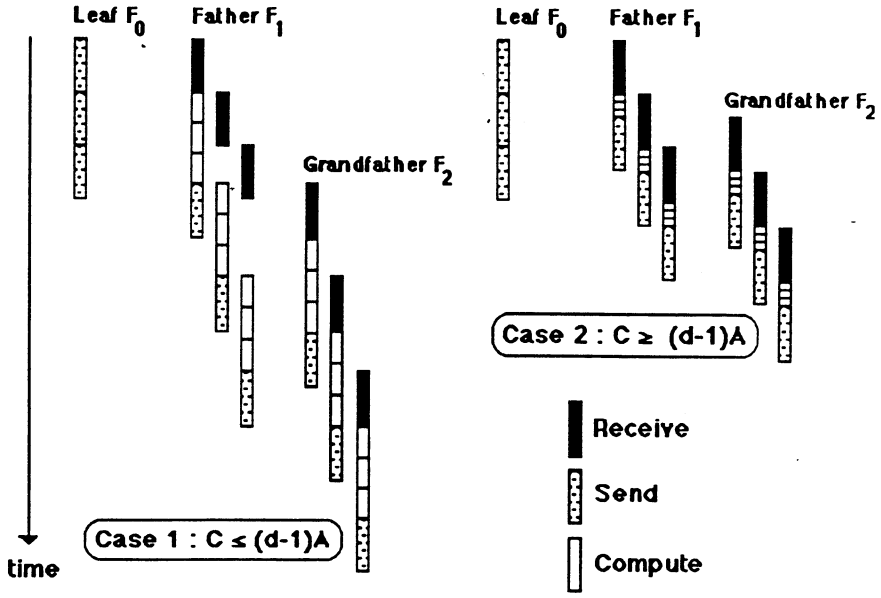


Figure 10 : Pipelining 3 reductions with d=4

Lemma 7 : The optimal time to perform the reduction of vectors of length L on a tree of degree d and height n is obtained by splitting the vectors into v_{opt} packets of length length $\frac{L}{v_{opt}}$ and by performing the reduction of these packets in a pipeline fashion, where

$$v_{opt} = \sqrt{\frac{(n-1)\tau_c + n(d-1)\tau_a}{\beta} L}$$

Proof : If $\tau_c \geq (d-1)\tau_a$, then $C = \beta + \frac{L}{v}\tau_c$ is greater than $A = \frac{L}{v}\tau_a$ for all values of v, hence $t_{pipe}(v) = n(C+(d-1)A) + (v-1)\,C = (n+v-1)(\beta + \frac{L}{v}\tau_c) + n(d-1)\frac{L}{v}\tau_a$, which is minimum for $v = v_1 = \sqrt{\frac{(n-1)\tau_c + n(d-1)\tau_a}{\beta} L}$.

If $\tau_c \leq (d-1)\tau_a$, we have two cases:

- $\max(C,(d-1)A) = A$ for $v \leq v_2$
- $\max(C,(d-1)A) = C$ for $v \geq v_2$, where $v_2 = \frac{(d-1)\tau_a - \tau_c}{\beta} L$.

For $v \leq v_2$, $t_{pipe}(v)$ is a non-increasing function of v, hence it is minimal for $v = v_2$. As can be checked in figure 11, the whole minimum is always achieved for $v = v_1$. ■

