Algorithmique parallèle : réseaux d'automates, architectures systoliques, machines SIMD et MIMD

Yves Robert

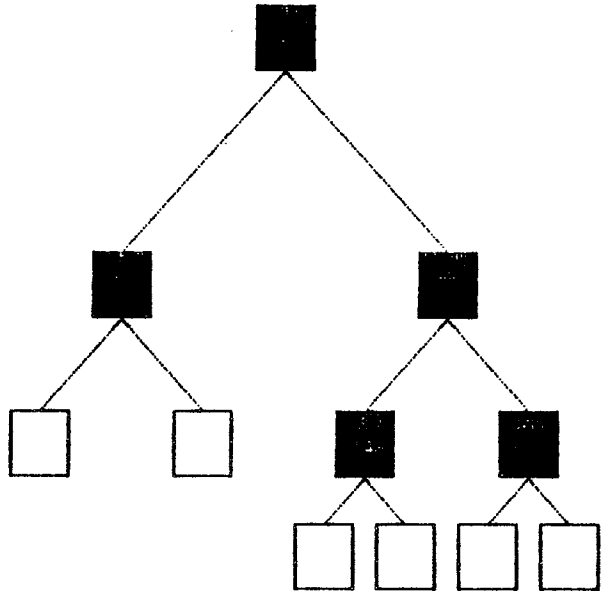
THESE

présentée à
l'Institut National Polytechnique de Grenoble
et à
l'Université Scientifique et Médicale de Grenoble

pour obtenir le grade de
DOCTEUR ES SCIENCES
« Informatique »

par

Yves ROBERT



ALGORITHMIQUE PARALLELE : réseaux d'automates, architectures systoliques, machines SIMD & MIMD

Date de soutenance : 6 Janvier 1986

Composition du Jury :

G. Mazare	Président
J. Berstel	
C. Delobel	
F. Jaeger	
P. Quinton	Examinateurs
F. Robert	
M. Tchuente	

Thèse préparée au sein du Laboratoire TIM3, U.A. au CNRS n° 397

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : M. TANCHE

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

(RANG A)

SAUF ENSEIGNANTS EN MEDECINE ET PHARMACIE

PROFESSEURS DE 1ère CLASSE

ARNAUD Paul	Chimie organique
ARVIEU Robert	Physique nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S.
AYANT Yves	Physique approfondie
BARBIER Marie-Jeanne	Electrochimie
BARBIER Jean-Claude	Physique expérimentale C.N.R.S. (labo de magnétisme)
BARJON Robert	Physique nucléaire I.S.N.
BARNOUD Fernand	Biosynthèse de la cellulose-Biologie
BARRA Jean-René	Statistiques - Mathématiques appliquées
BELORISKY Elie	Physique
BENZAKEN Claude (M.)	Mathématiques pures
BERNARD Alain	Mathématiques pures
BERTRANDIAS Françoise	Mathématiques pures
BERTRANDIAS Jean-Paul	Mathématiques pures
BILLET Jean	Géographie
BONNIER Jean-Marie	Chimie générale
BOUCHEZ Robert	Physique nucléaire I.S.N.
BRAVARD Yves	Géographie
CARLIER Georges	Biologie végétale
CAUQUIS Georges	Chimie organique
CHIBON Pierre	Biologie animale
COLIN DE VERDIERE Yves	Mathématiques pures
CRABBE Pierre (détaché)	C.E.R.M.O.
CYROT Michel	Physique du solide
DAUMAS Max	Géographie
DEBELMAS Jacques	Géologie générale
DEGRANGE Charles	Zoologie
DELOBEL Claude (M.)	M.I.A.G. Mathématiques appliquées
DEPORTES Charles	Chimie minérale
DESRE Pierre	Electrochimie
DOLIQUE Jean-Michel	Physique des plasmas
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques pures
GAGNAIRE Didier	Chimie physique

.../...

GASTINEL Noël	Analyse numérique - Mathématiques appliquées
GERBER Robert	Mathématiques pures
GERMAIN Jean-Pierre	Mécanique
GIRAUD Pierre	Géologie
IDELMAN Simon	Physiologie animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques pures
JULLIEN Pierre	Mathématiques appliquées
KAHANE André (détaché DAFCO)	Physique
KAHANE Josette	Physique
KOSZUL Jean-Louis	Mathématiques pures
KRAKOWIAK Sacha	Mathématiques appliquées
KUPTA Yvon	Mathématiques pures
LACAZE Albert	Thermodynamique
LAJZEROWICZ Jeannine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre	Mathématiques appliquées
DE LEIRIS Joël	Biologie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences nucléaires I.S.N.
LOUP Jean	Géographie
MACHE Régis	Physiologie végétale
MAYNARD Roger	Physique du solide
MICHEL Robert	Minéralogie et pétrographie (géologie)
MOZIERES Philippe	Spectrométrie - Physique
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (biologie végétale)
PAYAN Jean-Jacques (détaché)	Mathématiques pures
PEBAY PEYROULA Jean-Claude	Physique
PERRIAUX Jacques	Géologie
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
RASSAT André	Chimie systématique
RENARD Michel	Thermodynamique
RICHARD Lucien	Biologie végétale
RINAUDO Marguerite	Chimie CERMAV
SENGEL Philippe	Biologie animale
SERGERAERT Francis	Mathématiques pures
SOUTIF Michel	Physique
VAILLANT François	Zoologie
VALENTIN Jacques	Physique nucléaire I.S.N.
VAN CUTSEN Bernard	Mathématiques appliquées
VAUQUOIS Bernard	Mathématiques appliquées
VIALON Pierre	Géologie

PROFESSEURS DE 2ème CLASSE

ADIBA Michel	Mathématiques pures
ARMAND Gilbert	Géographie

.../...

AURIAULT Jean-Louis	Mécanique
BEGUIN Claude (M.)	Chimie organique
BOEHLER Jean-Paul	Mécanique
BOITET Christian	Mathématiques appliquées
BORNAREL Jean	Physique
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CHARDON Michel	Géographie
COHENADDAD Jean-Pierre	Physique
DENEUVILLE Alain	Physique
DEPASSEL Roger	Mécanique des fluides
DOUCE Roland	Physiologie végétale
DUFRESNOY Alain	Mathématiques pures
GASPARD François	Physique
GAUTRON René	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude (M.)	Sciences nucléaires I.S.N.
GUITTON Jacques	Chimie
HACQUES Gérard	Mathématiques appliquées
HERBIN Jacky	Géographie
HICTER Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKOVE Claude (M.)	Géologie
LE BRETON Alain	Mathématiques appliquées
LONGEQUEUE Nicole	Sciences nucléaires I.S.N.
LUCAS Robert	Physiques
LUNA Domingo	Mathématiques pures
MASCLE Georges	Géologie
NEMOZ Alain	Thermodynamique (CNRS - CRTBT)
OUDET Bruno	Mathématiques appliquées
PELMONT Jean	Biochimie
PERRIN Claude (M.)	Sciences nucléaires I.S.N.
PFISTER Jean-Claude (détaché)	Physique du solide
PIBOULE Michel	Géologie
PIERRE Jean-Louis	Chimie organique
RAYNAUD Hervé	Mathématiques appliquées
ROBERT Gilles	Mathématiques pures
ROBERT Jean-Bernard	Chimie physique
ROSSI André	Physiologie végétale
SAKAROVITCH Michel	Mathématiques appliquées
SARROT REYNAUD Jean	Géologie
SAXOD Raymond	Biologie animale
SOUTIF Jeanne	Physique
SCHOOL Pierre-Claude	Mathématiques appliquées
STUTZ Pierre	Mécanique
SUBRA Robert	Chimie
VIDAL Michel	Chimie organique
VIVIAN Robert	Géographie

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

A Sylvie
A mon petit Damien
A mes parents

Je voudrais exprimer toute ma reconnaissance aux membres du Jury:

- Monsieur Guy MAZARE, Professeur à l'ENSIMAG, pour l'honneur qu'il me fait en présidant le Jury de cette thèse
- Monsieur François ROBERT, Professeur à l'ENSIMAG, qui a dirigé mes recherches, pour ses conseils et encouragements tout au long de mon travail, pour les nombreuses heures qu'il m'a consacrées, et au-delà, pour tout ce que nous avons partagé ensemble depuis 6 ans
- Monsieur Jean BERSTEL, Professeur au LITP Paris, pour l'intérêt qu'il a manifesté pour mes recherches, et pour avoir accepté de juger cette thèse
- Monsieur Patrice QUINTON, Directeur de Recherches à l'IRISA Rennes, pour l'attention qu'il a portée au développement et à la mise en valeur de mon travail à l'occasion de son parrainage au CNRS, pour avoir accepté de juger cette thèse, et plus largement pour son action au service des Arcchitectures Systoliques dans le cadre du GRECO "C3" du CNRS
- Messieurs Claude DELOBEL, Professeur à l'USMG, François JAEGER, Directeur de Recherches au Laboratoire LSD, et Maurice TCHUENTE, Chargé de Recherches au Laboratoire TIM3, qui ont bien voulu faire parti du Jury.

Je remercie tout particulièrement Maurice TCHUENTE, et avec lui Michel COSNARD, pour notre longue collaboration et la complicité de notre trio depuis plus de 4 années. Merci également à Yves CLAUZEL, James H. DAVENPORT, Robert JAMIER, Ahmed A. JERRAYA, Jean-Michel MULLER, et Denis TRYSTRAM: je souhaite témoigner du plaisir que j'ai éprouvé à travailler avec chacun d'entre eux, et ... je suis prêt à renouveler l'expérience !

Je souhaite enfin remercier
- Jean DELLA DORA, Directeur du Laboratoire, pour le soutien actif qu'il a su donner aux activées centrées autour du parallélisme au sein de TIM3
- Claire DICRESCENZO et André EBERHARD, toujours prêts à secourir un utilisateur perdu devant sa console
- tous les membres du Groupe "Algorithmique Parallèle", et l'équipe d'Algorithmique Mathématique dans son ensemble.

Je ne saurais oublier toute l'Equipe du Service de Reprographie, Claude ANGUILLE, Daniel IGLESIAS, Claude LABORIE et Paul MOUNET, pour l'excellente qualité de leur travail, et pour leur constante gentillesse et disponibilité.

Il y a aussi mon ami MAC, toujours fidèle au poste. Nous avons rédigé cette thèse ensemble, et j'ai pu apprécier son savoir-faire et son humeur égale.

TABLE DES MATIERES

PARTIE 1 : EXPOSES DE SYNTHESE

PARTIE 2 : RECUEIL D'ARTICLES

INTRODUCTION

INTRODUCTION

Cette thèse traite d'algorithmique parallèle, sous divers aspects. Du comportement dynamique de réseaux d'automates à l'étude d'algorithmes parallèles pour machines SIMD et MIMD (selon la classification habituelle de FLYNN [25]), en passant par la conception d'architectures systoliques, l'idée directrice est toujours la même: définir de nouveaux algorithmes qui soient adaptés aux architectures étudiées, et analyser leur efficacité dans le cadre d'un modèle théorique bien défini.

La thèse comprend trois parties qui peuvent être considérées comme indépendantes, même si leur genèse ne l'est pas. Chacune d'elle comporte un exposé introductif, suivi d'un recueil d'articles. Les trois exposés ont pour but d'offrir une introduction, directement abordable par le non-spécialiste, à chacun des sujets traités dans les différentes parties. Pour plus de commodité pour le lecteur, ces exposés ont été rassemblés, permettant ainsi une présentation synthétique des thèmes abordés dans la thèse; les résultats plus spécialisés, regroupés par parties, font l'objet des chapîtres ultérieurs.

Le plan de la thèse est alors le suivant:

- chapître 1: contenu détaillé de la thèse
- chapîtres 2, 3 et 4: les trois exposés introductifs aux différentes parties
- chapîtres 5,6, et 7: recueils d'articles

Nous résumons très brièvement ci-dessous le contenu de chacune des trois parties, renvoyant aux exposés introductifs pour une présentation plus complète du sujet et au chapître 1 pour un exposé plus technique des résultats obtenus.

Partie 1
Comportement dynamique de réseaux d'automates monotones

Un réseau d'automates peut être défini de façon très générale comme un ensemble de cellules (automates finis) interconnectées localement, et évoluant dans un temps discret par interactions réciproques. Nous envisageons ici des automates déterministes, i.e. tels que la donnée d'une fonction de transition permet de déterminer l'évolution de toutes les cellules en fonction de la configuration initiale du réseau.

Si le réseau est fini, le nombre de configurations l'est également. A partir d'une configuration initiale, l'évolution se poursuit jusqu'à parvenir en régime périodique.

Une approche classique consiste à déterminer le comportement itératif du réseau à partir d'hypothèses sur la fonction de transition et sur le graphe de connexion (qui modélise les interactions entre cellules). Les principaux problèmes posés sont la détermination des cycles limites (nombre, périodes) et une majoration de la valeur du transitoire (nombre d'états avant le régime permanent).

Malgré l'introduction d'outils mathématiques sophistiqués, aucune approche unifiée n'a pu être développée pour répondre aux questions précédentes dans le cas général. En revanche, dans certains cas particuliers (automates à seuil par exemple, voir le chapitre 2 pour plus de précisions), on sait caractériser complètement l'évolution du réseau.

Nous nous intéressons au cas des fonctions de transition monotones: pour chaque cellule, l'état à l'instant t+1 est une fonction croissante des variables dont elle dépend. Nous établissons des résultats caractérisant la longueur des cycles limites à partir d'hypothèses sur le graphe de connexion. Dans la plupart des cas, les réseaux sont supposés booléens, mais nous donnons aussi des exemples où chaque cellule peut prendre $p \geq 3$ états (étude d'itérations à majorité généralisée notamment).

En outre, toujours dans le cas des réseaux monotones, nous proposons une approche inverse, permettant d'établir des résultats sur le graphe de connexion à partir d'hypothèses sur le graphe d'itération: nous montrons que la présence de certains cycles limites dans le graphe d'itération implique l'existence de circuits élémentaires d'une certaine longueur dans le graphe de connexion.

On considère ensuite des réseaux monotones booléens avec mémoire, et on montre - sur des exemples - l'influence du degré maximal du graphe de connexion sur la longueur maximale des circuits élémentaires du graphe d'itération.

Partie 2
Algorithmes et architectures systoliques

Les architectures systoliques sont constituées de réseaux de processeurs élémentaires localement et régulièrement agencés: en ce sens, ce sont des réseaux d'automates au plus pur sens du terme ! Leur particularité vient du fait qu'un intérêt technologique important motive et sous-tend leur étude: tout réseau systolique doit pouvoir être implémenté directement sur silicium, comme processeur intégré spécialisé relié à une structure hôte.

En vérité, la complexité des circuits intégrés disponibles à l'heure actuelle rend possible la réalisation à un faible coût de tels sytèmes parallèles. Deux restrictions cependant:
- il s'agit de processeurs spécialisés que l'on adjoint à un processeur hôte de type conventionnel
- la classe d'application est bien délimitée: les problèmes où le volume de calculs à effectuer prime largement sur les transferts de données à réaliser.

Le modèle systolique, introduit en 1978 par KUNG et LEISERSON [63], s'est révélé être un outil puissant pour la conception de processeurs intégrés spécialisés. Les réseaux systoliques se composent d'un grand nombre de cellules élémentaires identiques et localement interconnectées. Chaque cellule reçoit des données en provenance des cellules voisines, effectue un calcul simple, puis transmet les résultats, toujours aux cellules voisines, un temps de cycle plus tard. Pour fixer un ordre de grandeur, disons que chaque cellule a la complexité, au plus, d'un petit microprocesseur.

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

Les cellules évoluent en parallèle, en principe sous le contrôle d'une horloge globale (synchronisme total): plusieurs calculs sont effectués simultanément sur le réseau, et on peut "pipeliner" la résolution de plusieurs instances du même problème sur le réseau.

Nous présentons dans cette partie (la plus importante) de nombreux exemples d'algorithmes systoliques originaux, pour la résolution de problèmes issus surtout de l'informatique théorique et de l'Algèbre linéaire. Nous présentons également des architectures originales pour le Calcul Formel (PGCD de polynômes dans des corps finis). La variété des exemples abordés montre toute l'étendue du champ d'application de l'algorithmique systolique.

L'algorithmique systolique, pour reprendre ce terme qui vient d'être (imprudemment) employé, ne peut encore prétendre au titre de sous-discipline constituée de l'algorithmique parallèle. Cependant, après une période préliminaire qui a vu naître de nombreuses architectures systoliques grâce à la seule intuition de leurs auteurs, des méthodologies ont été développées, qui visent une conception automatique à partir de spécifications algorithmiques. Les réponses proposées sont encore insuffisantes, mais les outils se multiplient, tirant parti de la spécificité des architectures systoliques: des langages de description aux logiciels graphiques en passant par les modèles ad-hoc de test et de résistance aux pannes, l'arsenal du concepteur ne cesse de se développer.

Nous apportons notre (modeste) contribution à ces développements en complétant la méthode de systolisation des circuits synchrones de LEISERSON et SAXE [69] par une approche "divide-and-conquer" permettant d'augmenter l'efficacité des algorithmes implémentés.

Enfin, nous nous sommes initiés à la réalisation de circuits intégrés spécialisés (et nous avons testé par là-même la validité du modèle systolique) en utilisant les outils développés par l'équipe "Architectures d'ordinateurs et Sureté de fonctionnement" du Laboratoire TIM3 (dessin nMOS en LUCIE et compilateur de silicium APPOLON). Cet aspect pratique doit se développer prochainement avec la réalisation d'une cellule systolique programmable.

Avoir la possibilité d'étudier d'un point de vue théorique un modèle débouchant directement sur des applications concrètes est un rare privilège, et je fais bien volontiers miennes les lignes suivantes de C.E. LEISERSON, écrites en 1978: "Systolic systems are an attempt to capture the concepts of parallelism, pipelining and interconnexion structures in a unified framework of mathematics and engineering. They embody engineering techniques such as multiprocessing and pipelining together with the more theoretical ideas of cellular automata and algorithms, and

therefore are an excellent subject of investigation from a combined standpoint."

Partie 3
Algorithmes parallèles pour machines SIMD et MIMD

Dans cette partie, nous étudions l'influence du parallélisme sur la conception d'algorithmes s'exécutant efficacement sur les machines SIMD ou MIMD, en particulier pour la résolution de systèmes linéaires.

Lorsque l'on dispose de plus d'un processeur, une même méthode peut conduire à plusieurs implémentations différentes. Le nombre de processeurs n'est d'ailleurs pas le seul paramètre. Les contraintes imposées par la stucture de la machine peuvent influencer notablement les performances. De plus, le mode d'accès aux données prend une importance considérable et le rapport entre le temps de transmission et le temps de traitement devient une variable primordiale. Ces temps sont étroitement liés à la conception et à la fabrication de la machine parallèle sur laquelle est exécuté l'algorithme.

L'approche usuelle consiste à négliger les temps de transmission et stockage pour ne considérer que les opérations arithmétiques effectuées. Pour rester proche de la réalité, on suppose généralement pour la résolution d'un problème de taille n:
- l'existence d'un nombre de processeurs variant comme O(n) (au contraire du modèle systolique, un nombre quadratique de processeurs ne serait pas raisonnable au vu des perspectives actuelles sur les machines SIMD ou MIMD)
- la possibilité d'un accès simultané à tous les éléments d'une même ligne ou colonne d'une matrice, mais sans autoriser la modification d'un élément par plusieurs processeurs au même instant.

Ces contraintes étant définies, la première étape dans la parallélisation d'une méthode consiste en la définition des tâches élémentaires qu'exécuteront les processeurs et du graphe d'ordonnancement des tâches. Dans la classe des algorithmes admissibles (i.e. qui respectent les contraintes du graphe d'ordonnancement), on recherche alors des algorithmes optimaux, et, - quand on en trouve ! -, on tente d'évaluer leur temps d'exécution. On obtient alors la complexité de la méthode parallélisée.

On trouvera plusieurs illustrations de la démarche que nous venons de décrire dans cette partie. Sans entrer plus avant dans les résultats obtenus (renvoyant toujours au chapitre 1 pour des énoncés plus techniques), nous voulons souligner ici l'importance pratique de ce type d'étude: s'il est vrai que le modèle proposé est incomplet, notamment en matière de temps de communication, s'il est vrai encore que nous ne vivons pas "dans un monde de complexité asymptotique", les résultats que nous présentons permettent de disposer de critères fiables pour sélectionner certains algorithmes plutôt que d'autres lors d'une expérimentation concrète. Nous n'en prendrons pour preuve que la parfaite adéquation des évaluations théoriques et des résultats numériques obtenus par LORD, KOWALIK et KUMAR [70] sur l'ordinateur HEP de la firme Denelcor, Inc, lequel est pourtant de type MIMD (cas où le modèle s'avère le plus insuffisant).

Remerciements

La plupart des textes présentés dans cette thèse ont été écrits en collaboration. Je remercie vivement Yves CLAUZEL, Michel COSNARD, James H. DAVENPORT, Robert JAMIER, Ahmed A. JERRAYA, Jean-Michel MULLER, Maurice TCHUENTE, et Denis TRYSTRAM pour toutes les heures passées en commun, et souhaite témoigner du plaisir que j'ai éprouvé lors de chacune de ces collaborations.

CHAPITRE 1

CONTENU DE LA THESE

Comme nous l'avons dit, les chapitres 2, 3 & 4 sont des exposés introductifs à chacune des 3 parties de la thèse. Nous détaillons ici le contenu des résultats présentés dans les articles qui sont réunis aux chapîtres suivants.

1. Comportement dynamique de réseaux d'automates monotones

Les réseaux d'automates ont été introduits par John Von Neumann dans les années 1950. Informellement, un réseau d'automates peut s'interpréter comme un système dynamique discret dont le comportement global est la résultante des règles d'interaction locales des cellules élémentaires du réseau. Comme le souligne WOLFRAM [98], les réseaux d'automates ont été utilisés dans de nombreuses disciplines, car ils peuvent fournir des exemples de comportement dynamique complexe à partir de fonctions de transition très simples pour chacune des cellules.

Il n'existe pas de formalisme mathématique unifié pour modéliser le comportement dynamique de réseaux d'automates généraux, mais de nombreux outils ont été développés pour étudier certaines classes de réseaux: utilisation de l'algèbre polynomiale [72], introduction de fonctions d'énergie [13] [27] [37], arithmétique dans les corps finis [36], ... Nous renvoyons aux exemples développés dans le chapître 2 et aux travaux de TCHUENTE [91] pour un large tour d'horizon de ce type d'approches.

Dans cette partie, nous nous intéressons à la classe des réseaux d'automates monotones. Précisons tout d'abord quelques notations.

1.1. Notations

L'ensemble des cellules du réseau d'automates étudié $\mathcal{C}$ est noté $\mathcal{S}$ = {S1, S2, ..., Sn}. L'état Si(t) de la cellule Si à l'instant t est pris dans l'ensemble Q = {0,1,2,...,p}. Pour t≥0, on note $\mathcal{S}(t)$ = (S1(t),...Sn(t)) ∈ Q^n la configuration de $\mathcal{C}$ à l'instant t; $\mathcal{S}(0)$ est la configuration initiale de $\mathcal{C}$. La fonction de transition du réseau (qui détermine les règles d'évolution de chaque cellule) est une fonction

$$F = (f_1,...,f_n) : x = (x_1,...,x_n) \in Q^n \dashrightarrow F(x) \in Q^n$$

telle que chaque f_i ne dépend que d'un nombre restreint de variables x_j. On note FN(i) l'ensemble des indices j pour lesquels f_i dépend de x_j: FN(i) représente le voisinage fonctionnel de la cellule i.

On dit que F est monotone si

$$(x_i \leq y_i \ , \ 1 \leq i \leq n) \Rightarrow (f_i(x) \leq f_i(y) \ , \ 1 \leq i \leq n)$$

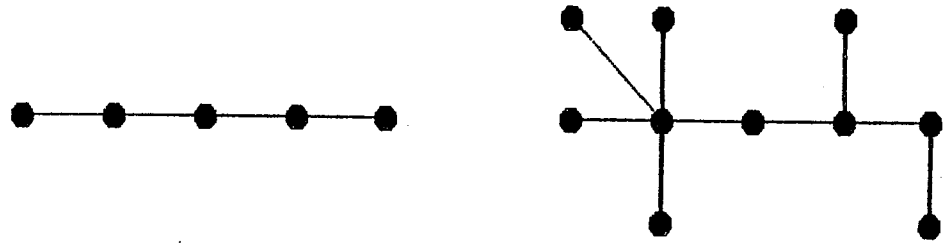
Enfin, le graphe de connexion $G_c(F)$ est défini par $G_c(F) = (\mathcal{S}, \Gamma)$, où $(j,i) \in \Gamma \Leftrightarrow j \in FN(i)$, et le graphe d'itération de F est le graphe orienté $G_i(F) = (Q^n, F)$, où tout sommet $x \in Q^n$ possède un seul arc sortant, (x,F(x)).

1.2. De $G_c(F)$ vers $G_i(F)$

L'approche usuelle consiste à établir des propriété sur le graphe d'itération $G_i(F)$ à partir d'hypothèses sur le graphe de connexion $G_c(F)$. En particulier, nous établissons la

Proposition 1 [T15] : si $G_c(F)$ est inclus dans une chenille, $G_i(F)$ est inclus dans un arbre.

Notons que pour établir cette proposition, l'hypothèse de monotonie joue un rôle clé, en particulier pour caractériser l'état des sommets pendants en régime permanent. On procède alors par récurrence sur le nombre de cellules non fixes. Exprimée en d'autres termes, la proposition 1 signifie que si $G_i(F)$ est inclus dans une chenille, alors toutes les cellules ont un cycle de longueur au plus 2 à la sortie du transitoire.



Une chaîne

Une chenille

Figure 1

Ce résultat ne peut pas s'étendre au cas général où $G_c(F)$ est inclus dans un arbre, mais on a la

Proposition 2 [T15] : si $G_c(F)$ est inclus dans un arbre, alors $G_i(F)$ est biparti.

Par contre, toujours dans le cas où $G_c(F)$ est inclus dans un arbre, la détermination de la longueur maximale des circuits élémentaires de $G_i(F)$ (en fonction du nombre n de cellules) est une question ouverte. Un résultat partiel est donnée par la

Proposition 3 [T15] : pour tout n, il existe une fonction monotone booléenne à n variables F telle que $G_c(F)$ est contenu dans un arbre, et $G_i(F)$ contient un circuit élémentaire de longueur O(log n).

1.3. De $G_i(F)$ vers $G_c(F)$

Nous abordons ici la démarche inverse, en montrant que certaines propriétés du graphe de connexion peuvent se déduire de l'observation du comportement dynamique du réseau. Le premier résultat (assez surprenant) est le suivant:

Proposition 4 [T15] : si le graphe d'itération $G_i(F)$ contient un circuit élémentaire de longueur $p \in [2,3]$, alors $G_c(F)$ contient un circuit élémentaire de longueur multiple de p

Le résultat est faux pour p = 4. En fait, on prouve dans [T15] que si le graphe d'itération contient un circuit élémentaire tel que toute cellule ait un cycle unimodal (de la forme aa...abb...baa...a), alors $G_c(F)$ contient un circuit élémentaire de longueur multiple de p. Pour p = 2 et p = 3 cette hypothèse est toujours vérifiée. Pour p = 4 nous exhibons dans [T15] un contre exemple où $G_i(F)$ contient un circuit élémentaire de longueur 4, alors que $G_c(F)$ ne contient pas de circuit de longueur supérieure à 2.

Nous obtenons également la proposition suivante

Proposition 5 [T15] : supposons que pour toute cellule i, f_i ne dépende pas de x_i: si $G_i(F)$ contient un circuit élémentaire de longueur strictement supérieure à 2, il en est de même pour $G_c(F)$.

De même, nous construisons dans [T15] un exemple où l'état d'une cellule i à l'instant t+1 peut dépendre de son état à l'instant t (soit $i \in FN(i)$), et mettant en défaut la propriété précédente.

1.4. Cas des fonctions de majorité généralisée

On considère ici un réseau $\mathcal{R}$ dont la fonction de transition F est définie comme suit:

> Pour tout $1 \leq i \leq n$, soit k(i) un entier arbitraire $\leq |FN(i)|$. Soit $t \geq 0$, on ordonne l'ensemble $\{Sj(t);\ j \in FN(i)\}$ en $x_1 \leq x_2 \leq \ldots \leq x_{|FN(i)|}$. On définit alors Si(t+1) par $Si(t+1) = x_{k(i)}$.

En d'autres termes, l'état de la cellule i à l'instant t+1 est la k(i)-ème valeur des états des cellules de son voisinage fonctionnel à l'instant t, ceux-ci étant totalement ordonnés dans l'ordre croissant.

De telles fonctions de transition généralisent les fonctions de majorité usuelles, où chaque cellule prend pour prochaine valeur la valeur médiane des cellules dont elle dépend. De telle fonctions permettent par exemple de modéliser l'évolution d'opinions en Sciences Sociales [37].

Supposant que $G_c(F)$ est symétrique ($i \in FN(j) \Leftrightarrow j \in FN(i)$), nous obtenons la

Proposition 6 [T26] : si $G_c(F)$ est symétrique, alors $G_l(F)$ est inclus dans un arbre.

Peut-être plus que le résultat lui-même, c'est la technique de démonstration qui est intéressante: nous utilisons une technique de morphisme pour nous ramener à une famille d'automates à seuil dont on sait caractériser le comportement dynamique par un théorème de [37].

1.5. Automates avec mémoire

On introduit maintenant des automates à mémoire: on suppose que l'état Si(t+1) d'une cellule i à l'instant t+1 peut dépendre non seulement des valeurs Sj(t) , $j \in FN(i)$, mais aussi de son état actuel Si(t) et des k précédents Si(t-1), Si(t-2), ... , Si(t-k). Le nombre k (supposé independant de i) est la longueur de la mémoire.

Nous commençons par la

Proposition 7 [T26] : soit $\mathcal{A}$ un automate cellulaire à mémoire de longueur 1. Si $G_c(F)$ est contenu dans une chaine, alors $G_l(F)$ est contenu dans un arbre.

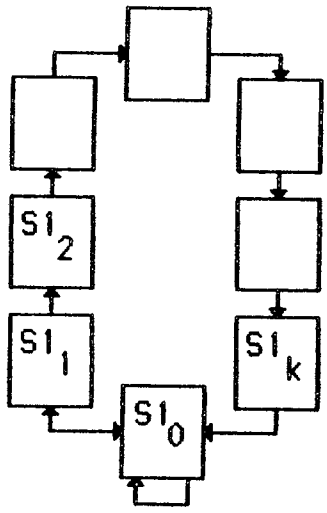
En fait, un automate $\mathcal{A}$ comprenant n cellules S1, ..., Sn à mémoire de longueur k peut être vu comme un automate sans mémoire comprenant (k+1)*n cellules, en remplaçant chaque cellule Si par (k+1) cellules Si_0, Si_1, ..., Si_k disposées en un circuit de longueur k+1. Chaque cellule Si_j, $j \geq 1$, agit simplement comme une cellule de transfert, qui copie la valeur précédente de la cellule Si_{j-1}, i.e. $Si_j(t) = Si_{j-1}(t-1)$ pour $t \geq 1$.

Cette interprétation permet de déduire directement la proposition 7 de la proposition 1.

Dans les deux propositions suivantes, nous illustrons la relation entre le degré maximal du graphe de connection $G_c(F)$ et la longueur maximale des circuits élémentaires du graphe d'itération $G_i(F)$. Nous nous restreignons au cas d'un automate comportant une seule cellule, celle-ci disposant d'une mémoire de longueur k.

Cas 1 : considèrons l'automate dont le graphe de connection est décrit figure 2. SI_1, SI_2, ..., SI_k sont des cellules de transfert, et pour tout $t \geq 1$, l'état $SI_0(t+1)$ est une fonction monotone de $SI_0(t)$, $SI_1(t) = SI_0(t-1)$ et $SI_k(t) = SI_0(t-k)$. Ainsi $FN(I_0) \subset \{ I_0, I_1, I_k \}$.

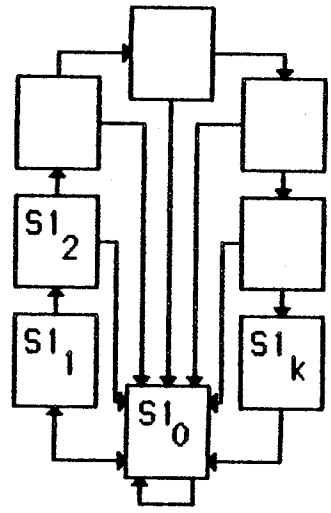
Proposition 8 [T26] : soit T la longueur d'un circuit élémentaire de $G_i(F)$. Alors T = 2 ou T divise k+1.



Automate à une seule cellule, avec mémoire de longueur 7

cas 1: SI_0 ne dépend que d'elle-même, SI_1 and SI_k

Figure 2



cas 2: SI_0 dépend de tous les SI_j, $0 \leq j \leq k$

Figure 3

Cas 2 : considérons maintenant l'automate de la figure 3: son graphe de connection est de degré maximal k+1. Cet automate a fait l'objet de nombreux travaux, dans le cas où la fonction de transition est linéaire sur le corps fini $Z_{/pZ}$, p premier [36] ou encore dans le cas où la fonction de transition est à seuil (modélisant alors une équation neuronale) [13]. On aborde ici le cas de fonctions de transition monotones, sans autres hypothèses. Pour les petites valeurs de k ($k \leq 5$), tous les circuits élémentaires de $G_i(F)$ sont de longueur $\leq k+1$. Mais cette observation est trompeuse, comme le montre la

Proposition 9 [T26] : on peut construire des fonctions de transition monotones telles que $G_i(F)$ contienne des circuits élémentaires dont la longueur croit exponentiellement avec k.

Remarquons enfin que le comportement dynamique de réseaux monotones généraux à mémoire semble beaucoup plus difficile à analyser (cf [T26]).

2. ALGORITHMES ET ARCHITECTURES SYSTOLIQUES

Le modèle systolique est présenté en détail au chapître 3: après une introduction générale (section 3.1), on propose de nombreux exemples d'architectures systoliques (section 3.2). La section 3.3 analyse les performances et l'intérêt du modèle, et se termine par un bref catalogue des circuits systoliques déjà réalisés. Enfin, la section 3.4, plus théorique, rappelle la méthodologie d' LEISERSON et SAXE [69], complétée par l'approche de type "divide-and-conquer" développée dans ROBERT et TCHUENTE [T16]. Nous renvoyons aux commentaires bibliographiques à la fin du chapître 3 pour les références à la littérature existante, mais nous avons inclu celles-ci dans la bibliographie générale qui conclut ce chapître.

Nous abordons alors directement la présentation des algorithmes originaux qui font l'objet de cette partie.

2.1. Algèbre matricielle

Cette section est centrée autour de la résolution systolique de systèmes linéaires d'équations. Ce problème est au centre du Calcul Scientifique traditionnel et malgré sa simplicité connait des développements toujours plux nombreux. Il en est pour les architectures systoliques comme pour les ordinateurs séquentiels, ainsi qu'en témoigne une abondante bibliographie [1] [9] [20] [35] [42] [50] [61] [63] [68] [73] [92].

Toutes les solutions présentées distinguent deux catégories de systèmes linéaires: ceux dont la matrice est dense, et ceux dont la matrice est à structure bande. Mais dans les deux cas, les solutions présentées reprennent l'approche séquentielle usuelle: triangularisation de la matrice bordée du second membre, puis résolution du système triangulaire obtenu. Le plus souvent dans l'étape de triangularisation, la décomposition LU ou QR de la matrice est également obtenue.

Commençons par les systèmes linéaires denses: les articles de GENTLEMAN et KUNG [35], AHMED et al. [1], proposent une architecture de $n^2/2$ cellules élémentaires IPS (pour "Inner product step", i.e. ces cellules sont capables de réaliser une multiplication/division éventuellement suivie d'une addition/soustraction) pour la triangularisation en temps 3n d'une matrice dense de taille n.

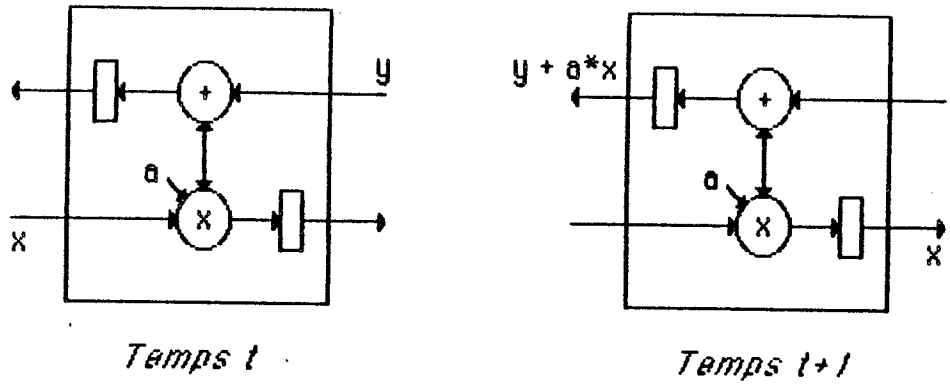


Figure 4 : Un processeur IPS

La résolution du système linéaire se fait en utilisant un autre réseau, celui de KUNG et LEISERSON [63], et nécessite un temps 2n. Cependant, il est nécessaire de stocker les résultats de la première étape dans la mémoire de la machine hôte et de les réordonner par diagonales avant de les réinjecter dans le deuxième réseau. Nous proposons au contraire dans [T17] d'utiliser directement la méthode de diagonalisation de Jordan pour résoudre le système linéaire en une seule étape; nous donnons deux implémentations différentes de cette méthode. La première reprend le réseau de [35] pour la phase de triangularisation et réactive les cellules pour amorcer la phase de diagonalisation (pipelinée avec la phase précédente):

Proposition 10 [T17] : on peut implanter la méthode de Jordan pour résoudre un système linéaire dense de n équations en temps 4n sur un réseau utilisant $n^2/2$ processeurs IPS.

Pour résoudre m systèmes de même matrice, le réseau s'étend directement en un réseau de taille $n^2/2 + mn$, pour un temps de résolution total de 4n+m. Nous proposons également une autre solution, qui a l'avantage de permettre une résolution pipelinée de m systèmes sur un réseau de taille inchangée:

Proposition 11 [T17] : on peut implanter la méthode de Jordan pour résoudre m systèmes linéaires denses de n équations (même matrice) en temps 4n+m sur un réseau utilisant n^2 processeurs IPS.

Cette dernière solution implémente directement la méthode de diagonalisation, en une seule phase. Pour inverser une matrice dense de taille n, elle est plus avantageuse que la précédente: elle nécessite le même temps 5n, mais n'utilise que n^2 processeurs (contre $3n^2/2$).

Passons maintenant à l'étude des sytèmes linéaires à matrice bande. Quand on manipule une matrice de largeur de bande w, la surface du réseau (qu'on peut mesurer approximativement par le nombre de processeurs) doit dépendre seulement de w, et non plus de la taille n de la matrice.

Ainsi, le réseau de décompostion LU de KUNG et LEISERSON [63] délivre en sortie les facteurs triangulaires L et U d'une matrice de taille n à structure bande de largeur w = p + q + 1 (p-1 sur-diagonales et q-1 sous-diagonales) en temps 3n + min(p,q) sur un réseau hexagonal comportant pq processeurs IPS.

Nous passons en revue dans [T18] les architectures systoliques permettant de trianguliariser une matrice bande, en distinguant les divers schémas d'élimination possibles. En particulier, nous expliquons pourquoi un schéma d'élimination par colonnes, avec pivots choisis sur la diagonale, ne peut atteindre qu'une efficacité e=1/3 (les cellules du réseau solution sont activées seulement une fois sur 3). Le temps d'exécution est alors en 3n + O(w), comme pour le réseau de KUNG et LEISERSON.

Pour améliorer l'efficacité, il faut recourir à des schémas d'élimination par diagonales successives, la sous-diagonale i étant utilisée pour éliminer la sous-diagonale i+1, en commençant par i=q-1. On obtient alors des réseaux d'efficacité e=1/2 comme le réseau de HELLER et IPSEN [42]. Malheureusement, des considérations numériques de stabilité imposent dans ce cas l'emploi de matrices de prémultiplication orthogonales (matrices de rotation de Givens). La détermination de chaque transformation élémentaire nécessite le calcul d'une racine carrée, et le calcul des coefficients d'une rotation dans le plan (i,j) est nettement plus long que la mise à jour des éléments des lignes i et j: comme la synchronisation globale s'effectue au rythme du processeur le plus lent, le handicap est considérable. Les remèdes proposés vont de la conception de processeurs de type Cordic, suggérés par AHMED et al [1], à l'emploi de méthodes d'élimination sans racine carrée plus sophistiqués (IPSEN [50]).

Nous proposons dans [T18] un réseau de triangularisation d'efficacité e=1/2 et dont les processeurs n'exécutent que les 4 opérations élémentaires. Ce réseau est basé sur la méthode de décomposition LU par blocs d'ordre 2:

Théorème 12 [T18] : soit A une matrice n x n de largeur de bande w = p+q-1. On peut calculer la décomposition LU par blocs de A en temps 2n + min(p,q) sur un réseau n'utilisant que pq processeurs IPS.

Qui plus est, l'utilisation d'une méthode par blocs 2 x 2 a l'avantage de conduire à une décomposition A = L.D.U, où
- L est triangulaire inférieure à diagonale unité et de première sous-diagonale nulle
- D est diagonale par blocs 2 x 2
- U est triangulaire inférieure à diagonale unité et de première sous-diagonale nulle

La résolution du système Ax = b est conduite en 3 étapes:
(1) résolution de U z = b
(2) résolution de D y = z
(3) résolution de L x = y

La résolution de (2) est immédiate (n/2 systèmes de 2 équations à 2 inconnues). La résolution de (3) est similaire à celle de (1), on se ramène donc à ce dernier système. Le réseau systolique pour la résolution d'un système triangulaire bande proposé par KUNG et LEISERSON [63] comporte q processeurs IPS et nécessite 2n+q unités de temps pour délivrer toutes les composantes du vecteur solution.

Nous montrons dans [T18] comment tirer parti de la présence d'une première sous-diagonale nulle pour diminuer de moitié ce temps d'exécution sans augmenter la taille de l'architecture solution:

Proposition 13 [T18] : on peut résoudre le système (3) en n + $\lceil q/2 \rceil$ + 1 sur un réeau de q processeurs IPS.

Nous généralisons ce résultat dans [T27] pour la résolution d'un système triangulaire quelconque. La démarche comprend maintenant 4 étapes:

soit L une matrice bande triangulaire inférieure de largeur de bande w et soit (S) L x = b un système à résoudre

(i) soit D la diagonale de L; effectuer le changement de variable y = Dx pour transformer (S) en (S') : B y = b
où B est triangulaire inférieure de largeur de bande w et de diagonale unité

(ii) effectuer le changement de variable $y = Q\,z$ pour transformer (S') en (S") : $C\,z = b$
où C est triangulaire inférieure de largeur de bande w+1, de diagonale unité et de première sous-diagonale nulle

La matrice Q est une matrice comportant seulement une diagonale unité et une première sous-diagonale non nulle.

(iii) résoudre le système (S") en tirant parti de la structure de C (proposition 13)

(iv) calculer la solution du système (S) $x = D^{-1} P\,z$

La concaténation des sous-réseaux systoliques réalisant chacune de ses étapes conduit au

Théorème 14 [T27] : si L est une matrice n x n triangulaire inférieure de largeur de bande w, le système $L\,x = b$ peut être résolu à l'aide du schéma précédent en temps $n + \max(\lceil w/2 \rceil + 2, 5)$ sur un réseau de $3w+1$ processeurs IPS. Ce réseau est complètement modulaire et son fan-out est égal à 2.

La modularité du réseau assure qu'il suffit d'ajouter des cellules IPS à la frontière du réseau pour la conception d'une architecture résolvant un problème de taille supérieure. Au contraire, la solution directe qui consiste à reporter dans toutes les cellules du réseau chaque nouvelle composante calculée exigerait que la cellule de division soit connectée à toutes les autres processeurs, ce qui rendrait impossible toute extension.

Le fan-out f d'un circuit est le nombre maximum de copies d'une variable donnée qui sont délivrées par un même processeur en entrées des autres cellules. Il est très important de le minimiser pour une réalisation physique. De même, la solution directe aurait un fan-out f égal à la largeur de bande w de la matrice, ce qui interdit toute réalisation dès que w n'est plus très petit.

On trouvera dans [T27] une généralisation du schéma d'élimination précédent pour des matrices dont plusieurs premières sous-diagonales sont nulles. Outre leur intérêt théorique (relatif à la complexité de la solution en circuit VLSI d'un système triangulaire), de tels schémas

peuvent être envisagés dans leur version par blocs pour certaines matrices issues de problèmes de discrétisation d'équations aux dérivées partielles.

L'idée sous-jacente au schéma d'élimination en 4 étapes peut être reprise pour d'autres problèmes de calculs de récurrences linéaires. Il s'agit toujours d'améliorer l'efficacité des architectures existantes. En fait, nous formaliserons plus loin cette approche dans un cadre général.

Pour l'instant, nous nous intéressons à l'évaluation d'une convolution récursive (ou filtrage à réponse impulsionnelle infinie en traitement du signal). Le problème s'énonce comme suit:

étant donnée une suite infinie $x_1, x_2, x_3, \ldots$ de données,

et pour des valeurs initiales $y_1, y_2, \ldots, y_h$,

calculer pour tout $i \geq \text{Max}(k, h+1)$ la valeur de

$$y_i = a_1 * x_i + a_2 * x_{i-1} + a_3 * x_{i-2} + \ldots + a_k * x_{i-k+1} + r_1 * y_{i-1} + r_2 * y_{i-2} + r_3 * y_{i-3} + \ldots + r_h * y_{i-h} \qquad (*)$$

où les poids $a_1, a_2, \ldots, a_k$ et $r_1, r_2, \ldots, r_h$ sont des coefficients fixés.

KUNG [59] propose l'architecture de la figure 5 pour résoudre ce problème. Ce réseau comporte k+h cellules de calcul et une cellule de retard:

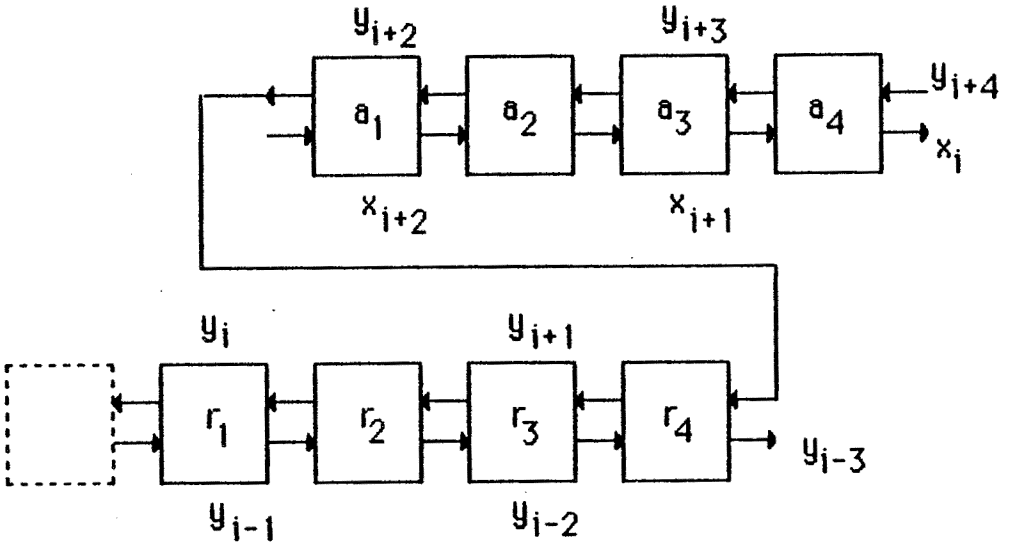


Figure 5 : Convolution récursive générale

Le réseau est d'efficacité e = 1/2: après un temps d'initialisation, un nouvel y_i est délivré en sortie une pulsation sur deux.

Pour obtenir une nouvelle solution d'efficacité e = 1, nous réécrivons la relation en faisant disparaître le terme en y_{i-1}. Pour cela, nous remplaçons dans (*) y_{i-1} par sa valeur

$$y_{i-1} = a_1 * x_{i-1} + \dots + a_k * x_{i-k} + r_1 * y_{i-2} + \dots + r_h * y_{i-h-1}$$

pour obtenir

$$y_i = a'_1 * x_i + a'_2 * x_{i-1} + a'_3 * x_{i-2} + \dots + a'_k * x_{i-k+1} + a'_{k+1} * x_{i-k} + r'_2 * y_{i-2} + r'_3 * y_{i-3} + \dots + r'_h * y_{i-h} + r'_{h+1} * y_{i-h-1} \qquad (**)$$

où (i) $a'_1 = a_1$; $a'_j = a_j + r_1 * a_{j-1}$ pour $2 \leq j \leq k$; $a'_{k+1} = r_1 * a_k$

(ii) $r'_j = r_j + r_1 * r_{j-1}$ pour $2 \leq j \leq k$; $r'_{k+1} = r_1 * r_k$

Avec cette nouvelle formulation, y_{i+1} n'a plus besoin de rencontrer y_{i-1}; cette transformation joue le même rôle que la création d'une première sous-diagonale nulle dans le problème précédent. Nous construisons à partir des équations (**) dans [T25] un réseau calculant un nouvel y_i toutes les pulsations (après initialisation), et ce sans augmenter le nombre de processeurs IPS nécessaires:

Théorème 15 [T25] : le problème de la convolution récursive peut être résolu sur un réseau de h+k+1 processeurs IPS avec une efficacité 1, i.e. un nouvel y_i est délivré en sortie du réseau toutes les pulsations.

2.2. Problèmes de mots

On aborde dans cette section la conception d'architectures intégrées spécialisées pour le traitement de chaînes de caractères. Les chaînes de caractères constituent un type de données rencontré très fréquemment en informatique, et tout laisse à penser que leur utilisation est encore appelée à croître dans les ordinateurs de la cinquième génération. Les chaînes de caractères sont bien sûr au centre des systèmes de traitement de texte, qui fournissent de nombreuses facilités pour leur manipulation. Plus généralement, les manipulations de chaînes de caractères jouent un rôle dans la plupart des environnements informatiques, et les développements récents soulignent l'importance des chaînes de caractères lisibles par l'homme comme type de données naturel [17].

Une manipulation performante des chaînes de caractères est à la base de telles opérations. Cependant, pour des problèmes de grande taille, les systèmes informatiques existants ne sont pas à même de traiter ce type de données de façon efficace, à cause du grand nombre d'accès à la mémoire centrale et de calculs à l'intérieur de boucles imbriquées que requiert ce genre de manipulations: pour améliorer la puissance et l'efficacité des opérations sur les chaines de caractères, il faut recourir à des processeurs spécialisés pour une implantation sous forme de circuit intégré des algorithmes les plus utilisés. L'emploi d'architectures systoliques pour le traitement de chaînes de caractères s'est révélé être un outil adapté à la réalisation de tels processeurs spécialisés [5] [16] [17] [29] [69] [90].

2.2.1. Approche "divide-and-conquer" et convolution généralisée

LEISERSON et SAXE [69] présentent une démarche constructive pour définir une architecture systolique à partir d'une architecture comprenant de la logique combinatoire se propageant en cascade (nous décrivons cette démarche au chapître 3). Le seul inconvénient de leur méthodologie est qu'elle débouche le plus souvent sur des réseaux où chaque cellule n'est activée qu'un top d'horloge sur k, k>=2, ce qui limite par un facteur k le degré de parallélisme obtenu et augmente le temps d'exécution.

Plusieurs solutions ont été proposées pour remédier à cette situation: KUNG et LEISERSON [63] proposent de fondre k cellules adjacentes en une seule, ce qui diminue la surface par un facteur k mais n'améliore pas le temps d'exécution. Une autre solution consisterait à entrelacer la résolution de k problèmes indépendants (de même nature) sur le réseau, mais ceci n'accélère pas la résolution de chaque problème.

Nous présentons ici une approche basée sur la technique classique "divide-and-conquer" qui peut s'énoncer ainsi [45]: étant donné un problème à résoudre, diviser ce problème en k sous-problèmes que l'on résoud séparément, puis combiner la solution de ces k sous-problèmes pour reconstruire une solution du problème initial. Cette technique est bien connue pour les algorithmes usuels, et nous allons illustrer son application aux architectures systoliques (pour augmenter la vitesse d'exécution) sur trois exemples issus de problèmes de mots: le calcul du nombre d'occurences d'un mot dans une chaîne, la reconnaissance des palindromes, et la reconnaissance des carrés.

Ces trois problèmes sont unifiés dans un formalisme unique, celui de convolution généralisée:

Soit E un ensemble non vide, (B,*) un monoide commutatif, et T une loi de composition externe sur E à valeurs dans B:

$$E \times E \dashrightarrow B$$
$$(e,e') \dashrightarrow e\ T\ e'$$

T est appele opérateur de base.
La convolution généralisée se définit comme suit:

Données :
- une suite $A = (a_i;\ i \geq 1)$ d'éléments de E, appelés symboles d'entrée
- une suite d'entiers $(n(i)\ ;i \geq 1)$
- une suite de vecteurs $W = (w^{(i)} \in E^{n(i)};\ i \geq 1)$; un vecteur $w^{(i)}$ est appelé vecteur de référence, ses composantes seront notées $w^{(i)} = (w_{i,1}\ w_{i,2} \ldots w_{i,n(i)})$

Problème :
calcul de la suite $(y_i;\ i \geq 1)$ définie par

$$y_i = (a_i\ T\ w_{i,1}) * (a_{i-1}\ T\ w_{i,2}) * \ldots * (a_{i-n(i)+1}\ T\ w_{i,n(i)})$$

La convolution classique

$$y_i = (a_i\ T\ w_1) * (a_{i-1}\ T\ w_2) * \ldots * (a_{i-k+1}\ T\ w_k) \quad ; i \geq k$$

correspond au cas particulier où la suite W est constante et égale à $(w_1, w_2, \ldots, w_k)$.

L'approche "divide-and-conquer" consiste à séparer les calculs en deux parties:

$$y_{i,1} = (a_i \quad T\, w_{i,1}) * (a_{i-2}\, T\, w_{i,3}) * (a_{i-4}\, T\, w_{i,5}) * \ldots$$

$$y_{i,2} = (a_{i-1}\, T\, w_{i,2}) * (a_{i-3}\, T\, w_{i,4}) * (a_{i-5}\, T\, w_{i,6}) * \ldots$$

On construit alors un réseau R comportant:

- un sous-réseau R1 pour le calcul de $y_{i,1}$
- un sous-réseau R2 pour le calcul de $y_{i,2}$
- une cellule qui reçoit $y_{i,1}$, $y_{i,2}$ en entrée et délivre en sortie

$$y_i = y_{i,1} * y_{i,2}$$

La raison de ce découpage est que, dans les réseaux systoliques proposés pour résoudre divers problèmes de convolution (au sens large où nous l'avons définie), les variables a_i et y_j circulent dans des directions opposées. Or on vérifie aisément (cf figure 6) que, dans un réseau linéaire, une variable y_i ne peut pas rencontrer deux éléments a_j, a_{j+1} qui se suivent. En conséquence, toute solution basée sur un réseau linéaire et où les variables a_j, y_i circulent dans des directions opposées, est d'efficacité 1/2, i.e. deux variables consécutives y_i, y_{i+1} sont séparées par deux tops d'horloge.