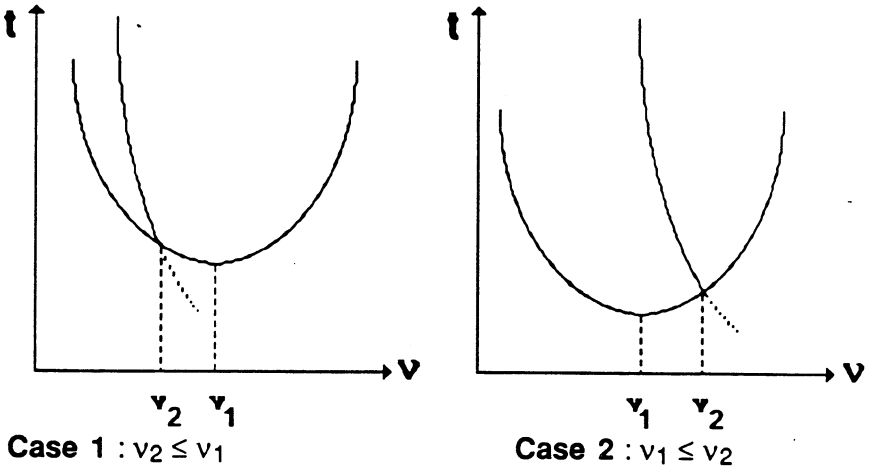


Figure 11 : Deriving the optimal number of packets

Under the hypotheses of Lemma 7, the optimal time is

$$t_{pipe}(v_{opt}) = (n-1)\beta + L\tau_c + 2\sqrt{((n-1)\tau_c + n(d-1)\tau_a)\beta L}$$

Lemma 8 : Any tree with p processors and of maximal degree $d \geq 3$ is of height n, where $n \geq \left\lceil \frac{\log_2 (d-2)p}{\log_2 d-1} \right\rceil - 1$.

Proof : For a complete tree of degree d and height n,

$$p = 1 + (d-1) + (d-1)^2 + \ldots + (d-1)^n = \frac{(d-1)^{n+1} - 1}{d-2},$$

hence the result. ■

Given p, we minimize n by arranging the interconnection tree as a perfect tree of degree d (all the levels are complete except possibly the last one, which is filled from left to right). The larger d, the smaller n, according to Lemma 8. For large p, we have $n = \frac{\log_2 p}{\log_2 d-1} + o(\log_2 p)$, hence we can rewrite the expression for the optimal time as

$$t_{pipe}(v_{opt}) = L\tau_c + \frac{\log_2 p}{\log_2 d-1}\beta + 2\sqrt{(\tau_c + (d-1)\tau_a)\,\beta L \frac{\log_2 p}{\log_2 d-1}} + o(\log_2 p)$$

For fixed L and large p, we deduce that the dominant term is $\frac{\log_2 p}{\log_2 d-1}\beta$, hence we should use perfect trees of maximum possible degree d. For fixed p and large L, the dominant term is always $L\tau_c$ (as expected), but we can minimize the second order term by decreasing $\sqrt{\frac{\tau_c + (d-1)\tau_a}{\log_2 d-1}}$. For instance if $\tau_c = \tau_a$, then this term is minimum for d=4. In the general case, given numerical values for p, L and the machine-dependent parameters β, τ_c and τ_a, it is easy to determine the best value d_{opt} of d from the expression of $t_{pipe}(v_{opt})$, and to arrange the p processors in a perfect tree of degree d_{opt}. Then the reduction is performed using packets of length v_{opt}.

5. Numerical experiments

In this section, we report numerical experiments performed on a Supernode machine with 32 processors. The interconnection network is reconfigurable, hence we can use any interconnection graph of maximal degree 4.

In this section, we only report upon a single experiment, which we believe to be the most important. We compare the performance of communication and computation trees for performing a single reduction operation (no pipelining). The idea is to let the arithmetic cost vary and to determine the breaking point at which computation trees become more efficient[1].

We experiment with a data structure of size large enough, namely 800 bytes, so that the communication cost can be precisely estimated through the value of τ_c (with the notations of section 2.1, β_0 and β_{link} are more difficult to estimate accurately). For ternary trees, there are three receives in parallel, so that

$$C = \beta_0 + 2\beta_{link} + 800\tau_c = 906.7\ \mu s$$

In table 2, we give experimental and theoretical (in italics) results obtained by letting τ_a vary[2]. We have $A = 800\tau_a$ and we let $\rho = C/A$ range from 0.1 to 10.

	communication tree	computation tree
$\rho = 0.1$	84500 *(81603)*	43600 *(40042)*
$\rho = 0.5$	17300 *(16230)*	11500 *(10920)*
$\rho = 1$	8900 *(8160)*	8000 *(7280)*
$\rho = 5$	2000 *(1632)*	5200 *(4638)*
$\rho = 10$	1200 *(816)*	4500 *(4004)*

Table 2 : Experimental *(predicted)* data with p = 32 processors

We check that the experimental data is in good accordance with the predicted value. In particular, the communication tree becomes less and less efficient as ρ increases, while it is the other way round for the computation tree. Theoretical values are computed using the results of sections 3.3 and 3.4: comm(32) = 3 and comp(32) = 6. The theoretical cross-point is given by:

$$\text{comm}(\rho)\ (C+3A) = \text{comp}(\rho)\ (C+A)$$

$$\Leftrightarrow 3(3+\rho) = 6(1+\rho)$$

$$\Leftrightarrow \rho = 1$$

which is in good accordance with the experimental data.

---

[1] Note that this experiment was motivated by results obtained for the implementation of the Z-buffer algorithm on a Supernode machine. Tiles are distributed to the processors, which simultaneously compute their contribution to the image. The partial contributions are then merged using a reduction tree [MLR].

[2] To this purpose, we generate a simple loop (assignment to the elements of a vector) whose size is adjusted by hand.

6. Conclusion

We have considered reduction operations with distributed memory machines whose interconnection networks are reconfigurable. We have assumed state-of-the-art processors for which computation and communication can be overlapped, and for which communication can occur in parallel on all the links.

We have focused on the 'compute-then-send' strategy which is very natural for performing the reduction operation. We have assumed a tree-based interconnection network, and we have supposed that each node sends the result of the reduction on its own subtree to its father. We have analytically determined the best tree-based interconnection topology of maximal degree d as a function of the number of processors and some machine-dependent parameters such as the elemental computation and communication times. We have also introduced two special families of trees that are very simple to generate and that perform well when the ratio communication/arithmetic is high (communication trees) or low (computation trees). Finally, we have analysed pipeline strategies when the data items are vectors. Experiments on Transputer-based networks are in good accordance with the theoretical results.

7. References

[FMR] P. FRAIGNIAUD, S. MIGUET, Y. ROBERT, Scattering on a ring of processors, Parallel Computing 13 (1990), 377-383

[GH] G.A. GEIST, M.T.HEATH, Matrix Factorization on a hypercube multiprocessor, Hypercube Multiprocessors 1986, M.T. Heath ed., SIAM (1986), 161-180

[HR] M.T. HEATH, C.A. ROMINE, Parallel solution of triangular systems on distributed-memory multiprocessors, SIAM J. Sci. Stat. Comput. 9, 3 (1988), 558-587

[HJ] C.T. HO, S.L. JOHNSSON, Optimum broadcasting and personalized communication in hypercubes, IEEE Trans. Computers 38, 9 (1989), 1249-1268

[LC] G.LI, T.F. COLEMAN, A parallel triangular system solver for a distributed-memory multiprocessor, SIAM J. Sci. Stat. Comput. 9, 3 (1988), 485-502

[MLR] J.J. LI, S. MIGUET, Y. ROBERT, S. UBEDA, Image processing algorithms on distributed memory machines, in "From pixels to features: parallelism in image processing", J.C. Simon ed., North Holland, to appear

[MV] O.A. MACBRYAN, E.F. VAN DE VELDE, Hypercube algorithms and implementations, SIAM J. Sci. Stat. Comput. 8, 2 (1987), s227-s287

[Nic] D. A. NICOLE, Esprit Project 1085, Reconfigurable Transputer Processor Architecture, in CONPAR 88, C. R. Jesshope et al. eds., Cambridge University Press (1989), 81-89.