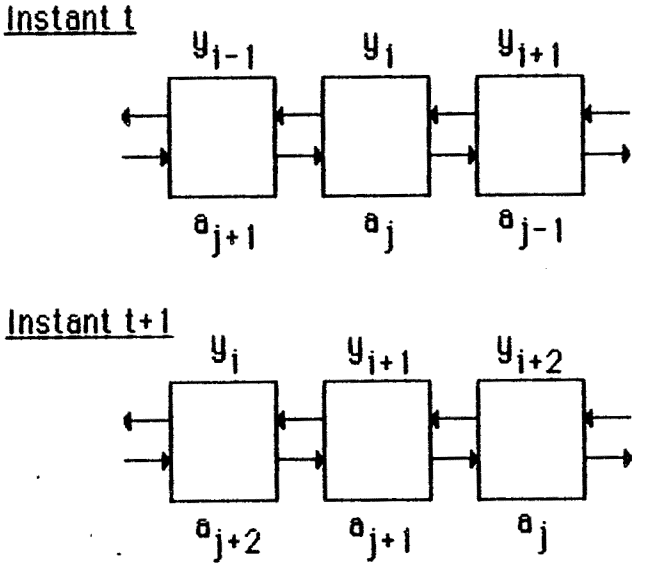


Figure 6 : y_i rencontre a_j et a_{j+2}, mais non a_{j+1}

L'approche "divide-and-conquer" permet de procéder ainsi:

- calcul de $y_{i,1} = (a_i \quad T\ w_{i,1}) * (a_{i-2}\ T\ w_{i,3}) * (a_{i-4}\ T\ w_{i,5}) * \ldots$
sur un réseau R1 d'efficacité 1
- calcul de $y_{i,2} = (a_{i-1}\ T\ w_{i,2}) * (a_{i-3}\ T\ w_{i,4}) * (a_{i-5}\ T\ w_{i,6}) * \ldots$
sur un réseau R2 d'efficacité 1
- calcul de $y_i = y_{i,1} * y_{i,2}$ avec une efficacité 1

Nous étudions les cas suivants qui sont de difficulté croissante :

• cas 1 : convolution avec une référence constante
on a alors $w^{(i)} = (w_1, w_2, \ldots, w_k)$ pour tout i.

Ce cas correspond au problème de détection des occurences d'un mot dans une chaîne de caractères: soit $A = a_1 \ldots a_N$ et $W = w_n \ldots w_2 w_1$, $n<N$, deux mots sur un alphabet S. Une occurence de W dans A est une sous-chaîne $a_i a_{i+1} \ldots a_{i+n-1}$ qui est égale à W. Le problème posé ici est celui de la détection des occurences de W dans A. Par exemple, si A = abababa et W = aba, alors $a_1a_2a_3$, $a_3a_4a_5$, $a_5a_6a_7$ sont des occurences de W dans A.

Ce problème est équivalent au calcul des variables booléennes

$$y_i = [a_i = w_1] \cdot [a_{i-1} = w_2] \cdot \ldots \cdot [a_{i-n+1} = w_n] \text{ pour } i \geq n$$

(et $y_i = 0$ pour $i<n$). L'approche "divide-and-conquer" conduit à la

Proposition 16 [T16] : le problème du calcul des occurences d'un mot dans une chaîne peut être résolu en efficacité 1 sur un réseau de n+1 processeurs élémentaires.

Notons que la solution peut s'étendre au calcul du nombre d'occurences disjointes de W dans A: deux occurences $a_i a_{i+1} \ldots a_{i+n-1}$, $a_j a_{j+1} \ldots a_{j+n-1}$ sont disjointes si elle n'ont aucune position commune, i.e. les intervalles [i,i+n-1] et [j,j+n-1] sont disjoints. Reprenant l'exemple avec A = abababa et W = aba, le nombre d'occurences disjointes de W dans A est 2. Le problème est alors analogue dans sa formulation à la convolution récursive de la section précédente, mais la solution apportée [T16] est de nature différente, puisque des techniques de réécriture n'auraient ici aucun sens.

• cas 2 : la référence s'allonge par la droite au cours du temps
on a alors $w^{(i)} = (w_1, w_2, \ldots, w_p)$ où $p = \lfloor i/2 \rfloor$: $w^{(i)}$ correspond aux p premiers termes d'une suite fixée $(w_j;\ j \geq 1)$. Cette situation où $w^{(i-1)}$ est préfixe de $w^{(i)}$ s'illustre par le problème de la reconnaissance des palindromes.

Une chaîne $a_1 a_2 \ldots a_n$ est un palindrome si elle est symétrique, i.e. si $a_i = a_{n-i+1}$ pour tout i. Les palindromes constituent un langage hors contexte très simple, et leur reconnaissance est un problème classique. BARZDIN [6] a montré que la reconnaissance d'un palindrome de taille n sur une machine de Turing ayant une seule tête de lecture-écriture nécessite un temps de l'ordre de n^2. Plus tard, SMITH [87] a exhibé un automate cellulaire uniforme de dimension un, capable de reconnaître un palindrome de longueur n en temps linéaire $t = \lceil n/2 \rceil$; le déploiement du diagramme espace-temps de cet algorithme donne la solution proposée récemment par CULIK et al. [15]. Cependant, le modèle de SMITH n'est pas adapté au pipeline et ne permet donc pas, pour une suite $(a_i;\ i \geq 1)$ donnée, le calcul en temps réel de la suite booléenne $(y_i;\ i \geq 1)$ définie par $y_i = 1$ si $a_1 a_2 \ldots a_i$ est un palindrome, et $y_i = 0$ sinon. Ce dernier problème a été résolu pour la première fois par COLE [12]. LEISERSON et SAXE [69] ont proposé une solution beaucoup plus simple mais dans laquelle deux symboles a_j consécutifs sont séparés par deux tops d'horloge, et ce pour les raisons que nous avons exposées plus haut.

Grâce à l'approche "divide-and-conquer", nous obtenons la

Proposition 17 [T16] : le problème de la reconnaissance des palindromes peut être résolu en efficacité 1 sur un réseau linéaire de processeurs élémentaires.

• cas 3 : la référence s'allonge par la droite au cours du temps
on a alors $w^{(i)} = (w_p, w_{p-1}, \ldots, w_1)$ où $i = 2p$: $w^{(i)}$ correspond aux p premiers termes, pris dans l'ordre inverse, d'une suite fixée $(w_j;\ j \geq 1)$. Cette situation où $w^{(i-1)}$ est suffixe de $w^{(i)}$ s'illustre par le problème de la reconnaissance des carrés.

Une chaîne $a_1 a_2 \ldots a_n$, $n=2p$, est un carré si $a_i = a_{i+p}$ pour $1 \leq i \leq p$. On doit donc évaluer la suite booléenne $y_i = 1$ si $a_1 a_2 \ldots a_i$ est un carré, et $y_i = 0$ sinon.

Ici, l'approche "divide-and-conquer" exige que la structure hôte fournisse les données de la manière suivante:
- à tout instant $t=2i+1$, elle délivre le symbole d'entrée a_{2i+1}
- à tout instant $t=2i$, elle délivre le symbole d'entrée a_{2i} et le symbole de référence a_i.

Il faut alors adjoindre au réseau solution un réseau de pré-traitement qui reçoit en entrée a_i à l'instant i et le délivre à la première cellule à l'instant $2i$. Ce réseau est implicite dans la construction proposée par COLE [12] pour la reconnaissance des carrés en temps réel.

Nous obtenons finalement la

Proposition 18 [T16] : le problème de la reconnaissance des carrés peut être résolu en efficacité 1 sur un réseau linéaire de processeurs élémentaires.

La méthode de systolisation de LEISERSON et SAXE [69], complétée par une transformation de type "divide-and-conquer" à son dernier stade, se révèle être un outil très efficace: ainsi pour le problème de convolution généralisée est-on conduit à une solution temps réel où le circuit reçoit une nouvelle donnée et délivre un nouveau résultat tous les tops d'horloge.

D'un point de vue méthodologique, on dispose d'une démarche qui, partant d'une solution évidente mais non adaptée au calcul en temps réel, conduit à un circuit systolique de performance maximale.

Notons enfin que, pour la reconnaissance des palindromes et des carrés, on peut retrouver les réseaux proposés par COLE [12] à partir des réseaux présentés ici, en effectuant des regroupements entre des cellules voisines. Notre approche cumule ainsi deux avantages:
- elle évite les preuves compliquées de COLE [12] grâce à l'utilisation du formalisme de LEISERSON et SAXE [69]
- elle conduit à une solution d'efficacité maximale, grâce à l'emploi d'une approche "divide-and-conquer", et remédie ainsi au manque de performance des solutions obtenues dans [69].

2.2.2. Plus longue sous-suite commune à deux chaînes

La détermination d'une plus longue sous-suite commune (PLSC) à deux mots A et B sur un alphabet S est un problème que l'on rencontre dans de nombreux domaines d'application, par exemples en traitement de données pour identifier deux fichiers qui sont semblables, ou en génétique pour mesurer le degré de corrélation de deux molécules [43] [76].

Une sous-suite d'un mot A est un mot obtenu en supprimant certains éléments (non nécessairement consécutifs). Une PLSC à deux mots A et B est une sous-suite commune à A et B, de longueur maximale. Par exemple si A = abcdbb et B = cbacbaab, alors A' = bcbb et A" = acbb sont les deux PLSC.

Tous les algorithmes connus exigent un temps de calcul proportionnel (pour le plus mauvais cas) au produit de la longueur des mots A et B [43] [48] [76], et ce même s'il s'agit de calculer la longueur d'une PLSC, et non pas d'exhiber une telle sous-suite. Notre but est de résoudre le problème en temps linéaire (i.e. proportionnel à la somme des longueurs des mots) à l'aide d'une architecture systolique.

Commençons par le problème plus simple de la détermination de la longueur p d'une PLSC de A = A(1)A(2)...A(m) et B = B(1)B(2)...B(n) (on suppose $m \leq n$ sans perte de généralité):

Lemme 19 [T6] [T23] : notant L(i,j) la longueur d'une PLSC de A(1)A(2)...A(i) et B(1)B(2)...B(j), avec L(i,0) = L(0,j) = 0 pour $0 \leq i \leq m$, $0 \leq j \leq n$, on a

$$L(i+1,j+1) = \mathrm{Max}\ (\ L(i,j+1)\ ,\ L(i+1,j)\ ,\ L(i,j) + [A(i+1)=B(j+1)]\ )$$

où [.=.] vaut 1 si l'égalité est vérifiée, et 0 sinon.

Ce lemme constitue la formulation récursive d'un algorithme séquentiel permettant le calcul en temps O(n*m) de p = L(m,n). Pour l'implantation parallèle de cet algorithme, nous considérons un réseau de n processeurs élémentaires identiques, conduisant au

Théorème 20 [T6] [T23] : soient A et B deux mots de longueur respectives m et n. La longueur d'une PLSC de A(1)A(2)...A(i) et B peut être calculée en temps $(i+n)*\tau$ sur un réseau systolique de n processeurs élémentaires de temps de cycle τ, pour tout $i \leq m$. En particulier, la longueur d'une PLSC de A et B est calculée en temps O(m+n).

Le problème de la détermination d'une PLSC à A et B est plus diificile. Nous obtenons la

Proposition 21 [T6] [T23] : soient A et B deux mots de longueur respectives m et n. La détermination d'une PLSC de A et B peut être faite en temps O(m+n) sur une architecture parallèle spécialisée de surface totale O(m*n).

L'idée-clé est d'adjoindre à chaque processeur j une pile systolique [40] dont le temps d'insertion/suppression est constant, ce qui assure que le temps de cycle de chaque processeur reste indépendant de m et n. Nous renvoyons à [T23] pour une construction détaillée du réseau solution.

2.2.3. Reconnaissance de langages

La reconnaissance des langages est un problème fondamental de l'informatique. Par exemple, la compilation qui consiste à traduire un langage évolué en langage machine, passe par une première phase au cours de laquelle le compilateur C vérifie si le programme source fourni par l'utilisateur est syntaxiquement correct, c'est-à-dire appartient au langage L(C) qui lui est associé.

Nous proposons ici des réseaux systoliques unidimensionnels pour la reconnaissance en temps réel des langages de la forme

$$L = \{ f_{n-1}(w)f_{n-2}(w)...f_1(w)w \; ; \; w \in A^+ \},$$

où A est un alphabet fini, A^+ l'ensemble des mots non-vides sur A, et pour tout i, $1 \leq i \leq n-1$, f_i est soit la fonction identité $f_i(w) = w$, soit la fonction miroir $f_i(w) = f_i(w_1 w_2 ... w_n) = w_n w_{n-1} ... w_1 = w^R$.

Ces réseaux généralisent ceux obtenus par Cole [12] pour la reconnaissance des palindromes (n=2, f_1=Rev) et des carrés (n=2, f_1=Id), et dont nous avons proposé des versions plus constructives (paragraphe précédent).

Dans le modèle de reconnaissance en temps réel introduit par Cole [12], on suppose que le réseau lit en entrée un nouveau symbole a_i tous les tops d'horloge; si le temps de cycle de la cellule de base est pris comme unité de temps. alors le réseau doit délivrer en sortie à tout instant t = i+k (k indépendant de i), la réponse booléenne y_i correspondant au mot $a_1 a_2 ... a_i$, c'est-à-dire $y_i = 1$ si $a_1 a_2 ... a_i \in L$ et $y_i = 0$ sinon.

La construction d'une architecture solution est très technique. Elle est basée sur l'approche "divide-and-conquer" et sur un partitionnement du réseau en sous-unités indépendantes. La structure générale du réseau est schématisée sur la figure 7. Nous obtenons le

Théorème 22 [T9] [T11] : Pour tout $n \geq 2$, pour tout alphabet fini A et pour tout ensemble de n-1 fonctions $f_1, f_2, \ldots, f_{n-1}$ égales soit à l'identité Id soit à la fonction miroir Rev, le langage $L = \{ f_{n-1}(w)\ldots f_2(w) f_1(w) w \; ; \; w \in A^+ \}$ peut être reconnu en temps réel par un réseau systolique périodique unidimensionnel, dont la période croit exponentiellement avec n.

Plus précisément, soit $J = \{r \; / \; 1 \leq r \leq n-1$ et $f_r = Id\}$ and $R = \{ r \; / \; 1 \leq r \leq n-1$ et $f_r = Rev\}$; la période T du réseau est

$T = ppcm \{ ppcm (\mu(n-r) ; r \in J) , ppcm(\mu(n-r-1) ; r \in R) \}$
où $\mu(x) = x/2$ si x est pair et $\mu(x) = x$ si x est impair.

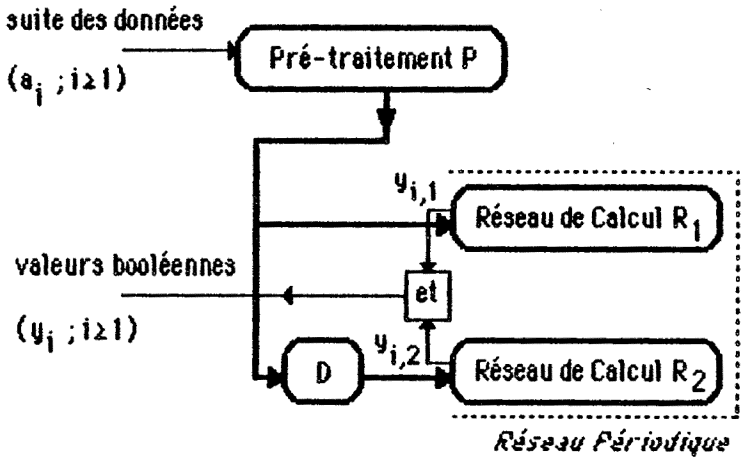


Figure 7 : Structure générale du réseau

Il est important de noter que la périodicité n'est relative qu'à la configuration des cellules de retard: les cellules de calcul sont toutes identiques, et le réseau peut être réalisé physiquement en dupliquant la structure d'une cellule de calcul élémentaire. Qui plus est, cette cellule de calcul peut elle-même s'obtenir en dupliquant deux réseaux combinatoires, à savoir le réseau de comparaison de symboles et le méchanisme de stockage dans les cellules; ceci permet d'envisager le dessin de cellules

programmables, ce qui nous permettrait de traiter une large classe de problèmes de reconnaissance de langages.

Enfin, la structure obtenue est à notre connaissance le premier exemple d'architecture systolique périodique. La périodicité pourrait être éliminée en regroupant plusieurs cellules adjacentes, mais le caractère exponentiel du réseau rend cette alternative illusoire.

2.2.4. Problème du chemin algébrique

Dans un article récent, ROTE [80] introduit l'algorithme d'élimination de Gauss-Jordan pour le problème général du plus court chemin algébrique (problème APP, mis pour Algebraic Path Problem). Ce concept de chemin algébrique formalise et unifie des algorithmes issus de domaines variés, comme le calcul de l'inverse d'une matrice réelle, la détermination de la fermeture réflexive et transitive d'une relation binaire, et le problème des plus courtes distances dans un graphe pondéré.

Le problème APP est défini comme suit [100]: étant donné un graphe pondéré G = (V,E,w), où V est l'ensemble fini des sommets, E l'ensemble des arcs, et w : E -> H une application à poids pris dans un semi-anneau (H,(+),(x)) de zéro (0) et d'unité (1), trouver pour toutes les paires de sommets (i,j) les valeurs

$$d_{ij} = \bigoplus_{p \in M_{ij}} w(p),$$

où M_{ij} représente l'ensemble de tous les chemins de i à j.

On associe au graphe G = (V,E,w) la matrice d'ordre n, $A = (a_{ij})$, où $a_{ij} = w(i,j)$ si $(i,j) \in E$ et $a_{ij} = 0$ sinon. On note $M_{ij}^{(k)}$ l'ensemble de tous les chemins de i à j qui ne contiennent comme sommets intermédiaires que des sommets d'indice x avec $1 \leq x \leq k$. Ainsi

$$a_{ij}^{(k)} = \bigoplus_{p \in M_{ij}^{(k)}} w(p),$$

est égal aux valeurs successives de a_{ij} que nous voulons calculer, partant de la valeur initiale $a_{ij}^{(0)} = a_{ij}$ jusqu'à $a_{ij}^{(n)} = d_{ij}$.

ROTE [80] présente un algorithme général pour la résolution d'une instance quelconque du problème APP. Il propose alors une implémentation de cet algorithme sur un réseau systolique hexagonal de $(n+1)^2$ processeurs

avec un temps de résolution égal à 7n-2, l'unité étant le temps nécessaire pour réaliser une multiplication suivie d'une addition dans l'algèbre sous-jacente.

Nous montrons dans [T29] comment implémenter de manière plus performante cet algorithme sur un réseau orthogonal:

Theorème 23 [T29] : toute instance du problème APP peut être résolue sur un réseau orthogonal de $(n+1)^2$ processeurs en 5n-2 unités de temps.

La valeur de 5n-2 est presque optimale: ROTE [80] montre que 5n-3 unités est le minimum possible, sous l'hypothèse que chacune des n^2 données est utilisée au plus une fois à chaque pas (aucune duplication autorisée). De plus, notre solution est uni-directionnelle (pas de circuit dans le cheminement des données), et donc plus avantageuse du point de vue de la résistance aux pannes, des possibilités de pipeline multi-niveaux et des techniques de décomposition en sous-problèmes [61].

Enfin, le réseau que nous proposons est très similaire à celui utilisé pour l'implémentation directe de la méthode de Jordan [T17]. Mais dans ce cas particulier du problème APP, nous avions tiré parti de la commutativité des opérateurs (X) et * pour exécuter l'algorithme d'élimination avec un autre ordonnancement que celui donné par ROTE [80] pour le cas général.

2.3. VLSI et Calcul Formel

Nous abordons dans cette section la conception d'architectures systoliques pour le Calcul Formel. A l'origine de ce travail, une constatation: les expressions manipulées dans les systèmes de Calcul Formel sont complexes · des fractions rationnelles à plusieurs indéterminées, par exemple -, et de ce fait les algorithmes de traitement souffrent d'une relative lenteur, même quand ils sont implantés sur les plus puissantes unités de calcul.

Peut-on espérer augmenter le rendement ou la capacité de ces systèmes en "cablant" certaines fonctions de base ? Il ne s'agit pas de transformer totalement les systèmes de Calcul Formel, mais de leur adjoindre des processeurs intégrés spécialisés performants afin d'augmenter la vitesse d'exécution (compte tenu de la nette supériorité du matériel sur le logiciel).

Une première proposition pour la conception de tels processeurs se trouve dans l'article de KUNG [58], qui propose des architectures systoliques pour la multiplication et la division euclidienne de deux polynômes à coefficients entiers. Le réseau de multiplication a été récemment repris dans [65] par LAM et MOSTOW, qui en présentent une nouvelle version. Les deux versions sont d'efficacité e = 1/2, et l'approche "divide-and-conquer" nous permet d'obtenir une solution d'efficacité 1:

Proposition 24 [T5] : le produit de deux polynômes de degré respectifs m et n ($m \leq n$) peut être calculé sur une architecture systolique de m processeurs IPS en temps $\lceil m/2 \rceil + n + 2$.

Notre architecture souffre cependant de la même lacune que les deux réseaux précédents: elle ne prend pas en compte les problèmes de taille des coefficients, problèmes qu'il est essentiel de maitriser en Calcul Formel: d'une manière générale, les polynômes manipulés sont souvent lacunaires, et leurs coefficients ne sont pas du même ordre de grandeur. Comme les dispositifs matériels sont de taille figée, il faut recourir à des méthodes plus sophistiquées pour garantir le non-dépassement de capacité et la pleine utilisation des multiplieurs.

Les méthodes modulaires répondent à notre objectif. L'approche est la suivante: pour résoudre un problème initial (I), on donne une solution intégrée uniquement pour l'étape modulaire (E). Réduction aux divers moduli et reconstruction via le théorème des Restes Chinois sont conduites

classiquement. Pour le produit de deux polynômes, on a le schéma de la figure 8.

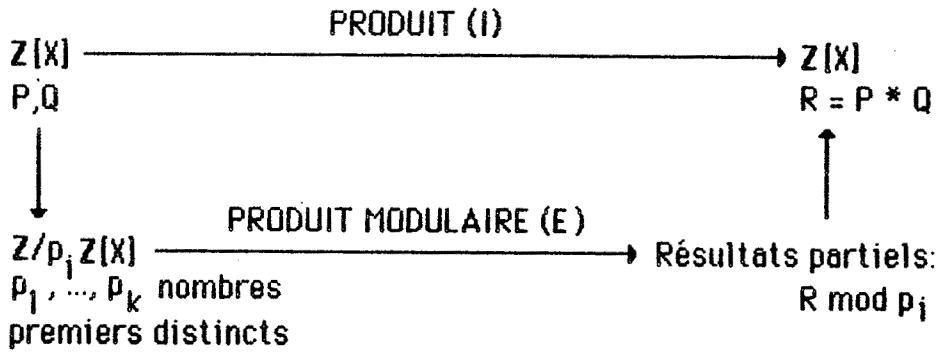


Figure 8 : Approche modulaire

On trouvera dans [T5] une discussion sur l'implémentation et le coût de l'arithmétique modulo p, ainsi qu'une description d'un processeur IPS calculant dans $Z_{/pZ}$.

L'exemple du calcul du PGCD de deux polynômes est encore plus probant: la croissance des résultats intermédiaires rend indispensable une approche modulaire. On donne dans [T10] un exemple où partant de deux polynômes de degré 8 et 6, à coefficients tous inférieurs à 100, le résultat fourni par l'algorithme d'Euclide est un entier naturel à 35 chiffres décimaux ... alors que la réponse attendue est 1: les polynômes sont premier entre eux.

L'approche modulaire se complique. Un algorithme détaillé pour le calcul du PGCD modulaire est indiqué dans [T5] et [T10], accompagné d'une étude de complexité [T10].

Se pose alors le problème de l'implantation de l'étape modulaire (E): calculer dans $Z_{/pZ}$ le PGCD de deux polynômes à coefficients modulo p (p premier). On choisit d'abord la méthode d'abaissement du degré, car elle susceptible de pipeline, au contraire d'une implémentation directe de l'algorithme d'Euclide. Mais on doit calculer des expressions du type

$$a - \alpha / \beta * b$$

et une nouvelle difficulté réside dans la division. Habituellement, on emploie une tables des inverses. Mais quitte à employer une table, nous proposons une approche performante permettant d'éliminer toutes les opérations bi-variables (multiplication et division): cette approche est

basée sur l'utilisation des logarithmes de Zech [99], et elle est résumée figure 9.

ARITHMETIQUE MODULAIRE

Un entier modulo p est usuellement représenté par un élément de l'ensemble

$$E_p = \{ 0, 1, ..., p-1 \}.$$

Nous fixons une racine primitive K de p (i.e. un generateur du groupe multiplicatif $Z_p \setminus \{0\}$). Alors tout élément non nul de E_p peut s'écrire de façon unique K^i, $0 \le i \le p-2$. Si on attache à 0 la représentation spéciale *, on peut représenter tout élément K^i de E_p simplement par i. Multiplication et division sont transformées en addition et soustraction, selon les règles:

$$i*j = \begin{cases} * \text{ si } i \text{ ou } j = * \\ i+j \bmod p-1 \text{ sinon} \end{cases}$$

$$i/j = \begin{cases} * \text{ si } i = * \\ \text{indéfini si } j = * \\ i-j \bmod p \text{ sinon} \end{cases}$$

Pour l'addition (et la soustraction), nous utilisons les logarithmes de Zech: on définit Z(u) par

$$K^{Z(u)} = 1 + K^u \text{ pour } u \in \{0, 1, ..., p-2\} \setminus \{(p-1)/2\}$$

$$Z((p-1)/2) = * \text{ et } Z(*) = 0$$

On en déduit les règles suivantes pour l'addition:

$$i+j = \begin{cases} j \text{ si } i = * \\ i \text{ si } j = * \\ * \text{ si } j-i = \pm (p-1)/2 \\ i + Z(j-i) \text{ sinon} \end{cases}$$

et des règles similaires pour la soustraction.

Ces représentations modulo un nombre premier p d'entiers de n bits ont n+1 bits, puisqu'un bit spécial est réservé pour *. L'emploi d'une table de logarithmes de Zech unidimensionnelle permet d'éliminer toutes les opérations bi-variables.

Figure 9

Nous obtenons ainsi une architecture systolique de n+m processeurs permettant de calculer le PGCD en temps linéaire:

Théorème 25 [T5] [T10] : le PGCD modulaire de deux polynômes de degrés respectifs m et n ($m \leq n$), à coefficents dans Z/pZ, p premier, peut être calculé en temps $(2n+m).\tau$ sur un réseau de n+m processeurs, où τ est le temps de cycle d'un processeur.

Au niveau macroscopique, l'architecture cellulaire de notre solution est très voisine du réseau de BRENT et KUNG [9] pour le calcul du PGCD de deux polynômes à coefficients entiers, mais l'intérêt de notre approche modulaire est double:
- c'est la seule qui règle les problèmes de croissance des résultats intermédiaires, condition "sine qua non" pour des problèmes issus du Calcul Formel
- elle permet, au niveau microscopique, une réalisation originale et concurrentielle de l'architecture interne de chaque cellule.

Une réalisation VLSI d'une telle cellule a d'ailleurs été entreprise pour tester sa faisabilité. L'implémentation de l'arithmétique modulaire que nous avons choisie est bien adaptée aux exigences du compilateur de silicium APPOLON [53] (pas de multiplieur, notamment), et nous présentons dans [T22] deux réalisations de la partie opérative, chacune des versions correpondant à un degré de parallélisme donné.

Il faut souligner que tout le dessin des masques a été effectué automatiquement par le compilateur à partir d'une description comportementale en instructions élémentaires.

2.4. Lissage par médiane

Le lissage par médiane est une technique utilisée en traitement d'images, qui consiste à remplacer chaque point d'une grille par la valeur médiane des points situés à l'intérieur d'un certain voisinage local. Contrairement à la convolution, ce lissage ne crée pas des valeurs intermédiaires qui ne se trouvaient pas dans la donnée originale.

Nous proposons dans [T2] un schéma de calcul récursif, permettant de calculer la médiane de 2k+1 points à partir d'un agencement régulier de cellules réalisant exclusivement des opérations du type a □ (b Δ c), où □ et Δ désignent le min ou le max de deux éléments.

Nous citons brièvement ici la réalisation d'un circuit systolique réalisé en technologie nMOS destiné au lissage par médiane sur 5 points (k=2) d'une suite unidimensionnelle de données. Il s'est agi d'un travail d'initiation à la conception des circuits intégrés. A l'aide des outils développés par l'équipe d'Architecture des Ordinateurs (système de CAO "Lucie" et microscope électronique) et des possibilités offertes par le projet CMP [3], nous avons dessiné les masques d'un circuit de dimension modeste. L'utilisation de cellules élémentaires programmables (minimum ou maximum de deux entiers) a permis de réduire le temps de conception d'une cellule du réseau, la structure de cette cellule étant elle-même répétée dans l'architecture.

Nous renvoyons à [T3] pour une description architecturale du circuit. Cette première expérience, si limitée soit-elle, nous a permis de vérifier de façon satisfaisante tout l'intérêt du modèle systolique.

3. ALGORITHMES PARALLELES POUR MACHINES SIMD & MIMD

Le chapître 4 constitue une introduction à l'algorithmique parallèle. Après une brève description des architectures SIMD et MIMD, on y présente les notions de tâches et de graphe d'ordonnancement (qui définit les contraintes temporelles liant l'exécution des tâches). Le développement d'ordinateurs parallèles a conduit à réévaluer la plupart des algorithmes usuels en fonction de nouveaux critères de viabilité et de performance. Pour paralléliser un algorithme, on commence par partitionner le problème en sous-tâches. L'élaboration du graphe de précédence permet de définir les contraintes temporelles pour l'exécution des tâches. Reste alors à affecter les tâches aux processeurs, en respectant les contraintes précédentes.

Nous rappellons brièvement les notions de facteur d'accélération et d'efficacité, renvoyant au chapître 4 pour une description du formalisme lié au graphe des tâches.

Considérons un algorithme qui s'exécute sur un ordinateur parallèle comportant p processeurs (identiques) en un temps T_p, et soit T_1 son temps d'exécution séquentielle (avec un seul processeur). On définit le facteur d'accélération par le rapport

$$S_p = \frac{T_1}{T_p} = \frac{\text{temps d'exécution avec 1 processeur}}{\text{temps d'exécution avec p processeurs}}$$

D'autre part, on introduit $e_p = S_p / p$ qui est l'efficacité de l'algorithme parallèle; e_p est une quantité inférieure à 1, car on peut toujours simuler séquentiellement l'exécution d'un algorithme parallèle en multipliant son temps d'exécution par le nombre de processeurs. En fait, e_p mesure le taux moyen d'utilisation des processeurs, et la parallélisation d'un algorithme est d'autant meilleure que e_p est proche de 1.

Pour amorcer une étude de complexité, nous avons besoin de simplifier le modèle. D'abord, nous nous préoccupons exclusivement de la parallélisation d'algorithmes "compute-bound". Dans un algorithme compute-bound, le nombre de calculs élémentaires est plus grand que le

nombre de données en entrée-sortie. Sinon, le problème est I/O-bound. Par exemple, la multiplication de deux matrices de taille n nécessite $O(n^3)$ multiplications et additions pour $O(n^2)$ données, c'est un problème compute-bound. Au contraire, l'addition de deux matrices exige n^2 additions pour $3n^2$ opérations d'entrée-sortie, et est donc I/O-bound.

Cette restriction étant faite, on peut idéaliser le temps d'exécution en négligeant les temps d'accès, de stockage, et d'échange des données, pour ne retenir que le coût des opérations arithmétiques. On choisit alors comme unité de temps le temps nécessaire à la réalisation de l'une des quatre opérations élémentaires (contrairement aux modèles de la décennie précédente, et pour suivre l'évolution de la technologie, on admet qu'une multiplication équivaut à une addition).

Enfin, si nous négligeons les temps de communication, nous imposons un certain mode d'accès aux données. Ainsi pour le traitement en parallèle des éléments d'une matrice A, nous supposerons par la suite que nous pouvons accéder aux éléments soit par lignes, soit par colonnes (comme les dispositifs matériels l'autorisent pour la plupart, cf. SCHENDEL [85] chap. 2.4.). Les opérations de transfert possibles seront alors :
- la transmission d'une ligne ou d'une colonne de A de l'organe de stockage vers un organe de traitement.
- la transmission d'une ligne ou d'une colonne de A d'un organe de traitement vers l'organe de stockage.

La duplication d'une ligne ou d'une colonne aura un coût nul, c'est à dire que nous supposons qu'il est possible de transférer simultanément une même donnée vers plusieurs processeurs. Dans ce cas, aucun processeur ne pourra modifier cette donnée. Inversement, un processeur ne pourra modifier une donnée que s'il est le seul à en posséder un exemplaire.

Etant donné un algorithme à paralléliser, supposons définies la segmentation en tâches et les contraintes d'ordonnancement. Voici plusieurs questions caractéristiques des problèmes qui se posent alors:
- (1) ne supposant pas de limite au nombre de processeurs disponibles, quel est le temps T_{opt} d'un algorithme optimal ?
- (2) quel est le nombre mimimal de processeurs permettant de réaliser un algorithme s'exécutant en temps T_{opt} ?

A priori, on sait seulement que T_{opt} est supérieur ou égal au plus long chemin du graphe d'ordonnancement, et qu'il est inutile de disposer de plus de D processeurs, ou D est le diamètre du graphe...

Enfin, pour un problème de taille n, se posent les questions relatives à la situation $n \to \infty$:

- (3) supposant un nombre de processeurs $p = \alpha n$, quel est le α maximisant l'efficacité ?
- (4) quel est le nombre mimimal de proceseurs permettant de réaliser un algorithme s'exécutant en temps asymptotiquement équivalent à T_{opt} ?

Nous allons donner une réponse partielle à ces questions pour plusieurs méthodes de résolution de systèmes linéaires denses et pour des problèmes aux moindres carrés.

2.3.1. Complexité de la factorisation QR en parallèle

Les problèmes aux moindres carrés sans contraintes se rencontrent dans de nombreux calculs scientifiques. On peut les formuler ainsi: soit A une matrice rectangulaire de taille m x n, avec $m \geq n$, et soit b un vecteur à m composantes; il s'agit de déterminer un vecteur x à n composantes minimisant $\|Ax-b\|$ en norme euclidienne.

On distingue usuellement deux cas, selon que A est de rang maximal où non. Mais on doit toujours calculer la factorisation orthogonale de A,

$$QA = (R,0)^T,$$

soit via des rotations planes (méthode des rotations de Givens), soit via des symétries par rapport à des hyperplans (réduction de Householder).

Nous nous intéressons à la parallélisation de la méthode des rotations de Givens pour une matrice de taille m x n. L'algorithme séquentiel nécessite $mn - n(n+1)/2$ étapes, chaque étape étant le temps nécessaire pour réaliser une rotation plane entre deux lignes de la matrice.

Une tâche élémentaire sera constituée par une rotation de Givens. Si l'on descend jusqu'au niveau des opérations arithmétiques, toutes les tâches n'ont pas le même coût, puisque les lignes comprennent de plus en plus d'éléments nuls au fur et à mesure du déroulement de l'algorithme. C'est cette approche qu'ont choisi LORD et al. [70].

Nous nous plaçons ici à un niveau macroscopique, et considérons que toutes les tâches ont la même durée d'exécution. Avec un tel modèle, et en supposant que le nombre de processeurs disponibles permet de réaliser jusqu'à $\lfloor m/2 \rfloor$ rotations en parallèle, SAMEH et KUCK [84] ont proposé un schéma de factorisation dont le nombre d'étapes est $m+n-2$ si $m>n$ et $2n-3$ si $m=n$. Récemment, MODI et CLARKE [75] ont introduit un algorithme "glouton" (que nous avons également proposé de façon indépendante [T4]).

Ces auteurs analysent partiellement le nouvel algorithme dans le cas particulier où m tend vers l'infini, n fixé. Le nombre d'étapes est alors asymptotiquement égal à $\log_2 m + (n-1)\log_2\log_2 m$. MODI et CLARKE présentent également d'autres schémas, appelés schémas de Fibonacci, et discutent de leurs performances.

Enfin, SAMEH et KUCK [84] et MODI et CLARKE [75] expliquent comment modifier leurs algorithmes pour traiter le cas où seulement p rotations de Givens ($p<\lfloor m/2 \rfloor$) peuvent être effectuées simultanément.

Au vu de ces résultats, plusieurs questions restent ouvertes:

1. Etant données les valeurs de m, n et p (le nombre de rotations de Givens pouvant être effectuées simultanément), trouver un algorithme optimal et évaluer le nombre d'étapes qu'il nécessite
2. Parmi les algorithmes précédents, déterminer ceux qui sont optimaux ou asymptotiquement optimaux; la réponse est-elle fonction des relations entre m, n et p ?

En outre, tous les algorithmes présentés par SAMEH et KUCK [84] et MODI et CLARKE [75] sont des suites de Givens, ce qui signifie que les zéros crées ne sont jamais détruits ultérieurement. La question de savoir si l'introduction temporaire de zéros qui seront détruits plus tard peut conduire à un meilleur parallélisme n'a jamais été discutée; c'est pourtant un bel exemple des questions spécifiques auxquelles l'algorithmique parallèle donne naissance.

Nous renvoyons à [T21] et [T24] pour une description des algorithmes de SAMEH et KUCK et MODI et CLARKE. De même, une définition précise de l'algorithme glouton se trouve dans [T4] ou [T24]. Pour ce dernier, disons simplement qu'il maximise à chaque étape le nombre d'éléments annulés (d'où son nom).

• On suppose d'abord que $p = \lfloor m/2 \rfloor$. Un algorithme M s'exécutant en T étapes est représenté par $M = (M(1),\ldots,M(T))$, où pour $1 \le t \le T$, M(t) est un groupe de r(t) rotations disjointes ($r(t) \le p$). On construit une suite A(t) telle que A(0)=A et pour $1 \le t \le T$, A(t) est obtenue en appliquant en parallèle les rotations de M(t) à A(t-1). On note brièvement $M = (M,T,R)$, avec $R = r(1)+\ldots+r(T)$.

On a déjà dit d'une suite de Givens est un algorithme où tout zéro créé est préservé. Une séquence de Givens parallèle standard (SGPS) est une suite de Givens qui réduit A sous forme triangulaire supérieure en annulant les éléments de gauche à droite et de bas en haut. Tous les

algorithmes cités plus haut, y compris l'algorithme glouton, sont des SGPS.

Enfin, on appelle r(i,t) le nombre de zéros introduit colonne i au temps t, et s(i,t) le nombre total de zéros colonne i après le temps t (pour des suites de Givens, on a donc $s(i,t) = r(i,1) + \ldots + r(i,t)$).

Lemme 26 [T4] : soit M une suite de Givens. M est une SGPS si

$s(0,t) = m$ pour $t \geq 0$; $s(i,0) = 0$ pour $1 \leq i \leq m$

$s(i,t) \leq s(i,t-1) + \lfloor (s(i-1,t-1) - s(i,t-1))/2 \rfloor$ pour $i,t \geq 1$.

Lemme 27 [T4] : dans l'ensemble des algorithmes optimaux il existe une SGPS.

Bien que le résultat paraisse intuitivement évident, la démonstration est longue et fastidieuse. Mais le lemme 27 est à la base du

Theorème 28 [T4] [T24] : pour toutes valeurs de m et n (avec $m \geq n$), et pour $p = \lfloor m/2 \rfloor$, l'algorithme glouton est optimal.

Le temps d'exécution OPT(m,n) de l'algorithme glouton donne donc la complexité de la factorisation QR en parallèle d'une matrice m x n. Nous n'avons pas réussi à obtenir une formule exacte pour OPT(m,n), mais nous avons le

Theorème 29 [T4] [T24] : soit $p = \lfloor m/2 \rfloor$.

1. Si n = m (A est une matrice carrée), alors $OPT(m,n) = 2n + o(n)$.
2. Si n est fixé, alors $OPT(m,n) = \log_2 m + o(\log_2 m)$.
3. Si $m = o(n^2)$, alors $OPT(m,n) = 2n + o(n)$.
4. Si $m = o(n^k)$, avec $k \geq 3$, alors $2n + o(n) \leq OPT(m,n) \leq 3n + o(n)$.

Notons ASE(m,n) l'efficacité maximale qu'il est asymptotiquement possible d'obtenir pour résoudre le problème:

Corollaire 30 [T4] [T24] : soit $p = \lfloor m/2 \rfloor$.

1. Si m = n, alors ASE(m,n) = 1/2.
2. Si $m = \beta n$, alors $ASE(m,n) = 1 - 1/2\beta$.
3. Si $m = o(n^2)$, alors ASE(m,n) = 1.
4. Si $m = o(n^k)$, alors $ASE(m,n) \geq 2/3$.
5. Si n est fixé, alors $ASE(m,n) = 2n/\log_2 m$.

L'importance du corollaire 30 n'est pas seulement théorique: la table de la figure 10 montre la validité des estimations obtenues, et ce même pour des tailles de matrices relativement faibles:

M=N	7	8	9	10	14	15	16	17	18
OPT	10	11	13	15	23	24	26	28	30

M=N	32	40	64	128	256	512	1024	2048	4096
OPT	56	72	118	243	495	1000	2015	4051	8129

Figure 10 : Nombre d'étapes de l'algorithme glouton pour une matrice carrée d'ordre N, à comparer avec l'estimation 2N

Corollaire 31 [T4] [T24] : soit $p = \lfloor m/2 \rfloor$.

1. Si m = n, alors le schéma de SAMEH et KUCH est asymptotiquement optimal.

2. Si $m = \beta n$, alors le schéma de Fibonacci d'ordre 1 est asymptotiquement optimal.

3. Si $m = o(n^2)$, alors le schéma de Fibonacci d'ordre 1 est asymptotiquement optimal

4. Si $m = o(n^k)$ et $m \geq n^2$, alors le schéma de Fibonacci d'ordre 2 est d'efficacité asymptotique 2/3.

• Dans le cas où seulement $p < \lfloor m/2 \rfloor$ rotations de Givens pouvent être effectuées simultanément, la situation est plus complexe: on peut toujours définir les SGPS, mais il existe plusieurs algorithmes gloutons. Les seuls résultats que nous avons pu obtenir sont donnés par les

Lemme 32 [T21] : pour $p < \lfloor m/2 \rfloor$, il existe une SGPS dans l'ensemble des algorithmes optimaux

Proposition 33 [T21]: étant donnés m, n et p, il existe un algorithme optimal dans l'ensemble des algorithmes gloutons

Cependant, il importe de noter que l'algorithme glouton "horizontal" comme l'algorithme glouton "vertical" (voir [T21]) peuvent ne pas être optimaux. En fait, nous ne connaissons pas de construction systématique d'un algorithme optimal dans le cas $p < \lfloor m/2 \rfloor$.

2.3.2. Parallélisation comparée de méthodes de diagonalisation

Pour résoudre un système linéaire dense Ax=b sur un ordinateur séquentiel classique, la méthode la plus utilisée est celle de Gauss. Celle-ci effectue deux opérations de nature apparemment différente: triangularisation de la matrice et inversion d'un système triangulaire. Une autre approche consiste à rassembler ces deux étapes en une seule: diagonaliser la matrice. En général, on utilise la méthode de Jordan dans ce cas. Cependant cette méthode a un coût supérieur à celle de Gauss ($n^3+O(n^2)$ opérations arithmétiques contre $2n^3/3+O(n^2)$). HUARD [47] a proposé une méthode, dite des paramètres, qui peut être vue comme une variante de la méthode de Gauss qui intègre dans l'étape de triangularisation, une étape de résolution et donc permet de diagonaliser la matrice en effectuant le même nombre d'opérations.

Nous étduons ici la parallélisation sur des machines SIMD ou MIMD de ces méthodes de diagonalisation

• méthode de Jordan

L'algorithme séquentiel est bien connu, et le mode d'accès aux données va déterminer la segmentation en tâches du programme. Nous avons ainsi deux parallélisations possibles:

```
{ JORDAN par lignes }
Pour k <- 1 à n
    exécuter T_kk : < Pour j <- k+1 à n+1,
                      faire a_kj <- a_kj /a_kk >
    Pour i <- 1 à n, i≠k
        exécuter T_ki : < Pour j <- k+1 à n+1,
                          faire a_ij <- a_ij - a_ik* a_kj >
```

Par contre, si on autorise un accès aux données par colonnes, on peut inverser l'ordre des boucles imbriquées en i et j pour avoir:

```
{ JORDAN par colonnes }
Pour k <- 1 à n
    Pour j <- k+1 à n+1
        exécuter T_kj: < a_kj <- a_kj /a_kk
                         Pour i <- 1 à n, i≠k
                         faire a_ij <- a_ij - a_ik* a_kj >
```

Nous avons choisi l'exemple de la méthode de JORDAN pour l'exposé introductif du chapitre 4. Nous nous contentons ici de rassembler les résultats obtenus:

Théorème 34 [T28] [T30] : pour JORDAN par lignes:

(i) $T_{opt} = 3n^2/2$ et $D = n$

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

(iii) pour $p \leq 2n/3$, $p = \alpha.n$, nous avons construit un algorithme optimal, d'efficacité 1, s'exécutant en temps n^2/α.

(iv) la valeur minimale de α permettant de réaliser un algorithme en T_{opt} est $\alpha = 2/3$.

(v) pour $p \geq 2n/3$, $p = \alpha.n$, $e_{p,\infty} = 2/3\alpha$.

Théorème 35 [T28] [T30] : pour JORDAN par colonnes:

(i) $T_{opt} = 2n^2$ et $D = n$

(ii) $e_{n,\infty} = 1/2$

(iii) parmi les algorithmes optimaux, il en existe qui satisfont à la contrainte d'ordonnancement (C): $T_{K,j} \leq\leq T_{K,j+1}$ pour $j>k+1$

(iv) pour toute valeur de p, l'algorithme glouton est optimal. Pour $p = \alpha.n$, $T_{glouton} = (\alpha + 1/\alpha)\ n^2$ et $e_{\alpha n,\infty} = 1 / (1+\alpha^2)$

(v) la seule valeur de α permettant de réaliser un algorithme en T_{opt} est $\alpha = 1$

Les deux théorèmes précédents montrent que la méthode par lignes est toujours préférable à la méthode par colonnes, et cela quel que soit le nombre de processeurs. L'étude théorique que nous avons menée permettrait de disposer de critères de selection précieux lors d'une expérimentation pratique de la méthode de JORDAN sur une machine multi-processeurs.

• méthode des paramètres

La méthode des paramètres comporte deux étapes par itération:

- annulation des k-1 premiers éléments de la ligne k et modifications correspondantes des n-k+1 éléments suivants: étape de descente.
- annulation des k-1 premiers éléments de la colonne k et modifications correspondantes des n-k derniers éléments des k-1 premières lignes: étape

de remontée.

L'algorithme séquentiel est le suivant:

```
{ PARAMETRES séquentiel }
Pour k <- 1 à n
(* élimination de la ligne k *)
    Pour i <- 1 à k-1
    Pour j <- k à n+1
        a_kj <- a_kj - a_ki * a_ij
    (* préparation de la ligne k *)
    c <- 1/a_kk
    Pour j <- k+1 à n+1
        a_kj <- a_kj * c
    (* mise à zéro colonne k, lignes 1 à k *)
    Pour i <- 1 à k-1
    Pour j <- k+1 à n+1
        a_ij <- a_ij - a_ik * a_kj
```

Suivant l'accés aux éléments de la matrice, on peut écrire chacune des deux étapes de deux manières différentes: par lignes ou par colonnes.

La descente en lignes n'est pas parallélisable, puisque, pour chaque valeur de i, les n-k+1 éléments a_{kj} sont modifiés. En effet, ceci contredit notre hypothèse d'interdire des modifications simultanées d'un même élément. Par conséquent, deux versions sont possibles: descente colonnes - remontée lignes et descente colonnes - remontée colonnes. On montre dans [T28] que cette dernière version est la meilleure.

On a alors les tâches P_{kj} suivantes :

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

Pour i <- 1 à k-1 $\quad k+1 \leq j \leq n+1$

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

Pour i <- 1 à k

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

On retrouve la même structure globale que pour l'algorithme de Jordan par colonnes, mais le temps d'exécution de chaque tâche est variable. P_{kj}

s'effectue en 4k-1 unités. Les contraintes temporelles restent inchangées, et on a la

Proposition 36 [T28] [T30] :

(i) L'algorithme glouton est optimal.

(ii) Pour m=1 et p = αn, le temps d'exécution de l'algorithme glouton est égal à $(-2\alpha^3+6\alpha^2+2)\, n^2 /3\alpha + O(n/\alpha)$

(iii) $e_{p,\infty} = 1/(1+ \alpha^2(3-\alpha))$

Le théorème suivant compare le temps d'exécution P_p de la méthode des paramètres avec le temps JL_p de la méthode de Jordan par lignes (toujours meilleur que celui de la version par colonnes, en vertu des théorèmes 34 et 35):

Théorème 36 [T28] [T30] : posons p = α.n. Pour $\alpha \leq 0.44$, $P_p < JL_p$ et pour $\alpha \geq 0.45$, $JL_p < P_p$

4. REFERENCES

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

[2] A.V. AHO, D.S. HIRSCHBERG, J.D. ULLMAN, Bounds on the complexity of the longest common subsequence problem, J.ACM 23,1,1-12 (1976)

[3] F. ANCEAU, A design methodology and a silicon compiler for circuits specified by algorithms, Proc. Third CALTECH Conf. on VLSI, R. Bryant ed., Computer Sc. Press 1983

[4] F. ANDRE, P. FRISON, P. QUINTON, Algorithmes systoliques: de la théorie à la pratique, RR 214, INRIA Rocquencourt, Mai 1983

[5] A. APOSTOLICO, A. NEGRO, Systolic algorithms for string manipulations, IEEE Trans. Computers C33, 4, 361-364 (1984)

[6] Y.M. BARZDIN, Complexity of recognition of symmetry in Turing machines, Problemy Kibernetiki 15 (1965).