[Saa] Y. SAAD, Gaussian elimination on hypercubes, in Parallel Algorithms and Architectures, M. Cosnard et al. eds., North-Holland (1986), 5-18

[SS] Y. SAAD, M.H. SCHULTZ, Data communication in parallel architectures, Parallel Computing 11 (1989), 131-150

[SW] Q.S. STOUT, B. WAGER, Intensive hypercube communication, Technical Report CRL-TR-9-87, Computing Research Laboratory, The University of Michigan (1987)

Bilan et perspectives

Comme nous l'avons indiqué dans l'introduction de la deuxième partie, les problèmes auxquels nous nous sommes intéressés ont au moins un point commun : les algorithmes qui les résolvent sont composés de tâches élémentaires très dépendantes entre elles, ce qui rend leur parallélisation difficile. En fonction de l'exemple étudié, différentes techniques ont été adoptées : stratégie d'allocation des données, choix du réseau d'interconnexion, choix de la granularité des opérations élémentaires ou utilisation de primitives de communication de haut niveau. Forts de nos expériences, nous tentons ici d'identifier des familles d'algorithmes, pour lesquels des techniques de parallélisation particulières peuvent être appliquées. Nous dégageons essentiellement deux grandes classes d'algorithmes :

4.1. Parallélisme "statique"

Dans la première classe, les opérations élémentaires à effectuer sont connues à l'avance et leur schéma d'exécution ne dépend pas dynamiquement des données. Pour les problèmes appartenant à cette classe, nous avons su trouver des fonctions d'allocation statiques qui offrent un bon compromis entre les deux exigences souvent contradictoires en algorithmique parallèle : l'équilibrage des calculs entre les processeurs et la minimisation du volume total de communication. Ces stratégies d'allocation reposent sur une modélisation de la machine cible, notamment en terme de temps élémentaires de calcul et de communication. Pour tous les problèmes traités appartenant à cette classe,

nous obtenons des algorithmes asymptotiquement optimaux[1] (pour les deux instances de programmation dynamique), voire optimaux (pour le produit d'une matrice symétrique par un vecteur et l'opération de distribution sur réseau linéaire).

Il est intéressant de noter que de nombreux algorithmes présentent cette particularité. En algèbre linéaire, ou en traitement d'image de bas niveau, par exemple, les programmes sont très souvent composés de boucles "pour" imbriquées, et ne comportent pas de tests. Des techniques similaires à celles que nous avons utilisées peuvent être employées pour bon nombre d'entre eux. B. Tourancheau [Tou] étudie ainsi dans sa thèse, la parallélisation de plusieurs algorithmes d'algèbre linéaire, et recourt pour les résoudre, au même type de solutions.

De même, les algorithmes systoliques, dédiés à des machines spécialisées, ou les algorithmes destinés aux machines SIMD, possèdent la régularité dans les calculs qui nous conduit à les inclure dans cette classe d'algorithmes. Néanmoins, il ne peuvent pas s'implémenter tels quels sur des ordinateurs MIMD à mémoire distribuée, car on ne dispose pas en général d'un nombre suffisant de processeurs. En revanche, on est beaucoup moins limité en place mémoire et on peut effectuer des tâches élémentaires plus complexes. Il faut donc adapter la granularité des opérations élémentaires, en regroupant sur un même processeur, des tâches qui auraient été allouées à des processeurs distincts du réseau systolique ou de la machine SIMD (techniques de partitionnement). On retrouve ici les idées qui sont à la base de l'allocation par blocs décrite en section 2.1.

Pour cette classe d'algorithmes, la topologie du réseau d'interconnexion joue un rôle primordial, et fait intrinsèquement partie de l'algorithme parallèle lui-même. Nous verrons que c'est moins vrai pour les algorithmes de la deuxième classe.

4.2. Parallélisme "dynamique"

Dans la deuxième classe de problèmes, lorsque le schéma de calcul dépend dynamiquement des données, il est bien plus difficile d'obtenir une parallélisation optimale. On ne connaît à l'avance ni les calculs qui

---

[1] Dans ce cadre, on considère qu'un algorithme parallèle est optimal si son efficacité (définie en section 1.4) est égale à 1, et asymptotiquement optimal si elle tend vers 1 quand la taille du problème et le nombre de processeurs tendent simultanément vers l'infini, avec un rapport constant.

doivent être effectués, ni les relations de dépendance entre les tâches. C'est le cas par exemple de l'algorithme du Z-Buffer. Pour paralléliser les problèmes de cette deuxième classe, deux techniques sont souvent employées :

4.2.1 Changer d'algorithme

La première approche consiste à paralléliser un autre algorithme[1] qui résout le même problème, et pour lequel on connait mieux les relations de dépendance entre tâches. On se ramène en quelque sorte à un problème de la première classe. C'est la solution que nous avons adoptée pour l'implémentation du Z-Buffer sur un arbre. En effet, l'étape de fusion est un calcul qui n'appartient pas à l'algorithme séquentiel. Mais elle permet d'introduire des nouvelles tâches élémentaires qui sont le calcul des bandes horizontales. Ces tâches peuvent être allouées statiquement aux processeurs, leur ordre d'évaluation est connu à l'avance et ne dépend pas dynamiquement des données. On est donc tout à fait dans la première classe d'algorithme. Par compte, le fait de rajouter ces calculs supplémentaires devient un handicap lorsque ceux-ci deviennent prépondérants. C'est le phénomène que l'on observe dans la version parallèle du Z-Buffer sur un arbre, quand la taille de l'image synthétisée est trop grande en regard de la complexité de la scène à traiter.

4.2.2. Utiliser des macro-communications

La deuxième approche consiste à mettre en oeuvre des primitives de communication de haut niveau évoquées en section 1.3. Dès lors que les processeurs peuvent avoir accès rapidement à des données d'autres processeurs éventuellement distants, on s'affranchit des problèmes de localité. C'est la solution que nous avons adoptée pour la version sur l'anneau du Z-Buffer, où nous réallouons dynamiquement la scène, par une opération de multi-diffusion. L'avantage essentiel de ce type de parallélisation est la portabilité des programmes écrits avec ces primitives, puisqu'ils deviennent indépendants de la topologie du réseau. Dès que ces macro-communications sont disponibles sur une machine, celle-ci pourra exécuter le code (plus ou moins efficacement, suivant la topologie d'interconnexion sous-jacente). Pour faire une remarque qui va

---

[1] Dans le sens où il fait d'autres calculs, et par opposition à un algorithme parallèle de la première classe, qui effectue (dans un ordre différent) les mêmes calculs sur les mêmes données.

dans ce sens, nous sommes en train d'étudier l'algorithme de multi-diffusion sur l'anneau-mélange-parait (voir section 1.2.2.). Lorsque cette primitive sera disponible, nous pourrons exécuter exactement le même programme sur l'anneau-mélange-parfait que sur l'anneau, mais de manière bien plus efficace.

Ici encore, lorsque ces communications intensives deviennent prépondérantes les performances de l'algorithme parallèle se dégradent. C'est le phénomène observé pour le Z-Buffer sur l'anneau, quand la scène à traiter est trop volumineuse par rapport à la taille de l'image produite.

4.3 Conclusion

L'algorithmique parallèle est une science récente. Elle a besoin de ses expérimentateurs pour élaborer des techniques nouvelles et des modèles théoriques adaptés. Les recherches passent parfois par des tâtonnements et des tentatives infructueuses. Mais elles sont nécessaires et apportent souvent des résultats encourageants, inaccessibles pour l'instant à des techniques automatisées. Verra-t-on un jour un compilateur capable de placer des processus sur un réseau de processeurs avec une efficacité surpassant celle de l'expert ? si c'est le cas dans le futur, ce sera grâce aux modèles dégagés par l'algorithmicien parallèle. Sera-t-il alors relégué dans l'oubli ? gageons qu'il sera en train d'étudier la nouvelle classe d'ordinateurs, la machine d'après demain.

En matière d'algorithmique parallèle, il n'y a pas de miracle. Le succès vient de l'association de la théorie et de l'expérience. Nous espérons en avoir convaincu à la fois le théoricien replié sur ses formules, et l'expérimentateur noyé sous ses données ! qu'on nous pardonne ce message final qui ne se veut pas du tout prétentieux.