[7] A. BOJANCZYK, R.P. BRENT, H.T. KUNG, Numerically stable solution of dense systems of linear equations using mesh-connected processors, Technical Report, Carnegie Mellon University, 1981

[8] A. BOSSAVIT, Préface des actes du colloque AFCET-GAMNI-ISINA, 17-18 Mars 1983, Paris, Bulletin de la direction des étude et recherches EDF, série C vol 1, 1983

[9] R.P. BRENT, H.T. KUNG, F.T. LUK, Some linear-time algorithms for systolic arrays, Technical Report TR CS 82-541 (1982), Cornell University Department of Computer Science

[10] S.C. CHEN, D.J. KUCK, Time and parallel processor bounds for linear recurrence systems, IEEE Trans. Computers, C-24, 701-717 (1975)

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

[12] S.N. COLE, Real-time computation by n-dimensional iterative arrays of finite-state machines, IEEE Trans. Computers C18, 4, 349-365 (1969)

[13] M. COSNARD, E. GOLES, Dynamical properties of an automaton with memory, NATO Advanced Research Workshop on Disordered systems and Biological Organization, 1985, à paraître Springer Verlag

[14] K. CULIK II, J. GRUSKA, A. SALOMAA, On a family of L languages resulting from systolic tree automata, Research Report CS-81-36, University of Waterloo

[15] K. CULIK II, J. GRUSKA, A. SALOMAA, Systolic treillis automata for (VLSI), Research Report CS-81-36, University of Waterloo

[16] K. CULIK II et al., Systolic tree acceptors, RAIRO Theoretical Computer Science 18, 1, 53-69 (1984)

[17] T.W. CURRY, A. MUKHOPADHYAY, Realization of efficient non-numeric operations through VLSI, Proc. of VLSI 83, F. Anceau and E.J. Aas eds, Elsevier Science Publishers B.V., North Holland, 327-336 (1983)

[18] J. H. DAVENPORT, VLSI et calcul formel, RR IMAG Grenoble 357, March 1983

[19] J. H. DAVENPORT, On the integration of algebraic functions, Lect. Noter in Computer Sc. 102, Springer Verlag, 1981

[20] J.M. DELOSME, Algorithms for finite shift-rank processes, Ph.D. Thesis, Technical Report M735-22, Sept. 1982, Stanford Electronics Laboratories

[21] J. ERHEL, W. JALBY, A. LICHNEWSKI, F. THOMASSET, Quelques progrès en calcul parallèle et vectoriel, 6ème coll. intern. sur les méthodes de calcul scientifique et technique, Glowinski et Lions eds, 1983

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

[23] M. FEILMEIER, Parallel computers - Parallel mathematics, IMACS North Holland, 1977

[24] A.L. FISHER, H.T. KUNG, K. SAROCKY, Experience with the CMU programmable systolic chip, Proc. SPIE Symp., vol. 495, Real-Time Signal Processing *VII* (1984)

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

[26] M. FLYNN, Some computer organisations and their effectiveness, IEEE Trans. Computers C21, 9, 948-960 (1972)

[27] F. FOGELMAN, Contributions à une théorie du calcul sur réseaux, Thèse, Grenoble (1985)

[28] M.J. FOSTER, H.T. KUNG, The design of special-purpose VLSI chips, IEEE Computer Magazine 13, 1, 26-40 (1980)

[29] M.J. FOSTER, H.T. KUNG, Recognize regular langages with programmable building-blocks, Proc. VLSI 198, Edinburgh

[30] J.M. FRAILONG, W. JALBY, J. LENFANT, XOR-schemes: a flexible data organization in parallel memories, 1985 International Conference on Parallel Processing

[31] J.M. FRAILONG, W. JALBY, Multigrid processing on an SIMD architecture, Copper Mountain Multigid Conference (1985)

[32] D.D. GAJSKI, J.K. PEIR, Parallel processing : problems and solutions, Preprint Univ. Illinois, 1984

[33] W.M. GENTLEMAN, Least squares computation by Givens transformations without square roots, J. Inst. Math. Appl. 12, 329-336 (1973)

[34] W.M. GENTLEMAN, Error analysis of QR decomposition by Givens tranformations, Linear Algebra and Appl. 10, 189-197 (1975)

[35] W.M. GENTLEMAN, H.T. KUNG, Matrix triangularisation by systolic arrays, Proc SPIE 298, Real-time Signal Processing *IV* (1981)

[36] A. GILL, Linear sequent l circuits, Analysis, Synthesis and Applications, Mc Graw-Hill Series in Computer Science, New York, 1966

[37] E. GOLES, Comportement dynamique de réseaux d'automates, Thèse, Grenoble (1985)

[38] G.H. GOLUB, R. PLEMMONS, Large-scale geodetic least-squares adjustment by dissection and orthogonal decomposition, Linear Algebra and Appl. 38, 3-28 (1980)

[39] L.J. GUIBAS, H.T. KUNG, C.D. THOMPSON, Direct VLSI implementation of combinatorial algorithms, Proc. Caltech Conf. on VLSI: Architecture, design, fabrication, California Instit. of technology, Pasadena (1979)

[40] L.J. GUIBAS, F.M. LIANG, Systolic stacks, queues and counters, Proc. of the 1982 Conference of Advanced Research in VLSI, M.I.T. Boston

[41] D. HELLER, A survey of parallel algorithms in numerical linear algebra, Siam Review 20, p 740-777, 1978

[42] D. HELLER, I. IPSEN, Systolic networks for orthogonal equivalence transformations and their applications, Proc. 1982 Conf. Advanced Research in VLSI, p 113-122, MIT 1982

[43] D.S. HIRSCHBERG, Algorithms for the longest common subsequence problem, J.ACM 24,4,664-675 (1977)

[44] D.S. HIRSCHBERG, An information-theoretic lower bound for the longest common subsequence problem, Inform. Process. Lett. 7,1, 40-41 (1978)

[45] E. HOROWITZ, A. ZORAT, Divide-and-conquer for parallel processing, IEEE Trans. on Computers C32(6), June 1983

[46] S. HORWARTH Jr., A new adaptative recursive LMS filter, Proc. Digital Signal Processing, Cappellini and Constantinides eds., 21-26, Academic Press (1980)

[47] P. HUARD, La méthode Simplex sans inverse explicite, Bulletin EDF série C, 79-98 (1979)

[48] J.W. HUNT, T.G. SZYMANSKI, A fast algorithm for computing longest common subsequences, C.ACM 20, 350-353 (1977)

[49] K. HWANG et F. BRIGGS, Parallel processing and computer architecture, Mac Graw Hill, 1984

[50] I. IPSEN, A parallel QR method using fast Givens'rotations, Research Report YALEU/DCS/RR 299, 1984, Yale University

[51] W. JALBY, J.M. FRAILONG, J. LENFANT, Diamond schemes: an organization of parallel memories for efficient array processing, RR 342, INRIA Oct. 1984

[52] W. JALBY, J. MAILLARD, F. MUSSI, Optimization and performance evaluation problems arising in parallelization of particle physics programs, Preprint INRIA, Avril 1985

[53] R. JAMIER, A.A. JERRAYA, The Appolon data-path silicon compiler, Proc. ICCD 85, New-York, Octobre 1985

[54] D.J. KUCK, The structure of computers and computations, J. Wiley & Sons, New York, 1978

[55] A.V. KULKARNI, D.W.L. YEN, Systolic processing and an implementation for signal and image processing, IEEE Trans. on Computers C31, 10, 1000-1009 (1982)

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

[57] H.T. KUNG, The structure of parallel algorithms, Advances in Computers 19 (1980), p 65-112

[58] H.T. KUNG, Use of VLSI in algebraic computation: some suggestions, in Proc. SYMSAC 81, P.S. Wang ed., ACM, NewYork, 1981

[59] H.T. KUNG, Why systolic architectures, IEEE Computer Magazine 15, 1, 37-46 (1982)

[60] H.T. KUNG, Programmable systolic chip, NATO Advanced Study Institute on Microarchitecture of VLSI computers, Sogesta, Italy, July 9-20, 1984

[61] H.T. KUNG, M.S. LAM, Fault-tolerance and two-level pipelining in VLSI systolic arrays, Technical Report, Carnegie Mellon University, November 1983

[62] H.T. KUNG, P.L. LEHMAN, Systolic (VLSI) arrays for relational database operations, Proceedings of the 1980 ACM/SIGMOD international conference on management of data, Los Angeles, USA

[63] H.T. KUNG, C.E. LEISERSON, Systolic arrays for (VLSI), in Proc. of the Symposium on Sparse Matrices Computations, I.S. Duff and G.W. Stewart eds, Knoxville, Tenn., 256-282 (1978)

[64] H.T. KUNG, W.T. LIN, An algebra for VLSI algorithmic design, Technical Report, Carnegie Mellon University, April 1983

[65] M. LAM, J. MOSTOW, A transformational model of VLSI systolic design, IEEE Computer 18, 2, 42-52 (1985)

[66] J. LARUS, A comparison of microcode, assembly code and high-level languages on the VAX11 and RISC1, Computer Architecture News 10, 5 (1982)

[67] C. LAWSON, R. HANSON, Solving least squares problems, Prentice-Hall (1974)

[68] C.E. LEISERSON, Area-efficient VLSI computation, PhD Thesis, Carnegie Mellon University, Pittsburgh, PA, USA, Octobre 1981

[69] C.E. LEISERSON, J.B. SAXE, Optimizing synchronous systems, Proc. of the 22-th Annual Symposium on Foundations of Computer Science, IEEE, October 1981, 23-36

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

[71] W.S. MAC CULLOCH, W.M. PITTS, A logical calculus of the ideas immanent in nervous activity, Bull. of Mathematical Biophysics 5, 115-133 (1943)

[72] O. MARTIN, A. ODLYZKO, S. WOLFRAM, Algebraic properties of cellular automata, Bell Laboratories Report (1983)

[73] L. MELKEMI, M. TCHUENTE M., Systolic arrays for connectivity and triangularisation problems, Proc. Conf. Dynamic Behaviour of Automata and Applications, J. Demongeot et al. eds, Academic Press (1985)

[74] J. MISKLOSKO et V. KOTOV, Algorithms, software and hardware of parallel computers, Springer Verlag 1985

[75] J.J. MODI, M.R.B. CLARKE, An alternative Givens ordering, Numer. Math. 43, 83-90 (1984)

[76] N. NAKATSU, Y. KAMBAYASHI, S. YAJIMA S., A longest common subsequence algorithm suitable for similar text strings, Acta Informatica 18,171-179 (1982)

[77] NCR note commerciale 45CG72, Geometric arithmetic parallel processor, 1984

[78] M.S. PATERSON, unpublished manuscript, University of Warwick, Coventry, England, 1974

[79] P. QUINTON, Synthèse automatique d'architectures systoliques, Rapport IRISA, Avril 1985

[80] G. ROTE, A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion), Computing 34, 191-219 (1985)

[81] A. SAMEH, Solving the linear least squares problem on a linear array of processors, Proc. Purdue Workshop on algorithmically-specialized computer organizations, W. Lafayettte, Indiana, September 1982

[82] A. SAMEH, Numerical parallel algorithms - a survey, in "High Speed Computer and Algorithm Organization", D.KUCK, D.LAWRIE and A.SAMEH eds, p 207-228, Academic Press, 1977

[83] A. SAMEH, An overview of parallel algorithms, Bulletin EDF, C1, 129-134 (1983)

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

[85] U. SCHENDEL, Introduction to Numerical Methods for Parallel Computers, Ellis Horwood Series, J. Wiley & Sons, New York, 1984

[86] R. SCHREIBER, Systolic arrays: high performance parallel machines for matrix computation, Proc. Elliptic problem solvers II, Academic Press 1984, 187-194

[87] A.R. SMITH III, Cellular automata theory, Technical Report 2, Dec 1969, Stanford Electronics Laboratories

[88] E. SPEDICATO, On the solution of linear least squares through the ABS class for linear systems, Quaderno DMSIA 8/84, Bergamo, 1984

[89] M. SRINIVAS, Optimal parallel scheduling of Gaussian elimination DAG's, IEEE Trans. Computers 32, 1109-1117 (1983)

[90] M. STEINBY, Systolic trees and systolic language recognition by tree automata, Theoretical Computer Science 22 (1983), p 219-232

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

[92] E. TIDEN, B. LISPER, R. SCHREIBER, Systolic arrays, Technical Report TRITA-NA-8315, The Royal Institute of Stockholm, Sweden

[93] P.C. TRELEAVEN, D.R. BROWNBRIDGE, R.P. HOPKINS, Data-driven and Demand-driven computers, ACM computing surveys, 14, 1982, 93-143

[94] D. TRYSTRAM, Expérimentation d'algorithmes de préconditionnement de grands systèmes creux pour un problème de puits de pétrole, Thèse de Docteur Ingénieur, INPG Grenoble (1984)

[95] J.D. ULLMAN, Computational aspects of VLSI, Chapter 5: Systolic algorithms, Computer Science Press, Rockville, Maryland, USA, 1984

[96] R.S. VARGA, Matrix iterative analysis, Prentice Hall, Englewoods Cliffs, N.J. 1962

[97] J. VUILLEMIN, Nouvelles structures d'ordinateurs, Proc. VLSI Algorithms and architectures, P. Bertolazzi and F. Luccio eds., Elsevier Science 1985, 137-151

[98] S. WOLFRAM, Preface to Physica 10D, Elsevier Science Publishers, North Holland (1984)

[99] H.G. ZIMMER, Computational methods in number theory, Lect. Note in Math. 272, Springer Verlag, 1972

[100] U. ZIMMERMANN, Linear and combinatorial optimization in ordered algebraic structures, Ann. Discrete Math. 10, 1, 1-380 (1981)

5. BIBLIOGRAPHIE DE LA THESE

Nous indiquons (par ordre chronologique) les références des travaux qui ont été résumés dans ce chapitre. Les textes précédés d'une étoile * sont reproduits dans les chapitres suivants.

[T1] Y. ROBERT, Regular incomplete factorizations of real positive definite matrices, Linear Algebra and its Applications 48, 105-117 (1982)

[T2]* Y. ROBERT, M. TCHUENTE, Calcul en parallèle sur des réseaux systoliques, Bulletin EDF, C1, 125-128 (1983)

[T3]* Y. CLAUZEL, Y. ROBERT, M. TCHUENTE, A case study for the design and layout of systolic architectures, RR IMAG Grenoble 369, Mars 1983

[T4]* M. COSNARD, Y. ROBERT, Complexité de la factorisation QR en parallèle, C. R. Acad. Sc. Paris, t. 297, série I, 137-139 (1983)

[T5]* J.H. DAVENPORT, Y. ROBERT, Implémentation VLSI d'algorithmes modulaires issus du Calcul Formel, Actes du Séminaire "Techniques de Calcul Formel pour l'automatique, l'analyse des systèmes et le traitement du signal", P. Chenin ed., Décembre 1983, IMAG Grenoble

[T6]* Y. ROBERT, M. TCHUENTE, Calcul en temps linéaire d'une plus longue sous-suite commune à deux chaînes sur une architecture systolique, C. R. Acad. Sc. Paris, t. 299, série I, 269-271(1984)

[T7] Y. ROBERT, M. TCHUENTE, Vous avez dit "Systolique" ? Bulletin de la SMAI, Mars 1984

[T8] Y. ROBERT, M. TCHUENTE, A systolic array for the longest common subsequence problem, Proc. Conf. International workshop on High-Level Computer Architecture, Los Angeles, CA, (1984)

[T9]* Y. ROBERT, M. TCHUENTE, Real-time recognition of the language L = { $f_{n-1}(w)...f_2(w)f_1(w)w$; $w \in A^+$ } on a special-purpose architecture, RR IMAG Grenoble 474, Novembre 1984

[T10]* J.H. DAVENPORT, Y. ROBERT, VLSI and Computer Algebra: the GCD example, Proc. Conf. Dynamical behaviour of cellular automata, J. Demongeot et al. eds, Academic Press, 1985

[T11]* Y. ROBERT, M. TCHUENTE , Reconnaissance de langages en temps réel sur une architecture parallèle spécialisée, C. R. Acad. Sc. Paris, t. 300, série I, 363-368 (1985)

[T12]* M. COSNARD, Y. ROBERT, Complexity of the Givens factorization for least squares problems, Proc. Conf. Vector and Parallel Processors for Scientific Computation", Academia Nazionale dei Lincei, IBM Italia, Rome, Mai 1985

[T13] Y. ROBERT, M. TCHUENTE, Special-purpose architectures for string processing, Actes du colloque COGNITIVA 85, Paris, Juin 1985

[T14] M. COSNARD, Y. ROBERT, Complexity of parallel algorithms: the example of the QR decomposition of a rectangular matrix, Actes du colloque COGNITIVA 85, Paris, Juin 1985

[T15]* Y. ROBERT, M. TCHUENTE, Connection-graph and iteration-graph of monotone boolean functions, Discrete Applied Mathematics 11, 245-253 (1985)

[T16]* Y. ROBERT, M. TCHUENTE, Réseaux systoliques pour des problèmes de mots, RAIRO Informatique Théorique 19, 2, 107-123 (1985)

[T17]* Y. ROBERT, M. TCHUENTE, Résolution systolique de systèmes linéaires denses
RAIRO Modélisation et Analyse Numérique 19, 2, 315-326 (1985)

[T18]* Y. ROBERT, Block LU decomposition of a band matrix on a systolic array, Intern. Journal Computer Math. 17, 295-315 (1985)

[T19] Y. ROBERT, M. TCHUENTE, Quelques exemples d'architectures systoliques, Colloque C3, Angoulème, 16-18 Septembre 1985, à paraître aux éditions du CNRS

[T20]* Y. ROBERT, Architectures systoliques: une introduction, Journées "Calculs sur réseaux: théorie et applications", F. Fogelman et al. eds, CNRS, Paris, 19-20 Septembre 1985

[T21] M. COSNARD, Y. ROBERT, Complexity of the parallel QR decomposition of a rectangular matrix, International Conference on Parallel Computing, Freie Universität Berlin, RFA, 23-25 Septembre 1985, à paraître North Holland

[T22]* R. JAMIER, A. JERRAYA, Y. ROBERT, Using a silicon compiler for computer algebra, Proc. Conf "Le Calcul ... demain", Grenoble, F. Robert et al. eds., Edition,s Masson, à paraître

[T23]* Y. ROBERT, M. TCHUENTE, A systolic array for the longest common subsequence problem, Information Processing Letters 21, 191-198 (1985)

[T24]* M. COSNARD, J.M. MULLER, Y. ROBERT, Parallel QR decomposition of a rectangular matrix, Numerische Mathematik, à paraître

[T25]* Y. ROBERT, M. TCHUENTE, An efficient systolic array for the 1D convolution problem, Journal of VLSI and Computer systems, à paraître

[T26]* Y. ROBERT, M. TCHUENTE, Iterative behaviour of monotone networks, Proceedings of the NATO Advanced Research Workshop on Disordered Systems and Biological Organization, Springer Verlag , à paraître

[T27]* Y. ROBERT, M. TCHUENTE, Parallel solution of band triangular systems on VLSI arrays with limited fan-out, Proc. Conf. International Workshop on Modeling and Performance Evaluation of Parallel Systems, M. Becker ed., Springer Verlag, à paraître

[T28]* M. COSNARD, Y. ROBERT, D. TRYSTRAM, Résolution parallèle de systèmes linéaires denses par diagonalisation, RR IMAG Grenoble 552, Juillet 1985, à paraître Bulletin de l'EDF, série C, 1986

[T29]* Y. ROBERT, D. TRYSTRAM, Un réseau systolique orthogonal pour le problème du chemin algébrique, RR IMAG Grenoble 553, Juillet 1985

[T30]* M. COSNARD, Y. ROBERT, D. TRYSTRAM, Comparaison des méthodes parallèles de diagonalisation pour la résolution de systèmes linéaires denses, C. R. Acad. Sc. Paris, à paraître

[T31] M. COSNARD, Y. ROBERT, Complexity results for parallel linear systems solvers, SIAM Conference on Parallel Processing for Scientific Computing, Nov 18-21, 1985, Norfolk, VA, USA,

PARTIE 1

Exposés de Synthèse

CHAPITRE 2

COMPORTEMENT DYNAMIQUE DE RESEAUX D'AUTOMATES: DES EXEMPLES

Yves ROBERT
CNRS - Laboratoire TIM3/INPG
BP 68
38402 Saint Martin d'Hères Cedex

Résumé :

Les réseaux d'automates cellulaires sont des systèmes dynamiques discrets dont la construction est simple, mais leur comportement dynamique peut s'avérer complexe. On présente dans ce texte l'utilisation d'outils mathématiques variés pour étudier le comportement itératif de quelques réseaux particuliers.

1. INTRODUCTION

Un réseau d'automates peut être défini de façon très générale comme un ensemble de cellules (automates finis) interconnectées localement, et évoluant dans un temps discret par interactions réciproques. Nous envisageons ici des automates déterministes, i.e. tels que la donnée d'une fonction de transition permet de déterminer l'évolution de toutes les cellules en fonction de la configuration initiale du réseau.

Comme le souligne WOLFRAM [10], les réseaux d'automates ont été utilisés dans de nombreuses disciplines, car ils peuvent fournir des exemples de comportement dynamique complexe à partir de fonctions de transition très simples pour chacune des cellules.

Si le réseau est fini, le nombre de configurations l'est également. A partir d'une configuration initiale, l'évolution se poursuit jusqu'à parvenir en régime périodique.

Une approche classique consiste à déterminer le comportement itératif du réseau à partir d'hypothèses sur la fonction de transition et sur le graphe de connexion (qui modélise les interactions entre cellules). Les principaux problèmes posés sont la détermination des cycles limites (nombre, périodes) et une majoration de la valeur du transitoire (nombre d'états avant le régime permanent).

Malgré l'introduction d'outils mathématiques sophistiqués, aucune approche unifiée n'a pu être développée pour répondre aux questions précédentes dans le cas général. En revanche, dans certains cas particuliers, des techniques ad-hoc permettent de caractériser complètement l'évolution du réseau. Nous en donnons ici plusieurs exemples représentatifs, basés notamment sur l'utilisation du calcul polynômial, de l'arithmétique modulaire et de fonctions d'énergie de type Liapounov.

Nous commençons par quelques définitions et notations qui nous seront utiles tout au long du texte.

2. NOTATIONS

Nous considérons un réseau d'automates $\mathcal{A}$. L'ensemble de ses n cellules est noté $\mathcal{S} = \{S_1, S_2, ..., S_n\}$. L'état $S_i(t)$ de la cellule S_i à l'instant t est pris dans l'ensemble $Q = \{0,1,2,...,q\}$.

Pour $t \geq 0$, on note $\mathcal{S}(t) = (S_1(t), S_2(t), ..S_n(t)) \in Q^n$ la configuration de $\mathcal{A}$ à l'instant t; $\mathcal{S}(0)$ est la configuration initiale de $\mathcal{A}$. La fonction de transition du réseau (qui détermine les règles d'évolution de chaque cellule) est une fonction

$$F = (f_1,...,f_n) : x = (x_1,...,x_n) \in Q^n \dashrightarrow F(x) \in Q^n$$

telle que chaque f_i ne dépend que d'un nombre restreint de variables x_j. On note FN(i) l'ensemble des indices j pour lesquels f_i dépend de x_j: FN(i) représente le voisinage fonctionnel de la cellule i.

Le graphe de connexion $G_c(F)$ est défini comme suit: un arc du graphe de connexion relie la cellule j à la cellule i si et seulement si la cellule i dépend de la cellule j. Formellement, ceci équivaut donc à $G_c(F) = (\mathcal{S}, \Gamma)$, où $(j,i) \in \Gamma \Leftrightarrow j \in FN(i)$.

Le graphe d'itération de F est le graphe orienté $G_i(F) = (Q^n, F)$, où tout sommet $x \in Q$ possède un seul arc sortant, (x,F(x)). Ce graphe matérialise donc les relations de succession entre toutes les Q^n configurations possibles du réseau.

Pour illustrer ces notions, considérons l'automate à 5 cellules dont le graphe de connexion est donné ci-dessous (figure 1).

Chaque cellule peut prendre 2 états: on choisit $Q = \{0,1\}$. Pour tout i, la cellule i possède un voisinage fonctionnel de cardinal 3. Par exemple, $FN(1) = \{2,3,4\}$. La fonction de transition est la même pour chaque cellule, c'est la fonction de majorité à 3 variables (les 3 états des cellules du voisinage fonctionnel). Ainsi $S_1(t+1) = MAJ [S_2(t) , S_3(t) , S_4(t)]$.

Partant de la configuration initiale $\mathcal{S}(t) = (0,1,1,1,0)$, la figure 2 montre l'évolution du réseau. Après une seule itération, on aboutit à un cycle de longueur 2. La longueur du transitoire est donc égale à 1. Un résultat de GOLES [5] montre que dans notre exemple, le graphe d'itération ne comporte que des points fixes ou des cycles élémentaires de longueur 2.

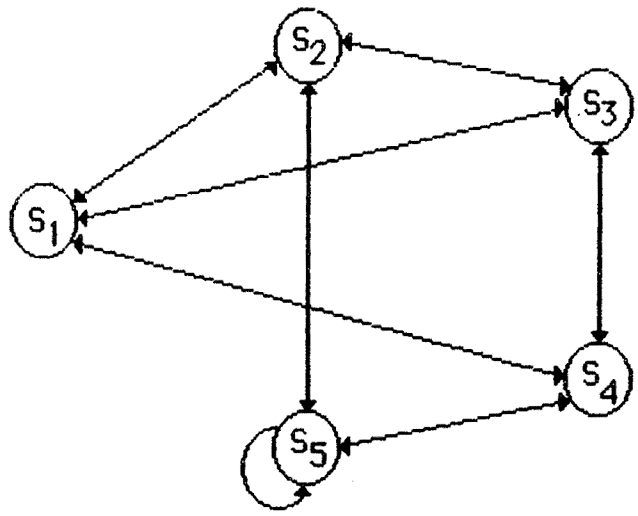


Figure 1 : Graphe de connexion

D'une manière générale, le graphe d'itération comprend plusieurs bassins d'attraction, chacun d'entre eux se terminant par un point fixe ou un cycle limite. Par contre, il existe des configurations n'admettant aucun prédécesseur: on ne peut les obtenir que comme configuration initiale, et non par itération du réseau. On appelle celles-ci des jardins d'Eden.

Pour définir de façon précise le transitoire et la période que l'on obtient à partir d'une configuration initiale $\mathcal{S}(0)$, nous observons qu'il existe deux nombres uniques T et P tels que

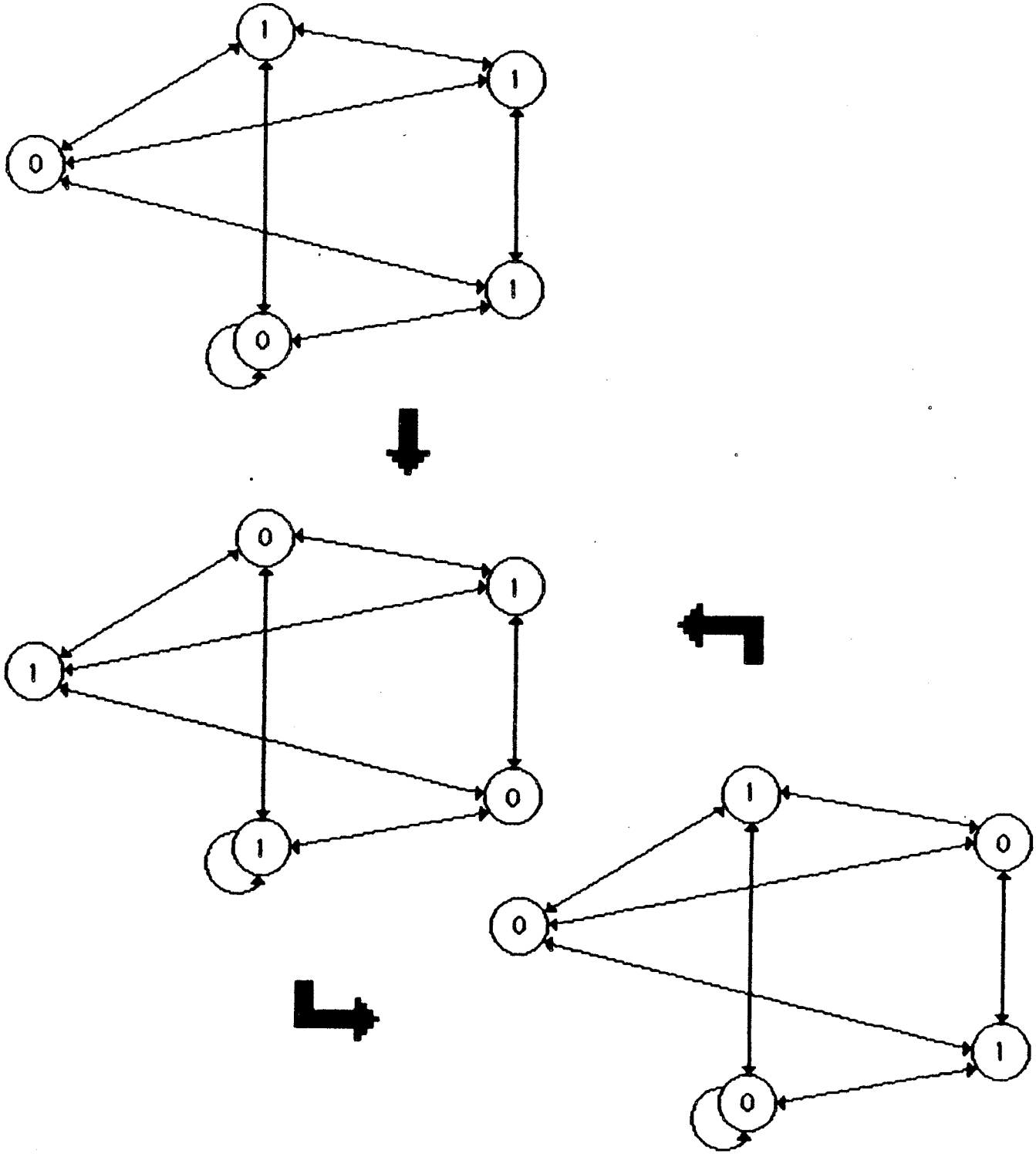
- $\mathcal{S}(t+P) = \mathcal{S}(t)$ pour tout $t \geq T$
- $\mathcal{S}(t+Q) \neq \mathcal{S}(t)$ pour tous $t < T$ et $Q < P$

T et P ainsi définis sont respectivement la longueur du transitoire et la période cherchées.

Enfin, nous diront qu'une fonction de transition F est monotone si

$$(x_i \leq y_i\ ,\ 1 \leq i \leq n) \Rightarrow (f_i(x) \leq f_i(y)\ ,\ 1 \leq i \leq n)$$

En d'autres termes, l'état d'une cellule à l'instant t+1 est une fonction croissante des variables dont elle dépend.



Convergence vers un cycle de longueur 2

Figure 2

3. AUTOMATES MODULAIRES

Le réseau $\mathcal{R}$ que nous considérons dans cette section comporte N cellules disposées en cercle, indicées de 0 à n-1. Ces indices seront toujours calculés modulo n dans ce qui suit.

Chaque cellule peut prendre pour valeur 0 ou 1. La configuration $\mathcal{S}(t) = (S_0(t),S_1(t),...,S_{n-1}(t))$ du réseau à l'instant t est représentée par le polynôme caractéristique

$$P(t) = \sum_{i=0}^{n-1} S_i(t).X^i$$

La fonction de transition est la suivante: l'état d'une cellule à l'instant t+1 est la somme modulo 2 des états de ses plus proches voisines, soit

$$S_i(t+1) = S_{i-1}(t) + S_{i+1}(t) \quad \text{modulo 2}$$

Cette règle correspond à la multiplication du polynôme caractéristique P(t) par

$$T(X) = X + X^{-1}$$

selon la relation

$$P(t+1) = T \,.\, P(t) \quad \text{modulo } (X^n-1)$$

La réduction modulo le polynôme X^n-1 permet d'éliminer les puissances négatives.

3.1. Principe de superposition additive

La règle d'évolution est additive, en ce sens que la configuration R(t) obtenue après t itérations à partir de la configuration initiale R(0) = P(0) + Q(0) esr R(t) = P(t) + Q(t), où P(t) et Q(t) sont les configurations obtenues en t itérations à partir de P(0) et Q(0), et où toutes les additions sont effectuées modulo 2.

Comme toute configuration initiale peut être représentée comme somme de configurations $\Delta_j(X) = X^j$ contenant une seule cellule j non nulle, le principe de superposition additive permet de déterminer n'importe quelle évolution en fonction de l'évolution de $\Delta_j(X)$. En vertu de la symétrie entre les cellules, il suffit de considérer le cas j=0.

Partant donc de $\Delta_0(X) = 1$, on obtient en t itérations du réseau

$$T(X)^t \cdot 1 = (X + X^{-1})^t = \sum_{i=0}^{t} \binom{n}{i} \cdot X^{2i-t} \quad \text{modulo } (X^n - 1)$$

Pour $t < n/2$ (avant que des "recouvrements" n'interviennent), la région des cellules non nulles croît linéairement avec le temps, et les états des cellules sont simplements donnés par les coefficients du binôme modulo 2.

3.2. Jardins d'Eden

Le théorème suivant précise la proportion de configurations qui n'ont pas de prédécesseurs:

lemme 1 : les configurations comportant un nombre impair de 1 sont des jardins d'Eden.

théorème 2 : le pourcentage de jardins d'Eden parmi les 2^n configurations possibles du réseau est 1/2 pour n impair et 3/4 pour n pair.

La démonstration fait appel à la factorisation du polynôme X^n-1 sur le corps $Z/_{2Z}$, et fait notamment intervenir les polynômes cyclotomiques.

Le théorème suivant montre que l'évolution du réseau est irréversible, puisqu'une configuration (qui n'est pas un jardin d'Eden) peut avoir plusieurs prédécesseurs: l'information sur l'état initial est donc perdue au cours du temps:

théorème 3 : les configurations qui ne sont pas des jardins d'Eden ont exactement 2 prédécesseurs pour n impair, et 4 pour n pair.

La démonstration utilise le lemme 4, qui a l'intérêt propre de montrer le type d'arguments algébriques que l'on utilise grâce à l'introduction du formalisme des polynômes:

lemme 4 : deux configuration P(t) et Q(t) conduisent à la même configuration $R(t+1) = T \cdot P(t) = T \cdot Q(t)$ après une itération du réseau si et seulement si $T \cdot (P(t) - Q(t)) = 0$ modulo X^n-1.

3.3. Longueur maximale des cycles

Le principe d'additivité conduit au

lemme 5 : la longueur des cycles du réseau divise la longueur t_n du cycle obtenu à partir d'une configuration initiale de type $\Delta_j(X)$.

Nous pouvons déterminer t_n suivant la forme de n:

proposition 6 : pour n de la forme $n = 2^m$, $t_n = 1$

Dans ce cas, toutes les configurations évoluent vers le même point fixe, qui est la configuration nulle. En effet,

$$(X + X^{-1})^n = X^{2^m} + X^{-2^m} \text{ (théorème de Fermat)}$$
$$= 0 \mod (X^n - 1)$$

proposition 7 . pour n pair mais non de la forme 2^m, $t_n = 2\, t_{n/2}$.

Reste à discuter le cas n impair:

théorème 8 : pour n impair, t_n divise $2^{sord(n)} - 1$, où sord(n) (mis pour "sous-ordre") est le plus petit entier j tel que $2^j = \pm 1 \mod n$.

Le théorème donne seulement une majoration. Le premier entier impair pour lequel on n'a pas égalité entre t_n et $2^{sord(n)}-1$ est n = 37, pour lequel $t_n = (2^{sord(n)}-1) / 3$.

Notons enfin que $sord(n) \leq (n-1)/2$, avec égalité possible si et seulement si n est premier. La période maximale du réseau est donc $2^{(n-1)/2} - 1$, valeur qui croît exponentiellement avec n.

Bien d'autres propriétés sont établies dans [7], en particulier sur la structure du graphe d'itération du réseau.

4. AUTOMATES A SEUIL SYMETRIQUES

La notion de réseau d'automates à seuil a été introduite en 1943 par Mac Culloch et Pitts [6] pour modéliser l'activité nerveuse: ces auteurs proposent un modèle de neurone formel, sous forme d'un automate à seuil dont les entrées représenteraient les neurones afférents, le sorties les neurones efférents et le seuil, le seuil d'excitabilité du neurone.

Pour un réseau $\mathcal{C}$ à n cellules, chaque composante f_i de la fonction de transition F est définie par

$$f_i(x_1,\ldots,x_n) = 1\Big(\sum_{j=1}^{n} a_{ij}.x_j - b_i\Big) \text{ pour tout } x = (x_1,\ldots,x_n) \in \{0,1\}^n$$

où $1(u) = \begin{cases} 1 \text{ si } u \geq 0 \\ 0 \text{ sinon} \end{cases}$

La matrice $A = (a_{ij})$ est une matrice d'ordre n à coefficients réels. Si $a_{ij}>0$, la cellule j excite la cellule i, et si $a_{ij}<0$, la cellule j inhibe la cellule i. Enfin, $b = (b_1, \ldots, b_n)$ est le vecteur des seuils.

Une première restriction technique est de se ramener à des fonctions à seuil strict: on modifie légèrement les coefficients de la matrice A pour assurer que $\Sigma_{1\leq j\leq n}\ a_{ij}.x_j - b_i$ ne s'annule jamais sur $\{0,1\}^n$.

Dans tout ce paragraphe, nous supposerons que la matrice A est symétrique. Autrement dit, l'influence d'une cellule j sur une cellule i est égale à l'influence réciproque de la cellule i sur la cellule j.

Pour une configuration initiale $\mathcal{S}(0)$, nous avons défini la longueur du transitoire $T(\mathcal{S})$ et la période $P(\mathcal{S})$. Notons $T(\mathcal{C})$ et $P(\mathcal{C})$ leur maximum respectif pour l'ensemble des 2^n configurations initiales possibles.

4.1. Cycles limites

théorème 9 : $P(\mathfrak{A}) \leq 2$

Ainsi le graphe d'itération du réseau ne comprend que des points fixes ou des circuits élémentaires de longueur 2. L'hypothèse de symétrie est capitale, car on peut exhiber des réseaux d'automates à seuil non symétriques pour lesquels la longueur maximale des cycles limites croît exponentiellement avec le nombre de cellules.

La démonstration du théorème 9 est basée sur l'utilisation d'une fonction d'énergie $E(x(t)) = \varphi(x(t),x(t-1))$, où le vecteur $x(t)$ représente la configuration $\mathcal{C}(t)$ et où φ est la forme quadratique définie par

$$\varphi(u,v) = -\sum_{i=1}^{n} u_i \sum_{j=1}^{n} a_{ij}.v_j + \sum_{i=1}^{n} b_i.(u_i + v_i)$$

On a $\Delta_t E = E(x(t+1)) - E(x(t))$

$$= -\sum_{i=1}^{n} [x_i(t+1) - x_i(t-1)].[\sum_{j=1}^{n} a_{ij}.x_j(t) - b_i]$$

(en utilisant la symétrie de A).

Par conséquent: - si $x_i(t+1) = x_i(t-1)$, on a $\Delta_t E = 0$

- si $x_i(t+1) \neq x_i(t-1)$, on a $\Delta_t E < 0$

Comme l'énergie E est une fonction décroissante, elle est nécessairement constante sur tout cycle limite, d'où le résultat.

4.2. Longueur du transitoire

L'utilisation de la fonction d'énergie E permet d'établir le

théorème 10 : (i) si A est une matrice à coefficients entiers, alors

$$T(\mathcal{R}) \leq \sum_{i,j}^{n} |a_{ij}| + 2 \sum_{i}^{n} |b_i|$$

(ii) si de plus $a_{ij} \in \{-1,0,1\}$ pour tous i et j, alors

$T(\mathcal{R}) \leq 3n^2$

En outre, si $|FN(i)| \leq v$ pour tout i, alors $T(\mathcal{R}) \leq 3nv$.

Ainsi, pour un réseau dont les coefficients sont tous égaux à -1, 0 ou 1, et dont chaque cellule a au plus v voisins (le degré du graphe de connexion $G_C(F)$ est v), la longueur du transitoire est linéaire en n.

Notons finalement que différents modes d'évolution sont possibles: plutôt que de calculer les nouvelles valeurs de toutes les cellules en parallèle, on peut envisager des itérations séquentielles, mixtes, ... Ici encore, des fonctions d'énergie ad-hoc ont été proposées [3] [5] pour analyser le comportement dynamique du réseau.

5. AUTOMATES A MEMOIRE

Nous considérons ici un réseau composé d'une seule cellule, mais avec mémoire: l'état S(t+1) de la cellule dépend de S(t), S(t-1), ..., S(t-n+1). En fait, le réseau peut être vu comme un réseau comprenant n cellules sans mémoire S_0, S_1, ..., S_{n-1} disposées en circuit comme indiqué figure 3. Chaque cellule S_i, $i \geq 1$, agit simplement comme une cellule de transfert, qui copie la valeur précédente de la cellule S_{i-1}, i.e. $S_i(t+1) = S_{i-1}(t)$ pour $t \geq 1$.

Le vecteur $Y(t) = (S(t) , S(t-1) , \ldots , S(t-n+1))$ représente alors l'état du réseau à l'instant t. Notant $y_0(t)$, ..., $y_{n-1}(t)$ ses composantes, la règle d'évolution est donnée par

$$y_0(t+1) = F(y_0(t) , y_1(t) , \ldots , y_{n-1}(t))$$
$$y_i(t+1) = y_{i-1}(t) \text{ pour } i \geq 1$$

Nous allons étudier le comportement dynamique de l'automate pour trois exemples de fonction de transition F: F fonction linéaire sur $Z_{/pZ}$, F fonction à seuil, F fonction booléenne monotone.

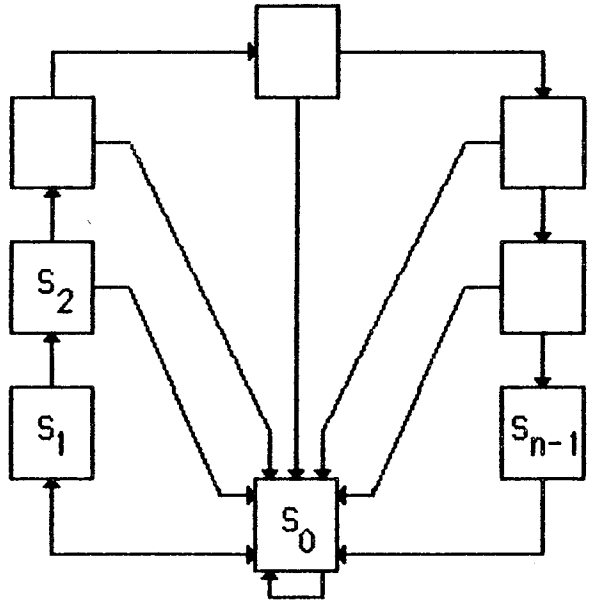


Figure 3 : Automate à mémoire

5.1. Circuits linéaires séquentiels

On choisit ici $F(y_0,y_1,\ldots,y_{n-1}) = -a_0.y_0 - a_1.y_1 - \ldots - a_{n-1}.y_{n-1}$ modulo un nombre premier p, ce qui correspond à la récurrence linéaire

$$S(t+1) = -a_0.S(t) - a_1.S(t-1) - \ldots - a_{n-1}.S(t-n+1)$$

sur le corps $Z_{/pZ}$.

On modélise ainsi un circuit linéaire séquentiel dont l'étude est importante dans de nombreux problèmes d'automatique [4]. Matriciellement, on a la formulation

$$Y(t+1) = M \,.\, Y(t),$$

où M est la matrice compagnon

$$M = \begin{bmatrix} -a_0 & -a_1 & -a_2 & \cdots & -a_{n-1} \\ 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ & & & & & 1 \end{bmatrix}$$

de polynôme caractéristique

$$\varphi(X) = a_{n-1} + a_{n-2}.X + \ldots + a_1.X^{n-2} + a_0.X^{n-1} + X^n$$

- cas où $a_{n-1} \neq 0$:

  On dit alors que le circuit est non-singulier. Soit T l'exposant du polynôme caractéristique $\varphi(X)$, qui est défini comme le plus petit entier i tel que $\varphi(X)$ divise $1-X^i$. Rappelons que toute l'arithmétique est calculée modulo p; l'existence de T est assurée par le fait que X ne divise pas $\varphi(X)$. Le théorème suivant caractérise la dynamique de l'automate:

  théorème 11 : le graphe d'itération ne comporte que des cycles (pas de transitoire). La longueur de tous les cycles divise T, et il existe au moins un cycle de longueur T.

  Bien mieux, si $\varphi(X)$ est irréductible, tous les configurations non nulles sont de période T.

• cas général

On a alors $\varphi(X) = X^r . \varphi'(X)$, avec φ' non divisible par X. Les suites générées par l'automate sont alors exactement celles générées par l'automate associé à φ', précédées de r valeurs arbitraires.

On a toujours $T \leq p^n - 1$. Mais pour toute valeur de n et pour tout entier premier p, il existe un polynôme irréductible de degré n et d'exposant maximal $T = p^n - 1$. Toutes les suites n'évoluant pas vers la configuration nulle sont alors de période maximale, en vertu du théorème précédent.

5.2. Equation neuronale

On choisit maintenant comme fonction F une fonction à seuil

$$F(y_0,y_1,\ldots,y_{n-1}) = 1\Big(b - \sum_{j=0}^{n-1} a_j.y_j\Big)$$

On modélise ainsi un "neurone formel" à 2 états 0 et 1, où pour calculer un nouvel état on pondère linéairement les n états précédents. L'approche du paragraphe 4 ne peut s'utiliser, puisque le graphe de connexion n'est plus symétrique. Le cas général où les coefficients a_j sont quelconques est ouvert, mais on trouvera dans [1] diverses conjectures sur la longueur maximale des cycles, motivées tant par des expérimentations numériques que par des résultats théoriques traitant de cas particuliers.

Nous allons présenter l'un d'entre eux, relatif aux cas d'une mémoire palindromique: les coefficients a_j vérifient $a_i = a_{n-1-i}$ pour $0 \leq i \leq n-1$.

théorème 12 : la période des cycles de l'automate divise n

Comme précédemment, on associe à chaque cycle de longueur T C=(y(0),y(1),...,y(T)) une fonction

$$E(C) = \sum_{t=0}^{T-1} y(t+1).\Big(\sum_{s=0}^{n-1} a(n-1-s).y(t+s) - \sum_{s=0}^{n-1} a(s).y(t-s)\Big)$$

où les indices sont pris modulo T.

On prouve alors que $E(C) > 0$ si T ne divise pas n, d'où le résultat cherché.

Réciproquement, il est facile de construire des cycles ayant pour longueur un diviseur quelconque de n [1].

5.3. Fonctions monotones

On suppose simplement ici que la fonction de transition F est booléenne monotone. Cette hypothèse est vérifiée pour l'automate précédent dans le cas où tous les coefficients a_j sont négatifs. Mais la classe des fonctions monotones est plus générale, comme le montre le

théorème 13 : on peut construire une fonction monotone F admettant des cycles de longueur $\geq 2^{n/3}$.

Signalons simplement que la démonstration repose sur la construction de suites booléennes incomparables de longueur maximale [8].

6. CONCLUSION

Les exemples que nous avons étudiés mettent en relief la richesse du comportement dynamique des réseaux d'automates, alors que ce comportement est la simple résultante d'interactions locales élémentaires entre les cellules du réseau.

La variété des outils mathématiques utilisés pour l'analyse itérative des réseaux d'automates reflète bien la complexité de ce modèle.

Commentaires bibliographiques

La présentation générale du modèle des réseaux d'automates s'inspire de nombreux textes: [2], [3], [5], [9] et [10]. Les résultats de la section 3 (automates modulaires) proviennent de [7], ceux de la section 4 (automates à seuil) sont tirés de [3] et [5]. L'étude des sections 5.1., 5.2. et 5.3. (automates à mémoire) est menée respectivement dans [4], [1] et [8].

Enfin ce texte doit beaucoup aux séances du groupe de travail "Comportement d'itérations", animé par F. ROBERT, où il est question de réseaux d'automates depuis ... de nombreuses années !

REFERENCES

[1] M. COSNARD, E. GOLES, Dynamical properties of an automaton with memory, NATO Advanced Research Workshop on Disordered systems and Biological Organization (1985), à paraître Springer Verlag

[2[M. COSNARD, F. ROBERT, Y. ROBERT, M. TCHUENTE, Les réseaux d'automates: une introduction, Journées CNRS "Calculs sur réseaux d'automates: théorie et applications", Paris, 19-20 Septembre 1985

[3] F. FOGELMAN, Contributions à une théorie du calcul sur réseaux, Thèse, Grenoble (1985)

[4] A. GILL, Linear sequential circuits, Analysis, Synthesis and Applications, Mc Graw-Hill Series in Computer Science, New York, 1966

[5] E. GOLES, Comportement dynamique de réseaux d'automates, Thèse, Grenoble (1985)

[6] W.S. MAC CULLOCH, W.M. PITTS, A logical calculus of the ideas immanent in nervous activity, Bull. of Mathematical Biophysics 5 (1943), 115-133

[7] O. MARTIN, A. ODLYZKO, S. WOLFRAM, Algebraic properties of cellular automata, Bell Laboratories Report (1983)

[8] Y. ROBERT, M. TCHUENTE, Iterative behaviour of monotone networks, NATO Advanced Research Workshop on Disordered Systems and Biological Organization (1985), à paraître Springer Verlag

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

[10] S. WOLFRAM, Preface to Physica 10D (1984), Elsevier Science Publishers

CHAPITRE 3

ARCHITECTURES SYSTOLIQUES : UNE INTRODUCTION

Yves ROBERT
CNRS - Laboratoire TIM3/INPG
BP 68
38402 Saint Martin d'Hères Cedex

Résumé :

On présente le modèle systolique introduit par H.T. KUNG et C.E. LEISERSON. Il s'agit de réseaux de processeurs intégrés spécialisés conçus pour des applications "temps réel". Après une introduction générale, on présente en détail quelques architectures systoliques simples. On discute ensuite des performances et de l'intérêt du modèle. Dans une dernière partie plus théorique, on s'intéresse à des méthodes systématiques de conception et de validation des réseaux systoliques.

1. INTRODUCTION

Pour augmenter les performances des calculateurs, l'approche traditionnelle consiste à rechercher une augmentation de la vitesse des composants matériels et repose sur un système d'exploitation sophistiqué pour minimiser les périodes d'horloge. Aujourd'hui, cette approche est remise en cause. Tout d'abord, l'industrie des semi-conducteurs est, d'une manière générale, plus à même de développer des technologies à très haute densité d'intégration que de réaliser des circuits ultra rapides. Ensuite, on a atteint des limites physiques qui semblent incontournables jusqu'à l'arrivée sur le marché des technologies très rapides de la prochaine génération (à l'Arsenide de Gallium GaAs); de plus, le prix à payer pour gagner un ordre de grandeur sur la vitesse des composants en utilisant ces nouvelles technologies sera très vraisenblablement prohibitif pour la plupart des applications.

Plutôt que de compter seulement sur l'augmentation de la vitesse des composants, une autre possibilité est d'opter pour des systèmes parallèles qui pourront être implantés efficacement sous forme de circuits intégrés VLSI (Very Large Scale Integration): la complexité de conception de ces circuits doit rester dans le cadre des meilleures possibilités industrielles.

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

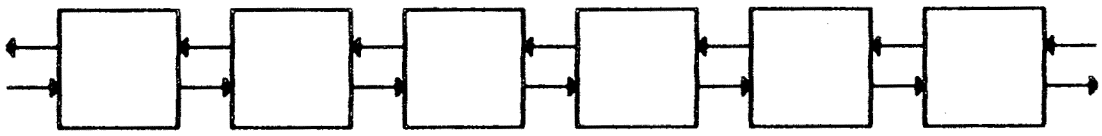
- il s'agit de processeurs spécialisés que l'on adjoint à un processeur hôte de type conventionnel
- la classe d'application est bien délimitée: les problèmes où le volume de calculs à effectuer prime largement sur les transferts de données à réaliser.

Le modèle systolique, introduit en 1978 par KUNG et LEISERSON, s'est révélé être un outil puissant pour la conception de processeurs intégrés spécialisés. En un mot, une architecture systolique est agencée en forme de réseau (voir les schémas de la page suivante). Ces réseaux se composent d'un grand nombre de cellules élémentaires identiques et localement interconnectées. Chaque cellule reçoit des données en provenance des cellules voisines, effectue un calcul simple, puis transmet les résultats, toujours aux cellules voisines, un temps de cycle plus tard. Pour fixer un ordre de grandeur, disons que chaque cellule a la complexité, au plus, d'un petit microprocesseur.

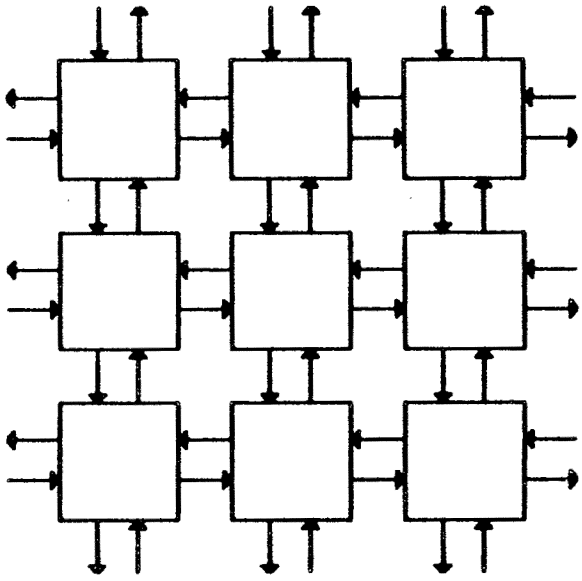
Les cellules évoluent en parallèle, en principe sous le contrôle d'une horloge globale (synchronisme total): plusieurs calculs sont effectués simultanément sur le réseau, et on peut "pipeliner" la résolution de plusieurs instances du même problème sur le réseau.

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

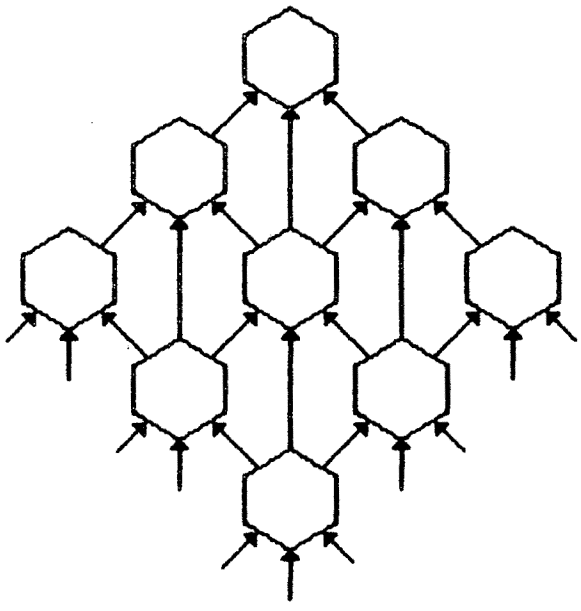
Avant de revenir en détail sur les caractéristiques de ce modèle, et sur son intérêt pour la réalisation de circuits spécialisés, nous présentons en détail le fonctionnement de réseaux systoliques résolvant quelques problèmes d'algèbre linéaire.



Réseau linéaire



Réseau orthogonal



Réseau hexagonal

2. QUELQUES EXEMPLES DETAILLES

Dans cette partie, on traite de l'exemple simple (et déjà classique) de la convolution non récursive, puis on envisage la modélisation de problèmes plus compliqués ...

2.1. Convolution non-récursive

Partons du problème suivant: étant donné une suite $x_1, x_2, x_3, \ldots$ de données, calculer pour tout $i \geq k$ la valeur de

$$y_i = a_1 * x_i + a_2 * x_{i-1} + a_3 * x_{i-2} + \ldots + a_k * x_{i-k+1},$$

où les poids $a_1, a_2, \ldots, a_k$ sont des coefficients fixés.

Une solution bien connue en théorie du filtrage est décrite figure 1 (avec k=4). Notons qu'un nouvel y_i est calculé tous les temps de cycle τ_k, mais que ce temps de cycle doit être assez long pour permettre la réalisation d'une multiplication et de k-1 additions.

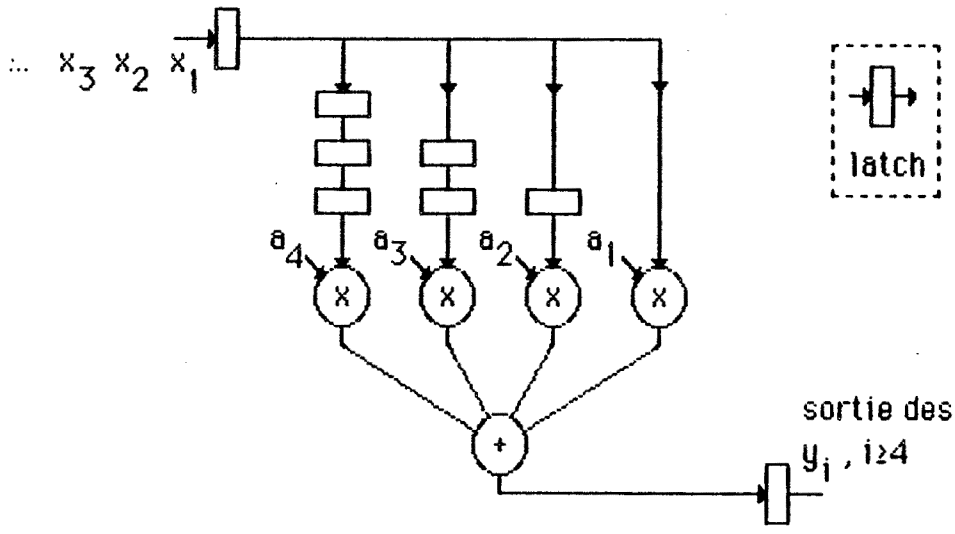
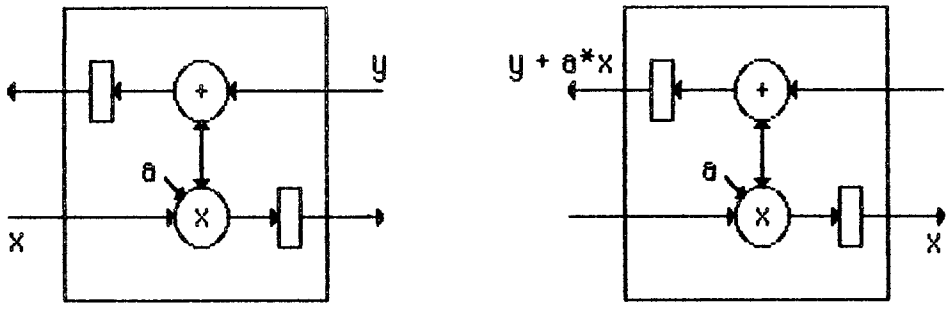


Figure 1 : Première solution

Le réseau (systolique !) de la figure 2 est composé de k cellules identiques dont le fonctionnement est le suivant:



Temps t *Temps t+1*

Sur ce réseau, un nouvel y_i n'est délivré en sortie qu'un temps de cycle sur 2, mais le temps de cycle τ_{syst} du réseau est celui de chaque cellule, correspondant à une multiplication suivie d'une addition. Pour k assez grand, $\tau_k > 2\tau_{syst}$, et le réseau systolique est plus performant. Bien mieux, τ_{syst} ne dépend pas de k, la taille du filtre. Quelques pulsations de ce réseau sont données figure 3, pour illustrer le principe du calcul "par accumulations successives": dans chaque cellule du réseau, y_i effectue un calcul partiel $y_i := y_i + a_j * x_{i-j+1}$, et plusieurs variables y_i sont calculées en parallèle: le réseau fonctionne en pipe-line.

La principale faiblesse du réseau présenté est que les cellules ne sont actives que tous les deux tops (on dit que l'efficacité est e = 1/2). On reviendra sur ce problème plus loin.

2.2. Convolution récursive

Soit maintenant à calculer l'expression récurrente suivante

$$y_i = a_1 * y_{i-1} + a_2 * y_{i-2} + a_3 * y_{i-3} + \ldots + a_k * y_{i-k}$$

pour $i \geq k+1$; $y_1, y_2, \ldots, y_k$ sont des valeurs initiales données; moyennant l'ajout d'une cellule de retard, le même réseau systolique peut être utilisé, comme indiqué figure 4: quand une variable y_i circule de droite à gauche, elle est en train d'accumuler sa valeur définitive, puis elle rebondit dans la cellule de retard pour circuler inchangée en sens inverse, jouant le rôle des x_i dans le réseau précédent.

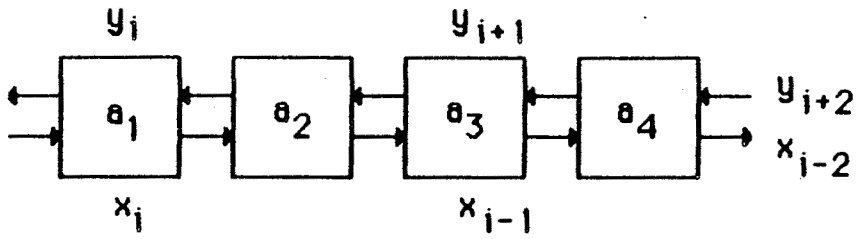


Figure 2 : Convolution non récursive

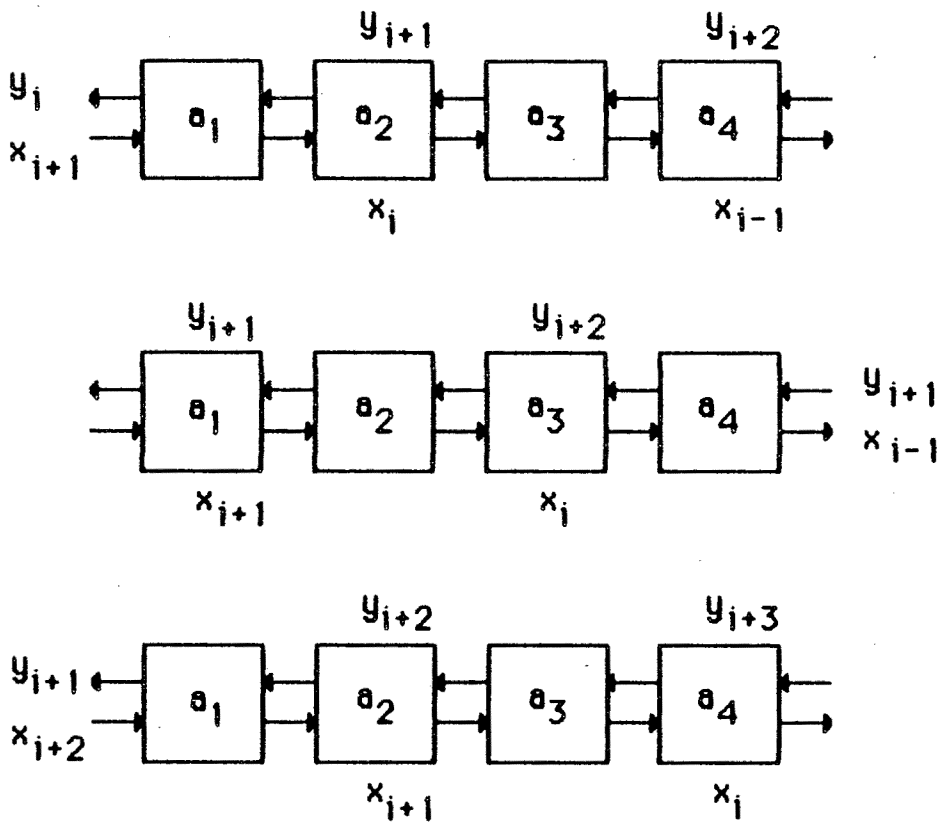


Figure 3 : Quelques pulsations

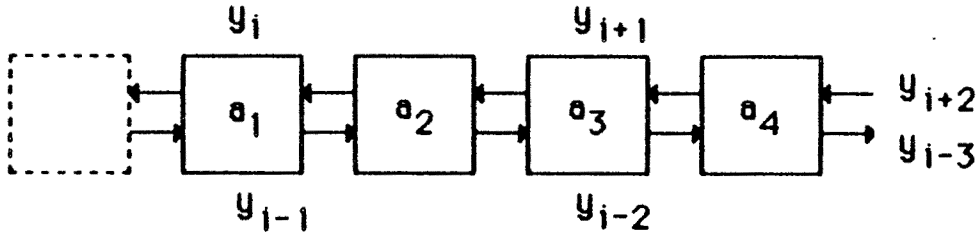


Figure 4 : Convolution récursive

Le problème général de la convolution récursive (ou filtrage à réponse impulsionnelle infinie)

$$y_i = a_1*x_i + a_2*x_{i-1} + a_3*x_{i-2} + \dots + a_k*x_{i-k+1} + r_1*y_{i-1} + r_2*y_{i-2} + r_3*y_{i-3} + \dots + r_h*y_{i-h}$$

pour $i \geq Max(k,h+1)$ et $y_1, y_2, \dots, y_h$ donnés, se résoud tout aussi simplement sur un réseau comportant k+h cellules de calcul et une cellule de retard (voir la figure 5).

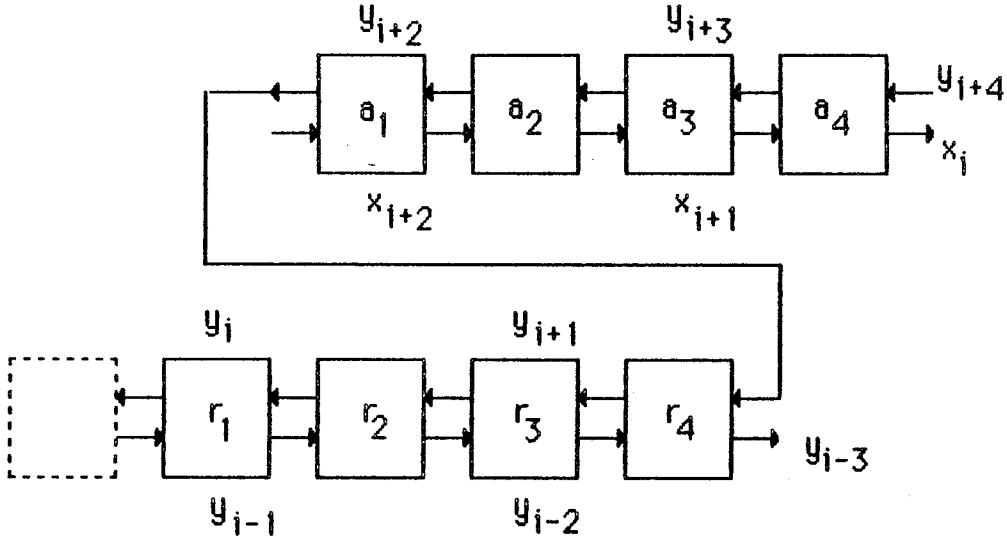


Figure 5 : Convolution récursive générale

2.3. Multiplication matrice bande-vecteur

Soit $A = (a_{ij})$ une matrice carrée d'ordre n à structure bande, de largeur $w = p+q-1$, et x un vecteur à n composantes. Les composantes du produit $y = Ax$ se calculent par

$$y_i = a_{i,i-p}*x_{i-p} + \dots + a_{i,i-1}*x_{i-1} + a_{i,i}*x_i + a_{i,i+1}*x_{i+1} + \dots + a_{i,i+q}*x_{i+q}$$

Le problème est analogue à la convolution récursive, mais les poids $a_1, \dots, a_k$ varient pour chaque y_i. On modifie simplement les cellules précédentes en leur adjoignant un port vertical pour recevoir les poids changés dynamiquement tous les deux temps de cycle, et on obtient le réseau de la figure 6.

Ce réseau qui comporte w cellules de calcul délivre en sortie toutes les composantes de y en 2n+w temps de cycle.

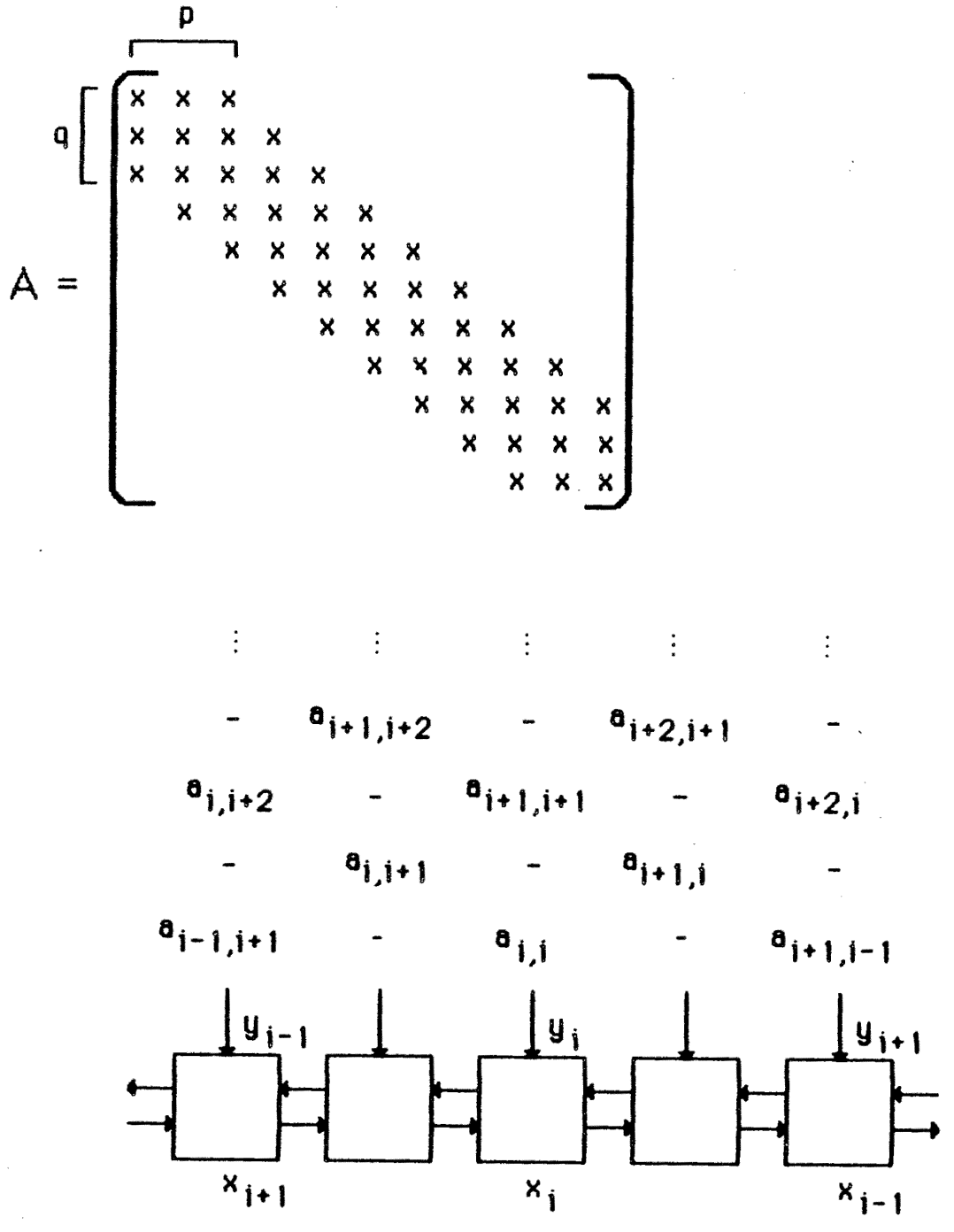


Figure 6 : Multiplication matrice-vecteur

2.4 Système linéaire triangulaire

C'est le problème inverse du précédent: étant donnée une matrice triangulaire inférieure A avec q sous-diagonales et un vecteur b, calculer la solution x du système linéaire Ax = b.

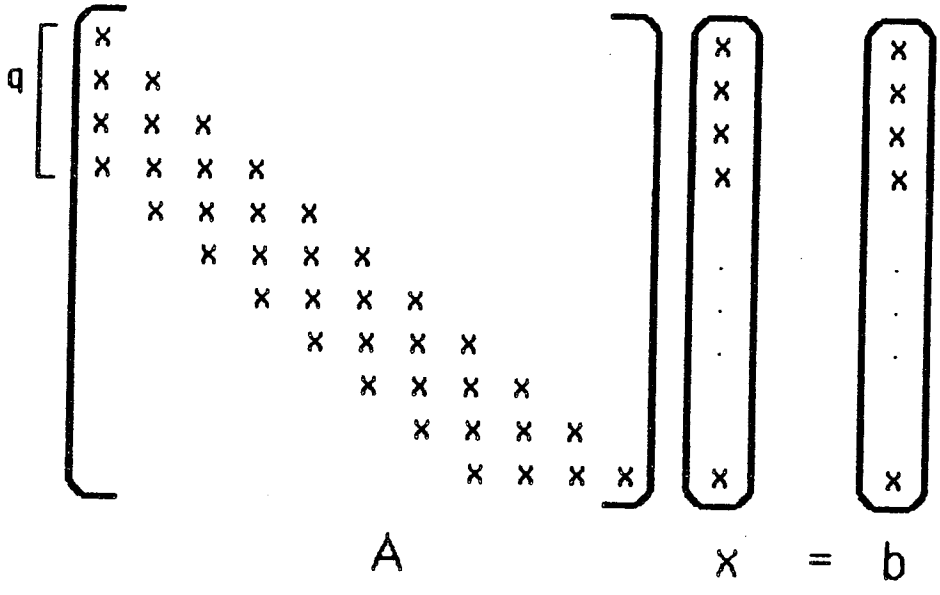
On commence par reprendre le réseau de la figure 6, en supprimant les cellules relatives à la partie triangulaire supérieure de la matrice. Puis on modifie la cellule qui reçoit la diagonale pour lui faire calculer une soustraction suivie d'une division au lieu d'une multiplication suivie d'une addition. Enfin, tout comme pour la convolution récursive, quand une variable x_i qui circule de droite à gauche (en train d'accumuler sa valeur définitive) atteint la cellule diagonale, elle est renvoyée après calcul pour circuler inchangée en sens inverse.

On obtient ainsi le réseau de la figure 7, qui comporte w = q cellules et résoud le système linéaire en temps 2n + w.

2.5 Un exemple bi-dimensionnel

Le dernier réseau présenté est bidimensionnel, il s'agit du produit de deux matrices denses. Soit donc à calculer le produit C = A . B, où A et B sont de taille 3x3. Pour calculer un élément de C, disons c_{11}, on doit réaliser le produit scalaire de la première ligne de A et de la première colonne de B. Un réseau R_{diag} de 3 cellules fera l'affaire (figure 8). Juxtaposant 2 autres réseaux identiques à la droite de R_{diag}, on effectue le produit scalaire de la première ligne de A par les colonnes 2 et 3 de B, calculant ainsi c_{12} et c_{13}. Pipelinant le calcul en envoyant les lignes 2 et 3 de A à la suite de la première, on peut aussi calculer les éléments c_{22}, c_{23}, et c_{33}. Enfin, pour obtenir la partie triangulaire inférieure du produit C, il suffit de juxtaposer, à gauche cette fois de R_{diag}, deux copies supplémentaires. Le réseau obtenu est décrit à la figure 9.

Dans le cas général, un réseau de (2n-1)*n cellules (2n-1 réseaux de produit scalaire, un par diagonale) permet le calcul du produit de deux matrices de taille n en temps 4n * τ_{syst} (n tops d'initialisation, n tops pour compléter le calcul de c_{11}, 2n tops pour obtenir c_{nn}).



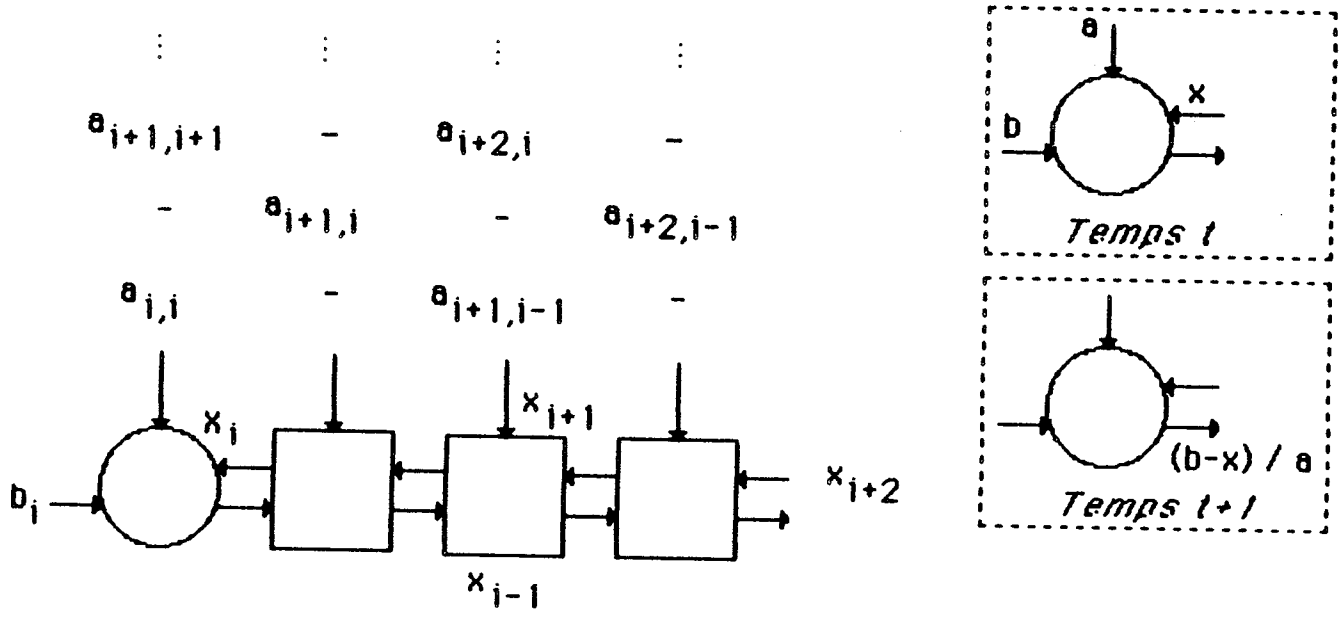


Figure 7 : Système triangulaire

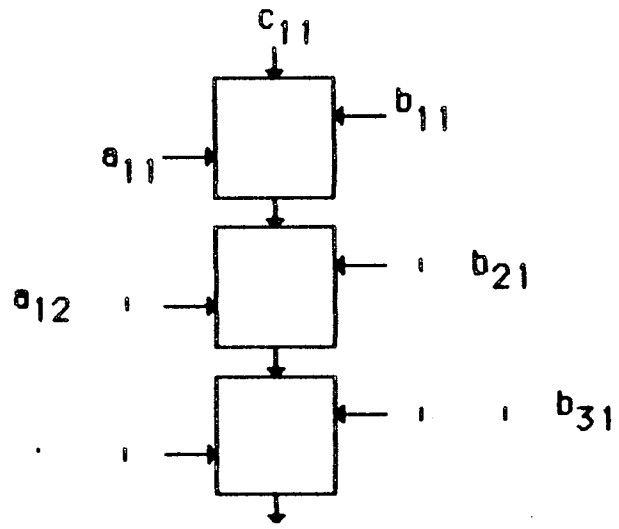


Figure 8 : Calcul d'un produit scalaire

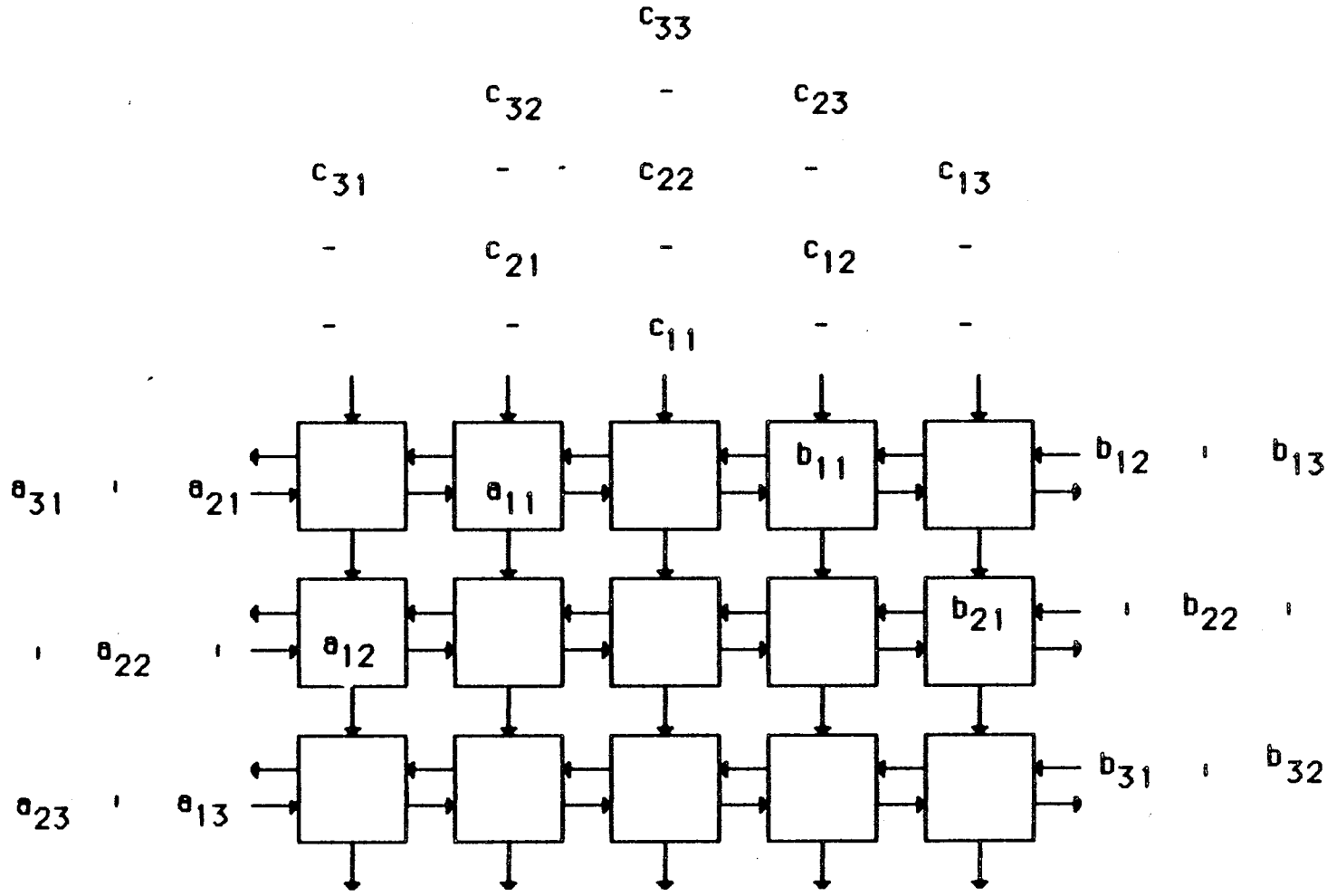


Figure 9 : Produit de matrices denses

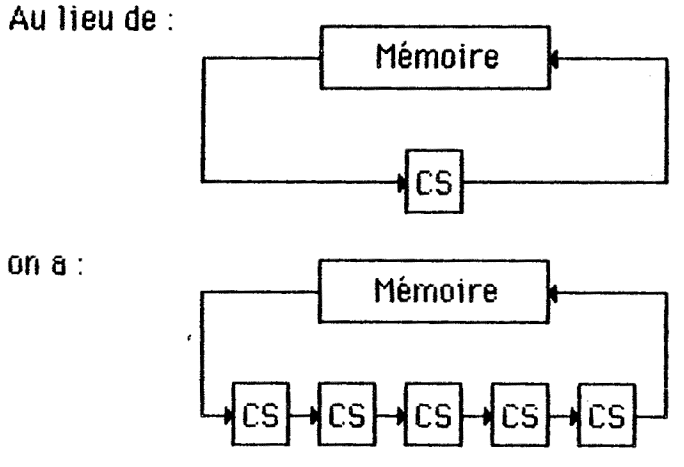
3. POURQUOI DES ARCHITECTURES SYSTOLIQUES ?

Le titre de ce paragraphe reprend (en Français) celui d'un article de H.T. KUNG qui fait référence pour présenter le modèle systolique. Pour l'instant, nous nous contentons d'une définition informelle du terme "systolique": une architecture systolique est un réseau régulier de processeurs relativement simples . Les données, c'est-à-dire aussi bien les entrées du réseau que les résultats partiels ou définitifs, circulent à travers la structure de façon synchrone et selon un cheminement fixe.

3.1. Concurrence et communication

Comme on l'a vu dans les exemples précédents, plusieurs calculs sont effectués sur une même donnée à l'intérieur du réseau: une fois qu'une donnée en provenance de la mémoire externe est lue par le réseau, elle passe de cellule en cellule et peut donc être utilisée de nombreuses fois. Ce mode de calcul est à l'opposé du fonctionnement basé sur un accès mémoire à chaque utilisation d'une donnée, caractéristique des machines de type Von Neumann.

En conséquence, un réseau systolique peut être étendu (en adjoignant des cellules) pour traiter un problème coûteux en nombre d'opérations sans qu'il faille imposer pour autant une augmentation correspondante de la largeur de bande de la mémoire externe.



CS = Cellule systolique

Figure 10 : Principe des réseaux systoliques

Cette propriété, illustrée par la figure 10, confère aux réseaux systoliques un avantage majeur sur les architectures traditionnelles, qui sont limitées par le "goulôt d'étranglement de Von Neumann".

3.2. Simplicité et régularité

Réaliser des circuits spécialisés à un coût raisonnable est une préoccupation majeure pour les concepteurs VLSI: le coût de conception d'un tel circuit doit être assez bas pour pouvoir être amorti sur un faible volume de production. Comme on l'a vu dans les exemples précédents, les architectures systoliques sont composées à partir d'un petit nombre de cellules de base, premier avantage sur une architecture composée d'une grande variété de cellules complexes. Deuxième avantage, l'interconnexion locale et régulière des cellules facilite grandement l'implantation topologique. Dans le cas le plus simple, celui des réseaux linéaires, on peut même dire que tout algorithme systolique conduit directement à un schéma de réalisation sur silicium.

Reprenons l'exemple de la figure 1: les données (x_i) sont propagées le long d'un bus de données, et dupliquées pour alimenter les quatre multiplieurs. Le dessin d'un circuit de convolution avec 5 coefficients (ou plus!) à partir du précédent n'est pas aisé, alors qu'avec la solution systolique de la figure 2 c'est immédiat. D'une manière générale, la modularité et le faible "fan-out" (nombre maximal de copies d'une même variable crées à l'intérieur du réseau) des réseaux systoliques leur donne une grande reconfigurabilité: elles sont adaptables à la taille et à la nature du problème traité.

Enfin, il importe de dire un mot de la testabilité et de la forte résistance aux pannes des architectures systoliques. Les mêmes arguments que précédemment jouent en leur faveur dans ces domaines:
- pour le test: si les cellules sont identiques, il suffit d'en tester une seule
- pour la résistance aux pannes: considérons un circuit comportant plusieurs rangées de cellules: on peut prévoir des mécanismes d'interrupteurs permettant de contourner les cellules qui se révéleraient défaillantes après réalisation. Il s'agit alors de reprogrammer le réseau en n'utilisant que les cellules valides, ce qui est facilité par la localité des connexions entre cellules.

Ces techniques prennent une importance grandissante avec l'arrivée prochaine des plaquettes "wafer-scale" où l'on intègre un très grand nombre de cellules sur la même puce de silicium, et où le taux de déchet (cellules défaillantes) reste important (pour l'instant, entre 50 et 80 %).

3.3 Calculs intensifs

Les systèmes VLSI sont adaptés à l'implantation d'algorithmes "compute-bound" (peut-on proposer une traduction satisfaisante?) plutôt qu'à la résolution de problèmes "I/O-bound". Dans un algorithme compute-bound, le nombre de calculs élémentaires est plus grand que le nombre de données en entrée-sortie. Sinon, le prolème est I/O-bound, et ne se prète pas à un traitement VLSI, où le nombre de ports d'entrée-sortie est limité.

Par exemple, la multiplication de deux matrices de taille n nécessite $O(n^3)$ multiplications et additions pour $O(n^2)$ données, c'est un problème compute-bound. Au contraire, l'addition de deux matrices exige n^2 additions pour $3n^2$ opérations d'entrée-sortie, et est donc I/O-bound.

Les caractéristiques des architectures systoliques conduisent dans la plupart des problèmes compute-bound à des calculs en temps réel (c.-à-d. où les sorties sont délivrées au même rythme que les entrées). De fait, de telles architectures se sont avérées très performantes pour la résolution de nombreux problèmes où le volume de calcul est très important, et le traitement local et régulier (voir la figure 11).

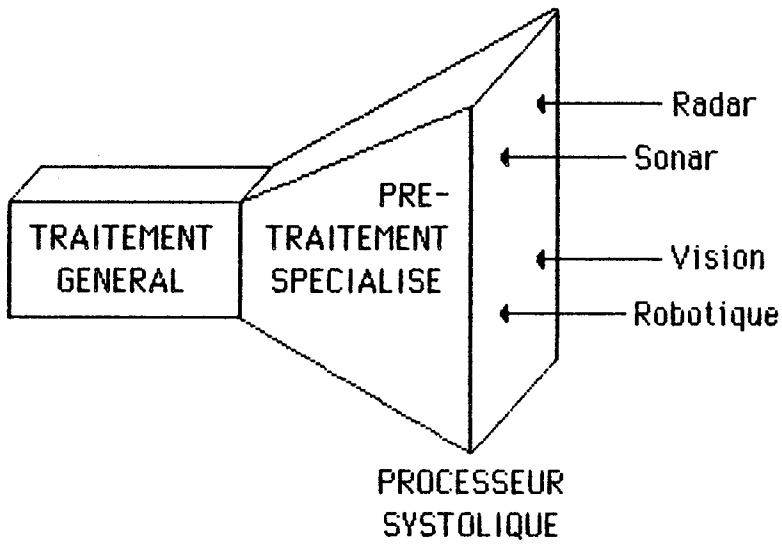


Figure 11

3.4 Peu de problèmes ont résisté à la systolisation

Même si tous les algorithmes proposés dans la littérature ne sont pas implémentés (ou implémentables) en VLSI, un tour d'horizon des réseaux existants a le mérite de montrer l'étendue du champ d'application de l'algorithmique systolique:

- traitement du signal et de l'image: filtrage, convolution 1D et 2D, corrélation, lissage par médiane 1D et 2D, transformée de Fourier discrète, projections géométriques, encodage/décodage pour les corrections d'erreur, ...

- arithmétique matricielle: multiplication matrice-vecteur; matrice-matrice, triangularisation de matrice (résolution de systèmes linéaires, calcul de l'inverse), décomposition QR (calcul aux moindres carrés, inversion de matrice de covariance), problèmes aux valeurs propres, décompsition SVD ("singular value decomposition"), ...

- applications non-numériques:

- structures de données: pile, file d'attente, recherche dans un dictionnaire, tri
- graphes et algorithmes géométriques: fermeture transitive, arbre de degré minimum, composantes connexes, enveloppes convexes
- manipulation de chaînes de caractères: occurences d'un mot, plus longue sous-suite, reconnaissance de langages réguliers
- programmation dynamique
- opérations sur des bases de données relationnelles
- algèbre des polynômes: multiplication, division euclidienne, PGCD
- arithmétique entière et dans des corps finis: multiplication, division, PGCD
- simulation Monte-Carlo

3.5 Quelques circuits systoliques

De nombreuses réalisations ont vu le jour. On peut distinguer 4 types de circuits systoliques:
- les circuits spécialisés: le corrélateur systolique de GEC (1983), le convolveur bit-série de CMU (1981), le réseau de convolution de Honeywell (1983), ...

- les circuits multi-applications: le processeur systolique ESL (1982), ...
- les systèmes programmables: le circuit de reconnaissance de la parole des Bell Labs (1981), celui de reconnaissance de langages réguliers CMU (1983), ...
- les blocs fonctionnels programmables: le PSC de CMU (1983), le circuit norvégien bit-sliced (1983), ...

Signalons que la firme NCR vient de mettre sur le marché un réseau systolique bi-dimensionnel à connections orthogonales. La configuration de base est 6 x 12, la technologie est CMOS double métallisation. D'autre part, l'université de Carnegie-Mellon annonce pour cette année une carte systolique programmable avec arithmétique flottante ... on est en train de passer du stade du prototype à celui des applications industrielles !

4. UN BRIN DE FORMALISATION

On aborde dans ce paragraphe une approche systématique qui permet de donner un sens précis au mot "systolique". On fait ensuite quelques remarques sur les problèmes de construction et validation automatique des architectures systoliques.

4.1. Une méthode de sytolisation

L'approche que nous présentons ici est due à C.E. LEISERSON et J.B. SAXE. Un circuit est défini formellement comme un graphe orienté G = (V,U), dont les sommets représentent les éléments fonctionnels du circuit. Un sommet particulier représente la structure hôte, qui permet au circuit de communiquer avec le monde extérieur.

Chaque sommet v de G est affecté d'un poids d(v) positif qui est le temps de cycle de la cellule qu'il représente; chaque arc e=(v,v') de U est affecté d'un poids entier qui représente le nombre de registres (ou latches) que doit traverser une donnée pour aller de v à v' (figure 12).

Les circuits systoliques sont alors ceux pour lesquels chaque arc porte au moins un registre; leur synchronisation peut ainsi s'effectuer globalement au rythme d'une horloge dont le temps de cycle est $Max_{v \in V} d(v)$.

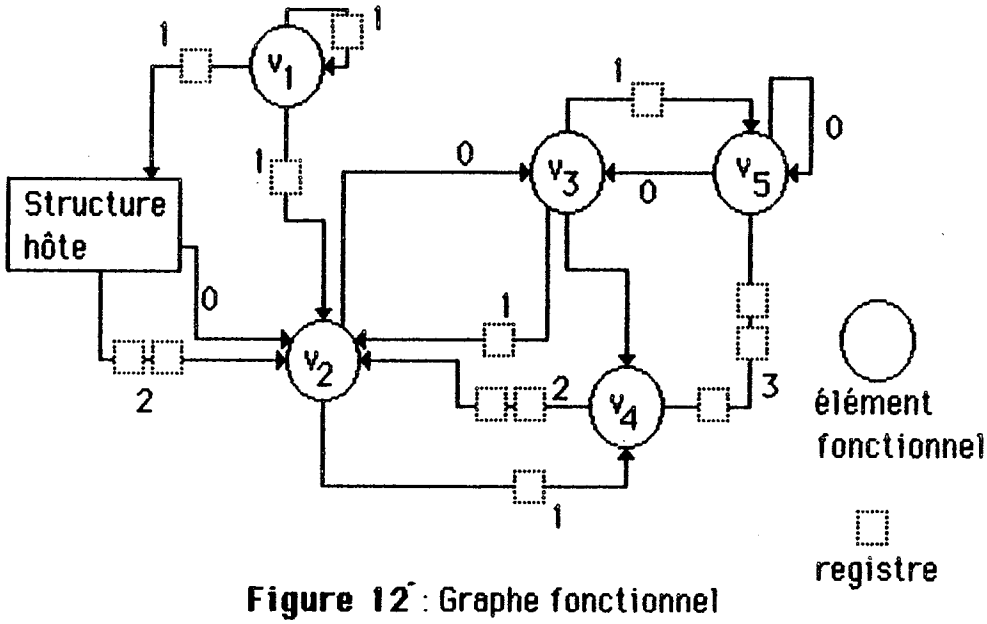
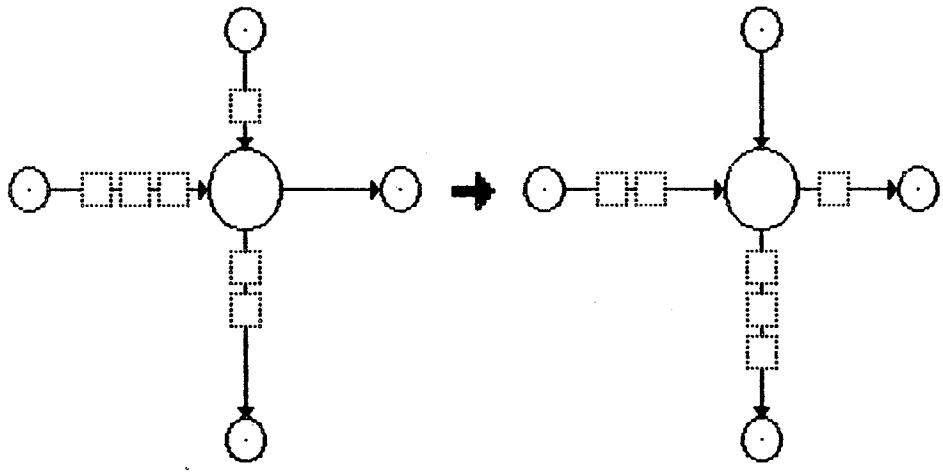


Figure 12 : Graphe fonctionnel

Il est clair que la transformation qui consiste à retirer un registre sur chaque arc entrant dans une cellule donnée, et à en ajouter un sur chaque arc sortant de cette cellule, ne modifie en rien le comportement de la cellule vis-à-vis des éléments voisins (cf la figure 13).



Transformation élémentaire du graphe de communication
Figure 13

Par ailleurs, on vérifie aisément que de telles transformations laissent invariant le nombre de registres sur tout circuit élémentaire; en conséquence, une condition nécessaire pour que de telles transformations conduisent à un circuit systolique, est que, sur chaque circuit élémentaire du graphe initial, le nombre de registres soit supérieur ou égal au nombre d'arcs. LEISERSON et SAXE ont montré que cette condition nécessaire est aussi suffisante. Ceci les amène à proposer la méthodologie suivante pour la conception d'architectures systoliques:

- étape 1: définir un réseau simple R (en général non systolique) où à chaque top d'horloge le résultat doit s'accumuler le long d'un chemin ne comportant pas de registres (logique combinatoire se propageant en cascade)
- étape 2: déterminer le plus petit entier k tel que le réseau R_k obtenu à partir de R en multipliant par k le poids de tous les arcs soit systolisable; R_k a alors le même comportement externe que R mais sa vitesse est divisée par k.

• étape 3: systoliser R_k, à l'aide des transformations précédentes (on peut définir des algorithmes polynomiaux performants pour celà)

Cette méthodologie conduit donc en général à un réseau qui délivre un résultat seulement une pulsation sur k.

Nous allons illustrer cette démarche en reprenant l'exemple de la convolution non récursive:

$$y_i = a_1 * x_i + a_2 * x_{i-1} + a_3 * x_{i-2} + \ldots + a_k * x_{i-k+1}$$

Le réseau de la figure 14 (a) correspond à la solution évidente (non systolique) obtenue à la figure 1. Etant donnée la présence dans ce réseau de circuits comportant deux arcs et un seul registre, on commence par doubler le nombre de registres sur chaque arc, ce qui conduit au réseau deux fois plus lent de la figure 14 (b): k=2 dans la deuxième étape de la construction. L'application de la méthode de LEISERSON et SAXE conduit alors au réseau de la figure 14 (c): on retrouve le réseau systolique de la figure 2, si ce n'est que pour des raisons de clarté les registres (latches) sont maintenant dessinés en dehors des cellules, lesquelles se composent uniquement d'un multiplieur et d'un aditionneur. Ce réseau est d'efficacité $e = 1/k = 1/2$ comme nous l'avions remarqué.

Y. ROBERT et M. TCHUENTE [24] ont proposé une quatrième étape purement algorithmique basée sur la stratégie classique du "divide-and-conquer" qui permet, à partir de R_k, d'obtenir un réseau R^* de même vitesse que R. Cette approche peut ainsi être considérée comme la dernière étape de la méthodologie de systolisation que nous venons de décrire brièvement.

Pour la convolution, l'approche "divide-and-conquer" consiste à séparer les calculs en deux parties:

$$y_{i,1} = a_1 * x_i + a_3 * x_{i-2} + a_5 * x_{i-4} + \ldots$$

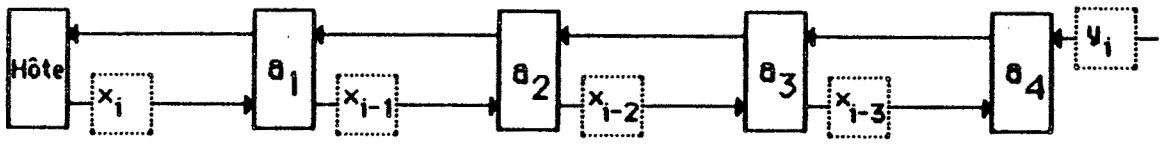
$$y_{i,2} = a_2 * x_{i-1} + a_4 * x_{i-3} + a_6 * x_{i-5} + \ldots$$

On construit alors un réseau R^* comportant:

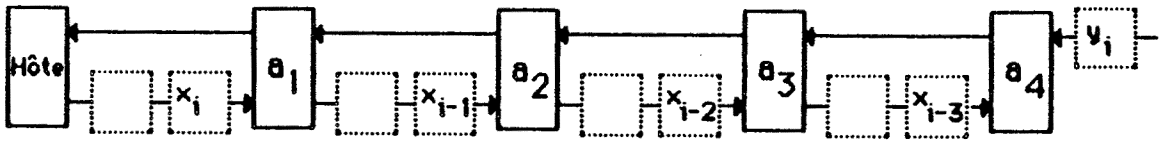
- un sous-réseau R1 pour le calcul de $y_{i,1}$
- un sous-réseau R2 pour le calcul de $y_{i,2}$
- une cellule qui reçoit $y_{i,1}$, $y_{i,2}$ en entrée et délivre en sortie

$$y_i = y_{i,1} + y_{i,2}$$

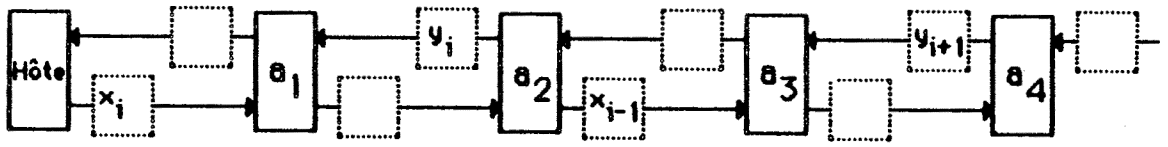
Réseaux de Convolution



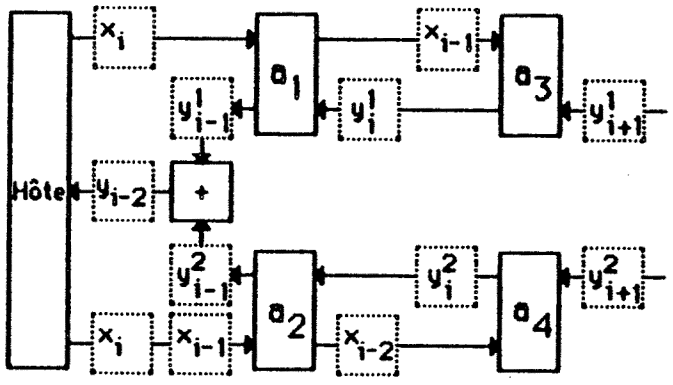
(a) la solution non systolique de la figure1



(b) solution précédente avec vitesse divisée par 2



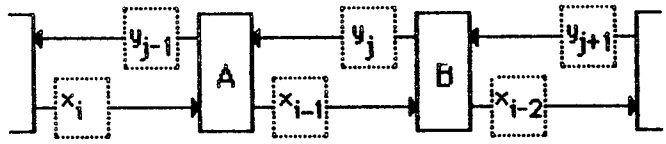
(c) version systolisée de (b): on retrouve le réseau de la figure 2



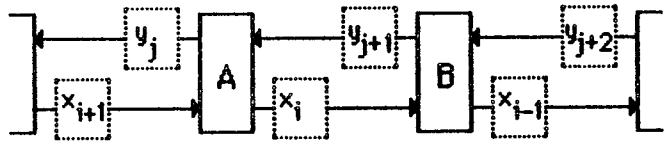
(d) réseau "divide-and-conquer" d'efficacité 1

Figure 14

La raison de ce découpage en deux sous-calculs est la suivante: pour obtenir un réseau d'efficacité 1, deux variables consécutives y_j, y_{j+1} ou x_i, x_{i+1} ne doivent être séparées que par un seul temps de cycle. Mais sur un réseau linéaire où les (y_j) circulent en sens inverse des (x_i), une variable y_j ne peut rencontrer qu'un x_i sur deux, comme on le voit sur le schéma ci-dessous. D'où l'idée d'utiliser deux réseaux linéaires, un pour chaque demi-calcul, qui conduit finalement au réseau de la figure 14 (d).



Instant t : y_{j+1} rencontre x_{i-1} cellule B



Instant t+1: y_{j+1} rencontre x_{i+1} cellule A
mais ne rencontre jamais x_i

Obtenir un réseau d'efficacité 1 pour la convolution récursive

$$y_i = a_1 * y_{i-1} + a_2 * y_{i-2} + a_3 * y_{i-3} + \ldots + a_k * y_{i-k} \qquad (E)$$

est plus compliqué: l'idée de renvoyer dans le réseau chaque variable y_i une fois calculée, pour servir au calcul des variables suivantes y_{i+1}, y_{i+2}, y_{i+3}, ... est toujours valide, mais

- comme pour le réseau de la figure 14 (d), il faut utiliser deux sous-réseaux linéaires, un pour chaque demi-calcul, puique deux variables consécutives ne sont plus séparées que par un seul temps de cycle
- y_i ne peut plus rencontrer y_{i+1}, toujours à cause de l'écart d'un seul top entre ces deux variables.

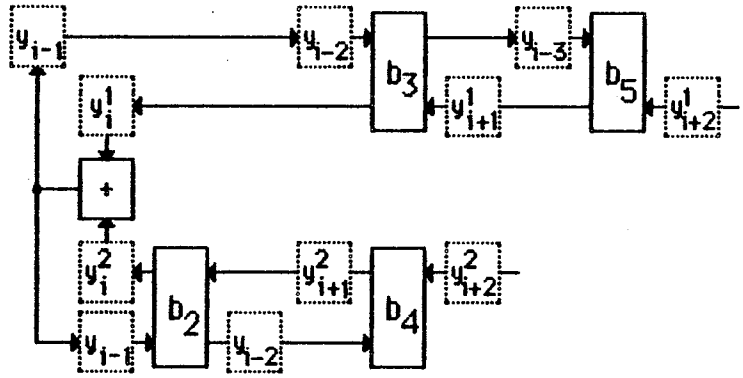
Si bien que la technique précédente permet seulement de résoudre le problème dans le cas où a_1 est nul: plus besoin de rencontre entre y_i et y_{i+1}, il suffit alors de renvoyer les variables calculées dans les deux sous-réseaux, comme indiqué figure 15.

Pour résoudre le problème quand a_1 est non nul, une solution consiste à réecrire la récurrence à un ordre supérieur: en remplaçant y_{i-1} par son expression récurrente dans (E), on obtient une formule du type

$$y_i = b_2 * y_{i-2} + b_3 * y_{i-3} + \ldots + b_k * y_{i-k} + b_{k+1} * y_{i-k-1}$$

On emploie alors la méthode précédente.

Bien sûr, des techniques similaires conduisent à des réseaux d'efficacité 1 pour la convolution récursive générale, la multiplication matrice-vecteur, la résolution d'un système triangulaire, ...



Convolution récursive (1er terme nul) en efficacité 1

Figure 15

La méthodologie présentée ici s'avère très puissante. Intellectuellement, elle est très satisfaisante, puisqu'elle permet de définir le concept de "systolique" dans un concept assez général. De plus, les transformations qu'elle introduit sont systématiques, et conduisent aux premiers réseaux systoliques non découverts à partir d'un seul et unique outil, ... l'intuition.

4.2. Autres méthodes de conception systématique

D'autres méthodologies ont été proposées: en particulier, celle de H.T.KUNG et W.T. LIN, basée sur des transformations algébriques, et celle de U. WEISER et A. DAVIS, basée sur une modélisation mathématique des retards, à l'aide d'opérateurs temporels. Dans les deux cas, ces approches ont pour elles l'avantage de la simplicité, en revanche leur portée est nettement moindre que la méthodologie de LEISERSON.

Enfin, des approches basées sur les dépendances des données entre elles, à partir d'équations récurrentes, permettent de générer automatiquement plusieurs réseaux systoliques pour un problème donné, et de comparer leurs performances. Ces approches font l'objet du papier de P. QUINTON [23].

4.3. Langages de description et de validation

Nous n'aborderons pas cet aspect ici, faute de place. Nous renvoyons au très bon article de synthèse de M. LAM et J. MOSTOW [20].

5. CONCLUSION

Ouvrant la thèse de C.E. LEISERSON, nous mettons en exergue le paragraphe suivant tiré de l'introduction:

"Systolic systems are an attempt to capture the concepts of parallelism, pipelining and interconnexion structures in a unified framework of mathematics and engineering. They embody engineering techniques such as multiprocessing and pipelining together with the more theoretical ideas of cellular automata and algorithms, and therefore are an excellent subject of investigation from a combined standpoint."

Ces lignes écrites en 1978 n'ont pas été démenties, bien au contraire ! Et s'il est vrai que le concept d'un réseau linéaire d'automates localement interconnectés et évoluants en parallèle se trouve déjà dans l'article de COLE daté de 1969, c'est bien l'émergence de la technologie VLSI qui a cristallisé la naissance d'une théorie adaptée aux nouvelles exigences de modularité et régularité, celle des processeurs intégrés spécialisés ... systoliques !

Commentaires bibliographiques

L'introduction au modèle systolique s'inspire des présentations de nombreux articles, citons [1], [6], [14], [15], [20], [21], [24] et [29]. Au paragraphe 2, les exemples de réseaux sont dus à [16] (figure 1), [14] (figures 2-5), [21] (figures 6-7), et [17] (figures 8-9). Le paragraphe 3 porte le titre de [14], et reprend brièvement le contenu de cet article. Les figures 10 et 11 s'inspirent de [15]. La méthodologie du paragraphe 4.1 est exposée dans [22], et complétée dans [25]. Le réseau de la figure 15 est extrait de [24]. Enfin, l'article auquel il est fait référence au paragraphe 4.3 est [20], et la thèse de LEISERSON porte le numéro [21].

Enfin, la présentation générale de cet article doit beaucoup à des travaux menés en commun avec M. TCHUENTE [24] [25] [27].

REFERENCES

[1] F. ANDRE, P. FRISON, P. QUINTON, Algorithmes systoliques: de la théorie à la pratique, RR 214, INRIA Rocquencourt, Mai 1983

[2] A. APOSTOLICO, A. NEGRO, Systolic algorithms for string manipulations, IEEE Trans. on Computers C33, 4 (1984), p 361-364

[3] R.P. BRENT, H.T. KUNG, F.T. LUK, Some linear-time algorithms for systolic arrays, Technical Report TR CS 82-541 (1982), Cornell University Department of Computer Science

[4] S.N. COLE, Real-time computation by n-dimensional iterative arrays of finite-state machines, IEEE Trans. on Computers C18, 4 (1969), p 349-365

[5] T.W. CURRY, A. MUKHOPADHYAY, Realization of efficient non-numeric operations through VLSI, in Proc. of VLSI 83, F. Anceau and E.J. Aas eds, Elsevier Science Publishers B.V. (North Holland), 1983, p 327-336

[6] M.J. FOSTER, H.T. KUNG, The design of special-purpose VLSI chips, IEEE Computer 13, 1 (1980), p 26-40

[7] M.J. FOSTER, H.T. KUNG, Recognize regular langages with programmable building-blocks, in the proceedings of the VLSI 1981 Conference,Edinburgh

[8] W.M. GENTLEMAN, H.T. KUNG, Matrix triangularisation by systolic arrays, Proc SPIE 298, Real-time Signal Processing IV, San Diego, California, 1981

[9] L.J. GUIBAS, F.M. LIANG, Systolic stacks, queues and counters, in Proc. of the 1982 Conference of Advanced Research in VLSI, M.I.T. Boston

[10] D. HELLER, I. IPSEN, Systolic networks for orthogonal equivalence transformations and their applications, Proc. 1982 Conf. Advanced Research in VLSI, p 113-122, MIT 1982

[11] A.V. KULKARNI, D.W.L. YEN, Systolic processing and an implementation for signal and image processing, IEEE Trans. on Computers C31 (Oct. 82), 1000-1009

[12] H.T. KUNG, The structure of parallel algorithms, Advances in Computers 19 (1980), p 65-112

[13] H.T. KUNG, Use of VLSI in algebraic computation: some suggestions, in Proc. SYMSAC 81, P.S. Wang ed., ACM, NewYork, 1981

[14] H.T. KUNG, Why systolic architectures, IEEE Computer 15, 1 (1982), p 37-46

[15] H.T. KUNG, Programmable systolic chip, NATO Advanced Study Institute on Microarchitecture of VLSI computers, Sogesta, Italy, July 9-20, 1984

[16] H.T. KUNG, M.S. LAM, Fault-tolerance and two-level pipelining in VLSI systolic arrays, Technical Report, Carnegie Mellon University, November 1983

[17] H.T. KUNG, P.L. LEHMAN, Systolic (VLSI) arrays for relational database operations, Proceedings of the 1980 ACM/SIGMOD International conference on management of data, Los Angeles, USA

[18] H.T. KUNG, C.E. LEISERSON, Systolic arrays for (VLSI), in Proc. of the Symposium on Sparse Matrices Computations, I.S. Duff and G.W. Stewart eds, Knoxville, Tenn., 1978, p 256-282

[19] H.T. KUNG, W.T. LIN, An algebra for VLSI algorithmic design, Technical Report, Carnegie Mellon University, April 1983

[20] M. LAM, J. MOSTOW, A transformational model of VLSI systolic design, IEEE Computer 18, 2 (1985), p 42-52

[21] C.E. LEISERSON, Area-efficient VLSI computation, PhD Thesis, Carnegie Mellon University, Pittsburgh, PA, USA, Octobre 1981

[22] C.E. LEISERSON, J.B. SAXE, Optimizing synchronous systems, in Proc. of the 22-th Annual Symposium on Foundations of Computer Science, IEEE, October 1981, p 23-36

[23] P. QUINTON, Synthèse automatique d'architectures systoliques, Rapport IRISA, Avril 1985

[24] Y. ROBERT, M. TCHUENTE, Designing efficient Systolic Algorithms, Rapport de Recherche 393, Sept 1983, Lab. IMAG, Grenoble, à paraître dans Journal of VLSI and Computer Systems

[25] Y. ROBERT, M. TCHUENTE, Réseaux systoliques pour des problèmes de mots, RAIRO Informatique théorique 1985

[26] Y. ROBERT, M. TCHUENTE, Calcul en temps linéaire de la plus longue sous-suite commune à deux chaînes sur une architecture systolique, C.R. Acad. Sc. Paris, t 297, Série I, n° 7, 1984, p 269-271

[27] Y. ROBERT, M. TCHUENTE, Resolution systolique de systemes lineaires denses, Research Report 420, 1983, Laboratoire TIM3/IMAG, Grenoble, France, à paraître dans RAIRO Modélisation et Analyse Numérique 1985

[28] R. SCHREIBER, Systolic arrays: high performance parallel machines for matrix computation, Proc. Elliptic problem solvers II, Academic Press 1984, 187-194

[29] E. TIDEN, B. LISPER, R. SCHREIBER, Systolic arrays, Technical Report TRITA-NA-8315, The Royal Institute of Technology, Stockholm, Sweden

CHAPITRE 4

ALGORITHMIQUE PARALLELE POUR MACHINES SIMD & MIMD

Yves ROBERT
CNRS - Laboratoire TIM3/INPG
BP 68
38402 Saint Martin d'Hères Cedex

Résumé :

Ce texte constitue une introduction à l'algorithmique parallèle. Après une brève description des architectures SIMD et MIMD, on présente les notions de tâches et de graphe d'ordonnancement (qui définit les contraintes temporelles liant l'exécution des tâches). On illustre alors sur un exemple la démarche qui, partant d'un algorithme séquentiel à paralléliser, le segmente en tâches, et affecte ensuite celles-ci aux processeurs, en respectant les contraintes du graphe d'ordonnancement. L'étude est axée sur la recherche d'algorithmes optimaux et de résultats de complexité.