Références

La bibliographie est divisée en deux parties. Nous donnons tout d'abord les références personnelles ayant donné lieu à des publications. Celles qui sont reproduites dans la thèse sont référencées par un chiffre compris entre 1 et 6, et les autres par une lettre comprise entre a et g.

Nous donnons ensuite une bibliographie plus générale sur le sujet de recherche, référencée dans la thèse par les initiales des auteurs.

Pour les références plus spécifiques à chacun des sujets traités, on se reportera à la bibliographie détaillée donnée à la fin des six articles présentés.

Publications personnelles

a) reproduites dans la thèse

[1] S. MIGUET, Y. ROBERT. Dynamic programming on a ring of processors, Hypercube and Distributed Computers, F. André et J.P. Verjus eds., North Holland (1989), 19-33

[2] S. MIGUET, Y. ROBERT. Path planning on a ring of processors, International Journal of Computer Mathematics, 32 (1990), 61-74

[3] K. GRIGG, S. MIGUET, Y. ROBERT. Symmetric Matrix Vector Product on a ring of processors. Information Processing Letters 35 (1990) 239-248

[4] P. FRAIGNIAUD, S. MIGUET, Y. ROBERT. Scattering on a ring of processors. Parallel Computing 13 (1990), 377-383

[5] J. J. LI, S. MIGUET. Z-Buffer on a Transputer-Based machine. Rapport de recherche LIP-IMAG, 90-30. Soumis pour publication

[6] S. MIGUET, Y. ROBERT. Reduction Operations on a distributed memory machine with a reconfigurable interconection network. Rapport de recherche LIP-IMAG 90-28. Soumis pour publication

b) ne figurant pas dans la thèse

[a] S. MIGUET. Optimal parenthesization of a matrix chain product on a hypercube. Parallel Computing 89. Leiden, The Netherlands, 29 August-1 September, 1989. Actes publiés par North-Holland.

[b] S. MIGUET. Distance transform on a ring of processors. IFIP W.G. 10.3. Working Conference on Decentralized Systems. Lyon, France, 11-13 Decembre, 1989. Actes publiés par North-Holland.

[c] S. MIGUET, Y. ROBERT. Path planning on distributed memory machines. 5th Distributed Memory Computing Conference. Charleston (Caroline du Sud), U.S.A, 8-12 Avril, 1990.

[d] K. GRIGG, S. MIGUET, Y. ROBERT. Complexity of the Symetric Matrix Vector product on a ring of processors. 5th Distributed Memory Computing Conference. Charleston (Caroline du Sud), U.S.A, 8-12 Avril, 1990.

[e] P. FRAIGNIAUD, S. MIGUET, Y. ROBERT. Complexity of scattering on a ring of processors. 5th Distributed Memory Computing Conference. Charleston (Caroline du Sud), U.S.A, 8-12 Avril, 1990.

[f] S. MIGUET. Parallel impementation of the Distance Transform algorithm. 5th European Signal Processing Conference. Barcelone, Septembre 1990. Actes publiés par North-Holland.

[g] S. MIGUET, Y. ROBERT. Parallélisation d'algorithmes de balayage d'image sur un anneau de processeurs. Technique et Science Informatiques. A paraître

Bibliographie générale

[BK] F. BITZ, H.T. KUNG, Path planning on the Warp computer: using a linear systolic array in dynamic programming, Intern. J. Computer Math. 25 (1988), 173-188

[BM] L. BRUNIE, S. MIGUET, Chirurgie du poignet assistée par ordinateur. Faculté de Médecine, La Tronche, Rapport de 3ème année. ENSIMAG (1990)

[BP] J.C BERMOND, PEYRAT. De Bruijn and Kautz networks : a competitor for the hypercube ? Hypercube and Distributed Computers, F. Andre & J.P. Verjus edt. Elseiver Science Publisher BV. North Holand, (1989) 279-293