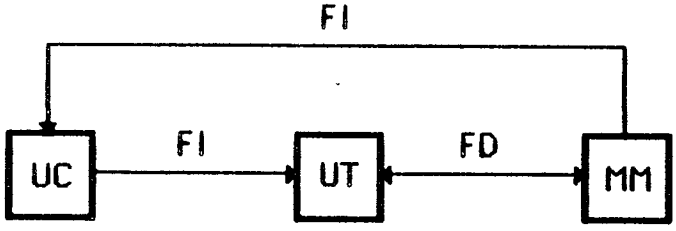
1. INTRODUCTION

Le calcul parallèle a motivé un nombre considérable de recherches ces dernières années. Ceci est dû en grande partie aux progrès réalisés dans la conception des circuits intégrés à très grande intégration: performance, fiabilité et faible coût de fabrication pour les dipositifs élémentaires tels que mémoires et microprocesseurs, autant d'arguments qui ont conduit au développement d'architectures multi-processeurs.

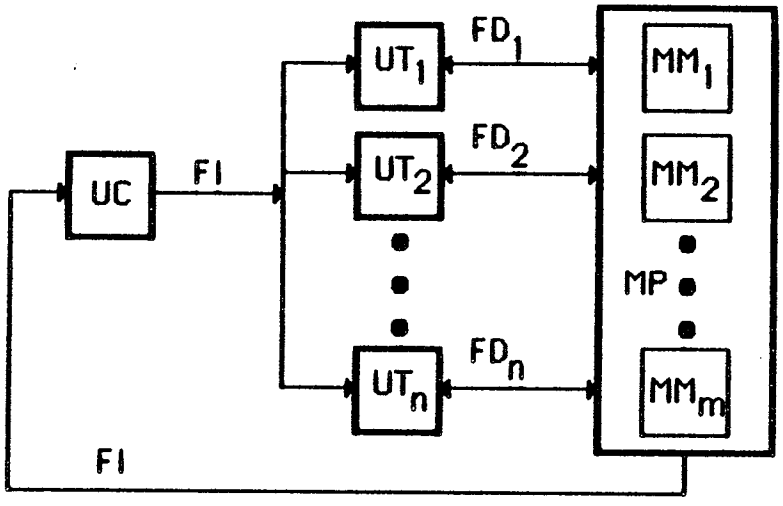
Le concept de parallélisme rompt avec l'approche classique qui consiste à gagner de la vitesse en effectuant plus vite chaque opération. En calcul parallèle, le gain de vitesse provient de la réalisation simultanée de plusieurs opérations.

Ces dernières années, plusieurs super-ordinateurs ont été construits suivant ce principe. De nombreuses variétés d'architectures ont vu le jour, et nous présenterons brièvement au paragraphe suivant la taxonomie la plus utilisée, celle de FLYNN [2]: celle-ci a pour critère de sélection le mode de contrôle des séquences d'opérations élémentaires effectuées par les différents processeurs.

Dans tous les cas, le développement d'ordinateurs parallèles a conduit à réévaluer la plupart des algorithmes usuels en fonction de nouveaux critères de viabilité et de performance. Pour paralléliser un algorithme, on commence par partitionner le problème en sous-tâches. L'élaboration du graphe de précédence permet de définir les contraintes temporelles pour l'exécution des tâches. Reste alors à affecter les tâches aux processeurs, en respectant les contraintes précédentes. Nous allons illustrer cette démarche en détaillant la parallélisation de la méthode de Jordan pour résoudre un système linéaire d'équations.



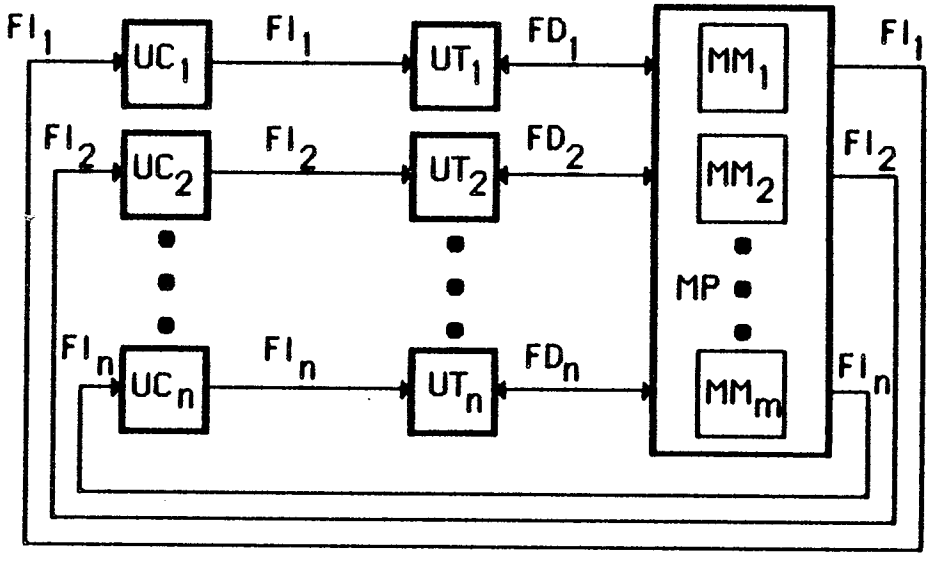
(a) ordinateur SISD



(b) ordinateur SIMD

UC : unité de contrôle
UT : unité de traitement (processeur)
MM : module mémoire
MP : mémoire partagée
FI : flux d'instructions
FD : flux de données

SISD : flux d'instructions et de données uniques
SIMD : flux d'instructions unique, flux de données multiples
MIMD : flux d'instructions et de données multiples



(c) ordinateur MIMD

La classification de FLYNN

d'après Hwang & Briggs, Computer Architecture & Parallel Processing, Mac Graw Hill 1984

Figure 1

2. MACHINES SIMD & MIMD

Le processus essentiel dans un ordinateur est l'exécution d'une suite d'instructions sur un ensemble de données. En général, les ordinateurs peuvent donc être classifiés en quatre catégories, selon la multiplicité des flux d'instructions et de données disponibles matériellement (FLYNN [2]).

En conservant les initiales anglaises consacrées par l'usage, on obtient les architectures suivantes, décrites figure1:

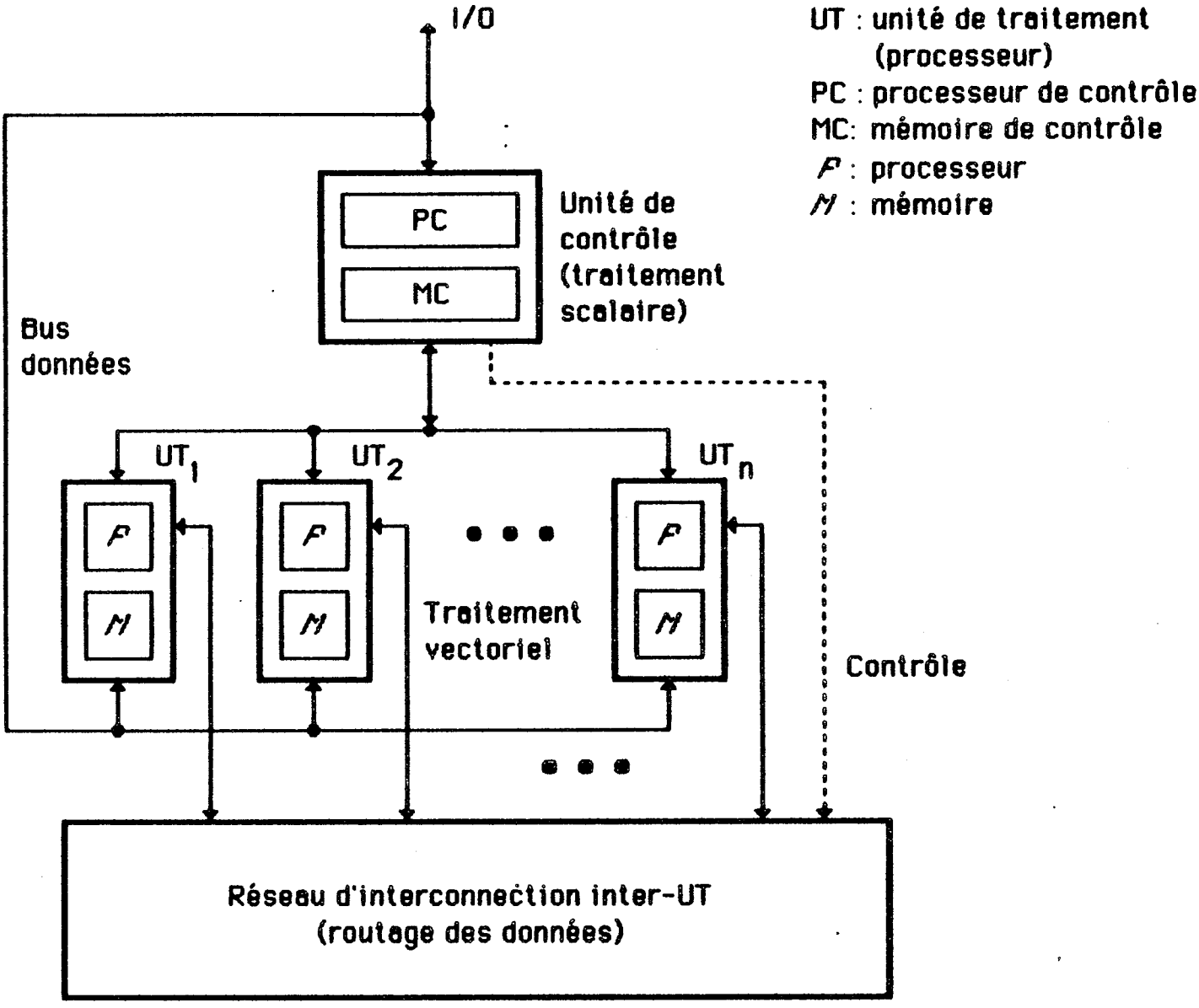
- SISD: un seul flux d'instructions et de données
- SIMD : un seul flux d'instructions, plusieurs flux de données
- MIMD : plusieurs flux d'instructions et de données

Stucture SISD : c'est l'ordinateur classique de Von Neumann. Les instructions sont exécutées séquentiellement mais peuvent être pipelinées (c'est effectivement le cas pour la plupart des machines d'aujourd'hui relevant de cette catégorie)

Structure SIMD : cette structure est détaillée figure 2. Plusieurs unités de traitement sont supervisées par la même unité de contrôle. Toutes les UT recoivent la même instruction (ou le même programme, auquel cas on parle d'une structure SPMD) diffusée par l'UC, mais opèrent sur des ensembles de données distincts, provenant de flux de données distincts. La mémoire partagée peut être subdivisée en plusieurs modules. Par exemple, les ordinateurs Illiac IV, BSP, STARAN, MPP, DAP sont des SIMD.

Structure MIMD : nous obtenons le schéma de la figure 3. Une structure MIMD "intrinsèque" implique des interactions entre les n processeurs, car tous les flux de mémoires sont issus d'une même unité allouée aux données. Si les n flux de données étaient issus de sous-unités disjointes de la mémoire partagée, nous aurions une structure dite "multiple-SISD", qui n'est rien d'autre que la juxtaposition de n systèmes uniprocesseurs SISD indépendants. On dit qu'un ordinateur MIMD est fortement couplé si les interactions entre les processeurs sont importantes. Sinon, on dit que l'ordinateur est faiblement couplé. La plupart des architectures MIMD commerciales sont faiblement couplées (IBM 370/168MP, Univac 1100/80, IBM 3081/3084, Cm*, ...). Parmi les MIMD fortement couplés, citons Cray-2, Cray X-MP, C.mmpp et surtout Denelcor HEP, la première structure multiprocesseur MIMD disponible commercialement, qui peut comporter jusqu'à 16 modules de traitement et 128 modules de mémoire.

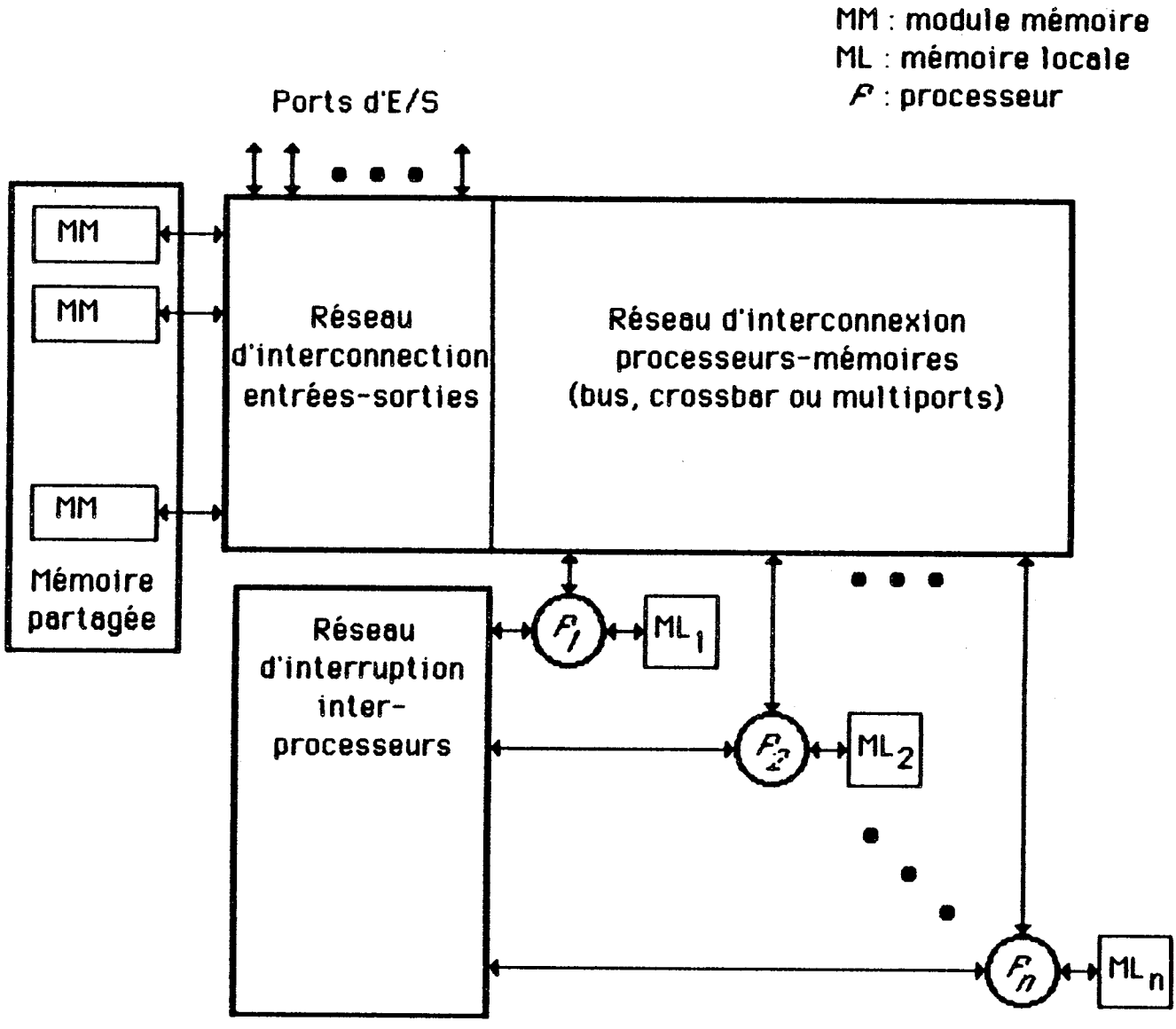
Enfin, la quatrième combinaison possible, qui comporterait n unités de traitement exécutant des instructions distinctes sur le même flux de données, ne semble pas présenter d'intérêt.



Structure fonctionnelle d'un ordinateur vectoriel SIMD avec traitement scalaire simultané dans l'unité de contrôle

d'après Hwang & Briggs, Computer Architecture & Parallel Processing, Mac Graw Hill 1984

Figure 2



Structure fonctionnelle d'un système multiprocesseur MIMD

d'après Hwang & Briggs, Computer Architecture & Parallel Processing, Mac Graw Hill 1984

Figure 3

3. PERFORMANCES

Considérons un algorithme qui s'exécute sur un ordinateur parallèle comportant p processeurs (identiques) en un temps T_p, et soit T_1 son temps d'exécution séquentielle (avec un seul processeur). On définit le facteur d'accélération par le rapport

$$S_p = \frac{T_1}{T_p} = \frac{\text{temps d'exécution avec 1 processeur}}{\text{temps d'exécution avec p processeurs}}$$

Le mieux que l'on puisse espérer est bien sûr $S_p = p$, mais c'est illusoire. STONE [8] propose la table suivante, évaluant (de façon empirique) S_p pour plusieurs problèmes usuels:

S_p	e_p	Exemples
kp	k	calculs matriciels, discrétisation
$kp/\log_2 p$	$k/\log_2 p$	tri, systèmes linéaires tridiagonaux, récurrences linéaires, évaluation de polynômes
$k\log_2 p$	$k\log_2 p/p$	recherche d'un élément dans un ensemble
k	k/p	certaines récurrences non linéaires, compilation de programmes

Dans ce tableau, k est un nombre positif plus petit que 1, et dépendant de la machine. D'autre part, on a introduit

$$e_p = S_p / p$$

qui est l'efficacité de l'algorithme parallèle; e_p est une quantité inférieure à 1, car on peut toujours simuler séquentiellement l'exécution d'un algorithme parallèle en multipliant son temps d'exécution par le nombre de processeurs. En fait, e_p mesure le taux moyen d'utilisation des

processeurs, et la parallélisation de l'algorithme est d'autant meilleure que e_p est proche de 1. La table précédente montre qu'une valeur constante de e_p quand p tend vers l'infini ne peut s'obtenir que pour des classes particulières de problèmes.

Pour amorcer une étude de complexité, nous avons besoin de simplifier le modèle. D'abord, nous nous préoccupons exclusivement de la parallélisation d'algorithmes "compute-bound". Dans un algorithme compute-bound, le nombre de calculs élémentaires est plus grand que le nombre de données en entrée-sortie. Sinon, le problème est I/O-bound. Par exemple, la multiplication de deux matrices de taille n nécessite $O(n^3)$ multiplications et additions pour $O(n^2)$ données, c'est un problème compute-bound. Au contraire, l'addition de deux matrices exige n^2 additions pour $3n^2$ opérations d'entrée-sortie, et est donc I/O-bound.

Cette restriction étant faite, on peut idéaliser le temps d'exécution en négligeant les temps d'accès, de stockage, et d'échange des données, pour ne retenir que le coût des opérations arithmétiques. On choisit alors comme unité de temps le temps nécessaire à la réalisation de l'une des quatre opérations élémentaires (contrairement aux modèles de la décennie précédente, et pour suivre l'évolution de la technologie, on admet qu'une multiplication équivaut à une addition).

Pour rester réaliste, il est maintenant absolûment nécessaire de borner le nombre de processeurs par un facteur linéaire. SAAD [6] montre comment les temps de communication prédominent pour résoudre un problème matriciel de taille n avec n^2 processeurs.

Enfin, si nous négligeons les temps de communication, nous imposons un certain mode d'accès aux données. Ainsi pour le traitement en parallèle des éléments d'une matrice A, nous supposerons par la suite que nous pouvons accéder aux éléments soit par lignes, soit par colonnes (comme les dispositifs matériels l'autorisent pour la plupart, cf. SCHENDEL [7] chap. 2.4.). Les opérations de transfert possibles seront alors :

- la transmission d'une ligne ou d'une colonne de A de l'organe de stockage vers un organe de traitement.
- la transmission d'une ligne ou d'une colonne de A d'un organe de traitement vers l'organe de stockage.

La duplication d'une ligne ou d'une colonne aura un coût nul, c'est à dire que nous supposons qu'il est possible de transférer simultanément une même donnée vers plusieurs processeurs. Dans ce cas, aucun processeur ne pourra modifier cette donnée. Inversement, un processeur ne pourra modifier une donnée que s'il est le seul à en posséder un exemplaire.

Nous ne nous préoccupons pas des mécanismes informatiques qui permettent de résoudre ce problème d'allocation des données sans conflit. Remarquons simplement que les contraintes précédentes et le mécanisme qui les gére permettent d'assurer un déroulement sans ambiguïté de l'algorithme et une certaine synchronisation.

4. GRAPHE DES TACHES

La notion de systèmes de tâches, telle que l'utilise KUMAR [4] permet d'introduire un formalisme qui sous-tendra toute nos études de parallélisations d'algorithmes.

Un programme séquentiel peut être segmenté en un ensemble de tâches. Par définition, une tâche est une unité indivisible de traitement caractérisée uniquement par son comportement extérieur: entrées, sorties et temps d'exécution.

Les tâches sont reliées par une relation d'ordre partiel, notée <<: T<<T' signifie que l'exécution de la tâche T doit être terminée avant que celle de T' ne puisse commencer. Le graphe de précédence est le graphe d'ordonnancement des tâches qui rend compte de ces contraintes temporelles.

Considérons par exemple [4] un ensemble de 8 tâches liées par les contraintes temporelles suivantes (la notation (i,j) signifie que T_i précède T_j, i.e. $T_i << T_j$):

(1,3)	(1,4)	(1,5)	(1,8)
(2,3)	(2,4)	(2,5)	(2,7)
(3,7)	(3,8)		
(4,6)	(4,7)	(4,8)	
(5,7)			
(6,8)			

Eliminant les contraintes redondantes, on obtient le graphe de prédédence suivant (figure 4).

Reste alors à affecter les tâches aux processeurs disponibles en respectant les contraintes du graphe (et celles liées aux temps d'exécution). Voici plusieurs questions caractéristiques des problèmes qui se posent:

(1) ne supposant pas de limite au nombre de processeurs disponibles, quel est le temps T_{opt} d'un algorithme optimal ?

(2) quel est le nombre mimimal de processeurs permettant de réaliser un algorithme s'exécutant en temps T_{opt} ?

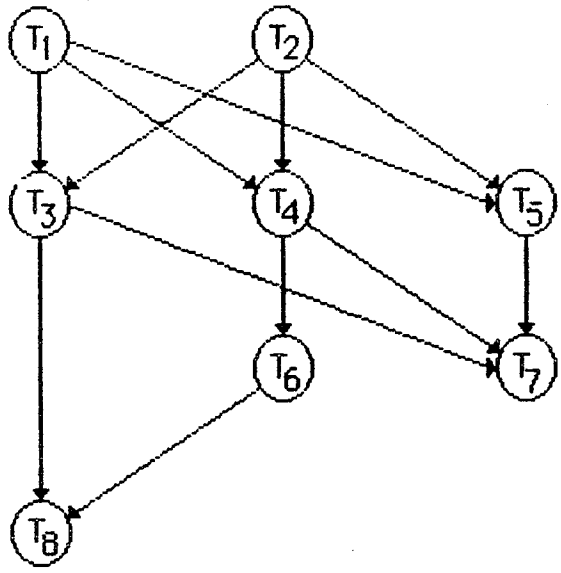


Figure 4

A priori, on sait seulement que T_{opt} est supérieur ou égal au plus long chemin du graphe, et qu'il est inutile de disposer de plus de D processeurs, ou D est le diamètre du graphe...

Dans l'exemple précédent, si toutes les tâches s'exécutent en temps 1, $T_{opt}=4$, D=3, et la réponse à la question 2 est p=2:

	tâches exécutées			
processeur 1	T_1	T_3	T_6	T_8
processeur 2	T_2	T_4	T_5	T_7

Enfin, pour un problème de taille n, se posent les questions relatives à la situation n -> ∞ :

(3) supposant un nombre de processeurs p = α n, quel est le α maximisant l'efficacité ?

(4) quel est le nombre mimimal de proceseurs permettant de réaliser un algorithme s'exécutant en temps asymptotiquement équivalent à T_{opt} ?

Autant de questions auxquelles nous essaierons de répondre pour l'exemple traité au paragraphe suivant !

5. ETUDE D'UN EXEMPLE

Nous étudions la parallélisation de la méthode de diagonalisation de Jordan pour résoudre un système linéaire dense à n équations de matrice A:

$$Ax = b$$

Il s'agit de transformer le couple (A,b) en un couple (D,c) où D est diagonale (nous laissons de côté la résolution du système diagonal). Les opérations élémentaires sont des prémultiplications par des matrices unicolonnes, et comme les transformations s'appliquent inchangées au second membre, nous considérerons A comme étant de taille n x (n+1). L'algorithme séquentiel est le suivant:

```
{ JORDAN séquentiel }
Pour k <- 1 à n
    c <- 1 / a_kk
    Pour j <- k+1 à n+1
        a_kj <- a_kj * c
    Pour i <- 1 à n, i≠k
        Pour j <- k+1 à n+1
            a_ij <- a_ij - a_ik * a_kj
```

Le mode d'accès aux données détermine la segmentation en tâches du programme séquentiel. Ainsi, si on autorise un accès aux données par lignes, on obtient immédiatement la décomposition en tâches suivante:

```
{ JORDAN par lignes }
Pour k <- 1 à n
    exécuter T_kk : < Pour j <- k+1 à n+1,
                      faire a_kj <- a_kj / a_kk >
    Pour i <- 1 à n, i≠k
        exécuter T_ki : < Pour j <- k+1 à n+1,
                          faire a_ij <- a_ij - a_ik * a_kj >
```

Par contre, si on autorise un accès aux données par colonnes, on peut inverser l'ordre des boucles imbriquées en i et j pour avoir:

```
{ JORDAN par colonnes }
Pour k <- 1 à n
    Pour j <- k+1 à n+1
        exécuter T_kj : < a_kj <- a_kj / a_kk
                        Pour i <- 1 à n, i≠k
                        faire a_ij <- a_ij - a_ik * a_kj >
```

5.1. JORDAN par lignes

Nous avons défini les tâches suivantes:

- T_{kk} : < Pour j <- k+1 à n+1,
  faire $a_{kj} \leftarrow a_{kj} / a_{kk}$ >
- T_{ki} : < Pour j <- k+1 à n+1, où $i \neq k$
  faire $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$ >

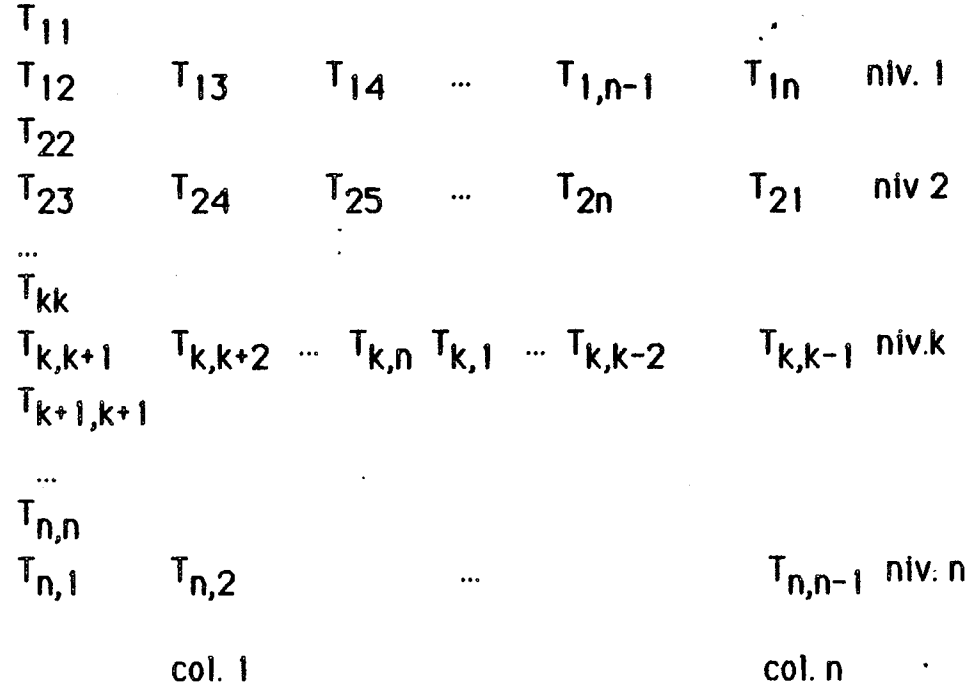
La tâche T_{kk} s'exécute en n+1-k unités de temps alors que T_{ki}, $i \neq k$, nécessite 2(n+1-k) unités.

Les contraintes d'ordonnancement sont les suivantes:

(A) $T_{kk} \ll T_{ki}$ pour $i \neq k$

(B) $T_{ki} \ll T_{k+1,i}$ pour $i \neq k$

ce qui conduit au graphe suivant:



Comme indiqué sur le schéma, nous appellerons niveau k la 2k-ème ligne du graphe, qui est constituée des tâches ($T_{k,i}$, $i \neq k$). De même, la colonne j sera constituée des tâches ($T_{k,k+j}$, $k \leq n$), où les indices sont pris modulo n.

i) détermination de T_{opt}

Le plus long chemin du graphe est constitué de l'ensemble

$$(T_{11}, T_{12}, T_{22}, T_{23}, \ldots, T_{kk}, T_{k,k+1}, \ldots, T_{n,n}, T_{n,1})$$

et donc $T_{opt} = \sum_{1 \leq k \leq n} 3(n+1-k) = 3n(n+1)/2$

D'autre part, le diamètre du graphe est D = n-1.

ii) efficacité asymptotique avec n-1 proceseurs

Il est évident de construire un algorithme s'exécutant en temps T_{opt} avec D = n-1 processeurs. L'efficacité asymptotique est alors

$$e_{n-1,\infty} = \frac{n^3}{n.(3n^2/2)} = 2/3$$

iii) cas où $p \leq \lfloor 2n/3 \rfloor$

On suppose ici que $p = \alpha.n$, avec $\alpha \leq 2/3$. Nous allons introduire un algorithme asymptotiquement optimal, et évaluer son temps d'exécution. Sans perte de généralité, posons p = 2q +1. Tout d'abord, on affecte toutes les tâches du plus long chemin

$$(T_{11}, T_{12}, T_{22}, T_{23}, \ldots, T_{kk}, T_{k,k+1}, \ldots, T_{n,n}, T_{n,1})$$

au premier processeur, qui les exécute dès que possible.

On répartit les autres tâches en q blocs constitués chacun de r = (n-2)/q colonnes consécutives, et on affecte leur exécution à un couple de processeurs. Par exemple, les processeurs 2 et 3 exécutent les tâches des colonnes 1 à r. Plus généralement, les processeurs 2i et 2i+1 auront en charge l'exécution des tâches des colonnes (i-1)r+1, (i-1)r+2, ... , ir.

Nous allons décrire l'action du premier couple de processeurs. Les autres couples opèrent de façon totalement similaire, et leur action s'obtient par un simple changement d'indices.

P2 et P3 ont en charge l'exécution des colonnes 1 à r. Par commodité, supposons r divisible par 3, r = 3s.

Pour s=1 (ce qui correspond à $\alpha = 2/3$), les processeurs opèrent selon le schéma suivant (ij dans la colonne Ps signifie que le processeur Ps effectue la tâche T_{ij}):

P1	P2	P3	temps alloué
11			
12	13	-	n+1
12	13	14	n
22	15	14	n
23	15	24	n
23	25	24	n-1
33	25	26	n-1
34	35	26	n-1
34	35	36	n-2
44	37	36	n-2
45	37	46	n-2
45	47	46	n-3
55	47	48	n-3
56	57	48	n-3
...	...	...	...

Comme les tâches T_{ki} coûtent le double des tâches T_{kk}, nous avons scindé (conceptuellement) leur exécution en deux parties. Notons que P1, P2 et P3 opèrent de façon synchrone, mais sont inactifs pendant au plus une unité de temps lorsqu'ils exécutent une tâche. Plus précisément, on perd 3 unités de temps tous les 2 niveaux sur un algorithme s'exécutant en T_{opt}. Nous en déduisons le temps d'exécution de l'algorithmé, qui est

$$T_{alg} = T_{opt} + \lceil 2n/3 \rceil.$$

L'algorithme est donc asymptotiquement optimal. Son efficacité est

$$e_{2n/3,\infty} = \frac{n^3}{(2n/3).(3n^2/2)} = 1$$

Pour s quelconque, P2 et P3 commencent par effectuer les mêmes tâches que précédemment niveaux 1 et 2. Mais avant d'aborder les niveaux 3 et 4, ils terminent l'exécution des tâches des niveaux 1 et 2 qui leur sont allouées, et ceci par groupes de 3 colonnes, comme indiqué sur l'exemple suivant pour s=2:

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

Le taux d'utilisation des processeurs (à l'exception de P1) est asymptotiquement égal à 1, ce qui établit que l'algorithme présenté est encore d'efficacité 1, et donc optimal. Son temps d'exécution est maintenant égal à s fois la valeur précédente T_{alg}, soit asymptotiquement

$$s.T_{alg} = (2n/3p).(3n^2/2) = n^3/p = n^2/\alpha$$

Notons qu'il est inutile de disposer de plus de 2n/3 processeurs (alors que le diamètre du graphe est n-1): nous avons montré qu'il existe un algorithme s'exécutant en temps miminal T_{opt} et n'utilisant que 2n/3 processeurs. Bien mieux, nous savons que $\alpha = 2/3$ est la valeur minimale pour construire un algorithme (utilisant $p = \alpha.n$ processeurs) s'exécutant en temps (équivalent à) T_{opt} : en effet pour toute valeur de $\alpha < 2/3$, nous connaissons le temps d'exécution d'un algorithme optimal, lequel est égal à n^2/α.

Enfin, les algorithmes présentés étant optimaux, les diverses évaluations d'efficacité sont en fait des résultats de complexité sur la parallélisation de la méthode étudiée. Nous pouvons rassembler ces résultats dans la

Proposition 1 :

(i) $T_{opt} = 3n^2/2$ et $D = n$

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

(iii) pour $p \leq 2n/3$, $p = \alpha.n$, nous avons construit un algorithme optimal, d'efficacité 1, s'exécutant en temps n^2/α.

(iv) la valeur minimale de α permettant de réaliser un algorithme en T_{opt} est $\alpha = 2/3$.

(v) pour $p \geq 2n/3$, $p = \alpha.n$, $e_{p,\infty} = 2/3\alpha$.

5.2. JORDAN par colonnes

Les tâches sont maintenant définies de la manière suivante:

- T_{kj} : < $a_{kj} \leftarrow a_{kj} / a_{kk}$
  Pour i <- 1 à n, i≠k
  faire $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$ >

Toutes les tâches T_{kj} s'exécutent dans le même temps $\tau = 2n-1$ unités de temps.

Les contraintes d'ordonnancement sont les suivantes:

(A) $T_{k,k+1} \ll T_{kj}$ pour $j>k+1$

(B) $T_{kj} \ll T_{k+1,j}$ pour $j>k$

ce qui conduit au graphe suivant:

T_{12}	T_{13}	T_{14}	....	$T_{1,n+1}$	niv. 1
	T_{23}	T_{24}	...	$T_{2,n+1}$	niv. 2
		T_{34}	...	$T_{3,n+1}$	niv. 3
				$T_{n,n+1}$	niv. n

En appelant niveau k l'ensemble { $T_{k,k+1}$, $T_{k,k+2}$, ..., $T_{k,n+1}$ }, nous avons maintenant un graphe où le nombre de tâches par niveau décroit, mais où toutes les tâches ont le même temps d'exécution τ.

i) détermination de T_{opt}

L'un des plus longs chemins du graphe est par exemple constitué de l'ensemble { T_{12}, T_{23}, ..., $T_{k,k+1}$, ..., $T_{n,n+1}$ }, d'où la valeur de T_{opt}:

$$T_{opt} = n.\tau = n(2n-1)$$

D'autre part, le diamètre du graphe est D = n. On supposera donc $p \leq n$.

ii) efficacité asymptotique avec n proceseurs

Construisant un algorithme qui s'exécute en temps T_{opt} avec D = n processeurs, l'efficacité asymptotique est alors

$$e_{n,\infty} = \frac{n^3}{n.(2n^2)} = 1/2$$

iii) restriction à une certaine classe d'algorithmes

Nous allons montrer que parmi les algorithmes optimaux, il en existe qui satisfont de plus à la contrainte d'ordonnancement (C):

(C) $T_{k,j} \lesssim T_{k,j+1}$ pour $j>k+1$

où la notation $T \lesssim T'$ signifie que l'exécution de la tache T' ne peut débuter avant celle de la tache T (mais la simultanété est possible).

Pour cela, considérons un algorithme quelconque M. Nous construisons à partir de M un algorithme M* de même temps d'exécution que M et qui respecte la contrainte supplémentaire (C):

- si à un instant t, l'algorithme M commence l'exécution de q tâches niveau k, M* fait de même, mais exécute les q premières tâches (selon l'ordre imposé par la contrainte (C)) non encore commencées du niveau k
- si à un instant t, l'algorithme M ne commence l'exécution d'aucune tâche niveau k, M* non plus.

Ainsi, à tout instant au cours de l'exécution, M* aura exécuté le même nombre de tâches par niveau que l'algorithme M. Nous devons cependant démontrer que les contraintes d'ordonnancement (A) et (B) sont satisfaites pour M*. Pour la contrainte (A), c'est évident puisque $T_{k,k+1}$ est la première tâche exécutée par M* niveau k. Pour la contrainte (B), nous montrons que le nombre de tâches exécutées par l'algorithme M niveau k à l'instant t est supérieur ou égal au nombre de tâches exécutées par M niveau k+1 à l'instant $t + \tau$. Ce résultat assure que M* tel que nous l'avons construit respecte bien la contrainte (B).

Nous procédons par récurrence sur t. Pour t=1, la seule tâche qui puisse être commencée par M est T_{21} en vertu de (A), d'où le résultat. Supposons le résultat vrai à l'instant t; pour toute tâche T_{kj} du niveau k, on a

$$fin(T_{kj}) > t+1 \Rightarrow fin(T_{k+1,j}) > t + 1 + \tau$$

où fin(T) désigne l'instant où l'exécution de la tâche T est terminée. Par suite, card {j / $fin(T_{kj}) \leq t+1$} > card {j / $fin(T_{k+1,j}) \leq t+T_k+1$}.

iv) l'algorithme glouton

Nous introduisons maintenant l'algorithme glouton G, défini informellement de la manière suivante: l'algorithme G exécute les tâches en balayant les lignes du graphe d'ordonnancement de gauche à droite, en commençant à tout instant le maximum de tâches (d'où le nom de glouton). Plus précisément, l'algorithme G exécute les tâches dans l'ordre suivant:

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

A l'instant t=1, l'algorithme G commence l'exécution des tâches T_{12}, T_{13}, ..., T_{1p}. A tout instant $t \geq 1$, si $q \leq p$ processeurs sont disponibles, l'algorithme G les affecte à l'exécution des q tâches suivantes, du moins tant que la contrainte (B) le permet (sinon, on effectue le nombre maximum de tâches exécutables).

Plutôt que d'expliciter formellement l'algorithme glouton dans le cas général, nous allons détailler son exécution sur un exemple. Choisissant n=5 et p=3, nous avons le séquencement suivant (avec les notations employées au paragraphe précédent):

P1	P2	P3	instant où l'exécution débute
12	13	14	0
15	16	23	τ
24	25	26	2τ
34	35	36	3τ
45	46	-	4τ
56	-	-	5τ

Quel que soit le nombre de processeurs $p \leq n$, nous montrons que l'algorithme glouton G est optimal. En appliquant le résultat établi en iii), nous nous restreignons à la classe des algorithmes qui vérifient les contraintes (A), (B) et (C). Sachant maintenant qu'une tâche T_{kj} doit être exécutée avant (ou en même temps que) la tache $T_{k,j+1}$, on montre par récurrence que l'algorithme glouton débute le plus tôt possible l'exécution des tâches $T_{1,n+1}$, $T_{2,n+1}$, ..., $T_{n,n+1}$, ce qui termine la démonstration.

Pour évaluer le temps d'exécution de l'algorithme glouton, nous notons que jusqu'à l'exécution de la tâche $T_{n-p+1,n+1}$ tous les p processeurs sont actifs sans interruption: en effet il y a plus de p tâches par niveau jusqu'au niveau n+1-p. Le temps d'exécution jusqu'à $T_{n-p+1,n+1}$ est donc égal au temps d'exécution séquentielle de toutes les tâches précédentes, divisé par le nombre de processeurs p (en d'autres termes G est d'efficacité 1 jusqu'au niveau n+m-p). Nous obtenons une première valeur,

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

à laquelle on ajoute le temps d'exécution des tâches $T_{n-p+2,n+1}$, $T_{n-p+3,n+1}$, ..., $T_{n,n+1}$, soit $(p-1).\tau = (p-1)(2n-1)$. Soit finalement

$$T_{glouton} = [\,(n+p)(n+1-p)/2p + p-1\,].(2n-1)$$

Posant $p = \alpha.n$, on obtient

$$T_{glouton} = (\alpha + 1/\alpha)\, n^2 + O(n)$$

et $$e_{\alpha n,\infty} = 1 / (1+\alpha^2)$$

Ici encore, les résultats sont des résultats de complexité, puisque tous les algorithmes présentés sont optimaux.

La fonction $\alpha \mapsto \alpha + 1/\alpha$ est strictement décroissante sur [0,1], et la valeur miminale de α permettant de construire un algorithme s'exécutant en temps (équivalent à) T_{opt} est ici $\alpha = 1$. D'où la

Proposition 2:

(i) $T_{opt} = 2n^2$ et $D = n$

(ii) $e_{n,\infty} = 1/2$

(iii) parmi les algorithmes optimaux, il en existe qui satisfont à la contrainte d'ordonnancement (C): $T_{k,j} \lesssim T_{k,j+1}$ pour $j>k+1$

(iv) pour toute valeur de p, l'algorithme glouton est optimal. Pour $p = \alpha.n$, $T_{glouton} = (\alpha + 1/\alpha)\, n^2$ et $e_{\alpha n,\infty} = 1 / (1+\alpha^2)$

(v) la seule valeur de α permettant de réaliser un algorithme en T_{opt} est $\alpha = 1$

5.3. Comparaison

Les propositions 1 et 2 montrent que la méthode par lignes est toujours préférable à la méthode par colonnes, et cela quel que soit le nombre de processeurs. En effet, pour $p = \alpha.n$, les temps d'exécution d'un algorithme optimal pour chacune des deux méthodes sont liés par:

- si $\alpha \leq 2/3$,

$$T_{\alpha.n,\ colonnes} - T_{\alpha.n,\ lignes} = (\alpha + 1/\alpha)n^2 - n^2/\alpha = \alpha n^2 > 0$$

- si $\alpha \geq 2/3$,

$$T_{\alpha.n,\ colonnes} - T_{\alpha.n,\ lignes} = (\alpha + 1/\alpha)n^2 - 3n^2/2 \geq n^2/2 > 0$$

Pour p donné, il en résulte aussi que l'efficacité de la parallélisation est toujours meilleure pour la version par lignes que pour celle par colonnes.

L'étude théorique que nous avons menée permettrait de disposer de critères de selection précieux lors d'une expérimentation pratique de la méthode de JORDAN sur une machine multi-processeurs.

6. CONCLUSION

L'algorithmique parallèle est en plein développement, et les outils d'analyse sont encore peu nombreux. L'approche que nous avons présentée sur un exemple est relativement générale, et permet d'aborder une large classe de problèmes. Encore faut-il respecter les limites du modèle théorique où nous nous sommes placés, modèle qui privilégie les opérations arithmétiques et néglige les temps de communication.

A tout le moins, une analyse de complexité asymptotique telle que nous l'avons menée permettra de disposer de critères fiables pour sélectionner certains algorithmes plutôt que d'autres lors d'une expérimentation concrète. Nous n'en prendrons pour preuve que la parfaite adéquation des évaluations théoriques et des résultats numériques obtenus par LORD, KOWALIK et KUMAR [5] lors de la parallélisation d'autres méthodes de résolution de systèmes linéaires (GAUSS et rotations de GIVENS), sur l'ordinateur HEP de la firme Denelcor, Inc, lequel est pourtant de type MIMD (cas où le modèle s'avère le plus insuffisant).

Commentaires bibliographiques

L'introduction au architectures multiprocesseurs s'inspire de plusieurs articles, citons [2], [3], [4] et [7]. Au paragraphe 2, les figures sont dues à [3] dans leur version anglaise originale. Le formalisme de la section 4 est introduit dans [4] (voir aussi la liste de références proposées dans cette thèse). Toute l'étude de la méthode de JORDAN (section 5) est directement tirée de [1]. D'une manière générale, ce texte doit beaucoup aux recherches développées en commun avec M. COSNARD et D. TRYSTRAM.

REFERENCES

[1] M. COSNARD, Y. ROBERT, D. TRYSTRAM, Résolution parallèle de systèmes linéaires denses par diagonalisation, RR 552, TIM3/IMAG Grenoble (1985)

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

[3] K. HWANG et F. BRIGGS, Parallel processing and computer architecture, MC Graw Hill (1984)

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

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

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

[7] U. SCHENDEL, Introduction to Numerical Methods for Parallel Computers, Ellis Horwood Series, J. Wiley & Sons, New York (1984)

[8] H.S. STONE, Problems of parallel computation, Proc. Conf. "Complexity of sequential and parallel numerical algorithms", J.F. Traub ed., Academic Press (1973)

PARTIE 2

Recueil d'articles

CHAPITRE 5

Discrete Applied Mathematics 11 (1985) 245-253
North-Holland

CONNECTION-GRAPH AND ITERATION-GRAPH OF MONOTONE BOOLEAN FUNCTIONS

Y. ROBERT and M. TCHUENTE

CNRS Laboratoire TIM3, Institut I.M.A.G., BP 68, 38402 Saint Martin D'Heres Cedex, France

Received 3 August 1983
Revised 10 July 1984

In this paper, we establish some relationships between the circuits of the connection-graph $G_C(F)$, and the circuits of the iteration-graph $G_I(F)$, of a monotone boolean function F. We first show that if $G_I(F)$ contains an element circuit of length multiple of $p \in [2,3]$, then $G_C(F)$ contains an elementary circuit of length multiple of p; then we prove that if $G_C(F)$ is a subgraph of a caterpillar, then $G_I(F)$ is a subgraph of a tree; at last we exhibit an infinite family of monotone boolean functions $\{F_n; n = 2 \times 5^q, q \geq 1\}$ such that any $G_C(F_n)$ is a subgraph of a tree, and $G_I(F_n)$ contains a circuit of length 2^{q+1}, i.e., of the order $n^{\log 2/\log 5}$.

1. Introduction and notations

The iterative behaviour of networks of automata has been extensively studied in the literature [2-7]. The underlying combinatorial problem is the study of the iteration-graph associated with the transition function of the network.

In this paper, we are interested in the particular case of monotone boolean networks. First we adopt the classical scheme which consists of deriving properties of the iteration-graph as a consequence of some hypothesis on the connection-graph; then we show that, conversely, some properties of the connection-graph can be derived from the observation of the dynamical behaviour of the network.

A directed graph (or digraph) of order n is denoted $G = (V, \Gamma)$ where $V = \{v_1, \ldots, v_n\}$ is the set of vertices and $\{(v_i, \Gamma(v_i)); 1 \leq i \leq n\}$ is the set of arcs. Every other notation or definition of graph theory will be consistent with that in [1].

A boolean function $F = (f_1, \ldots, f_n)$: $\{0,1\}^n \to \{0,1\}^n$ is said to be monotone if $(x_i \leq y_i,\ 1 \leq i \leq n)$ implies $(f_i(x) \leq f_i(y),\ 1 \leq i \leq n)$.

The *connection-graph* of F is defined by $G_C(F) = ([n], \Gamma)$ where $[n] = \{1, 2, \ldots, n\}$ and $\Gamma(i) = \{j: f_j \text{ depends on } x_i\}$, $1 \leq i \leq n$.

The *iteration-graph* of F is the functional digraph $G_I(F) = (\{0,1\}^n, F)$ where any vertex $x \in \{0,1\}^n$ has a unique out-going arc $(x, F(x))$.

In all the figures, the loops are omitted, and two opposite arcs $(x, y), (y, x)$ are represented by a single undirected edge $[x, y]$.

In the sequel, we shall make an extensive use of the following property:

Property 1. *Let $G_C(F) = ([n], \Gamma)$ be the connection-graph of a monotone boolean*

function. If $f_i(x)=0$ and $f_i(y)=1$, then there exists $j\in\Gamma^{-1}(i)$ such that $x_j=0$ and $y_j=1$.

2. From $G_1(F)$ to $G_C(F)$

Proposition 1. *Let F be a monotone boolean function. If $G_1(F)$ contains an elementary circuit $[x^0,\dots,x^{p-1},x^0]$, $p\geq 2$, such that, for any i, $1\leq i\leq n$, $(x_i^0,x_i^1,\dots,x_i^{p-1})$ is of the form $(a,a,\dots,a,b,b,\dots,b,a,a,\dots,a)$, then $G_C(F)=([n],\Gamma)$ contains an elementary circuit of length multiple of p.*

Proof. Let $A_k=\{i: x_i^k=0 \text{ and } x_i^{k+1}=1\}$, $0\leq k\leq p-1$ (upper indices are computed modulo p). Clearly, $A_h\cap A_k=\emptyset$ for $h\neq k$. Let us assume that $A_k=\emptyset$; then $x_i^k\geq x_i^{k+1}$ for any i, and since F is monotone, it follows that $x^k\geq x^{k+1}\geq\cdots\geq x^{k+p}=x^k$, hence $x^k=x^{k+r}$ for any $r\geq 1$, and this contradicts the hypothesis. As a consequence, $A=A_0\cup A_1\cup\cdots\cup A_{p-1}$ is a partition.

Consider the subgraph of $G_C(F)$ defined by $G_1=(A,\Gamma_1)$, where $\Gamma_1^{-1}(i)=\Gamma^{-1}(i)\cap A_{k-1}$ for any $i\in A_k$, $0\leq k\leq p-1$. Any arc of G_1 is of the form $(j,i)\in A_{k-1}\times A_k$, $0\leq k\leq p-1$. Since, from Property 1, $\Gamma_1^{-1}(i)\neq\emptyset$ for any $i\in A$, it follows that G_1 contains an elementary circuit of length multiple of p. □

As a direct consequence of Proposition 1, we obtain the following result:

Corollary 2. *If $G_1(F)$ contains an elementary circuit of length $p\in\{2,3\}$ then $G_C(F)$ contains an elementary circuit of length multiple of p.*

The Example 1 in Fig. 1 below shows that Corollary 2 cannot be extended to the case $p=4$.

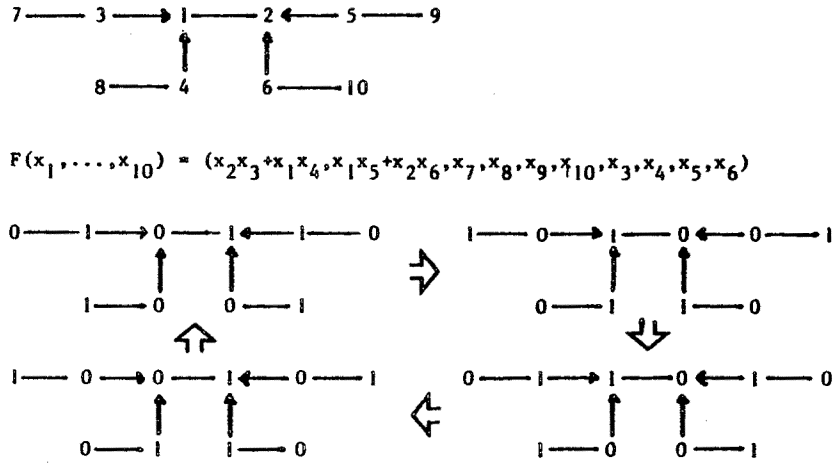


Fig. 1. Example 1.

Proposition 3. *Let $F=(f_1,\dots,f_n):\{0,1\}^n\to\{0,1\}^n$ be a monotone function such that for any i, f_i does not depend on x_i. If $G_I(F)$ contains an elementary circuit of length greater than two, then the same property holds for $G_C(F)=([n],\Gamma)$.*

Proof. Let $[x^0,x^1,\dots,x^{p-1},x^0]$, $p>2$ be an elementary circuit of $G_I(F)$. As in the proof of Proposition 1, it is easily shown that if there does not exist h such that $x_i^h=0$ and $x_i^{h+2}=1$, then $x^h=x^{h+2}$ for any $h\geq 0$ and this contradicts the hypothesis. Let (i,k) be such that $x_i^k=0$ and $x_i^{k+2}=1$. Using Property 1, we construct inductively a sequence

$$i_0=i,i_1,\dots,i_r,\dots$$

such that

$$x_{i_r}^{k-r}=0,\quad x_{i_r}^{k+2-r}=1\quad\text{and}\quad i_r\in\Gamma^{-1}(i_{r-1})$$

(i.e. $f_{i_{r-1}}$ depends on x_{i_r}). It is easily checked that $i_r\neq i_{r+1}$ and $i_r\neq i_{r+2}$ for every r. Therefore $G_C(F)$ contains an elementary circuit of length greater than two. □

Example 1 shows that this result cannot be extended to functions $F=(f_1,\dots,f_n)$ where, for some i, f_i depends on x_i.

3. From $G_C(F)$ to $G_I(F)$

Property 2. *Let $G_C(F)=([n],\Gamma)$ be the connection-graph of a monotone boolean function F, and let i,j be such that $\Gamma^{-1}(i)\subset\{i,j\}$. If $f_i(x)\neq x_j$, then in any elementary circuit $[x^0,x^1,\dots,x^{p-1},x^0]$ of $G_I(F)$, the sequence $\{x_i^r;\ r>0\}$ is constant.*

Proof. Since f_i is a monotone function of two variables x_i,x_j, it is easily verified that if $f_i(x)\neq x_j$, then either $f_i(x)\geq x_i$ or $f_i(x)\leq x_i$.

Therefore, in any elementary circuit $[x^0,x^1,\dots,x^{p-1},x^0]$, the sequence $\{x_i^r;\ r\geq 0\}$ is monotone and cyclic, hence it is constant. □

A caterpillar is a tree obtained from a chain by adding pending vertices (see Fig. 2 below).

Proposition 4. *Let F be a monotone boolean function. If $G_C(F)$ is a subgraph of a caterpillar, then $G_I(F)$ is a subgraph of a tree.*

Fig. 2.

Proof. Since $G_1(F)$ is a functional digraph, it is sufficient to prove that any elementary circuit is of length two. We proceed by induction on n. The case $n \leq 2$ is obvious. Let us assume that the result holds for $n-1$, and let $F=(f_1,\dots,f_n)$ be a monotone boolean function such that $G_C(F)$ is a subgraph of a caterpillar H. Let $[x^0,x^1,\dots,x^{p-1},x^0]$, $p\geq 2$ be an elementary circuit of $G_1(F)$.

Case 1: There exists i, say $i=n$, such that the sequence $\{x_i^r;\ r\geq 0\}$ is constant and equal to b. The connection-graph $G_C(F')$ of the function

$$F': x'=(x_1,\dots,x_{n-1})\to(f_1(x_1,\dots,x_{n-1},b),\dots,f_{n-1}(x_1,\dots,x_{n-1},b))$$

is obtained from $G_C(F)$ by deleting vertex n and its incident arcs; since $[x'^0,\dots,x'^{p-1},x'^0]$, where $x'^k=(x_1^k,\dots,x_{n-1}^k)$, is an elementary circuit of $G_C(F')$, it follows from the induction hypothesis that $p=2$.

Case 2: For any i, $\{x_i^r;\ r\geq 0\}$ is not constant. It follows from Property 2 that if i is pending in $G_C(F)$ and $\Gamma^{-1}(i)\subset\{i,j\}$, then $f_i(x)=x_j$. We are going to proceed by contradiction. So, let us assume that $p>2$. Clearly, there exists h,k, $1\leq h\leq n$, $0\leq k\leq p-1$ such that $x_h^k=1$ and $x_h^{k+2}=0$. From Property 1, we construct a sequence $j_0=h, j_1,\dots,j_r,\dots$ such that $j_r\in\Gamma^{-1}(j_{r-1})$ (i.e. $f_{j_{r-1}}$ depends on x_{j_r}), $x_{j_r}^{k-r}=1$ and $x_{j_r}^{k-r+2}=0$ for $r\geq 0$; furthermore, if there exists $j\in\Gamma^{-1}(j_{r-1})-\{j_0,\dots,j_{r-1}\}$ such that $x_j^{k-r}=1$ and $x_j^{k+2-r}=0$, then we choose such a j for j_r. Clearly, $j_r\neq j_{r+2}$ for $r\geq 0$. Let s denote the smallest integer such that

$$\{j_0,\dots,j_{s+1}\}=\{j_0,\dots,j_{s+3}\}$$

Since $G_C(F)$ is a caterpillar, it is easy seen that $j_{s+1}=j_{s+2}$ and $j_s=j_{s+3}$ (see Fig. 3 below).

Fig. 3.

We shall denote $j_s=j$ and $j_{s+1}=i$. The situation can therefore be summarized as follows:

r	x_j^r	x_i^r	x_l^r
$k-s-3$	1	-	b_l
$k-s-2$	-	1	a_l
$k-s-1$	0	1	b_l'
$k-s$	1	0	a_l'
$k-s+1$	-	0	
$k-s+0$	0	-	

$(a_l,a_l')\neq(1,0)$,
$(b_l,b_l')\neq(1,0)$
where l is any vertex
adjacent to i
and different from j.

$x_i^{k-s+1}=0$, thus there exists $l\in\Gamma^{-1}(i)$ such that $x_l^{k-s}=0$. On the other hand, $x_j^{k-s}=1$ and $x_l^{k-s}=x_i^{k-s-1}=1$ for any pending vertex $l\in\Gamma^{-1}(i)$; thus l is different

from j and is not pending. Clearly, since $G_C(F)$ is a caterpillar, j and $\bar{I}$ are the only non-pending vertices of $\Gamma^{-1}(i)$.

The application of Property 1 to $(x_i^{k-s-1}, x_i^{k-s}) = (1,0)$ shows that there exists $l^* \in \Gamma^{-1}(i)$ such that $x_{l^*}^{k-s-1} = 0$ and $x_{l^*}^{k-s-2} = 1$. We are going to prove that $l^* = j$.

- l^* is not pending since $x_{l^*}^{k-s-1} \neq x_i^{k-s-2}$.
- $l^* \neq \bar{I}$ since $(x_{\bar{I}}^{k-s-2}, x_{\bar{I}}^{k-s}) = (a_{\bar{I}}, a'_{\bar{I}}) \neq (1,0)$ and $x_{\bar{I}}^{k-s} = 0$.

The situation can be summarized as follows:

r	x_j^r	x_i^r
$k-s-3$	1	-
$k-s-2$	1	1
$k-s-1$	0	1
$k-s$	-	0

We can construct from Property 1 a sequence $i_0 = i$, $i_1 = j, \ldots, i_r, \ldots$ such that

$$i_r \in \Gamma^{-1}(i_{r-1}), \qquad x_{i_r}^{k-s-r} = 0, \qquad x_{i_r}^{k-s-r-1} = 1 = x_{i_r}^{k-s-r-2}$$

for any $r \geq 1$. It is easily verified that i_r is not pending $i_r \neq i_{r+1}$ and $i_r \neq i_{r+2}$ for every r, and this contradicts the fact that $G_C(F)$ is a caterpillar. We have therefore proved that any elementary circuit of $G_I(F)$ is of length two. □

Proposition 5. *Let F be a monotone boolean function. If $G_C(F)$ is a subgraph of a tree, then for any elementary circuit $[x^0, \ldots, x^{p-1}, x^0]$, $p \geq 2$ of $G_I(F)$, there exists i such that $\{x_i^r; r \geq 0\}$ is of period two.*

Proof. We proceed by induction on n. The case $n \leq 2$ is obvious. Let us assume that the result holds for $n-1$, and let $F = (f_1, \ldots, f_n)$ be a monotone boolean function such that $G_C(F)$ is a subgraph of a tree. Let $[x^0, x^1, \ldots, x^{p-1}, x^0]$ be an elementary circuit of $G_I(F)$.

Case 1: There exists a vertex i, say $i = n$, such that $\{x_i^r; r \geq 0\}$ is constant and equal to b. If F' denotes the monotone function

$$F' : (x_1, \ldots, x_{n-1}) \to (f_1(x_1, \ldots, x_{n-1}, b), \ldots, f_{n-1}(x_1, \ldots, x_{n-1}, b))$$

then, as in the proof of Proposition 4, it is easily verified that $G_C(F')$ is a subgraph of a tree, and

$$[x'^0, x'^1, \ldots, x'^{p-1}, x'^0] \quad \text{where } x'^k = (x_1^k, \ldots, x_{n-1}^k) \quad \text{for } 0 \leq k \leq p-1$$

is an elementary circuit of $G_C(F')$; the result follows from the induction hypothesis.

Case 2: For every i, $\{x_i^r; r \geq 0\}$ is not constant. It follows from Property 2 that for any pending vertex i, $f_i(x) = x_j$, where j is the only vertex adjacent to i.

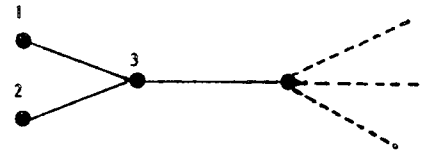


Fig. 4.

Subcase 2a: There exists a vertex i, say $i=3$, adjacent to two pending vertices, say $j=1$ and $k=2$ (see Fig. 4).

$f_1(x)=f_2(x)=x_3$. Let F' be the monotone function defined by

$$F'(x_2,x_3,\ldots,x_n)=(f_2(x_2,x_2,x_3,\ldots,x_n),\ldots,f_n(x_2,x_2,x_3,\ldots,x_n))$$

and let $x''^r=(x_2^r,\ldots,x_n^r)$. Clearly, $G_C(F')$ is a subgraph of a tree, and $[x'^0,\ldots,x'^{p-1},x'^0]$ is an elementary circuit of $G_1(F')$. Hence the result follows from the induction hypothesis.

Subcase 2b: Any vertex is adjacent to at most one pending vertex. Therefore there exists a vertex i of degree two, say $i=2$, adjacent to two vertices j and k, where j is pending, say $j=1$ and $k=3$ (see Fig. 5 below).

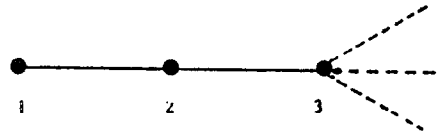


Fig. 5.

Let T_i denote the period of $\{x_i^r;\ r\geq 0\}$. From the hypothesis, $T_i>1$ for any i. If $T_1=2$, then the result holds, otherwise $T_1>2$ and there exists k such that $x_1^{k+1}=x_1^{k+2}$ and $x_1^{k+2}\neq x_1^{k+3}$; we shall assume that

$$x_1^{k+1}=1 \quad \text{and} \quad x_1^{k+3}=0.$$

Since $f_1(x)=x_2$, it follows that $x_1^{r+1}=x_2^r$ for every r. The application of Property 1 to the couples (x_2^{k+2},x_2^{k+1}) and (x_2^{k+2},x_2^k) gives the following situation:

r	x_1^r	x_2^r	x_3^r
$k-1$	-	-	1
k	-	1	1
$k+1$	1	1	0
$k+2$	1	0	-
$k+3$	0	-	-

Since f_2 is monotone and $f_2(x_1^{k+1},x_2^{k+1},x_3^{k+1})=f_2(1,1,0)=0$, it follows that $f_2(x_1^r,x_2^r,0)=0$ for any $r\geq 0$:

We are going to show by contradiction that

$$f_2(x_1^r, x_2^r, 1) = 1 \quad \text{for any } r \geq 0.$$

- Let $x_1^r = 1$ and $f_2(1, x_2^r, 1) = 0$. Since f_2 is monotone, it follows that $x_2^s = 0$ for any $s \geq k$ and this is a contradiction.
- Let $x_1^r = x_2^r = 0$ and $f_2(0, 0, 1) = 0$. Since $f_1(x) = x_2$, it follows that $x_1^s = x_2^s = 0$ for any $s \geq r$, and this is a contradiction.
- Let $x_1^r = 0$, $x_2^r = 1$ and $f_2(0, 1, 1) = 0$. We show easily by induction that $0 = x_1^r = x_1^{r+2h} = 0$ for $h \geq 0$ and this is not compatible with $x_2^k = x_2^{k+1} = 1$.

This shows that $f_2(x_1^r, x_2^r, x_3^r) = x_3^r$ for any $r \geq 0$.

Let us denote $F'(x_2, \ldots, x_n) = (f_2(x_2, \ldots, x_n), \ldots, f_n(x_2, \ldots, x_n))$ and $x'^r = (x_2^r, \ldots, x_n^r)$. Clearly, $G_C(F')$ is contained in a tree of order $n-1$, and $[x'^0, \ldots, x'^{p-1}, x'^0]$ is an elementary circuit of $G_1(F')$, hence the result follows from the induction hypothesis.

This ends the proof of Proposition 5. □

Corollary 6. *Let F be a monotone boolean function. If $G_C(F)$ is a subgraph of a tree, then $G_1(F)$ is bipartite.*

Proof. Let $[x^0, x^1, \ldots, x^{p-1}, x^0]$, $p \geq 2$ be an elementary circuit of $G_1(F)$. Let T_i denotes the period of $\{x_i^r; r \geq 0\}$, then p is the least common multiple of the T_i, $1 \leq i \leq n$, and the result is a direct consequence of Proposition 5. □

We are now interested in the maximum length of the circuits of $G_1(F)$, where F is monotone and $G_C(F)$ is a subgraph of a tree.

Let $G = ([n], U)$ be a digraph of order n. We denote by $G^* = (V^*, U^*)$ the digraph of order $5n$ defined by $V^* = \{1, 2, \ldots, n, n+1, \ldots, 5n\}$ and

$$U^* = U \cup \{(n+i, 2n+i), (2n+i, n+i), (2n+i, i), (3n+i, i), (3n+i, 4n+i), (4n+i, 3n+i),\ 1 \leq i \leq n\}$$

(see Fig. 6 below).

Proposition 7. *Let $F = (f_1, \ldots, f_n)$ be a monotone boolean function. If $G_1(F)$ contains an elementary circuit of length p, then there exists a monotone boolean function $F^* = (f_1^*, \ldots, f_{5n}^*)$ such that $G_C(F^*) = (G_C(F))^*$ and $G_1(F^*)$ contains an elementary circuit of length $2p$.*

Proof. We define

$$f_i^*(x) = x_{2n+i} f_i(x) + x_i x_{3n+i}, \qquad f_{n+i}^*(x) = x_{2n+i},$$

$$f_{2n+i}^*(x) = x_{n+i}, \qquad f_{3n+i}^*(x) = x_{4n+i},$$

$$f_{4n+i}^*(x) = x_{3n+i} \quad \text{for } 1 \leq i \leq n.$$

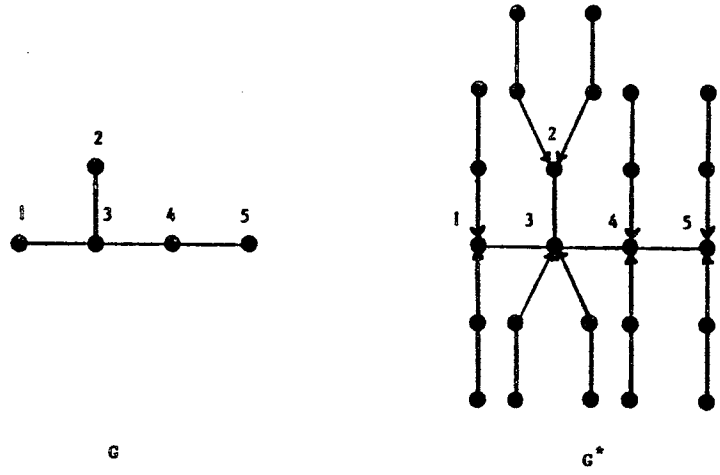


Fig. 6.

Let $[x^0, x^1, \ldots, x^{p-1}, x^0]$ be an elementary circuit of $G_1(F)$. Let us denote for any $x \in \{0,1\}^n$:

$$x_a = (x_1, \ldots, x_n, \underbrace{1, \ldots, 1}_{n}, \underbrace{0, \ldots, 0}_{n}, \underbrace{1, \ldots, 1}_{n}, \underbrace{0, \ldots, 0}_{n}),$$

$$x_b = (x_1, \ldots, x_n, 0, \ldots, 0, 1, \ldots, 1, 0, \ldots, 0, 1, \ldots, 1).$$

Clearly, $F^*(x_a^i) = x_b^i$ and $F^*(x_b^i) = x_a^{i+1}$, hence

$$[x_a^0, x_b^0, x_a^1, x_b^1, \ldots, x_a^{p-1}, x_b^{p-1}, x_a^0]$$

is an elementary circuit of $G_1(F^*)$. □

Corollary 8. *For any $q \geq 0$ there exists a monotone boolean function F from $\{0,1\}^{2.5^q}$ into itself such that $G_C(F)$ is a subgraph of a tree and $G_1(F)$ contains an elementary circuit of length 2^{q+1}.*

Proof. Let $\phi: \{0,1\}^2 \to \{0,1\}^2$ be such that $\phi(x_1, x_2) = (x_2, x_1)$. F is obtained inductively from ϕ by the algorithm exhibited in Proposition 6. □

Comment. Corollary 8 shows that if $G_C(F)$ is contained in a tree of order n, then $G_1(F)$ may contain circuits of length $O(n^{\log 2/\log 5})$. However, if we consider the set of monotone boolean functions $F = (f_1, \ldots, f_n)$ whose connection-graph $G_C(F)$ is a subgraph of a tree, the problem of determining the asymptotic expression of the maximum length for the circuits of $G_1(F)$ is still an open question.

References

[1] C. Berge, Graphes et Hypergraphes (Dunod, Paris, 1974).
[2] G. Galperin, One-dimensional local monotonic operators with memory, Soviet. Math. Dokl. 17 (3) (1976).
[3] E. Goles and J. Olivos, Comportement périodique des fonctions à seuil binaires et applications, Discrete Applied Math. 3 (1981) 93–105.
[4] J.M. Greenberg, C. Greene and S.P. Hastings, A combinatorial problem arising in the study of reaction-diffusion equations, SIAM J. Algebraic Discrete Methods 1 (1980) 34–42.
[5] T. Kitagawa, Cell space approaches in biomathematics, Math. Biosciences 19 (1974) 27–71.
[6] R. Shingai, Maximum period of 2-dimensional uniform neural networks, Information and Control 41 (1979) 324–341.
[7] M. Tchuente, Contribution à l'étude des méthodes de calcul pour des systèmes de type coopératif, Thèse d'Etat, Grenoble (1982).

Dynamical behaviour of monotone networks

Yves ROBERT et Maurice TCHUENTE
CNRS - Laboratoire TIM3/IMAG
BP 68 - 38402 St Martin d'Hères Cedex - FRANCE

ABSTRACT

We study the dynamics of some monotone networks. First we consider cellular automata whose transition functions are generalized majority functions. Then we consider boolean monotone networks with memory, and we show how the maximal degree of the connection-graph can influence the maximal length of an elementary circuit of the iteration-graph.

A short version of this paper is to appear in the Proceedings of the NATO Advanced Research Workshop on Disordered Systems and Biological Organization, SPRINGER VERLAG 1985

1. INTRODUCTION

Cellular automata were first introduced in the 1950's by John Von Neumann. Informally, a cellular automaton can be viewed as a discrete dynamical system whose global behaviour is generated by the (simple) local interactions of its elementary cells, hereafter called the sites of the automaton. As pointed out by Wolfram [8], cellular automata have arisen in several disciplines, because they provide examples in which the generation of complex behaviour by the cooperative effects of simple components may be studied. No unified mathematical framework has been yet developped to modelize the iterative behaviour of general networks of automata, but some tools such as algebraic operators [4], Lyapounov functions [2] [4], modular arithmetic, polynomial algebra [5], arithmetic in finite fields [7] have been introduced to analyze special classes of cellular automata. In this paper, we characterize the dynamics of some monotone cellular automata. First we use a morphism technique to derive the iterative behaviour of automata whose transition functions are generalized majority functions. Then, we study some relationships between the connection-graph and the iteration-graph of boolean automata with memory, assuming that the transition function of each site is monotone with regard to its inputs.

We begin with some notations and definitions.

2. NOTATIONS AND DEFINITIONS

Throughout the paper, the set of sites of the cellular automaton $\mathcal{C}$ that we study is denoted $\mathcal{S} = \{S1, S2, ..., Sn\}$. The value $Si(t)$ of Si at step t belongs to $Q = \{0,1,2,...,p\}$. For $t \geq 0$, we let $\mathcal{S}(t) = (S1(t),...Sn(t)) \in Q^n$ be the configuration of $\mathcal{C}$ at step t; $\mathcal{S}(0)$ is called the initial configuration of $\mathcal{C}$. The transition function of $\mathcal{C}$ is a function

$$F = (f_1,...,f_n) : x = (x_1,...,x_n) \in Q^n \dashrightarrow F(x) \in Q^n$$

such that each f_i depends only on a restricted number of variables x_j. We let $FN(i)$ be the set of indexes j such that f_i depends on x_j: $FN(i)$ stands for the Functional Neighbourhood of the site i.

Finally, the connection-graph $G_C(F)$ is defined by $G_C(F) = (\mathcal{S},\Gamma)$, where $(j,i) \in \Gamma \Leftrightarrow j \in FN(i)$, and the iteration-graph of F is the functional digraph $G_I(F) = (Q^n,F)$ where any vertex $x \in Q$ has a unique out-going arc $(x,F(x))$.

3. USING A MORPHISM TECHNIQUE

In this section, we consider a cellular automaton $\mathcal{A}$ whose transition function F is defined as follows:

For $t \geq 0$ and $1 \leq i \leq n$, let k(i) be an arbitrary nonnegative integer $\leq |FN(i)|$ and let $\{S_j(t);\ j \in FN(i)\}$ be ordered as $x_1 \leq x_2 \leq \ldots \leq x_{|FN(i)|}$. Then we define $S_i(t+1) = x_{k(i)}$.

That is to say, the next value of site i is the k(i)-th value of the current states of its functional neighbourhood when totally ordered (see the Example 1 below).

Assuming that $G_c(F)$ is symmetric, that is $i \in FN(j) \Leftrightarrow j \in FN(i)$, we can state the following:

proposition 1 : if $G_c(F)$ is symmetric, then $G_i(F)$ is a subgraph of a tree.

proof : we prove that any elementary circuit of $G_i(F)$ is of length two; we proceed by contradiction. So, assume there exists an elementary circuit $[x_1, x_2, \ldots, x_T] = [\mathcal{S}(t+1), \mathcal{S}(t+2), \ldots, \mathcal{S}(t+T)]$ of $G_i(F)$ with $T \geq 3$; for $q \in Q$ we introduce the function $\varphi_q : Q \dashrightarrow \{0,1\}$

$$x \dashrightarrow \begin{cases} 1 \text{ if } x \geq q \\ 0 \text{ otherwise} \end{cases}$$

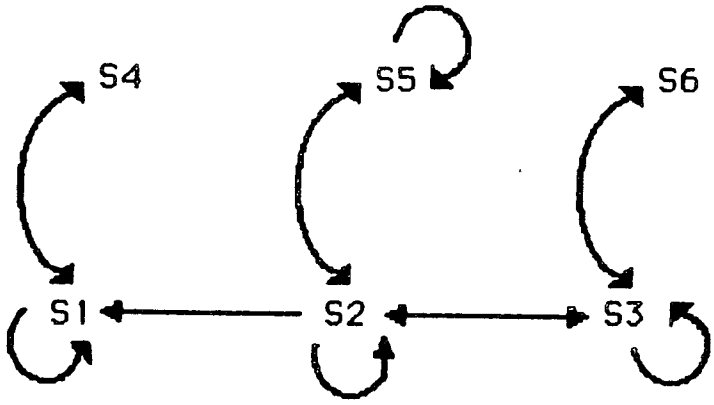
The proof of the following two lemmas is straigthforward :

lemma 1 : there exists $q \in Q$ such that $\{ \varphi_q(x_i) ;\ 1 \leq i \leq T \}$ is a periodic sequence of $\{0,1\}^n$ of length greater than 2.

Let $A = (a_{ij})$ be the n by n matrix defined by $a_{ij} = 1$ if $j \in FN(i)$ and $a_{ij} = 0$ otherwise. We introduce the threshold function $H = (h_1, \ldots, h_n) : \{0,1\}^n \dashrightarrow \{0,1\}^n$ defined by $h_i(y_1, \ldots, y_n) = \begin{cases} 1 \text{ if } \Sigma_{1 \leq i \leq n}\ a_{ij} \cdot y_j \geq |FN(i)| - k(i). \\ 0 \text{ otherwise} \end{cases}$

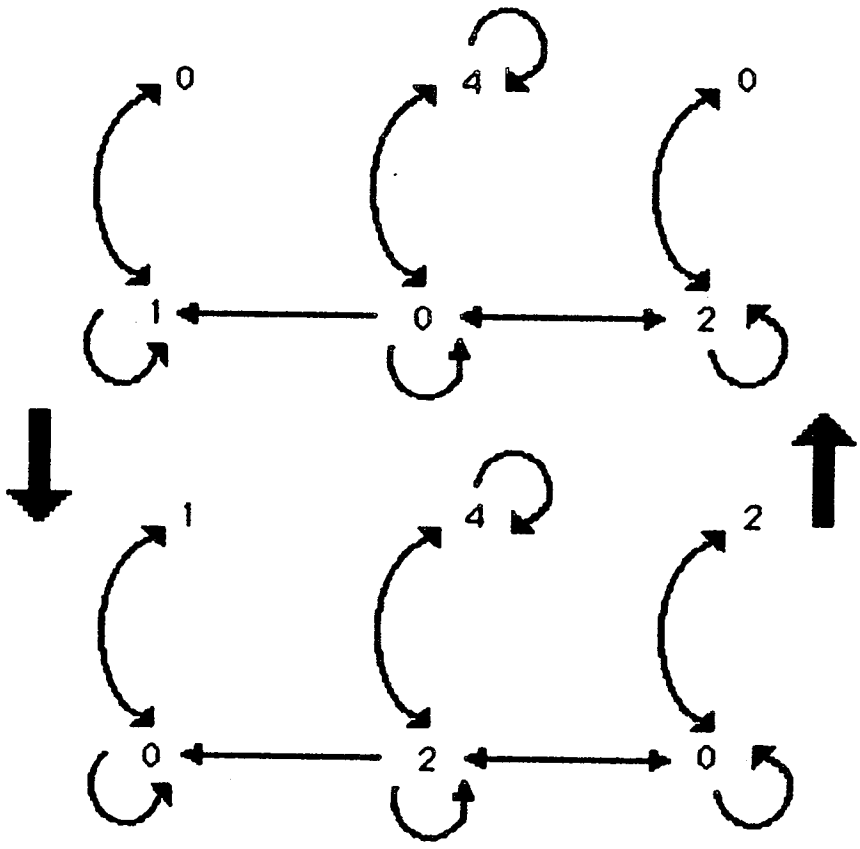
lemma 2 : for any $q \in Q$, $\varphi_q \circ F = H \circ \varphi_q$

Thus $\varphi_q \circ F^t = H^t \circ \varphi_q$ for all $t \geq 1$, and the contradiction follows from a theorem of [4] which states that any elementary circuit of $G_i(H)$ is of length less than or equal to two.



k(1) = 2, k(2) = 3, k(3) = 2, k(4) = 1, k(5) = 2, k(6) = 1

Connection-graph



Elementary circuit of the iteration-graph

Example 1

4. MONOTONE BOOLEAN AUTOMATA

In this section we concentrate on boolean networks whose transition functions are monotone. Thus Q = {0,1} and we assume that the transition function $F : (f_1,...,f_n) : \{0,1\}^n \dashrightarrow \{0,1\}^n$ satisfies to:

$$(x_i \leq y_i \ , \ 1 \leq i \leq n) \Rightarrow (f_i(x) \leq f_i(y) \ , \ 1 \leq i \leq n)$$

We consider automata with memory, i.e. we assume that the value Si(t+1) of a site i can depend on the value of Sj(t), $j \in FN(i)$, and on Si(t), Si(t-1), Si(t-2), ... , Si(t-k). The number k (independent of i) is the length of the memory.

We point out that any network $\mathcal{R}$ with n sites S1,...,Sn and a k-length memory can be viewed as an automaton with (k+1)*n sites and no memory, by replacing each site Si by (k+1) sites $Si_0, Si_1, ..., Si_k$ arranged along a circuit of length k+1, as illustrates in the figure 1. Each site Si_j, $j \geq 1$, is only a transfer cell wich copy the previous value of the site Si_{j-1}, i.e. $Si_j(t) = Si_{j-1}(t-1)$ for $t \geq 1$.

proposition 2 : let $\mathcal{R}$ be a cellular automaton with a memory of length 1. If $G_c(F)$ is contained in a chain, then $G_i(F)$ is contained in a tree.

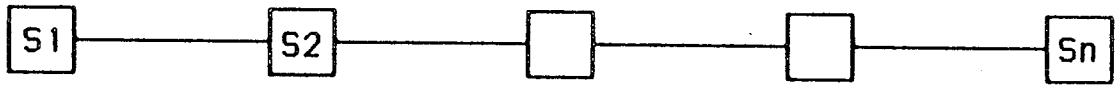
proof : owing to the above interpretation, we can consider $\mathcal{R}$ as a cellular automaton with no memory; the associated connection-graph is then contained in a caterpillar, and the result follows from a theorem of [6].

In the following two paragraphs, we illustrate the relation between the maximal degree of the connection-graph $G_c(F)$ and the maximal length of an elementary circuit of the iteration graph $G_i(F)$. We restrict ourselves to the case of a single site monotone automaton with a memory of length k.

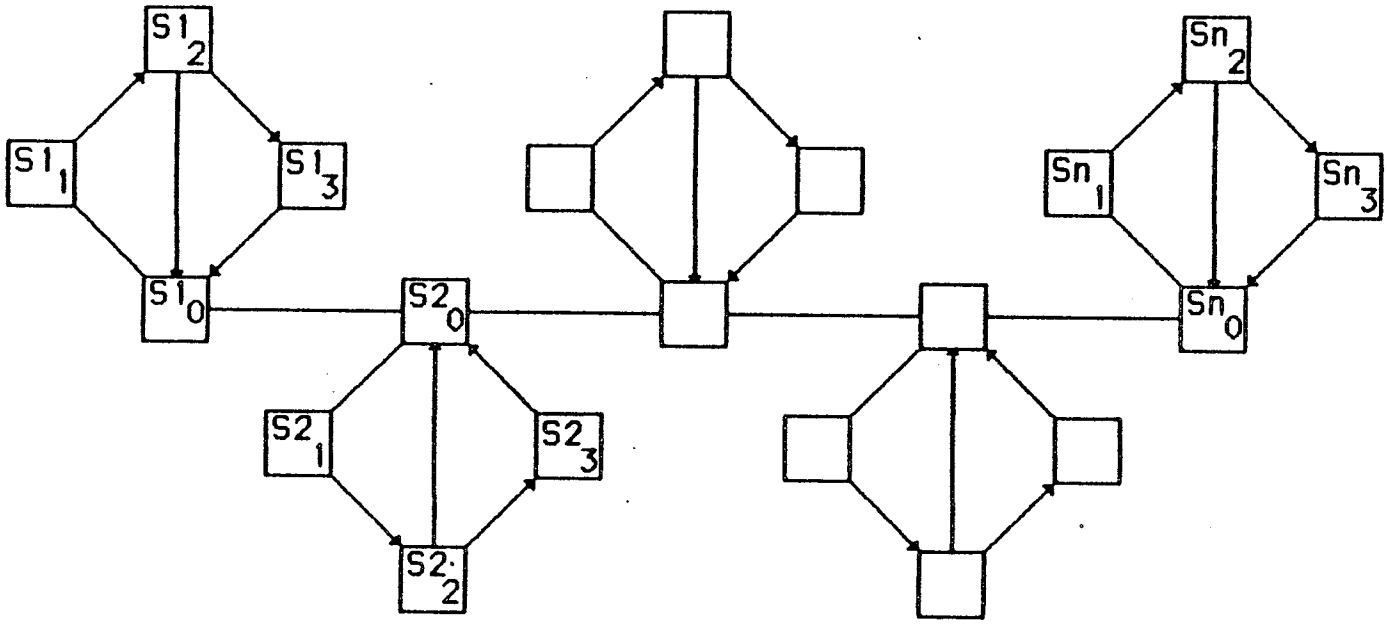
CASE 1

We consider the automaton whose connection-graph is depicted in figure 2. Sl_1, Sl_2, ..., Sl_k are transfer cells, and for any $t \geq 1$, the value $Sl_0(t+1)$ is a monotone function of $Sl_0(t)$, $Sl_1(t) = Sl_0(t-1)$ and $Sl_k(t) = Sl_0(t-k)$. Thus $FN(l_0) \subset \{ l_0, l_1, l_k \}$.

proposition 3 : let T be the length of an elementary circuit of $G_i(F)$. Then either T = 2 or T divides k+1.



a) n sites and k-length memory

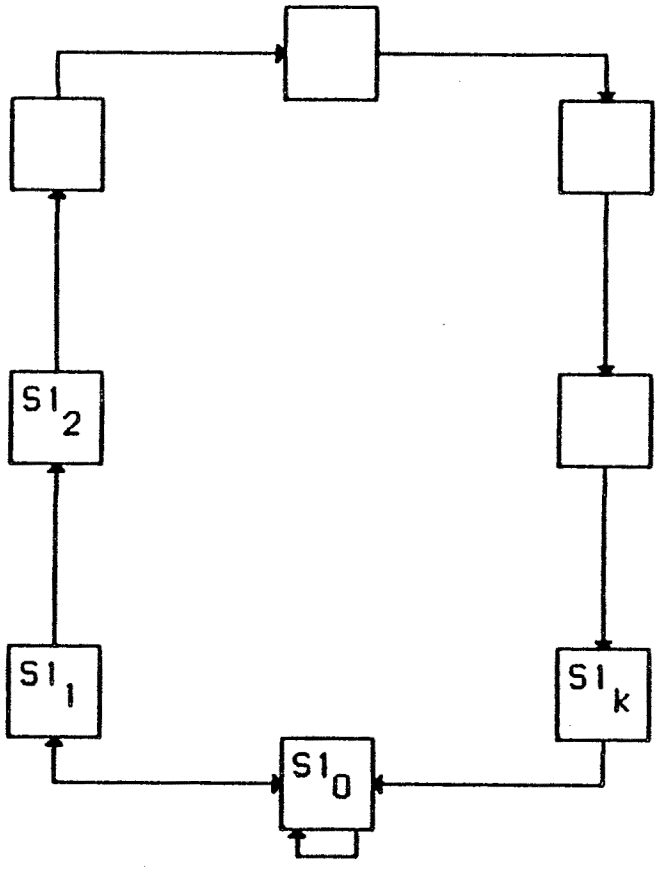


b) (k+1)*n sites, no memory (with k=3)

two opposite arcs are represented by a single undirected edge

loops are omitted

Figure 1



Single site with k-length memory (k=7)
$S1_0$ may only depend on itself, $S1_1$ and $S1_k$

Figure 2

proof : assume there exists an elementary circuit of $G_i(F)$ of length T different from 2. Then we can choose a time-step t such that

$$Sl_0(t) = Sl_0(t+1) \text{ and } Sl_0(t+2) \neq Sl_0(t)$$

For instance $Sl_0(t) = Sl_0(t+1) = 1$ and $Sl_0(t+2) = 0$.

Necessarily, $l_k \in FN(l_0)$ and $Sl_k(t+1) = 0$, otherwise by monotony we would have $Sl_0(t+u) = 0$ for all $u \geq 2$.

Using again the monotony of the transition function of Sl_0, we deduce that

$$Sl_0(t+u) = 0 \Rightarrow Sl_k(t+u-1) = 0 \text{ for } u \geq 2$$

and finally we have $Sl_0(t+u) = Sl_k(t+u-1) = 0$ for all $u \geq 2$, which proves that T divides k+1.

CASE 2

Consider now the automaton of figure 3: its connection-graph is of maximal degree k+1. Such an automaton has been extensively studied in the literature when the transition function is a linear function over the finite field Z/qZ, q prime [3] or when the transition function is a threshold function [1]. We deal here with a general monotone transition function. For small values of k ($k \leq 5$), all elementary circuits of $G_i(F)$ are of length smaller than or equal to k+1. However, we show that there exists elementary circuits of $G_i(F)$ whose length grows exponentially with k :

proposition 4 : there exist single monotone boolean automata with k-length memory whose iteration-graph contain elementary circuits of length growing exponentially with k (as k goes to infinity)

proof : we let (without loss of generality) k = 3 (m+2), m even, and we consider the following sequence

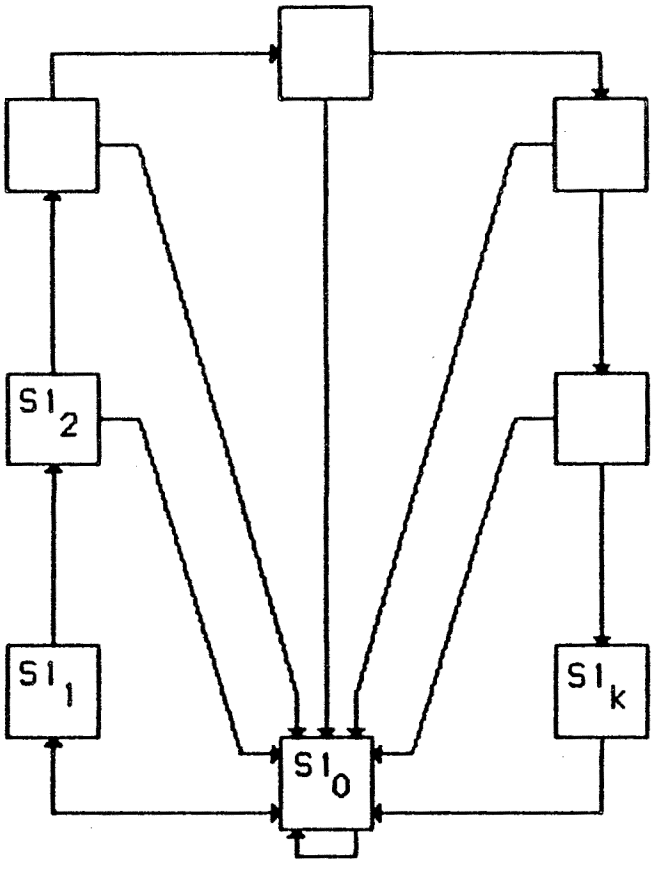
$$a_0\, a_1\, a_2 \ldots a_{T-1} = 1\, u_1\, 1\, 0^{m+2}\, 1\, u_2\, 1\, 0^{m+2} \ldots 1\, u_p\, 1\, 0^{m+2}$$

where power means concatenation and

- $a_i \in [0,1]$ for $0 \leq i \leq T-1$
- $u_1, u_2, \ldots, u_p$ are uncomparable slices of m digits : $u_i \in [0,1]^m$ for $1 \leq i \leq p$, and $(u_i \leq u_j \Rightarrow i=j)$ for $1 \leq i,j \leq p$.

Clearly, the maximal value of p for which these conditions hold is

$$p = \binom{m}{m/2}$$



Single site with k-length memory (k=7)
$S1_0$ may depend on every site $S1_i$, $0 \le i \le k$

Figure 3

lemma 3 : $a_i a_{i+1} \ldots a_{i+k}$ and $a_j a_{j+1} \ldots a_{j+k}$ are uncomparable slices of k+1 digits for all $i \neq j$, $0 \leq i,j \leq T-(k+1)$

proof of lemma 3 : assume that $i \neq j$. Since $k = 3(m+2)$, $a_i a_{i+1} \ldots a_{i+k}$ can be written as

$$I = a_i a_{i+1} \ldots a_{i+k} = A\ 1\ u_r\ 1\ 0^{m+2}\ B$$

for some sequences A, B, and some $r \leq p$. Writing in the same way

$$J = a_j a_{j+1} \ldots a_{j+k} = A'\ x\ U\ y\ Z\ B'$$

where $x,y \in \{0,1\}$, length(A') = length(A), and length(B') = m+2, we discuss the two cases:

- $x = y = 1$ then $U = u_s$ for some $s \leq p$, and u_r and u_s are uncomparable
- $x = 0$ or $y = 0$ then Z contains at least one "1"

In both cases, I and J are uncomparable.

Now, we simply define the transition function f by letting

$$f(a_t, a_{t+1}, \ldots, a_{t+k}) = a_{t+1} \text{ for all } t \geq 1$$

We extend this definition by monotony to the whole lattice $\{0,1\}^{k+1}$. In order to ensure that f is monotone, it is sufficient to show that any two different slices of k+1 consecutive elements of $a_1\ a_2\ \ldots\ a_T$ are uncomparable, and this follows from lemma 3. Then, we have

$$\begin{aligned} Sl_j(t) &= f(Sl_0(t-1), Sl_1(t-1), \ldots, Sl_k(t-1)) \\ &= f(a_{t-1}, a_{t-2}, \ldots, a_{t-k-1}) \\ &= a_t \end{aligned}$$

for all $t \geq k+1$ (all the indices are computed mod T).

According to Stirling's formula, we have $p_{max} \geq 2^{m+1}/m$ for m large enough and therefore

$$T = 2m * p_{max} \geq 2^{m+2} = 2^{k/3}$$

thereby establishing the proposition.

REMARK

The dynamics of general monotone networks with memory seems much more difficult to analyze. For instance, consider the automaton of figure 4 : it is composed of three sites with memory, and satisfies to the conditions of proposition 2 except that one of the site has a memory of length 2. However, there exists an elementary circuit of $G_1(F)$ of length 6.

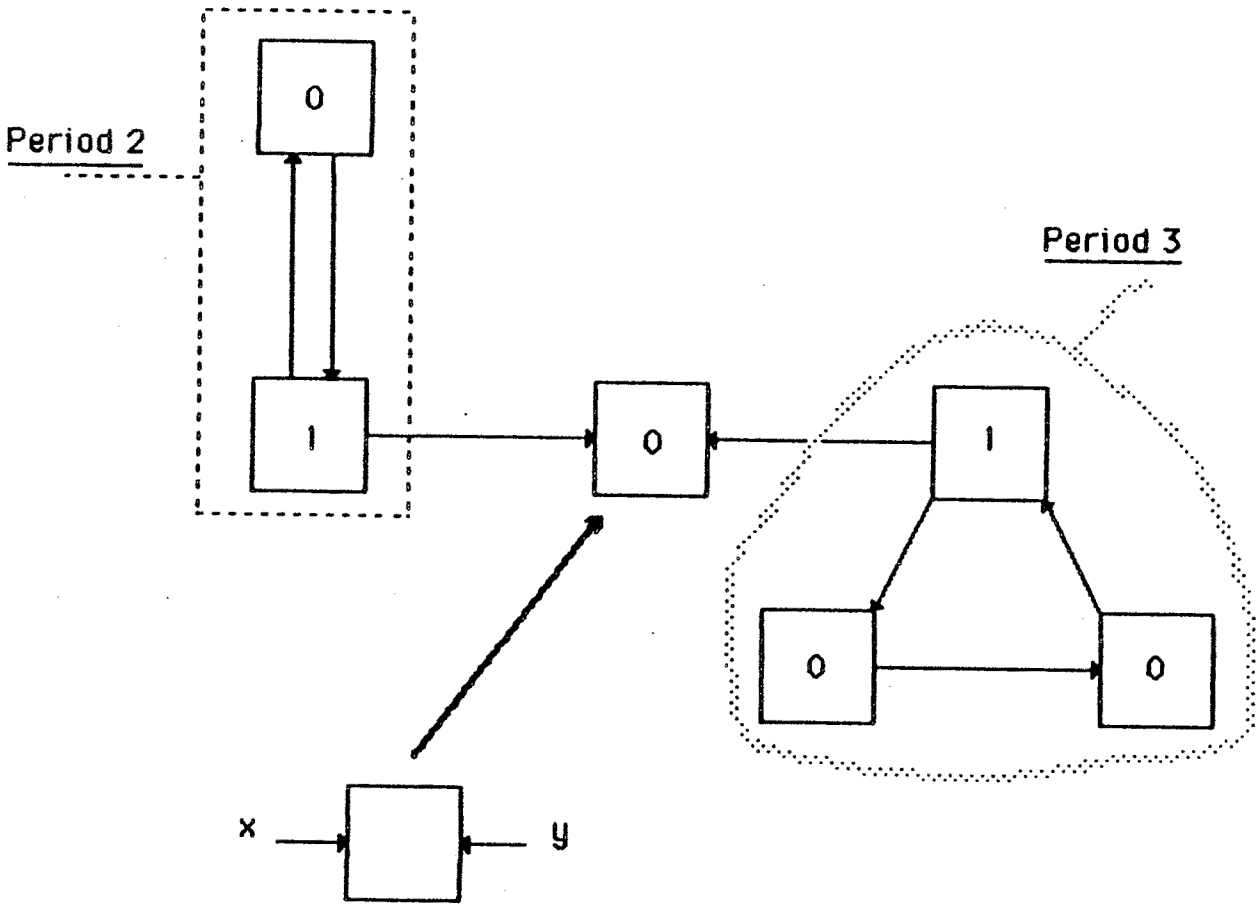


Figure 4

REFERENCES

[1] M. COSNARD, E. GOLES, Dynamical properties of an automaton with memory, NATO Advanced Research Workshop on Disordered systems and Biological Organization, Springer Verlag 1985

[2] F. FOGELMAN, Contributions à une théorie du calcul sur réseaux, Thesis, Grenoble (1985)

[3] E. GOLES, Comportement dynamique de réseaux d'automates, Thesis, Grenoble (1985)

[4] A. GILL, Linear sequential circuits, Analysis, Synthesis and Applications, Mc Graw-Hill Series in Computer Science, New York, 1966

[5] O. MARTIN, A. ODLYZKO, S. WOLFRAM, Algebraic properties of cellular automata, Bell Laboratories Report (1983)

[6] Y. ROBERT, M. TCHUENTE, Connection-graph and iteration-graph of monotone boolean functions, Discrete Applied Mathematics 11 (1985)

[7] M. TCHUENTE, Contribution à l'étude des méthodes de calcul pour des systèmes de type coopératif, Thesis, Grenoble (1982)

[8] S. WOLFRAM, Preface to Physica 10D (1984), Elsevier Science Publishers

CHAPITRE 6

SOUS-CHAPITRE 6.1.

MATHEMATICAL MODELLING AND NUMERICAL ANALYSIS
MODÉLISATION MATHÉMATIQUE ET ANALYSE NUMÉRIQUE
(vol. 19, n° 2, 1985, p. 315 à 326)

RÉSOLUTION SYSTOLIQUE DE SYSTÈMES LINÉAIRES DENSES (*)

par Yves ROBERT et Maurice TCHUENTE (1)

Communiquée par F. ROBERT

Résumé. — *On présente deux algorithmes systoliques pour la résolution de systèmes linéaires denses. Plutôt que de procéder en une étape de triangularisation suivie de la résolution d'un système triangulaire (comme les solutions systoliques déjà connues), nous utilisons directement la méthode de diagonalisation de Jordan. Malgré son coût élevé en nombre d'opérations élémentaires (ce qui explique sa non-utilisation dans le cas séquentiel), cette méthode conduit à des réseaux performants, et s'avère être la plus adaptée au calcul parallèle.*

Abstract. — *Two systolic arrays are proposed for the solution of dense linear systems. Instead of triangularising and then solving a triangular system (as suggested in previous systolic solutions), the Jordan diagonalisation method is used. In view of the better performances of the resulting networks, the Jordan method appears to be well-suited to parallel computation (contrarily to its high cost on sequential machines).*

1. INTRODUCTION

Les algorithmes numériques actuellement utilisés pour résoudre les problèmes qui se posent lors de l'étude en simulation du comportement de réacteurs nucléaires ou d'écoulements hydrodynamiques — pour ne reprendre que deux exemples dus à [16] — conduisent à une saturation quasi totale des machines séquentielles disponibles. Il est communément admis qu'il faut absolument, pour répondre aux besoins toujours croissants du Calcul Scientifique, gagner un facteur de cent au moins sur les vitesses de traitement, au cours de la prochaine décennie [3] : le parallélisme est à l'ordre du jour. Une première approche consiste à programmer au mieux les algorithmes séquentiels existants sur des machines vectorielles comme le CRAY 1 ou le CYBER

(*) Reçu en janvier 1984.
(1) CNRS, Laboratoire TIM3, IMAG, BP 68, 38402 Saint Martin d'Hères Cedex.

M² AN Modélisation mathématique et Analyse numérique, 0399-0516/85/02/315/12/$ 3,20

205. Une autre démarche reconsidère les algorithmes usuels pour leur donner une structure à fort degré de parallélisme, adaptée aux exigences des nouvelles architectures d'ordinateurs (SIMD, MIMD, processeurs intégrés spécialisés, voir [6] pour cette classification).

La première approche est plus directement orientée vers la pratique (puisque les machines vectorielles sont d'ores et déjà largement répandues), tandis que la deuxième reste encore principalement du domaine de la recherche (il existe cependant des réalisations, comme l'ordinateur HEP de Denelcor, Inc de type MIMD, ou le processeur intégré systolique de l'Université de Carnegie Mellon).

Nous nous intéressons ici à la résolution de problèmes linéaires denses. En ce qui concerne des architectures de type SIMD ou MIMD, cette question a déjà fait l'objet de nombreuses études, comme en témoignent les abondantes bibliographies des articles de synthèse de Sameh [16] et Heller [10]. L'apparition de solutions intégrées est plus récente (1978), puisque c'est seulement depuis quelques années que les progrès de la technologie VLSI permettent d'entrevoir la réalisation de cartes spécialisées que l'on adjoindra à un calculateur dit « hôte » pour accélérer le traitement de certains types de problèmes comportant un grand volume de calculs.

Le modèle systolique introduit par Kung et Leiserson [13] permet de concevoir des circuits intégrés spécialisés performants ayant un facteur de régularité et de modularité élevé. En un mot, il s'agit de structures-réseaux composés de processeurs (ou cellules) simples, régulièrement agencés, et combinant les deux types classiques de parallélisme :

— dans le temps (pipe-line),

— dans l'espace : plusieurs calculs sont effectués simultanément sur le réseau.

Ces caractéristiques conduisent à des calculs en temps réel (i.e. où les sorties sont délivrées au même rythme que les entrées). Kung décrit en détail dans [12] les avantages du modèle systolique, et on peut se référer à [2] pour une introduction et un exemple d'application.

Dans le paragraphe suivant, nous rappelons brièvement les méthodes de triangularisation et de diagonalisation pour la résolution de systèmes linéaires denses, qui nécessitent $0(n^3)$ opérations, et comment ces méthodes ont été parallélisées sur des architectures MIMD comportant $0(n)$ processeurs pour obtenir une résolution en $0(n^2)$ unités de temps. Nous présentons alors de nouvelles architectures systoliques composées de $0(n^2)$ cellules et réalisant des temps de calcul linéaires, en les comparant aux architectures systoliques déjà connues (Ahmed *et al.* [1]; Gentleman et Kung [9]; Kung et Leiserson [13]).

2. RÉSOLUTION EN PARALLÈLE DE SYSTÈMES LINÉAIRES

Soit A une matrice carrée d'ordre n et $Ax = b$ un système linéaire à résoudre. La plupart des méthodes directes sont des algorithmes séquentiels qui procèdent en deux étapes : triangularisation de la matrice A bordée du vecteur colonne b par prémultiplication par des matrices convenables, puis résolution du système triangulaire obtenu. Formellement, on peut écrire :

Programme de triangularisation du système

Soit $A' = (A, b)$ matrice de taille $n \times (n + 1)$
pour $i := 1$ à n faire
pour $j := i + 1$ à n faire la tâche T_{ij} :

$$\begin{pmatrix}\text{ligne } i\\ \text{ligne } j\end{pmatrix} := M_{ij}\begin{pmatrix}\text{ligne } i\\ \text{ligne } j\end{pmatrix}$$

où M_{ij} est déterminée de manière à annuler le coefficient a_{ji}.

Par exemple, en choisissant

$$(*)\quad M_{ij} = \begin{pmatrix}1 & 0\\ l_{ji} & 1\end{pmatrix},\quad l_{ji} = -a_{ji}/a_{ii}$$

on retrouve la méthode de Gauss sans pivotage. En choisissant pour M_{ij} une matrice de rotation plane, on obtient la méthode de Givens. Toutes ces méthodes ont un coût de $n^3/3$ opérations élémentaires, mais il faut résoudre ensuite un système triangulaire.

Une autre possibilité est de diagonaliser A directement, en annulant à l'étape i tous les coefficients a_{ji}, $j \neq i$. Le nombre d'opérations élémentaires est alors $n^3/2$, et la solution du système est directement obtenue en effectuant n divisions. Le choix précédent (*) pour les M_{ij} conduit à la méthode de Jordan.

Tous ces algorithmes peuvent être parallélisés en recherchant le maximum de tâches indépendantes, que l'on mènera simultanément. Les contraintes de séquencement imposées par le programme précédent sont les suivantes :

(R) T_{ij} précède $T_{i+1,j}$ ($j > i$ pour la triangularisation, et $j \neq i$ pour la diagonalisation).

La fermeture transitive de la relation (R) conduit au graphe de parallélisme maximal. Par exemple, $T_{12}, T_{13}, \ldots, T_{1n}$ peuvent être exécutées en parallèle. Pour la triangularisation d'une matrice de taille n, Sameh et Kuck [17] intro-

duisent un algorithme parallèle (asymptotiquement optimal [4]) s'exécutant en temps linéaire sur une machine SIMD comportant $[n/2]$ processeurs, tandis que Lord *et al.* [14] décrivent une parallélisation optimale utilisant le même nombre de processeurs pour une architecture MIMD. Enfin, des réseaux systoliques ont été proposés par Gentleman et Kung [9], Ahmed *et al.* [1] pour résoudre ce problème, et seront décrits en détail ci-dessous.

3. RÉSEAUX SYSTOLIQUES POUR LA MÉTHODE DE JORDAN

Pour définir une architecture systolique réalisant la triangularisation ou la diagonalisation des matrices, il faut se doter de trois types de cellules :

— des cellules de génération, capables de calculer les coefficients des matrices M_{ij},

— des cellules de propagation, capables de recevoir, combiner et transmettre les informations en provenance des cellules voisines,

— éventuellement, des cellules de retard.

D'autre part, comme pour les machines SIMD ou MIMD, il faut définir un séquencement pour l'exécution des tâches élémentaires, et organiser le croisement des flots de données régis par le séquencement.

Ainsi, pour la triangularisation d'une matrice A de taille n, bordée du vecteur b dont les composantes sont notées $b_i = a_{i,n+1}$, Gentleman et Kung [9] définissent un réseau comportant $n(n+3)/2$ cellules, dont n cellules de génération (voir *fig.* 1). La matrice A est introduite par la partie supérieure du réseau, un port d'entrée recevant tous les éléments d'une même colonne. Lorsqu'à l'instant i, a_{1i} ($1 <= i <= n+1$) arrive dans le réseau, il est stocké dans le registre interne de la cellule qui le reçoit en entrée (i-ième cellule de la première ligne); l'instant d'après, cette cellule dispose de a_{1i}, a_{2i} comme données, et aussi de la matrice M_{21} générée par la cellule recevant a_{11} : elle dispose ainsi de tous les éléments pour modifier a_{2i} suivant la transformation M_{21} conduisant à l'annulation de a_{21}. Le même raisonnement s'applique pour les M_{i1}, $i >= 3$. De manière analogue, les cellules de la ligne j permettront l'annulation des coefficients a_{kj}, $k > j$, et la modification correspondante des éléments a_{1m}, 1 et $m >= j+1$.