[DP] M. L. DUPLAQUET, P. POUSSET. Spot Image Mosaïc and Dynamic Programming. Signal Processing V : Theory and applications. Elseiver Science Publisher BV. (1990) 1031-1034

[Des] F. DESPREZ. Algèbre Linéaire Distribuée. Rapport de DEA. ENS-Lyon (1990)

[Fly] M.J. FLYNN Some Computer Organizations and Their Effectiveness. (1972) IEEE Trans. on Computers C-21, 9 (1972) 948-960

[GH] G.A.GEIST, M.T.HEATH, Matrix Factorization on a hypercube multiprocessor, Hypercube Multiprocessors 1986, M.T. Heath ed., SIAM (1986), 161-180

[Gus] J.L. GUSTAFSON, Reevaluating Amdahl's law, Communications of the A.C.M. 31, 5 (1988), 532-533

[Mon] A. MONTANVERT, Medial line : graph representation and shape description, 8th International Conference on Pattern Recognition, Paris, (1986) 430-432

[MV] O.A. MAC BRYAN, E.F. VAN DE VELDE, Hypercube algorithms and implementations, SIAM J. Sci. Stat. Comput. 8, 2 (1987), s227-s287

[Saa] Y. SAAD, Gaussian elimination on hypercubes, in Parallel Algorithms and Architectures, M. Cosnard et al. eds., North-Holland (1986), 5-18

[Sak] M. SAKAROVITCH, Optimisation combinatoire, vol 2 : Programmation discrète. Chap 8 : Programmation dynamique. (1984) 185-223

[Tou] B. TOURANCHEAU, Algorithmique Parallèle pour les machines à mémoire distribuée (application aux algorithmes matriciels). Thèse de l'INPG. (1989)

[Ube] S. UBEDA, Parallélisation d'algorithmes d'amincissement d'image. Rapport de DEA. ENS-Lyon (1990)

[YBR] A. Y. WU, S. K. BHASKAR, A. ROSENFELD, Parallel Computation of Geometric Properties from the Medial Axis Transform, Comput. Vision, Graphics Image Process. 41, (1988) 323-332

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Grenoble, le 14 Décembre 1990

DÉPARTEMENT DES ÉTUDES DOCTORALES

Affaire suivie par
Tél : 76.57.

N/Réf. :

Objet :

AUTORISATION de SOUTENANCE

Vu les dispositions de l'arrêté du 23 Novembre 1988 relatif aux Etudes Doctorales
Vu les rapports de présentation de :

- **Monsieur CINOUIN**
- **Monsieur COSNARD**

Monsieur MIQUET Serge

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"

Programmation dynamique et traitement d'images sur machines parallèles à mémoire distribuée.

Mots-clé

parallélisme, machines à mémoire distribuée, programmation dynamique, algèbre linéaire, traitement d'images, synthèse d'images.

Résumé

Nous étudions la mise en œuvre d'algorithmes parallèles sur des ordinateurs à mémoire distribuée. A travers plusieurs exemples issus de la programmation dynamique, de l'algèbre linéaire et du traitement d'images, nous exposons les problèmes liés à la programmation de ces machines : topologie d'interconnexion, stratégie d'allocation des données, équilibrage des calculs et minimisation du volume de communication inter-processeurs. Les exemples étudiés sont pour la plupart des algorithmes séquentiels coûteux en temps de calcul et en place mémoire, et pour lesquels il est très intéressant d'avoir une parallélisation efficace. Nous avons choisi des problèmes dont l'implémentation sur des machines à mémoire distribuée n'est pas aisée, essentiellement à cause de la grande interdépendance entre les différentes tâches composant les algorithmes.

Abstract

We study the implementation of parallel algorithms on distributed memory computers. By taking into account several examples from dynamic programming, linear algebra and image processing, we expose the specific problems that occur when programming these machines : interconnection topologies, data allocation strategies, workload balancing and interprocessor communication. The examples that we study are compute-intensive and memory-consuming algorithms for which an efficient parallelization is very interesting. Furthermore, they are difficult to parallelize because of the great interdependecy between the tasks composing the algorithms.