A la fin de cette étape (au bout de $3n$ unités de temps, une unité correspondant au temps de cycle maximal d'une cellule), on obtient un système triangulaire $Tx = c$, la matrice $S = (T, c)$ est stockée dans le réseau selon le schéma de la figure 2. Gentleman et Kung proposent alors d'utiliser un réseau de n cellules défini dans [13] pour la résolution des systèmes triangulaires. Cette

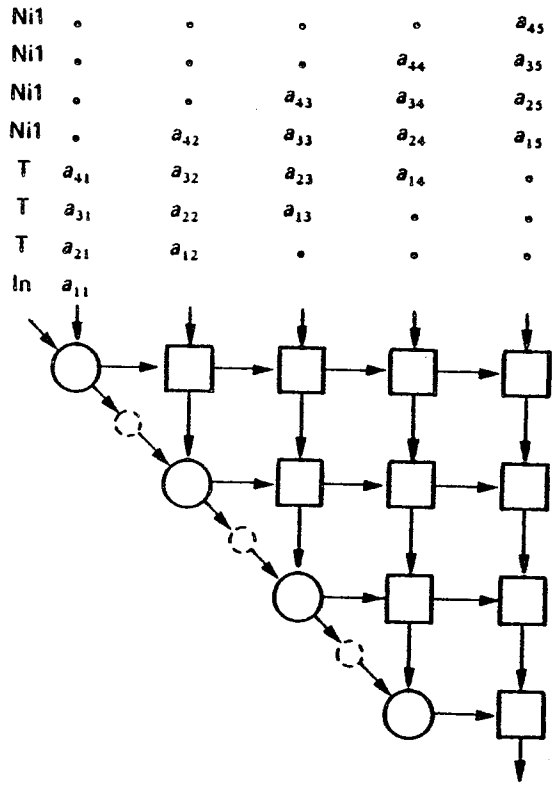


Figure 1*a*. — Structure générale du réseau.

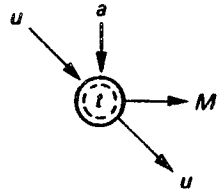
solution nécessite en plus l'emploi de $n(n-1)/2$ cellules de retard, et un temps de calcul additionnel de $2n$ tops d'horloge. Nous allons ici résoudre directement le système triangulaire obtenu par une deuxième phase d'activation du réseau.

On commence par envoyer un signal à la cellule contenant s_{11} : à la réception de ce signal, cette cellule génère la matrice

$$\begin{pmatrix} 0 & 0 \\ 1/s_{11} & 0 \end{pmatrix}.$$

Si les cellules de la première ligne continuent à fonctionner comme dans la phase 1, alors chaque élément s_{1i} sera divisé par s_{11} puis envoyé dans la cellule voisine du bas. Notons $s'_{1i} = s_{1i}/s_{11}$; étant donné que s'_{1i+1} est calculé un top d'horloge après s'_{1i}, tout se passe comme si on avait la situation de la figure 3. En conséquence, si toutes les cellules de génération fonctionnent

Cellule de génération

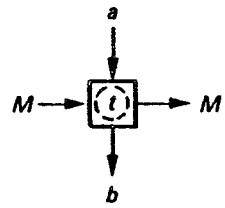


$a, u \in R$; $M \in M_{2\times 2}(R)$; $u \in \{$ Nil, In, Ga, Pv, Gi $\}$ où In, Ga, Pv, Gi sont respectivement les abréviations de Initialisation, Gauss, Pivots-voisins et Givens :

si u = Nil alors $M := \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ sinon
si u = In alors $M := \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ sinon
si u = Ga alors $M := \begin{pmatrix} 1 & 0 \\ -a/t & 1 \end{pmatrix}$ sinon
si u = Pv alors
début si $abs(a) > abs(t)$ alors $M := \begin{pmatrix} 1 & -t/a \\ 0 & 1 \end{pmatrix}$ sinon
si $a = t = 0$ alors $M := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ sinon
$M := \begin{pmatrix} 1 & 0 \\ -a/t & 1 \end{pmatrix}$
fin
sinon
si u = Gi alors
début si $a = t = 0$ alors $M := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ sinon
$M := \begin{pmatrix} a/a^2 + t^2 & -t/a^2 + t^2 \\ t/a^2 + t^2 & a/a^2 + t^2 \end{pmatrix}$
fin ;
$t := m_{11}.t + m_{12}.a$;

Cellule de combinaison

$$\begin{pmatrix} t \\ b \end{pmatrix} := M.\begin{pmatrix} t \\ a \end{pmatrix}.$$



Cellule de retard

Figure 1b. — Structure des cellules.

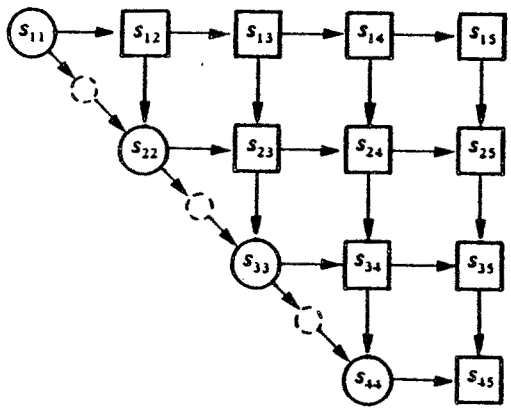


Figure 2.

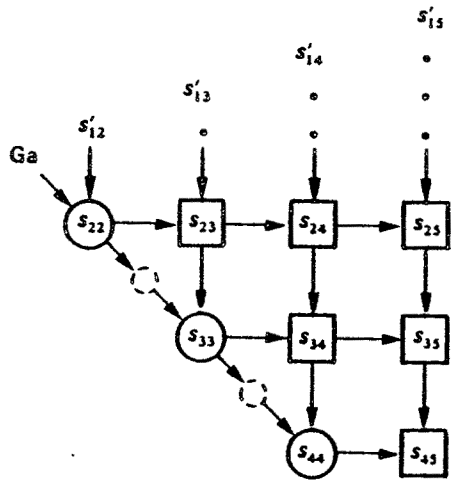


Figure 3.

comme dans la phase 1, mais en générant cette fois-ci les matrices 2×2 correspondant exclusivement à l'élimination de Gauss, alors on peut éliminer successivement s'_{12}, s'_{13}, ..., s'_{1n}; à la sortie du réseau, on aura $s'_{1n+1} = x_1$. Le même raisonnement est valable pour toutes les autres lignes. De plus :

— au cours de cette deuxième phase, les traitements correspondant aux lignes 1, 2, ..., n peuvent être pipelinés,

— la phase 1 et la phase 2 peuvent également être pipelinées.

On obtient finalement le réseau de la figure 4. Le temps total de calcul de la solution x du système linéaire est de $4n$ tops d'horloge (au lieu de $5n$ pour

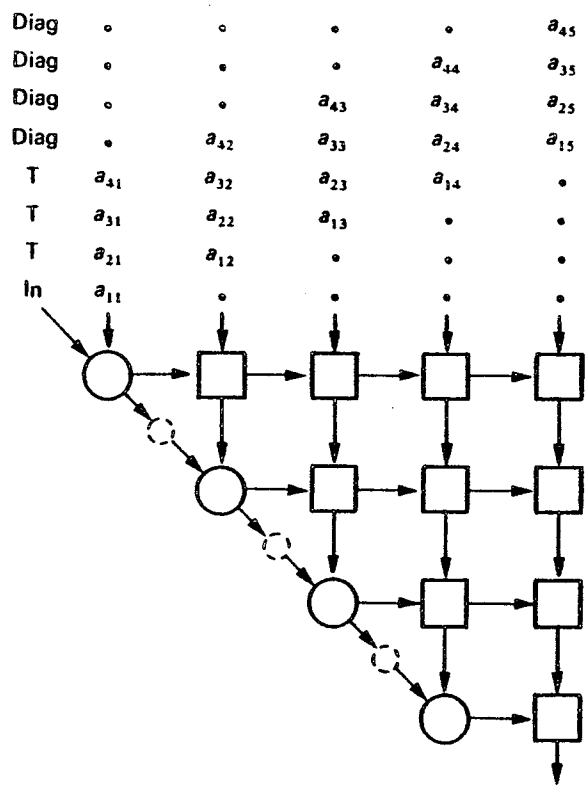
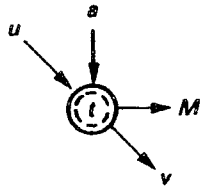


Figure 4a. — Structure générale du réseau.

Cellule de génération

$u \in$ { Nil, In, Ga, Gi, Pv, Diag }. Diag est le signal de diagonalisation. La cellule exécute en plus l'instruction suivante :



si u = Diag alors
début si $t \neq 0$ alors
début $t := 0$;
$v :=$ Nil;
$M := \begin{pmatrix} 1/t & 0 \\ 0 & 0 \end{pmatrix}$
fin
sinon
$v :=$ Diag
fin
sinon
$v := u$;

Figure 4b. — Structure de la nouvelle cellule de génération.

[9]) et nous n'avons utilisé que les $n(n + 3)/2$ cellules du réseau de triangularisation.

A condition de doubler le nombre de cellules, il est possible de diagonaliser la matrice A en seulement $4n$ tops d'horloge, en employant directement la méthode de Jordan. Compte tenu du fait que dans l'algorithme, toutes les colonnes de la matrice jouent maintenant des rôles similaires (on doit annuler n coefficients dans chacune d'elles), l'idée-clé est de préparer un réseau rectangulaire de $n(n + 2)$ cellules (dont n de génération et n de retard) dans lequel les informations correspondant aux matrices M_{ij} seront stockées dans les cellules, alors que les coefficients de la matrice circuleront ligne par ligne dans le réseau. En ne conservant que la partie triangulaire inférieure de ce nouveau réseau, on obtient le réseau de triangularisation que présentent Ahmed *et al.* dans [1].

Cette deuxième solution pour diagonaliser A est décrite figure 5 (noter que les éléments de la matrice diagonale obtenue sont stockés dans la dernière ligne du réseau). On vérifie bien que le système $Ax = b$ est maintenant résolu en $4n$ tops d'horloge, le prix à payer pour l'emploi de cette méthode étant le doublement de la surface.

Remarques

A. *Choix de la méthode de triangularisation*

Si la matrice A n'est pas symétrique définie positive ou à diagonale dominante, il peut être préférable, pour des raisons de stabilité, d'opter pour la méthode de Givens. Les cellules de propagation doivent alors calculer des racines carrées, et leur temps de cycle devient nettement supérieur à celui des autres cellules. Deux remèdes à cette situation peuvent être envisagés :

— l'emploi de la méthode de Givens sans racines carrées, due à Gentleman [8]. Gentleman et Kung [9] discutent cette solution, qui complique le réseau :

— l'emploi de processeurs spécialisés, de type Cordic [1], pour lesquels multiplication et extraction de racine carrée sont du même ordre de complexité.

Notons que le dernier réseau proposé peut réaliser sans difficulté la triangularisation par la méthode de Givens : il suffit pour cela de programmer les cellules du réseau, celles de la partie supérieure réalisant des transformations de Givens et celles de la partie inférieure des transformations de Gauss.

B. *Résolution de plusieurs systèmes*

Si on a p systèmes linéaires $Ax_i = b_i$, $1 <= i <= p$, à résoudre avec la même matrice A :

— pour la première solution, il est nécessaire d'ajouter $p - 1$ colonnes à la droite du réseau, pour un temps total de résolution de $4n + p$ tops d'horloge,

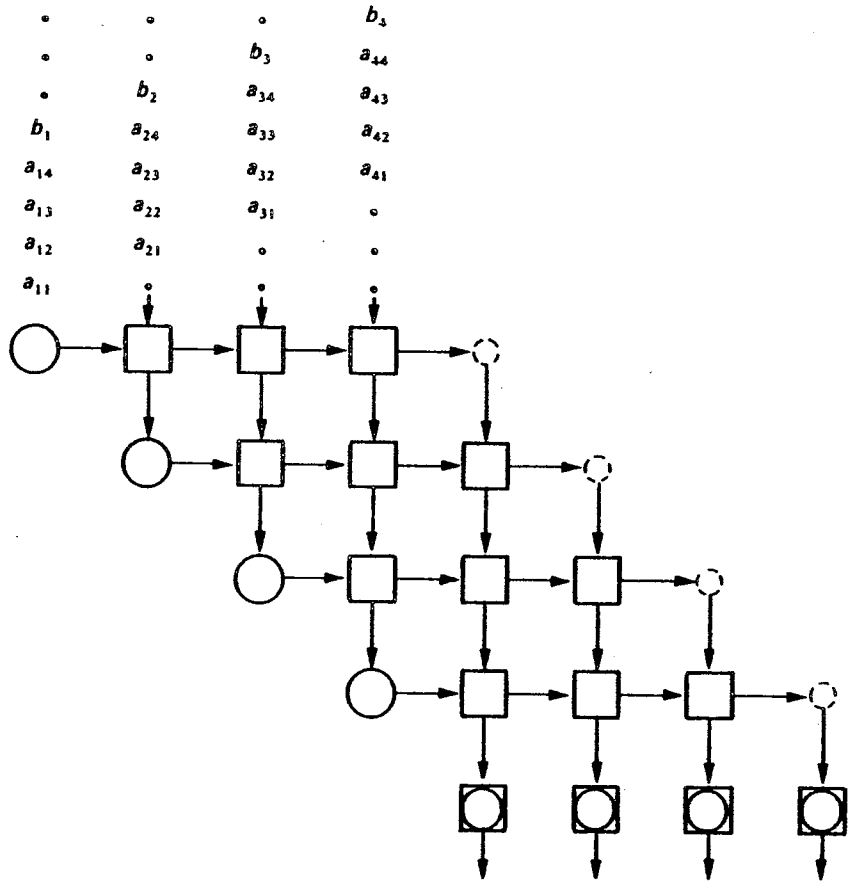


Figure 5*a*. — Structure générale du réseau.

Cellule de division

t est initialisé à 0.

si $t = 0$ alors $t := x$
sinon
$y := x/t$;

Figure 5*b*. — Structure de la cellule de division.

— pour la deuxième solution, il suffit de compléter chaque ligne j entrant dans le réseau par les j-ièmes composantes des vecteurs $b_2, ..., b_p$: le nombre de cellules reste inchangé, et le temps de calcul est encore de $4n + p$. Cette solution s'avère donc préférable si p est grand en particulier, le calcul de l'inverse d'une matrice peut se faire en temps $5n$ sur le réseau de la figure 5.

C. *Cas où la matrice du système est symétrique*

Dans le cas des matrices symétriques, Delosme [5] propose un réseau de $n^2/2$ cellules pour résoudre le système $Ax = b$; le facteur de Choleski L de la matrice $A = L^t L$ est tout d'abord déterminé en $2n$ tops d'horloge (par utilisation d'une extension de l'algorithme de Schur-Levinson). Toutes les composantes du produit $L^{-1} b = c$ sont alors disponibles au top suivant, et renvoyées dans le réseau qui délivre le vecteur solution x après n tops d'horloge supplémentaires; le système est ainsi résolu en temps total $3n + 1$. Ce réseau réalise donc le meilleur compromis temps-surface dans le cas symétrique.

D. *Cas des matrices bandes*

Plusieurs réseaux ont été proposés pour la triangularisation des matrices bandes ($a_{ij} = 0$ si $i - j > p$ ou $j - i > q$) : voir [9, 11, 13, 15]. Tous ont l'intérêt d'utiliser un nombre de cellules dépendant uniquement de la largeur $w = p + q - 1$ de la bande, et non de la taille de la matrice. Aucun ne semble cependant pouvoir se prêter à la résolution directe d'un système linéaire à structure bande.

CONCLUSION

Pour la résolution des systèmes linéaires, l'emploi de la méthode de Jordan conduit à des réseaux systoliques plus performants que ceux obtenus par l'utilisation d'une méthode de triangularisation suivie de la résolution d'un système triangulaire. Cette conclusion s'inscrit à l'opposé du critère séquentiel qui au vu du nombre d'opérations, conduit à préférer la deuxième méthode (pour un coût de l'ordre de $n^3/3$) à la première (nécessitant $n^3/2$ transformations élémentaires); elle illustre bien le fait que les critères d'évaluation des algorithmes, traditionnellement basés sur les opérations arithmétiques et ignorant les temps de communication et la structure des données, doivent être révisés en calcul parallèle. Dans le domaine particulier de l'implantation VLSI, les bons algorithmes ne sont pas ceux qui nécessitent le moins de calculs : comme il est souligné dans [7], les calculs sont peu coûteux en VLSI, ce sont les communications qui déterminent les performances.

REFFRENCES

[1] H. M. AHMED, J. M. DELOSME, M. MORF, *Highly concurrent computing structures for matrix arithmetic and signal processing*. Computer magazine, January 1982, pp. 65-82.

[2] F. ANDRÉ, P. FRISON, P. QUINTON, *Algorithmes systoliques : de la théorie à la pratique*, Rapport de Recherche INRIA n° 214, 1983.

[3] A. BOSSAVIT, *Préface des actes du colloque AFCET-GAMNI-ISINA*, 17-18 mars 1983, Paris, Bulletin de la direction des études et recherches EDF, série C, vol. 1, 1983.

[4] M. COSNARD, Y. ROBERT, *Complexité de la factorisation QR en parallèle*, C. R. Acad. Sc. Paris, t. 297, Série I, pp. 137-139 (septembre 1983).

[5] J. M. DELOSME, *Algorithms for finite shift-rank processes*, Ph. D., Technical Report M735-22, September 1982, Stanford Electronics Laboratories.

[6] M. FLYNN, *Some computer organisations and their effectiveness*, IEEE Trans. on Computers C21, 9 (1972), pp. 948-960.

[7] M. J. FOSTER, H. T. KUNG, *The design of special-purpose VLSI chips*, IEEE Computer 13, 1 (January 1980), pp. 26-40.

[8] W. M. GENTLEMAN, *Least squares computation by Givens transformations without square roots*, J. Inst. Math. Appl. 12 (1973) pp. 329-336.

[9] W. M. GENTLEMAN, H. T. KUNG, *Matrix triangularisation by systolic arrays*, Proc. SPIE 298, Real-time Signal Processing IV, San Diego, California, 1981.

[10] D. HELLER, *A survey of parallel algorithms in numerical linear algebra*, Siam Review 20, pp. 740-777, 1978.

[11] D. HELLER, I. IPSEN, *Systolic networks for orthogonal equivalence transformations and their applications*, Proc. 1982 Conf. Advanced Research in VLSI, pp. 113-122, MIT 1982.

[12] H. T. KUNG, *Why systolic architectures*, IEEE Computer 15, 1 (January 1982), pp. 37-46.

[13] H. T. KUNG, C. E. LEISERSON, *Systolic Arrays for (VLSI), in the proceedings of the Symposium on sparse matrix computations and their applications*, Knoxville, 1978.

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

[15] L. MELKEMI, M. TCHUENTE, *Systolic arrays for connectivity and triangularisation problems*, to appear in Proc. « Dynamical Systems and Cellular Automata », J. Demongeot, E. Coles et M. Tchuente eds., Academic Press, 1985.

[16] A. SAMEH, *Numerical parallel algorithms — a survey*, in « High Speed Computer and Algorithm Organization », D. Kuck, D. Lawrie and A. Sameh eds., pp. 207-228, Academic Press, 1977.

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

Intern. J. Computer Math., 1985, Vol. 17, pp. 295–315
0020-7160/85/1703-0295 $18.50/0

Block *LU* Decomposition of a Band Matrix on a Systolic Array

Yves ROBERT

CNRS, Laboratoire TIM3, Institut IMAG, BP 68, 38402 Saint Martin d'Heres Cedex, France

(*Received October, 1984*)

After a brief discussion on systolic arrays for band matrix *LU* or *QR* decomposition, we introduce a new systolic array for the block 2×2 *LU* decompositon of a band matrix A. This array is an hexagonally connected systolic array whose efficiency is $e = \frac{1}{2}$ although its hardware requirement is the same as the *LU* decomposition array of Kung and Leiserson [8]. In the last section we consider the systolic solution of linear systems of matrix A: we show that computing the block *LU* decomposition of A is twice more efficient than computing the usual *LU* decomposition of A.

KEY WORDS: Systolic array, *LU* decomposition, band matrix, block method, area–time performance.

C.R. CATEGORIES: F.1.1 (Computation by abstract devices): Models of Computation—*Systolic array*; F.2.1 (Analysis of algorithms and problem complexity): Numerical algorithms and problems—*LU decomposition of a band matrix*; B.7.1 (Integrated Circuits): Types and design styles—*Algorithms implemented in hardware, VLSI*.

1. INTRODUCTION

In this paper we present a new systolic array for band matrix block 2×2 *LU* decomposition. Introducing block 2×2 methods seems the only way to improve the efficiency $e = \frac{1}{3}$ of the hexagonally connected

LU decomposition array of Kung and Leiserson [8]. Our new array has the same number of cells as that of Kung and Leiserson and its efficiency is $e=\frac{1}{2}$. Moreover the block 2×2 *LU* decomposition of a matrix A appears better suited to the resolution of linear systems $Ax=b$ than the usual *LU* decomposition, as discussed in the last section.

1.1 Preliminary concepts

In discussing parallel processor arrays we use terms and conventions that have become standard (see [3], [7] for background). In particular, the term efficiency will denote the fraction of processor cycles during which a typical processor is actively employed in the array. Although coalescing adjacent processors or interleaving on the array the solution of different instances of the same problem to be solved (as suggested in [8]) could increase the area–time performance of the array, we point out that the efficiency really evaluates the intrinsic degree of parallelism which the array can achieve.

For matrix computations on systolic arrays, the fundamental operation is a multiply-and-add performable by an inner-product-step (IPS) cell. A unit of time is thus the time necessary to achieve such a multiply-and-add. We assume that addition and subtraction on the one hand, multiplication and division on the other hand have the same order of complexity: an IPS cell may thus perform a divide-and-subtract.

Throughout the paper, A is a n by n band matrix with p superdiagonals and q subdiagonals. We let $w=p+q-1$ denote the band width of A, i.e. the number of its non-zero diagonals:

$$A=\begin{array}{c} \overbrace{}^{p} \\ \left(\begin{array}{ccccccccc} x & x & x & x & & & & & \\ x & x & x & x & x & & & & \\ x & x & x & x & x & x & & & \\ & x & x & x & x & x & x & & \\ & & x & x & x & x & x & x & \\ & & & x & x & x & x & x & x \\ & & & & x & x & x & x & x \\ & & & & & x & x & x & x \end{array}\right) \end{array} \begin{array}{l} \\ q \end{array}$$

Although dense matrices may be considered as band matrices with maximum band width, there is a great difference between designing systolic arrays whose inputs are band matrices and arrays whose inputs are general dense matrices. Indeed, when designing arrays for band matrices, the hardware requirements should be proportional to the width w of the band, and the coefficients of the matrix will be fed into the array diagonal by diagonal, each diagonal corresponding to an input port.

1.2 On systolic arrays for band matrix decomposition

For the *LU* or *QR* decomposition of a matrix A, A is premultiplied by some simple matrices, either elementary Gaussian matrices (which leads to the *LU* decomposition of A) or Givens' plane rotations (leading to the *QR* decomposition). For both algorithms, the elementary computation consists of combining two rows of A. This complicates the flow of data if the coefficients are entered in a diagonal fashion, and explains the fact that band matrix decomposition arrays are less efficient than those designed for dense matrices, such as [2], [4] which are both of maximal efficiency $e=1$.

Let us discuss further the usual algorithms to triangularise A: for the purpose of illustration we choose a matrix A with $p=q=3$:

$$A=\begin{pmatrix} x & x & x & & & & \\ x & x & x & x & & & \\ x & x & x & x & x & & \\ & x & x & x & x & x & \\ & & x & x & x & x & x \\ & & & x & x & x & x \\ & & & & \ddots & \ddots & \ddots \end{pmatrix}$$

In order to zero a coefficient a_{ij}, $i>j$, the row i will be combined with some row k ($k<i$):

$$\begin{pmatrix} a_{kj} \\ 0 \end{pmatrix} := M_{ij} \cdot \begin{pmatrix} a_{kj} \\ a_{ij} \end{pmatrix}$$

$$M_{ij}=\begin{pmatrix} 1 & 0 \\ l_{ij} & 1 \end{pmatrix} \rightarrow LU \text{ decomposition}$$

$$M_{ij}=\begin{pmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{pmatrix} \rightarrow QR \text{ decomposition}$$

There are two possible strategies for choosing k: $k=i-1$ leads to a diagonal elimination scheme, and $k=j$ leads to a column elimination scheme.

* *combining row i and row i−1*

Using this algorithm the subdiagonals of A will be successively annihilated: let t denote the time-step during which a_{41} is annihilated:

Step t	annihilate a_{41} with a_{31} by combining row 3 and row 4;
Step t+1	modify a_{42} with a_{32} and the parameters of M_{41};
Step t+2	modify a_{43} with a_{33} and the parameters of M_{41} annihilate a_{52} with a_{42} by combining row 4 and row 5.

It is possible to pipeline these annihilations: at Step $t+1$ we may begin the annihilation of the next subdiagonal, i.e. zero a_{31} with a_{21} by combining row 2 and row 3; at Step $t+2$ we shall modify a_{32} with a_{22} and the parameters of M_{31}.

The resulting array will lead to the decomposition of A in time $2n+0(w)$. Using such a strategy, which we name a diagonal elimination scheme, the Heller–Ipsen QR decomposition array [5] achieves the decomposition in time $2n+2q-1$.

* *combining row i and row j*

Here the columns of A will be successively annihilated. The same coefficient a_{jj} will be used as pivot to zero all the $a_{j,j+k}$, for $k=1$ to $k=q-1$.

Let t denote now the time-step during which a_{21} is annihilated:

Step t	annihilate a_{21} with a_{11} by combining row 1 and row 2;
Step t+1	modify a_{22} with a_{12} and the parameters of M_{21} annihilate a_{31} with a_{11} by combining row 1 and row 3;
Step t+2	modify a_{23} with a_{13} and the parameters of M_{21} modify a_{32} with a_{12} and the parameters of M_{31} annihilate a_{41} with a_{11} by combining row 1 and row 4;

Step $t+3$ modify a_{24} with a_{14} and the parameters of M_{21}
modify a_{33} with a_{13} and the parameters of M_{31}
modify a_{42} with a_{12} and the parameters of M_{41}
annihilate a_{32} with a_{22} by combining row 2 and row 3.

Unless we accept the possibility to broadcast the pivot a_{11} to all the rows within the same time-step, the elimination of the coefficients in the second column of A cannot begin before Step $t+3$. The resulting array will thus lead to the decomposition in time $3n+0(w)$. We call this method a column elimination scheme. Kung and Leiserson [8] introduce such an array: they obtain the *LU* decomposition in time $3n+\min(p, q)$.

Assume that p and q are much smaller than n: $w \ll n$, i.e. A is really banded. The first (diagonal) scheme leads to systolic arrays of efficiency $e=\frac{1}{2}$, whereas the second (column) scheme only permits to achieve a decomposition with efficiency $e=\frac{1}{3}$.

However, there are situations where the second strategy is to be preferred: most decomposition methods require partial or global pivoting (i.e. exchanging rows) for reasons of stability. To avoid pivoting with the first method it is usually necessary to perform plane rotations which are much more stable than Gaussian combinations of rows (adding to some row the multiple of another). These rotations entail the computation of square roots and as a consequence the cycle delay of an elementary processor will be longer than that of an IPS cell. We refer the reader to the paper of Ahmed *et al.* [1] who suggest the use of Cordic processors to fasten the cycle delay, and to the paper of Ipsen [6] where more sophisticated elimination matrices are introduced.

Conversely, it is well known that some matrices do not require pivoting for *LU* decomposition, such as irreducible diagonally dominant or symmetric positive definite matrices. These are cases where column strategies could be chosen. Let us point out that in most applications (e.g., signal processing [1]) the matrices to be triangularised fulfil one of these properties.

The aim of this paper is to present a band matrix decomposition array whose efficiency is $e=\frac{1}{2}$ and whose processors only perform addition, subtraction, multiplication or division but no square root computations: this array will be obtained by the use of a block 2×2 *LU* decomposition method.

2. A BAND MATRIX BLOCK *LU* DECOMPOSITION ARRAY

2.1 The block 2×2 *LU* decomposition algorithm

During the first elimination step of the usual *LU* decomposition (without pivoting) of an $n \times n$ band matrix A with band width $p+q-1$, we choose a_{11} for pivot and write

$$A = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ & & 1 & \\ l_{q1} & & & \\ & & & 1 \end{bmatrix} \cdot \begin{bmatrix} a_{11} & a_{12} & & a_{1p} \\ 0 & & & \\ & & & A^{(2)} \\ 0 & & & \\ 0 & & & \end{bmatrix}$$

where $l_{i1} = -a_{i1}/a_{11}$, $2 \leq i \leq q$.

Equivalently, we can write too

$$A = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ & & 1 & \\ l_{q1} & & & \\ & & & 1 \end{bmatrix} \cdot \begin{bmatrix} a_{11} & 0 & & \\ 0 & A^{(2)} & & \\ & & & \end{bmatrix} \cdot \begin{bmatrix} 1 & u_{12} & & u_{1p} \\ & 1 & & \\ & & 1 & \\ & & & \\ & & & 1 \end{bmatrix}$$

where $u_{1i} = a_{1i}/a_{11}$, $2 \leq i \leq p$.

This last decomposition will lead after $n-1$ elimination steps to the *LDU* decomposition of A.

Let us assume (without any loss of generality) that n is even whereas p and q are odd: $n=2m$, $p=2r+1$ and $q=2s+1$. We partition A into 2×2 sub-blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} & & A_{1r} \\ A_{21} & A_{22} & \ddots & \ddots \\ & \ddots & \ddots & \\ A_{s1} & \ddots & & \end{pmatrix}$$

Now, to perform the first elimination step, we choose A_{11} for pivot:

$$A=\begin{bmatrix} I & & & \\ L_{21} & I & & \\ & & \ddots & \\ L_{s1} & & & I \end{bmatrix} \cdot \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1r} \\ 0 & & & \\ 0 & & & \\ & & & \\ 0 & & & \end{bmatrix}$$

where $L_{i1} \cdot A_{11} = -A_{i1}$, $2 \leq i \leq s$.

Or, equivalently

$$A=\begin{bmatrix} 1 & & & \\ L_{21} & I & & \\ & & I & \\ L_{s1} & & & I \end{bmatrix} \cdot \begin{bmatrix} A_{11} & 0 \\ 0 & A^{(2)} \end{bmatrix} \cdot \begin{bmatrix} I & U_{12} & \cdots & U_{1r} \\ & I & & \\ & & I & \\ & & & I \end{bmatrix}$$

where $A_{11} \cdot U_{1i} = A_{1i}$, $2 \leq i \leq r$.

This process of computation is repeated iteratively, leading either to a block *LU* decomposition or to a block *LDU* decomposition of A.

Before discussing the parallel implementation of this block decomposition, let us point out that [10]

- any matrix which admits a *LU* decomposition admits a block *LU* decomposition,
- just as for the usual *LU* decomposition, no pivoting is required for the block decomposition when the matrix is symmetric positive definite or irreducible and diagonally dominant.

2.2 Computing the coefficients of the block *LU* decomposition

We concentrate our work on the first four rows and columns of the matrix A (deriving then directly the general case). So we consider

$$A=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad (A_{ij} \text{ is a } 2 \times 2 \text{ sub-block}).$$

We have

$$\begin{bmatrix} I & 0 \\ L_{21} & I \end{bmatrix} \cdot \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22}+L_{21}\cdot A_{12} \end{bmatrix}$$

where

$$L_{21} = \begin{pmatrix} l_{31} & l_{32} \\ l_{41} & l_{42} \end{pmatrix} \text{ is a solution of } L_{21}\cdot A_{11} = -A_{12}.$$

Thus, setting $D = a_{11}a_{22} - a_{12}a_{21}$:

$$\begin{cases} D\cdot l_{31} = a_{21}a_{32} - a_{31}a_{22} \\ D\cdot l_{32} = a_{12}a_{31} - a_{32}a_{11} \end{cases}.$$

Similar relations holds for $D\cdot l_{41}$ and $D\cdot l_{42}$.

Finally:

$$(R)\begin{cases} (a_{33}, a_{34}) := (a_{33}, a_{34}) + (l_{31}, l_{32})\cdot A_{12} \\ \qquad\qquad := (a_{33}, a_{34}) + (D\cdot l_{31}, D\cdot l_{32})\cdot (A_{12}/D) \\ \text{with } A_{12} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix}. \end{cases}$$

To get the value of U_{12} whenever a block LDU decomposition is needed, we have to premultiply A_{12} by the inverse of A_{11}. Since the coefficients of A_{12} have been already divided by the determinant D of A_{11}, we only have to compute four determinants:

$$U_{12} = \begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix} \cdot (A_{12}/D).$$

We have in mind the parallel implementation of this block decomposition. We shall use a hexagonally connected systolic array: the same geometry as the LU decomposition array of [8]. The array of [8] involves IPS cells and a division cell which computes the reciprocal of its input. We shall need here IPS cells, division cells

and another type of cells, which we call determinant cells, to handle the treatment of the 2×2 sub-blocks.

A determinant cell is a cell with four inputs a, b, c, d and which can compute the determinant $ad-bc$. For physical realisation of such a cell, two multipliers, one subtracter and some latches are needed, as shown in Figure 1. The two multipliers are controlled by the same internal clock and hence the subtraction can be pipelined with the two multiplications, i.e. it can begin when the first bits of ad and bc are available. Thus, the cycle delay of a determinant cell is exactly the same as the one of an IPS cell.

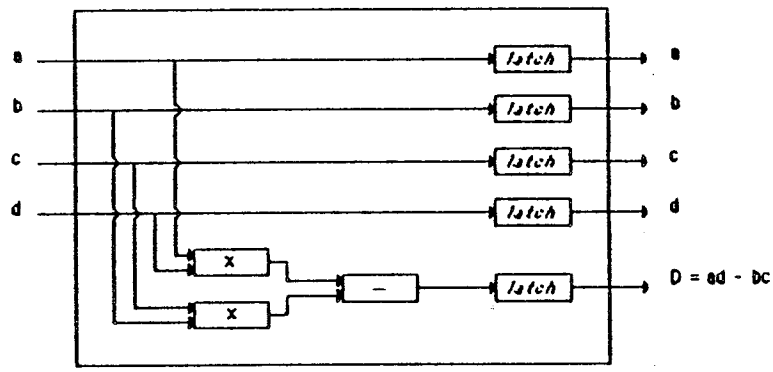


FIGURE 1 A determinant cell.

Now, getting back to the expressions (R), let us evaluate the number of time steps we need to update the coefficients of A_{22}. Consider for instance the updating of (a_{33}, a_{34}):

Step 1 compute the determinant $D=a_{11}a_{22}-a_{12}a_{21}$;
Step 2 compute the determinant $(D \cdot l_{31})$; compute $a_{13}/D, a_{14}/D$;
Step 3 update a_{33} and a_{34} according to

$$a_{33} := a_{33} + (D \cdot l_{31}) \cdot (a_{13}/D);$$
$$a_{34} := a_{34} + (D \cdot l_{31}) \cdot (a_{14}/D)$$

compute the determinant $(D \cdot 1_{32})$; compute $a_{23}/D \cdot a_{24}/D$;
Step 4 update a_{33} and a_{34} to get their final values.

Of course, a similar computation is done in parallel to update

(a_{43}, a_{44}). We conclude that four steps are necessary to update a 2×2 sub-block.

Then, consider again the general problem of computing the block 2×2 LU decomposition of the matrix A: at the fifth time-step the sub-block A_{22} will be taken for pivot to begin the second elimination step of the decomposition. We deduce that A_{11} and A_{22} must be separated by four time-steps. A_{nn} will thus enter the array $2n$ time-steps after A_{11}. Adding $0(w)$ time-steps to go through the array leads to a global computation time of $2n+0(w)$. In the next section we discuss the implementation of such an array, where $0(w)$ is equal to $\min(p, q)$—the same initialization delay as for the LU decomposition array of [8].

2.3 The systolic array

The systolic array we design requires an unusual input format: the four coefficients of any 2×2 sub-block must enter the array simultaneously. Assuming that the coefficients of the matrix A are directly delivered from the host, this restriction is not important. However, whenever necessary, we can easily design a preprocessing array to convert a usual format of data into the one we need. When thinking to a usual format of data we refer to the input format of the band matrix–vector multiplication array of [8], which seems to be the most frequently encountered for systolic arrays of efficiency $e=\frac{1}{2}$. Adopting the usual format of [8] will reduce the number of input pins by avoiding that two coefficients of a same diagonal enter the array simultaneously. We will detail first the preprocessing array and then concentrate on the computational array itself.

2.3.1 The preprocessing array The matrix is assumed to be fed in the array diagonal by diagonal, a new input of any diagonal every second step. a_{11} enters the array at the first step, $a_{i1}(2 \leq i \leq q)$ and $a_{1i}(2 \leq i \leq p)$ at the ith step. When output from the preprocessing array the four coefficients of any 2×2 sub-block are regrouped, so that they will enter simultaneously a cell of the computational array. These input/output formats are shown in Figure 2.

The design of such an array is straightforward. An example with $p=3$ and $q=5$ is given in Figure 3. The array is composed of $2w$

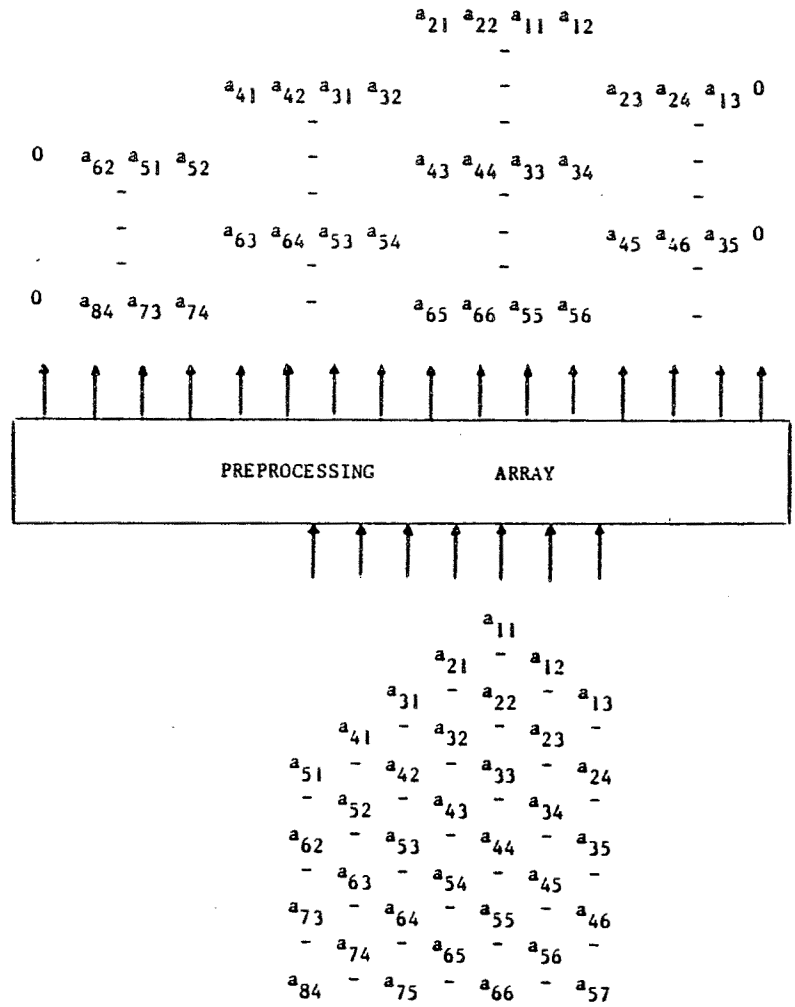


FIGURE 2 Input/output format for the preprocessing array.

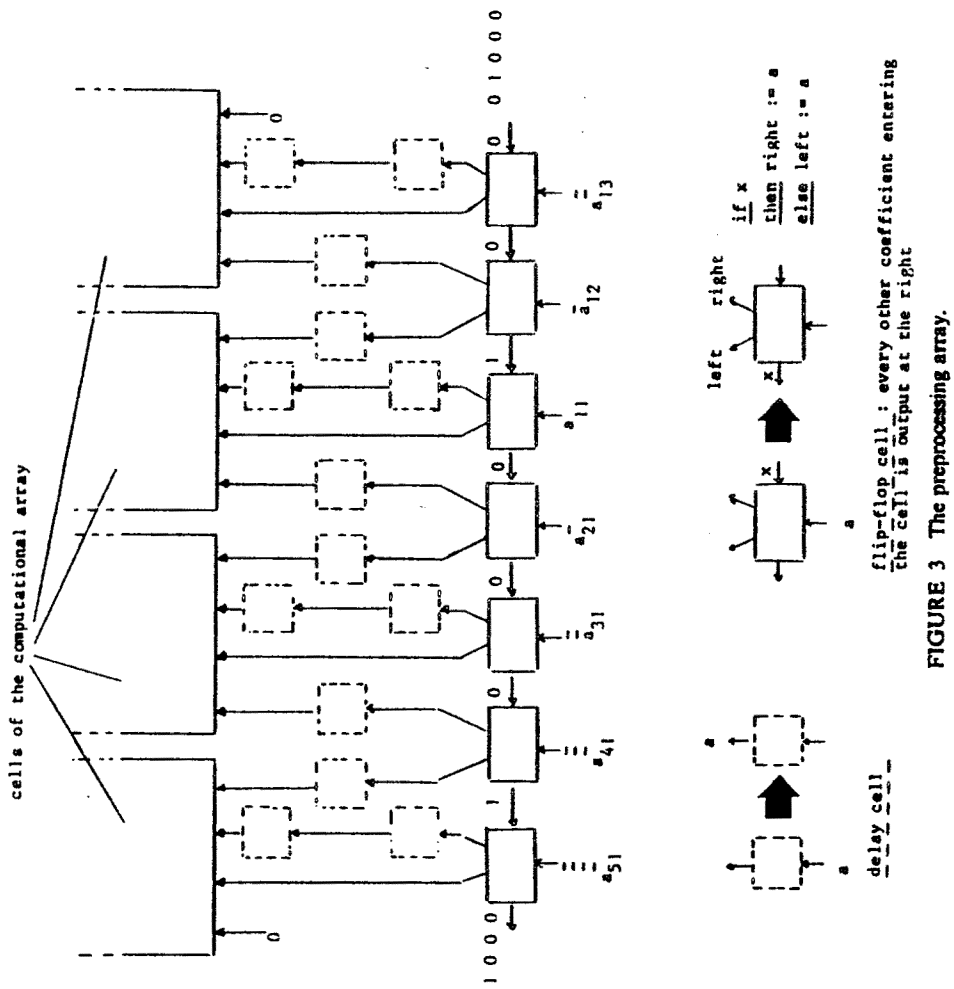


FIGURE 3 The preprocessing array.

delay cells and w "flip-flop cells" (which are delay cells whose output is controlled by the global clock).

2.3.2 The computational array A global view of the computational array is shown in Figure 4. This hexagonally connected array is constructed as follows:

— the upper boundary is composed of two lines of w processors; those of the second line perform elementary computations according to the scheme of Section 2.2 whereas those of the first line update the coefficients of L and U to get the final decomposition of A:

  * *second line of the upper boundary* (represented with circles in Figure 4): the processor at the center of the line is a determinant processor, whose operation is detailed in Figure 5(a). The processors on the left are composed of two determinant cells, their program is given in Figure 5(b). The processors on the right are composed of two division cells, see Figure 5(c).

  * *first line of the upper boundary* (represented with dotted squares in Figure 4): the processor at the center is just a delay cell. The processors on the left are composed of two division cells: they divide the coefficients $D \cdot l_{ij}$ by the corresponding determinant D (see Figure 5(d)). If the block decomposition $A = LU$ is needed, the processors on the right are deleted, since the coefficients of U can be picked up when they enter the circle cell below. But if the decomposition $A = LDU$ is needed, any 2×2 sub-block on the right of the array has to be premultiplied by the inverse of the corresponding pivot: in this case, the processors will be composed of four determinant cells (see Figure 5(d) too).

— all processors except for those on the upper boundary are "macro" IPS cells, i.e. they are composed of four IPS cells. A description of such processors is given in Figure 5(e).

Figure 6 shows five consecutive steps of the operation of the array (with $p = q = 5$). The first block A_{11} of A enters the array at Step 1. Nothing happens before A_{11} reaches the top processor at Step 3.

If we estimate that a determinant cell is equivalent to two IPS cells from a hardware point of view, then we check easily that the

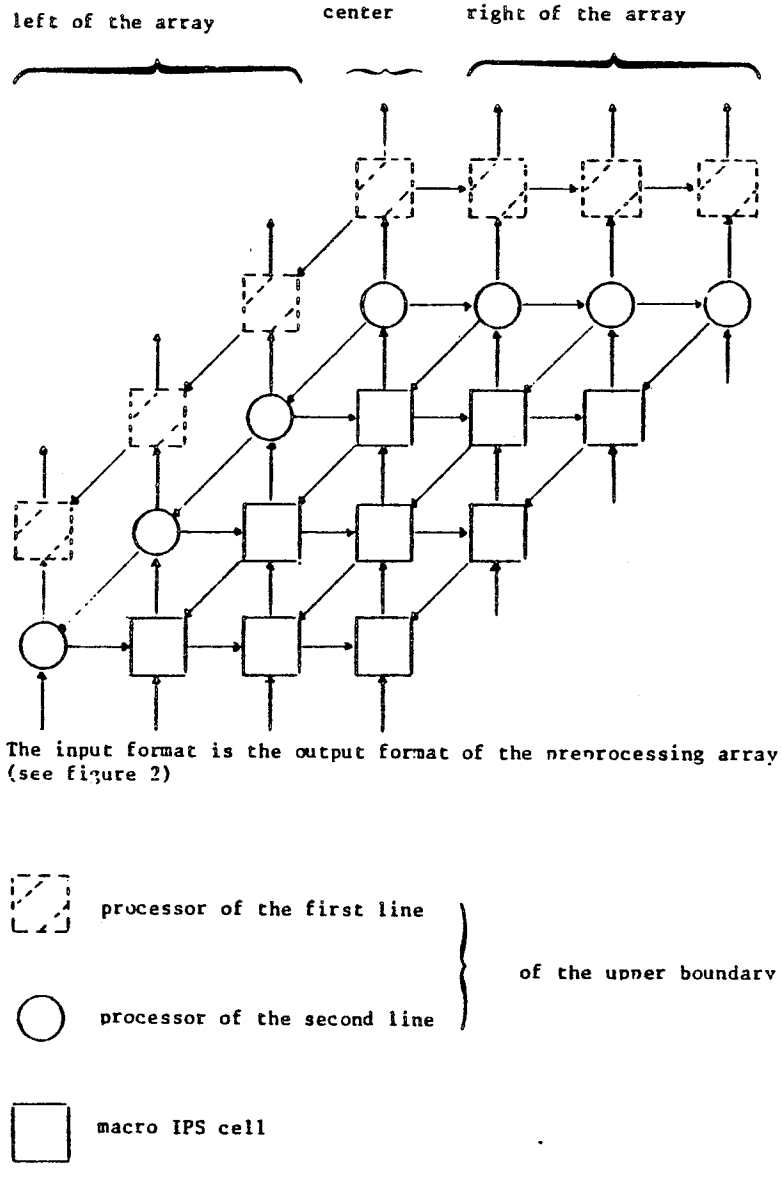


FIGURE 4 The computational array (with $p=q=7$).

1. General comments for Figure 5

Any processor receives data every fourth step, eventually during two steps. We describe the operation of a given processor by starting at a Step t where it receives coefficients as inputs: we assume the processor was idle at Step $t-1$, thus the processor may receive other inputs at Step $t+1$. The outputs will be delivered either at Step $t+1$ or at Step $t+2$, depending on the program of the cell. In any case, the processor will be idle at Step $t+3$.

2. Second line of the upper boundary:

a) center processor

Step t In—S = a, b, c, d
Step t+1 Out—N = $a, b, c, d, a*d-b*c$
Out—E = $a*d-b*c$
Out—SW = a, b
Step t+2 Out—SW = c, d

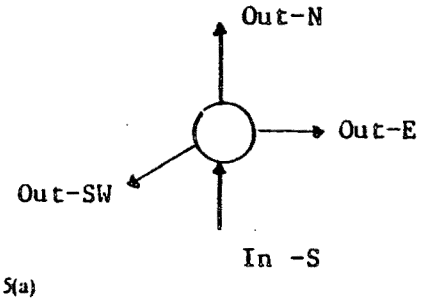


Figure 5(a)

b) left processor

Step t In—S = l, m, n, p
In—NE = a, b
Step t+1 In—NE = c, d
Out—N = $a*p-n*b, a*m-l*b$
Out—E = $a*p-n*b, a*m-l*b$
Out—SW = a, b
Step t+2 Out—N = $d*n-c*p, d*l-c*m$
Out—E = $d*n-c*p, d*l-c*m$
Out—SW = c, d

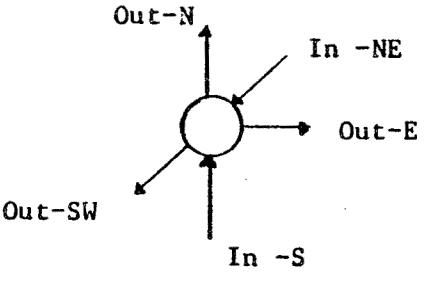


Figure 5(b)

c) right processor

Step t In—S = u, v, w, x
In—W = D
Step t+1 Out—N = $w/D, x/D$
Out—E = D
Out—SW = $w/D, x/D$
Step t+2 Out—N = $u/D, v/D$
Out—SW = $u/D, v/D$

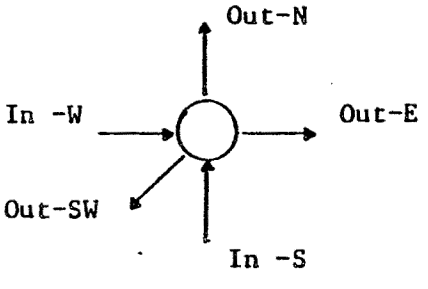


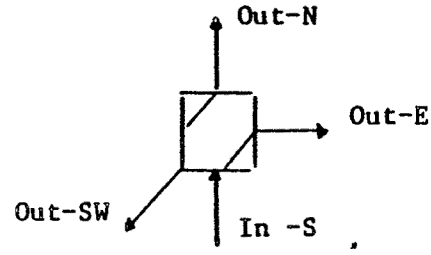
Figure 5(c)

3. First line of the upper boundary

The output format of the computational array is the same as its input format. The lower block triangular part of A is now equal to L, its block diagonal is equal to D and its upper block triangular part is equal to U.

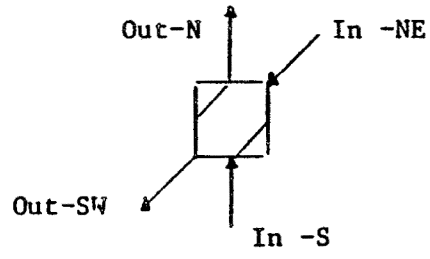
a) center processor

Step t In—S = a, b, c, d, D
Step t+1 Out—E = a, b, c, d
Out—SW = D
Step t+2 Out—N = a, b, c, d



b) left processor

Step t In—S = n, l
In—NE = D
Step t+1 In—S = p, m
$l := l/D, n := n/D$
Out—SW = D
Step t+2 $m := m/D, p := p/D$
Out—N = l, m, n, p



c) right processor

Step t In—S = w, x
In—W = a, b, c, d
Step t+1 In—S = u, v
Out—E = a, b, c, d

Step t+2
$$\begin{pmatrix} w \\ x \\ u \\ v \end{pmatrix} := \begin{pmatrix} wb-ud \\ xb-vd \\ uc-wa \\ vc-xa \end{pmatrix}$$

Out—N = u, v, w, x

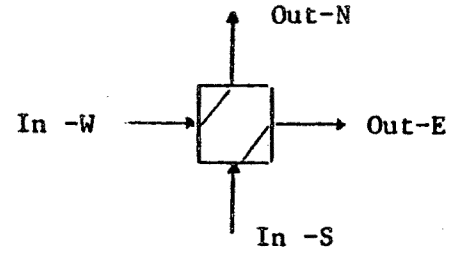


Figure 5(d)

4. Macro IPS cell

Step t In—S = a, b, c, d
In—W = n, l
In—NE = w, x
Step t+1 In—W = p, m
In—NE = u, v
$a := a + l \circ w, b := b + l \circ x$
$c := c + n \circ w, d := d + n \circ x$
Out—E = n, l
Out—SW = w, x
Step t+2 $a := a + m \circ u, b := b + m \circ v$
$c := c + p \circ u, d := d + n \circ v$
Out—N = a, b, c, d
Out—E = p, m
Out—SW = u, v

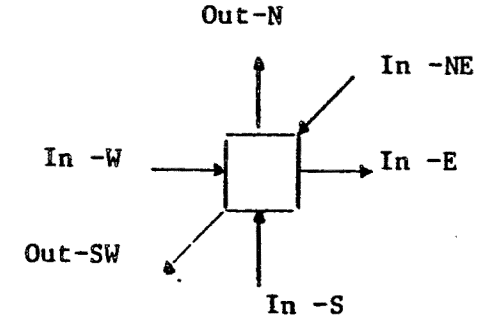
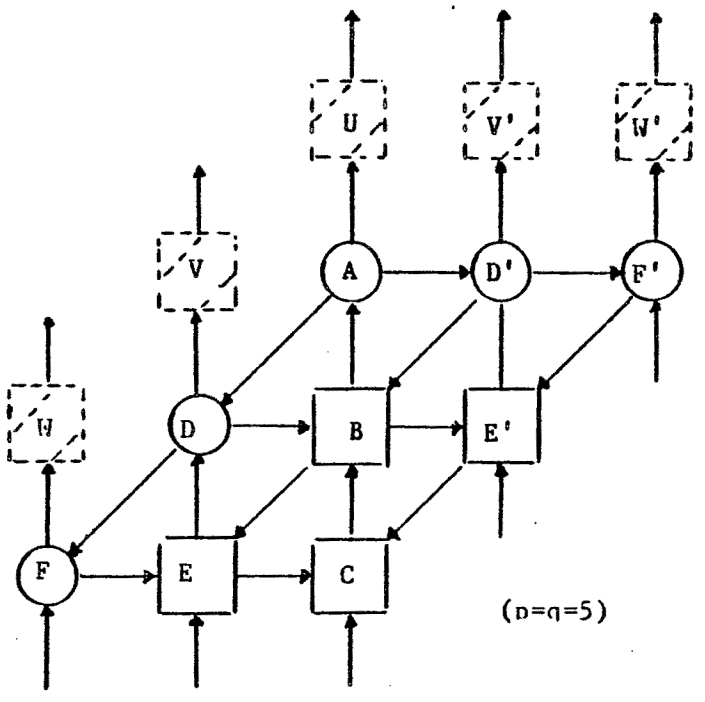


Figure 5(e)

FIGURE 5 Operation of the cells of the array.



Step 4 cell A $D=a_{11}a_{22}-a_{12}a_{21}$

Step 5 cell D $D\cdot l_{31}=a_{21}a_{32}-a_{31}a_{22}, D\cdot l_{41}=-a_{21}a_{42}-a_{41}a_{22}$

cell D' $a_{13}:=a_{13}/D, a_{14}:=a_{14}/D$

Step 6 cell B $A_{22}:=A_{22}+\begin{pmatrix}D\cdot l_{31}\\ D\cdot l_{41}\end{pmatrix}\cdot(a_{13}/D, a_{14}/D)$

cell D computation of $D\cdot l_{32}$ and $D\cdot l_{42}$

cell D' computation of a_{23}/D and a_{24}/D

cell F computation of $D\cdot l_{51}$ and $D\cdot l_{61}$

cell F' computation of a_{15}/D and a_{16}/D

Step 7 cell B $A_{22}:=A_{22}+\begin{pmatrix}D\cdot l_{32}\\ D\cdot l_{42}\end{pmatrix}\cdot(a_{23}/D, a_{24}/D)$

cell E updating of A_{32} (phase 1)

cell E' updating of A_{23} (phase 1)

cell F computation of $D\cdot l_{52}$ and $D\cdot l_{62}$

cell F' computation of a_{25}/D and a_{26}/D

Step 8 a new elimination step starts now. For instance cell A computes the determinant of the new pivot:
$D=a_{33}a_{44}-a_{34}a_{43}$

Output of the 2×2 *blocks*

cell U outputs A_{11} at Step 5; cell V outputs L_{21} and cell V' outputs U_{12} at Step 7; cell W outputs L_{31} and cell W' outputs U_{13} at Step 9, just when cell U outputs $A_{22}^{(2)}$.

FIGURE 6 The operation of the computational array.

array corresponds to pq IPS cells, just as the LU decomposition array of [8]. Hence we have proven the following theorem:

THEOREM *If A is an $n \times n$ band matrix with band width $w = p+q-1$, an array having no more than pq processors can compute the block LU decomposition of A in $2n+\min(p,q)$ units of time (which includes I/O time).*

3. SOLVING THE LINEAR SYSTEM $Ax=b$

We want to show in this section that the block LDU decomposition of A is well-suited to the resolution of the linear system of equations $Ax=b$. In the preceding section we have obtained the following decomposition for an $n \times n$ band matrix A with band width $w = p+q-1$ (n is even, p and q are odd):

$$A = LDU$$

where

- L is a band lower triangular matrix with band width q, a unit diagonal and a null first subdiagonal;
- D is a 2×2 block diagonal matrix;
- U is a band upper triangular matrix with band width p, a unit diagonal and a null first superdiagonal.

The solution x of $Ax=b$ is computed in three steps:

- back solving of $Uz=b$;
- solving of $Dy=z$;
- forward solving of $Lx=y$.

Of course the second step is very easy to solve: the system $Dy=z$ is composed of $n/2$ independent systems of two equations in two unknowns. And since the first upper triangular system can be rewritten as a lower triangular system, we only have to discuss the solution of the system $Lx=b$. A linearly connected systolic array of q IPS cells has been introduced in [8] to solve such a system in $2n+q$ units of time: after an initialisation delay of q steps, a new compo-

nent of x is delivered every second time-step. It is shown in [9] how to take advantage of the presence of a null first subdiagonal to solve the system in $n+q/2+1$ units of time (about twice faster) using q IPS cells too: two consecutive components of x will be separated by a single unit of time, but since x_i does not have to meet x_{i+1} (because $a_{i+1,i}=0$), x_i once computed can be fed back into the array just in time to meet $x_{i+2}, x_{i+3}, \ldots, x_{i+q-1}$. The systolic network is depicted in Figure 7. To compute x_i we need to accumulate the value of the expression

$$y_i = \sum_{j=i-q+1}^{i-2} a_{ij} \cdot x_j \quad \text{(then } x_i = b_i - y_i\text{)}.$$

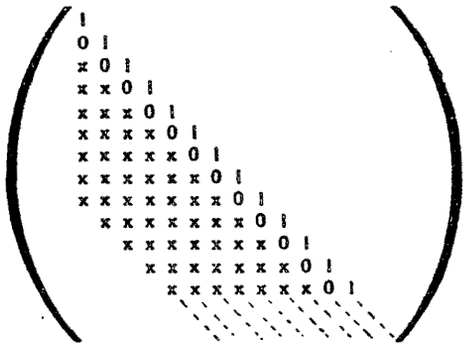
This computation is divided into two subcomputations

$$\begin{aligned} y_i^1 &= \sum_{i-j \text{ even}} a_{ij} \cdot x_j \\ y_i^2 &= \sum_{i-j \text{ odd}} a_{ij} \cdot x_j \end{aligned} \quad \rightarrow \quad x_i = b_i - (y_i^1 + y_i^2)$$

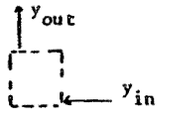
which are added in the leftmost special cell. The systolic network is thus composed of two linearly connected arrays of IPS cells (each corresponding to one of these subcomputations) and the leftmost special cell. The reason for decomposing the evaluation of

$$\sum_{j=i-q+1}^{i-2} a_{ij} \cdot x_j$$

into two subcomputations is the following: consider a linearly connected systolic array where the data is moving from right to left and sent back into the array when reaching the leftmost cell. If two consecutive elements are separated by two units of time, any of them, say x_i, will meet all the elements coming after it; if two elements are separated by a single unit of time, then x_i will never meet $x_{i+1}, x_{i+3}, x_{i+5}, \ldots$. This is why we have duplicated the array: when fed back, any element meets one-half of the elements following it on the first array, and the other half on the second array. The only element which it cannot meet is the next one, but this does not matter here since $a_{i+1,i}=0$.

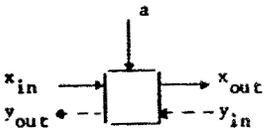


Structure of the matrix L (q=9)



$y_{out} = y_{in}$

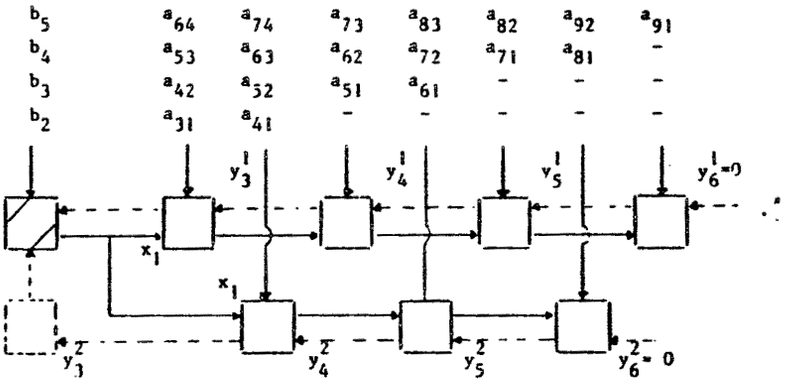
delay cell

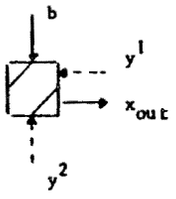


$x_{out} = x_{in}$

$y_{out} = y_{in} + a.x_{in}$

IPS cell





$x_{out} = b - (y^1 + y^2)$

Special leftmost cell

FIGURE 7 Solution of the system $Lx = b$.

4. CONCLUDING REMARKS

The block 2×2 *LU* or *LDU* decomposition array for band matrices that we have introduced has two noteworthy features:

— the efficiency of the decomposition array is increased from $\frac{1}{3}$ to $\frac{1}{2}$ whereas the number of IPS cells is the same as for the usual *LU* decomposition array of [8];

— once the block triangular factors L and U of the block decomposition $A = LDU$ are known, the linear system $Ax = b$ can be solved twice as fast as when having computed the usual *LU* or *LDU* decomposition, owing to the special structure of the block triangular factors L and U whose second diagonal is always null.

References

[1] H. M. Ahmed, J. M. Delosme and M. Morf, Highly concurrent computing structures for matrix arithmetic and signal processing, *IEEE Computer* **15**, 1 (1982), 65–82.

[2] A. Bojanczk, R. P. Brent and H. T. Kung, Numerically stable solution of dense systems of linear equations using mesh-connected processors, Technical Report, May 1981, Carnegie-Mellon University Department of Computer Science.

[3] R. P. Brent, H. T. Kung and F. T. Luk, Some linear-time algorithms for systolic arrays, Technical Report TR CS 82-541 (1982), Cornell University Department of Computer Science.

[4] W. M. Gentleman and H. T. Kung, Matrix triangularization by systolic arrays, *Proceedings of the SPIE*, vol. 298, Real time signal processing 4, August 1981.

[5] D. Heller and I. Ipsen, Systolic networks for orthogonal equivalence transformations and their applications, *Proc. 1982 Conf. Advanced Research in VLSI*, pp. 113–122, MIT, 1982.

[6] I. Ipsen, A parallel *QR* method using fast Givens' rotations, *Research Report YALEU/DCS/RR 299*, 1984, Yale University.

[7] H. T. Kung, Why systolic architectures, *IEEE Computer* **15**, 1 (1982), 37–46.

[8] H. T. Kung and C. E. Leiserson, Systolic arrays for (VLSI), *Sparse Matrix Proc. 1978*, I. Duff and G. W. Stewart (eds.), pp. 256–282, SIAM 1979.

[9] Y. Robert and M. Tchuente, Resolution systolique de systemes lineaires denses, *Research Report 420*, 1983, Laboratoire TIM3/IMAG, Grenoble, France.

[10] R. S. Varga, *Matrix Iterative Analysis*, Prentice-Hall, Englewoods Cliffs, N.J., 1962.

Parallel solution of band triangular systems on VLSI arrays with limited fan-out

Yves ROBERT & Maurice TCHUENTE
CNRS, Laboratoire TIM3/IMAG
BP 68, 38402 Saint Martin d'Hères Cedex

to appear (North Holland) in the Proceedings of the
INTERNATIONAL WORKSHOP ON MODELING
AND PERFORMANCE EVALUATION OF PARALLEL SYSTEMS
December 13-18, 1984, Grenoble,
M. BECKER editor

Abstract :

We are interested in the solution of band lower triangular linear systems of n equations Ax = b of band width w, on modular VLSI arrays whose fan-out f is independent of w and n. Kung and Leiserson have introduced a network such that f=1 and which works in time T = 2n+w. We exhibit here an array of the same area such that f=2 and T = $n+\lceil w/2 \rceil$. We also show that, for any p>=1, there exists a modular VLSI array of fan-out f=2p which solves the system in time T.<= $\lceil n/p \rceil + \lceil w/p \rceil$ provided that a sufficient number q >= $(p+2)\lceil \log p \rceil - 1$ of first subdiagonals of the matrix A are zero. We point out that a limited fan-out is a very important property for the VLSI implementation of a parallel algorithm.

1. INTRODUCTION

In this paper, we are interested in the designing of a VLSI array for the solution of linear systems of equations

(1) $$A\,x = b$$

where b, x are vectors with n components and A is an $n \times n$ band lower triangular matrix of band width w, ie $a_{ij} = 0$ for $i-j \notin [0, w-1]$.

Such systems are frequently encountered in numerical linear algebra or signal processing. Using a sequential computer, the solution x can be obtained in a number of steps $O(w*n)$. For dense matrices ($w=n$), the number of steps is thus quadratic in the size n of the matrix A. This motivates the development of parallel algorithms devoted as well to high-performance SIMD, MIMD machines as to special-purpose processors: see for instance Chen, Kuck and Sameh [2], Evans and Dunbar [3], Kung and Leiserson [5], Sameh and Kuck [8] or refer to the survey papers of Heller [4] and Sameh [6], [7].

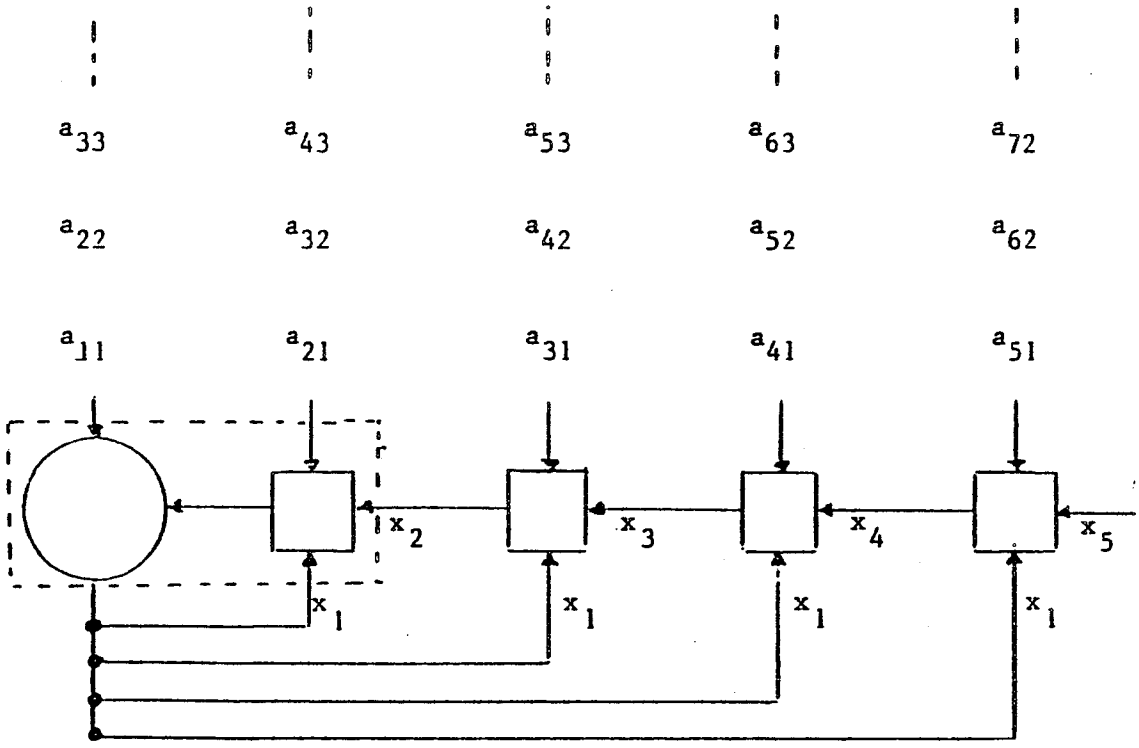
System (1) can be easily solved by a forward substitution process which can be stated as follows:

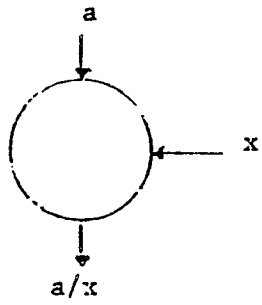
For i := 1 to n do

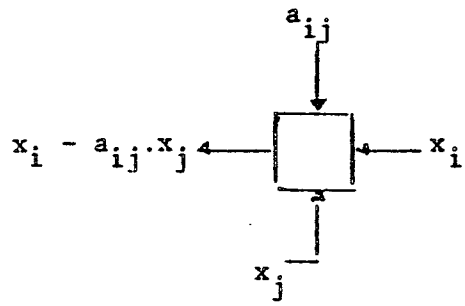
$$x_i := \left(b_i - \sum_{j=i-w+1}^{i-1} a_{ij} \cdot x_j \right) / a_{ii}$$

Hereafter, we take as unit step the time necessary to perform an accumulation $y := y \pm a.x$ or a division $y := y / x$.

The straightforward VLSI implementation of this process consists of a linearly connected array of w cells depicted in the figure 1: the leftmost cell computes a new component x_i of x each cycle time and sends it to all the other cells which update $x_{i+1}, x_{i+2}, \ldots, x_{i+w-1}$ by performing an accumulation. The cycle delay of the array is thus the time necessary for a division and an accumulation, hence the solution x is delivered after $2n + w$







Straightforward VLSI implementation

Figure 1

time steps. If the matrix A is preprocessed in order to have a unit diagonal (we discuss later a VLSI implementation of such a preprocessing), the number of steps can be reduced to n + w.

However, two major disadvantages of this approach are its high degree of fan-out and its lack of modularity. The fan-out f of a chip is the maximum number of copies of a given variable which are sent from a single cell to the others: it is equal here to the band width w of the matrix A, and this prohibits the use of the previous solution whenever w is not very small. On the other hand, since the leftmost cell is connected to all the others, the previous solution is neither local nor modular ... while locality and modularity are highly deasirable features for a physical realization.

Kung and Leiserson [5] have shown that the forward substitution process can be performed in time 2n + w on a linearly connected array of w cells. This array is thus slower than the previous one but it is completely modular. Moreover its fan-out f is minimum, ie f = 1 (no variable is duplicated).

We shall concentrate here on the designing of a special-purpose VLSI implementation, involving a limited number of processors O(w), and twice faster than the systolic array of Kung and Leiserson [5]: in the next section, we introduce an algorithm which is computation-expensive but which can be realized in time $n + \max \{ \lceil w/2 \rceil + 2 , 5 \}$ on an array of size 3w + 1. When applied to dense matrices, this method requires O(n) processors and works in time $\lceil 3n/2 \rceil$. Our array is modular and its fan-out is limited to 2. Finally, we make some comments and discuss some extensions.

2. THE SCOPE-2 SUBSTITUTION PROCESS

Let us consider a system of linear equations

(1) $A\,x = b$

where $b \in R^n$ and $A \in M_{nxn}(R)$ is a lower triangular matrix of band width $w >= 2$, ie $a_{ij} = 0$ for $i-j \notin [0,w-1]$

The idea is to make a change of variable

$x = P\,z$

where $P \in M_{nxn}(R)$ is a bidiagonal lower triangular matrix,

in order to obtain an equivalent system

(2) $C\,z = b$

where $C \in M_{nxn}(R)$ is a lower triangular matrix of band width $w + 1$ and such that $c_{ii} = 1$ and $c_{i,i-1} = 0$ for any i

The system (2) is then solved by the scope-2 substitution process described as follows:

(3) $z_1 = b_1$; $z_2 = b_2$;

For $i := 3$ to n do

$$z_i := b_i - \sum_{j=i-w}^{i-2} c_{ij} \cdot z_j$$

We say that this substitution process is of scope 2 because, for any i, z_i depends only of variables z_j such that $i-j >= 2$.

Finally, the recovering of a solution x of system (1) is obtained by a matrix-vector multiplication $x = P\,z$.

The change of variable introduced here is crucial. Indeed, a VLSI implementation of the classical scope-1 substitution process described in the introduction yields a solution of efficiency 1/2 [5]: a new component x_i is delivered every second time step [5]. On the contrary, we shall exhibit here an array which computes (3) with efficiency 1, ie a new z_i (and a new x_i too) is delivered every time step.

3. VLSI IMPLEMENTATION

The VLSI implementation of the algorithm described in section 1 consists of four different phases.

Phase 1

Let D denote the diagonal of matrix A. By a change of variable $y = D\,x$, we transform system (1) into an equivalent system

(1') $B\,y = b$

where B is a lower triangular matrix of band width w and unit diagonal

$B = A.D^{-1}$ is obtained by dividing any column A_j of A by a_{jj}. The parallel implementation of this phase on an array of w processors is depicted in figure 2. The coefficients a_{ij} such that $i-j = k >= 0$ are fed in the top input port of cell k. The operation of the array can be described as follows:

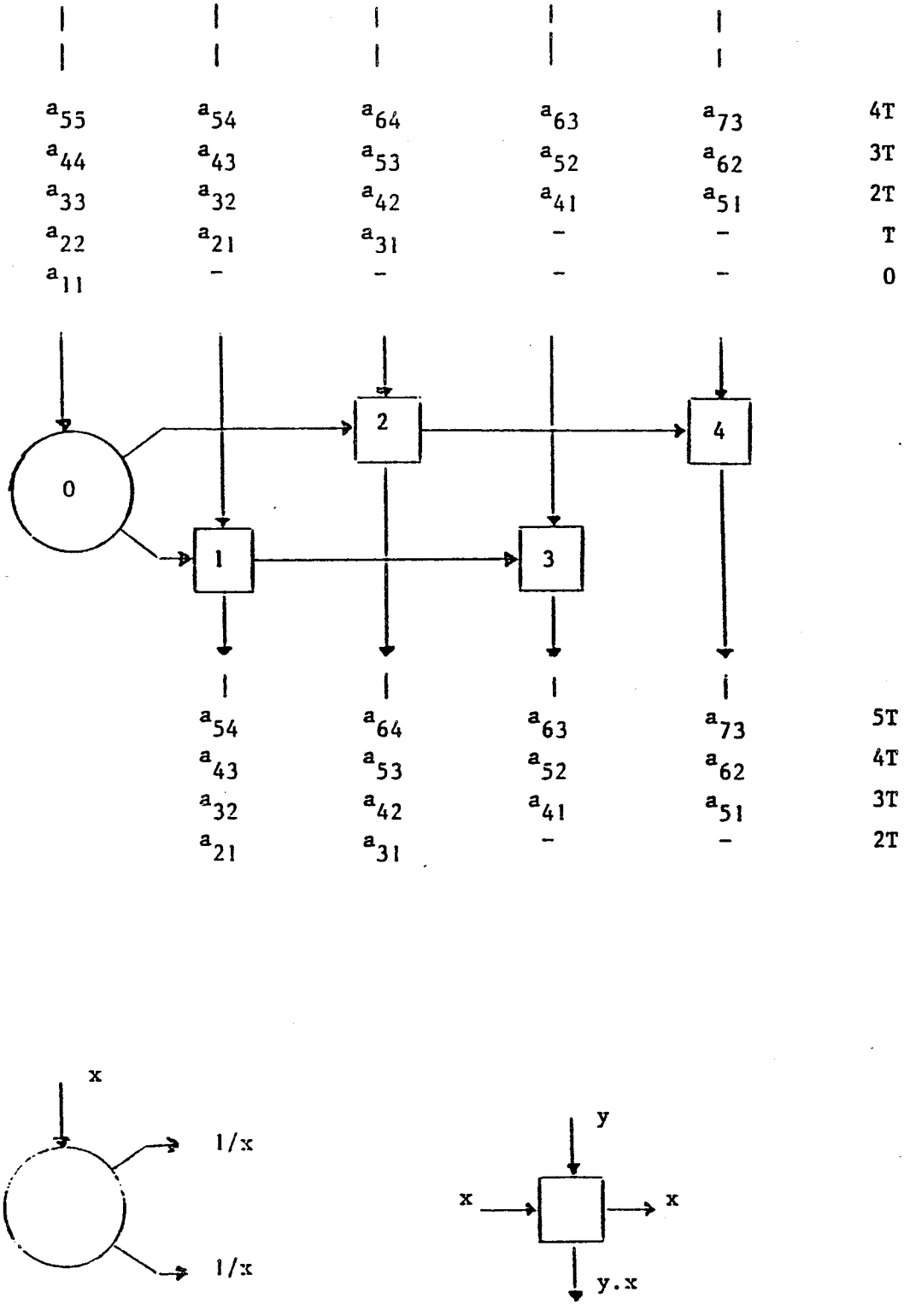
- when processor 0 receives an element a_{jj}, it computes its reciprocical $1/a_{jj}$ and sends it to the right
- when a processor k ($k >= 1$) receives a_{ij}, $i-j = k$, on its top input port and a_{jj} on its left input port, $b_{ij} = a_{ij} / a_{jj}$ is computed and sent downwards while $1/a_{jj}$ is propagated to the right.

Phase 2

In this phase, we perform a change of variable $y = Q\,z$ in order to transform the system (1') into an equivalent system

(2) $C\,z = b$

where C is a lower triangular matrix of band width $w + 1$ such that $c_{ii} = 1$ and $c_{i,i-1} = 0$ for any i.



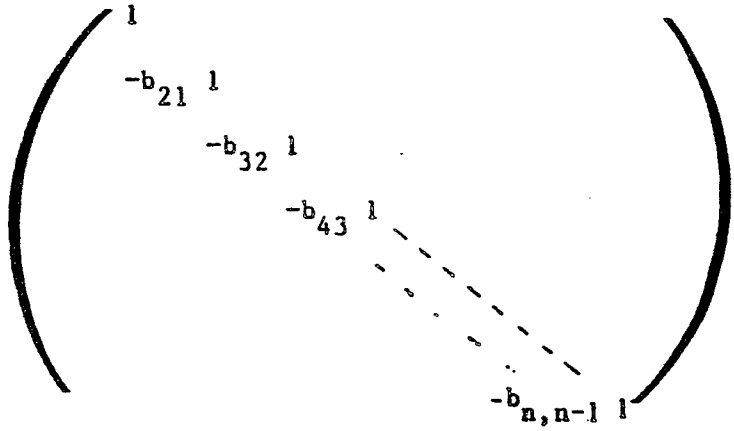
Phase 1

Figure 2

That is to say, C has an unit diagonal and a null first subdiagonal. In order to obtain such a matrix C, any column B_j of B is replaced by

$$C_j = B_j - b_{j+1,j} B_{j+1}$$

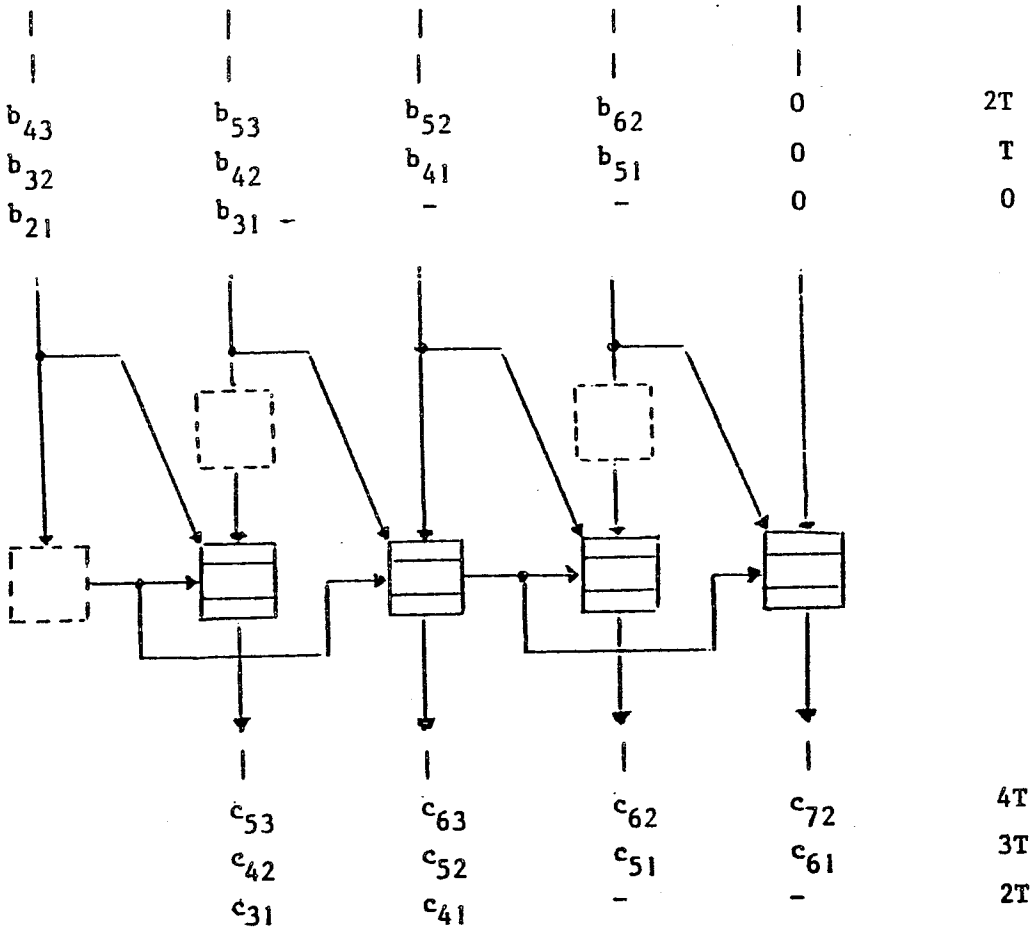
(the last column of B remains unchanged). This corresponds to a post-multiplication of matrix B by the matrix P

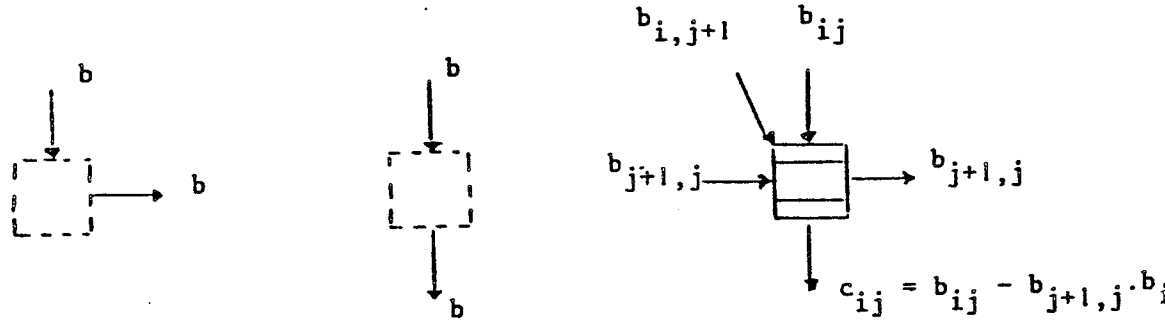


The parallel implementation of this phase on an array of w-1 processors and $\lceil w/2 \rceil$ delay cells is shown in figure 3. When processor k receives b_{ij} on its vertical input port, $b_{i,j+1}$ on its diagonal input port, and $b_{j+1,j}$ on its horizontal input port, it propagates $b_{j+1,j}$ to the right and sends $c_{ij} = b_{ij} - b_{j+1,j} \cdot b_{i,j+1}$ downwards.

Phase 3

In this phase, the solution of the triangular system C z = b is computed. A linearly connected systolic array of w cells has been introduced in [5] to solve such a system in 2n + w units of time: after an initialization delay of w steps, a new component of z is delivered every second time-step. We shall show here how to take advantage of the presence of a null first subdiagonal to solve the system in n + $\lceil w/2 \rceil$ units of time (about twice faster) using w cells: two consecutive components of z will be separated by a single





Phase 2

Figure 3

unit of time, but since z_i does not have to meet z_{i-1} (because $c_{i,i-1} = 0$), z_i once computed can be fed back into the array just in time to meet z_{i+2}, z_{i+3}, ..., z_{i+w}. The VLSI array is depicted in figure 4. To compute z_i we need to accumulate the value of the expression

$$z_i = b_i - \sum_{j=i-w}^{i-2} c_{ij} \cdot z_j \quad (j>=3)$$

where $z_1 = b_1$ and $z_2 = b_2$

This computation is divided into two subcomputations:

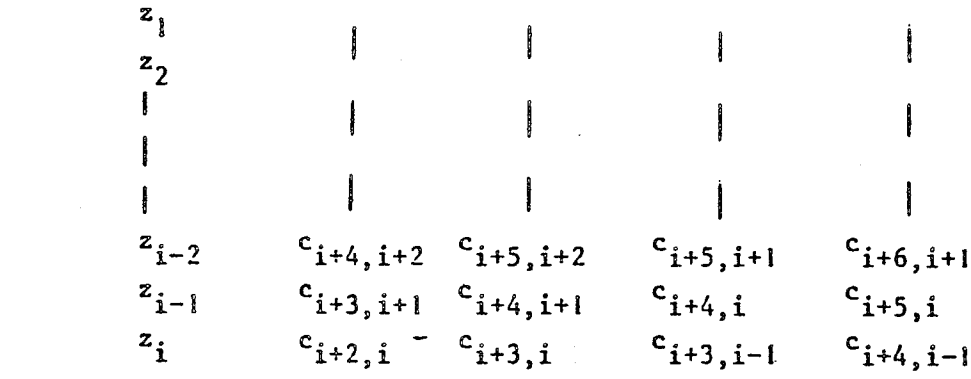
$$z_i^1 = b_i - \sum_{i-j \text{ even}} c_{ij} \cdot z_j$$

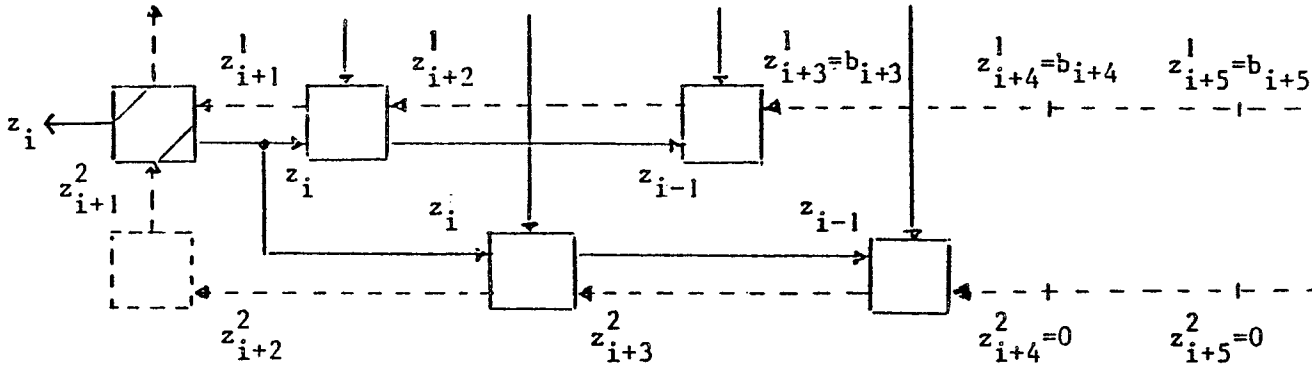
$$z_i^2 = - \sum_{i-j \text{ odd}} c_{ij} \cdot z_j$$

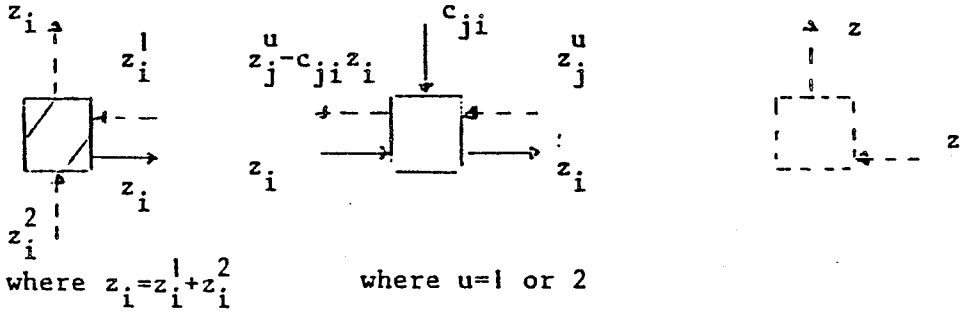
which are added in the leftmost special cell. The systolic network is thus composed of two linearly connected arrays (each corresponding to one of these subcomputations) and the leftmost special cell. The reason for decomposing the evaluation of

$$\sum_{j=i-w}^{i-2} -c_{ij} \cdot z_j$$

into two subcomputations is the following: consider a linearly connected systolic array where the data is moving from right to left and sent back into the array when reaching the leftmost cell. If two consecutive elements are separated by two units of time, any of them, say z_i, will meet all the elements coming after him; if two element are separated by a single unit of time, then z_i will never meet z_{i+1}, z_{i+3}, z_{i+5},... This is why we have duplicated the array: when fed back, any element z_i meet one half of the following elements z_{i+2}, z_{i+3}, ... on the first array, and the other half on the second array. The only element that it can not meet is z_{i+1}, and this does not matter here since $c_{i,i+1} = 0$.







Phase 3

Figure 4

Phase 4

In this last phase the solution $x = D^{-1} P z$ of system (1) is computed:

$x_1 = z_1 / a_{11}$

For i := 2 to n do $x_i := (z_i - c_{i,i-1} \cdot z_{i-1}) / a_{ii}$

This simple computation is described in figure 5.

The concatenation of the arrays corresponding to the four phases of the algorithm leads to the global network depicted in the figure 5. We can now state our main result:

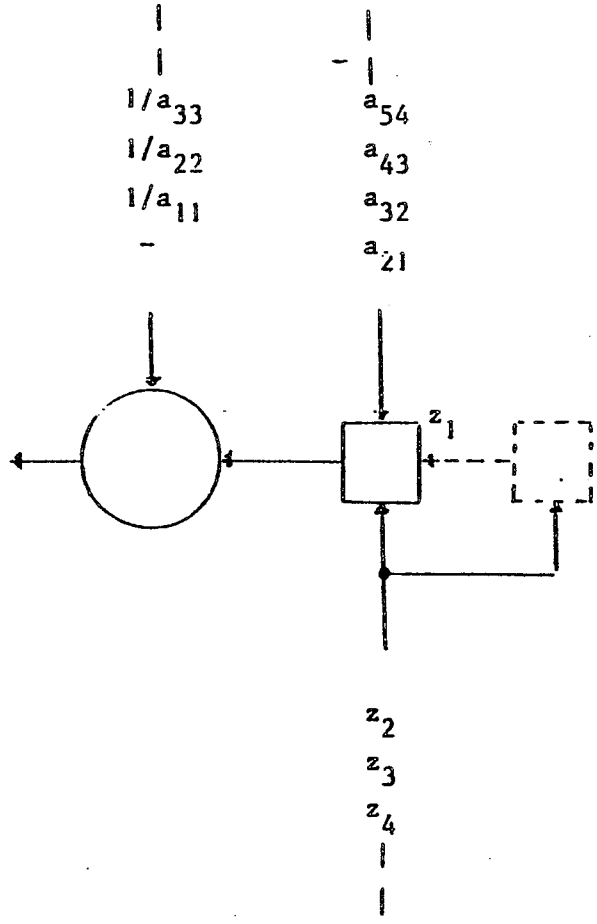
Theorem : if A is an nxn band matrix of band width w, the scope-2 substitution process can solve the linear system A x = b in time $n + \max \{ \lceil w/2 \rceil + 2 , 5 \}$ on a systolic array involving 3w + 1 computation cells. This arrray is completely modular and its fan-out is equal to 2.

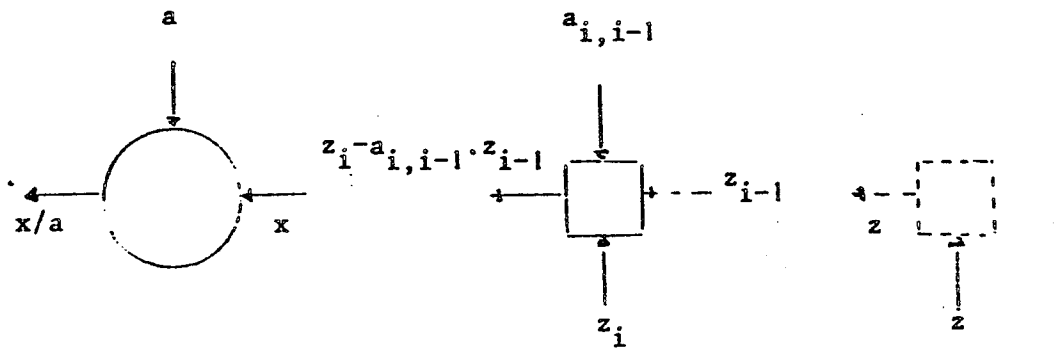
Corollary : if A is an nxn dense matrix, the system A x = b can be solved in time $\lceil 3n/2 \rceil$ on an array containing $\lceil n/2 \rceil$ computation cells and of fan-out f = 2.

Proof of the Theorem : Let us count the number of computation cells required by each subarray:

- Phase 1 : we need as many computation cells as the band width of A, ie w cells
- Phase 2 : any non-zero subdiagonal is input to a computation cell, thus we need w-1 cells
- Phase 3 : the two identical sub-arrays require an amount of w-1 computation cells, and we have to add the special cell, which leads to a total of w cells
- Phase 4 : only two computation cells are required for this phase.

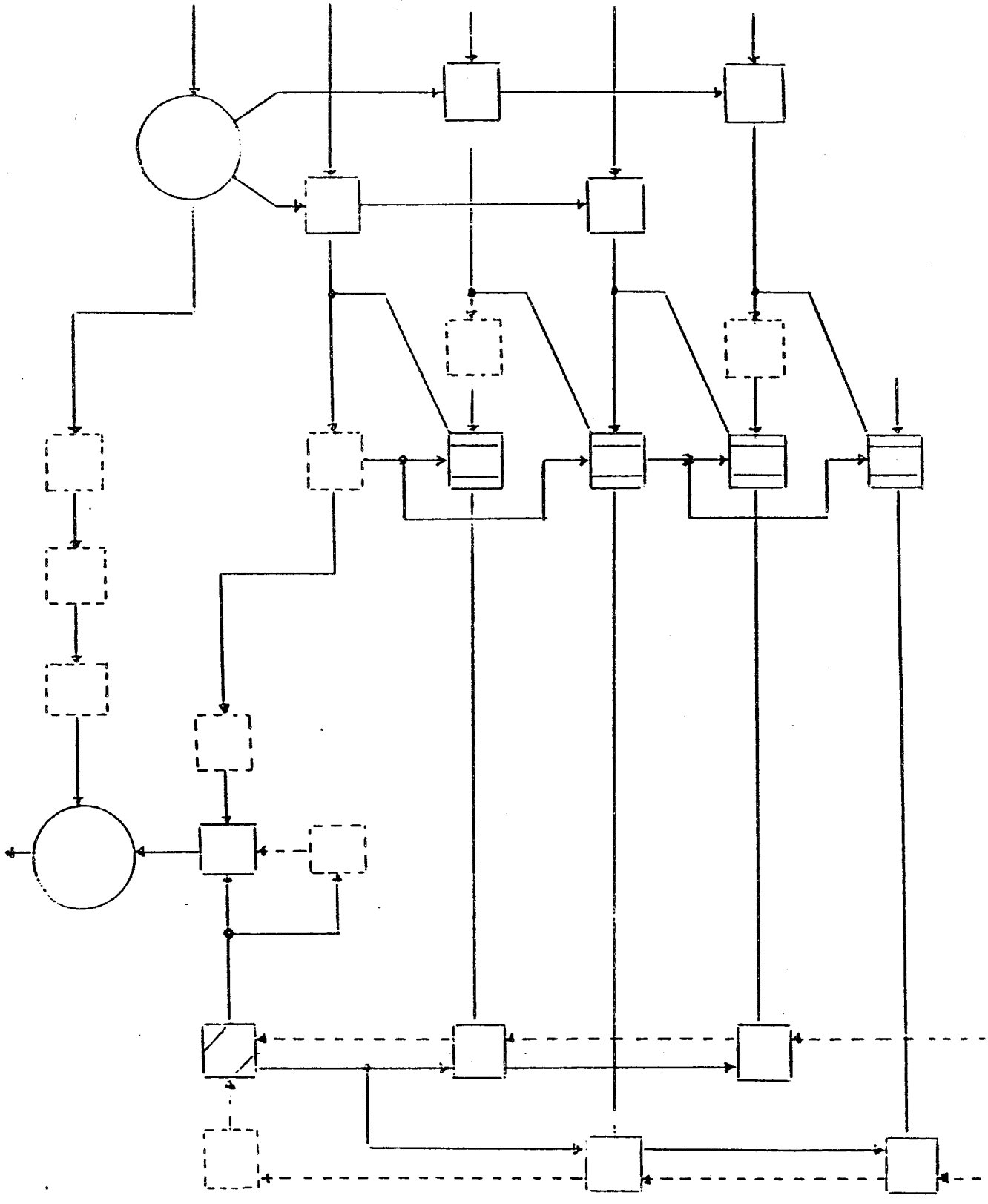
This shows that the scope-2 substitution process can be executed on an





Phase 4

Figure 5



The whole array

Figure 6

array involving 3w+1 computation cells for a matrix of band width w.

Now, a_{11} and b_1 go respectively through 5 and $\lceil w/2 \rceil + 2$ cells before being output from the leftmost cell of Phase 4 which communicates with the host: hence the first component of the solution x can be delivered at time T = Max { $\lceil w/2 \rceil + 2$, 5 } + 1. Since a new component is delivered every time-step, this proves the theorem.

Eliminating the first subdiagonal of the system matrix in order to fasten the solution can be done by the first step of the diagonal elimination scheme of Chen and Kuck [1]. However this scheme uses combinations of rows rather than columns: the reason for choosing to combine the columns of the system matrix by post multiplication is only the simplicity of the induced VLSI design.

4. AN EXTENSION OF THE ALGORITHM

The theorem above can be generalized in the following way:

Theorem : let $Cx = b$ be a lower triangular system of n linear equations, of band width $q + w + 1$ such that

$$c_{ii} = 1 \text{ for any } i \text{ and } c_{ij} = 0 \text{ for } i-j \notin [q+1, q+w]$$

If p is an integer satisfying to the relation

$$q >= p \lceil \log 2p \rceil + p - 1$$

(all logarithms are in base 2) then there exists a VLSI array composed of $w + p(2p-1)$ computation cells and of maximum fan out 2p, which solves the problem in time

$$T <= \lceil n/p \rceil + \lceil \log 2p \rceil + \lceil w/p \rceil$$

Proof : we consider an array R which can be described as follows:

Spatial organization of computation cells : we can distinguish four levels in the array:

level 1 : R is composed of p sub-arrays $R_1, R_2, \ldots, R_p$. Any R_s is devoted to the computation of the components x_i such that $i = s$ modulo p.

level 2 : any R_s, $1<=s<=p$, is composed of two parts:

- it contains p sub-arrays $R_{s,1}, R_{s,2}, \ldots, R_{s,p}$. Any $R_{s,r}$ is devoted to the accumulation of expressions

$$e_{ir} = \Sigma_{j=r \bmod p,\ q<i-j<=q+w} -c_{ij} \cdot x_j \text{ where } i=s \bmod p$$

- the outputs of the sub-arrays $R_{s,1}, \ldots, R_{s,p}$ are summed up by a tree-connected collection of $p - 1$ adders.

level 3 : any $R_{s,r}$, $1<=s,r<=p$, is composed in turn of two sub-arrays $R_{s,r,0}$ and $R_{s,r,1}$. Let us denote

$$u_{r,s} = \min \{ k \ / \ q<k<=q+w,\ s-k=r \bmod p \}$$

Any $R_{s,r,a}$ is devoted to the accumulation of the expressions

$$c_{i,r,s} = \sum_{j=i-u_{r,s}-\theta p,\ q<i-j<=q+w,\ \theta=a \bmod p} -c_{i,j}.x_j$$

level 4 : any $R_{s,r,a}$, $1<=s,r<=p$, $a=0,1$, is a copy of the linear array introduced by Kung and Leiserson [5] for band triangular systems.

Flow of data :

- for any i, $1<=i<=n$, $x_{i,1,0}$ is initialized to b_i and the other variables $x_{i,r,0}$, $2<=r<=p$, and $x_{i,r,1}$, $1<=r<=p$, are initialized to 0
- at instant t, the variables

$$\{ x_{pt+1,r,a}, x_{pt+2,r,a}, \ldots, x_{(p+1)t,r,a},\ 1<=r<=p,\ a=0,1 \}$$

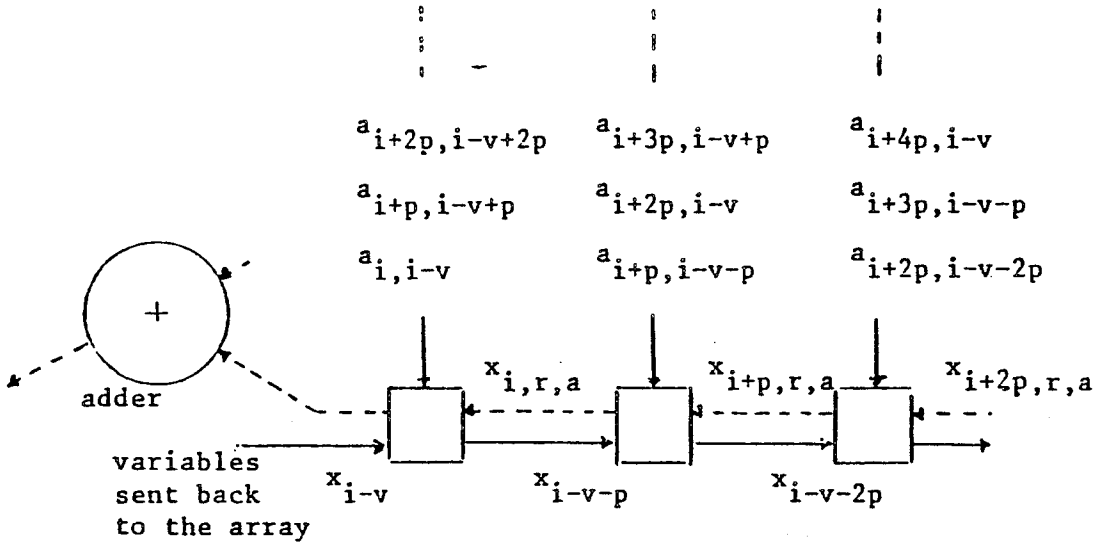
are read in parallel by the array R. More precisely, $x_{pt+s,r,a}$ is an input to the linear sub-array $R_{s,r,a}$

- when a variable x_j such that $j=r \bmod p$ is output from R_r, 2p copies are sent back into the array. For any s, $1<=s<=p$, the two copies of x_j which are sent to R_s are used by $R_{s,r,0}$ and $R_{s,r,1}$

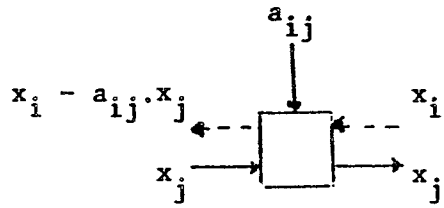
Validity of the algorithm : since the array contains exactly $\lceil \log 2p \rceil$ layers of adders, and since $q >= p \lceil \log 2p \rceil + p - 1$, it is easily verified that for any (i,r,a) such that $i=s \bmod p$, the variables $\{ x_j ; j<i-q \}$ are output from R before the moment where x_i is output from R_s. As a consequence, by adding a suitable number of delay cells between the output of R_r and the input of R_s, we can supply any sub-array $R_{s,r,a}$, $1<=s,r<=p$, $a=0,1$, with the flow of variables which are necessary to realize

the computations illustrated in figure 7.

The major advantage of the preceding construction is to introduce a fast triangular system solver which is modular and whose fan-out is bounded by 2p. However, the algorithm cannot be applied directly on any band matrix: we have assumed above that the first q subdiagonals of the system matrix where all zero. Contrarily to the case p=1 described in the previous section, it is not possible to design a locally interconnected VLSI array to annihilate these subdiagonals, unless enough hardware is available to perform the columns interchanges which are needed to preserve local connections in the array. Rather, such a preprocessing should be performed on the host computer. Finally, we point out that annihilating 2p supplementary subdiagonals of the matrix would permit to reduce the fan-out to 2 in the array, simply by using a binary tree to send back any variable to the 2p sub-arrays of the network.



where $v = u_{r,s} - a.p$



Computations performed by $R_{s,r,a}$

Figure 7

5. CONCLUSION

We have introduced a scope-2 substitution process which is computation expensive but which leads to a modular and locally interconnected VLSI array for the solution of band triangular linear systems. This VLSI array runs twice faster than the one of Kung and Leiserson [5], and its fan-out is limited to 2. In the last section, we have extended this result by discussing how to balance speed against fan-out in the VLSI design of a triangular system solver, provided that a sufficient number of first diagonals of the system matrix are zero.

The importance of designing VLSI arrays with a limited fan-out has been emphasized. On the other hand, for SIMD or MIMD algorithms, the fan-out corresponds to the maximum number of processors which must read simultaneously a given variable. The difficulty of a simultaneous access of a variable by different processors in a parallel computer greatly reduces the performance of usual parallel algorithms when implemented on SIMD and MIMD machines . As a consequence, the SIMD or MIMD implementation of the algorithms introduced here may run faster than the classical schemes proposed in the literature.

REFERENCES

[1] CHEN S.C., KUCK D.J., Time and parallel processor bounds for linear recurrence systems, IEEE Trans. Computers, C-24 (1975), p 701-717

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

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

[4] HELLER D., A survey of parallel algorithms in numerical linear algebra, SIAM Review 20 (1978), p 740-777

[5] KUNG H.T., LEISERSON C.E., Systolic arrays for (VLSI), Sparse Matrix Proc. 1978, I.DUFF and G.W. STEWART eds., p 256-282, SIAM 1979

[6] SAMEH A.H., Numerical parallel algorithms - a survey, in High Speed Computer and Algorithm Organization, D. KUCK, D. LAWRIE and A. SAMEH eds., p 207-228, Academic Press, 1977

[7] SAMEH A.H., An overview of parallel algorithms in numerical linear algebra, Bull. EDF C, 1 (1983), p 129-134

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

Journal of VLSI and Computer Systems

Authors:

Yves ROBERT and Maurice TCHUENTE
C.N.R.S. Laboratoire TIM 3 - IMAG
BP 68
F - 38402 SAINT MARTIN D'HERES CEDEX
FRANCE

Title:

AN EFFICIENT SYSTOLIC ARRAY FOR THE 1D RECURSIVE CONVOLUTION PROBLEM

Abstract:

This paper is concerned with the design of a new systolic array for the one-dimensional recursive convolution problem. The solution we obtain runs twice faster than the previously known arrays without any significant increase in area. The approach we use to derive this new solution is based on a divide-and-conquer strategy and appears to be a useful tool for the designing of systolic arrays of maximal efficiency.

1. INTRODUCTION

Systolic arrays, as defined by KUNG and LEISERSON (9), are a useful tool for designing special-purpose VLSI chips. A chip based on a systolic design consists of essentially only one type of simple cells which are mesh-interconnected in a regular and modular way, and achieves high performance through extensive concurrent and pipeline use of the cells.

KUNG carefully explains in (7) how such architectures should result in cost-effective and high-performance special-purpose systems, and he gives three main arguments to justify the advantages of the systolic model:
- systolic systems are easy to implement because of their simple and regular design
- they are easy to reconfigure because of their modularity
- they permit multiple computations for each memory access, which speeds up the execution of compute-bound problems without increasing I/O requirements.

Indeed, such designs have been shown to be very suitable for the solution of compute-bound problems such as signal processing or numerical linear algebra : see for instance (1), (3), (4), (6), (8) or (9).

Unfortunately, most of the yet introduced systolic arrays suffer from the drawback that the elementary cells are alternatively activated and idle, which limits the degree of parallelism and increases the computation time. Two solutions have been proposed to eliminate these drawbacks: considering a problem (P) solved on a systolic array where each cell is activated only every k-th time step, KUNG and LEISERSON (9) suggest to coalesce k adjacent processors into a single one. This reduces the area by a factor of k but doesn't improve the computation time. A second solution consists of interleaving k independent samples of (P), which doesn't speed up either the execution of each sample of (P).

In this paper, we introduce an approach based on the classical divide-and-conquer (DAC for short) technique which can be stated as follows (5): given a problem and a problem instance, divide it into k subproblems; once these subproblems have been solved, combine their solutions into a solution for the original problem. This technique is well-known for classical algorithms (5) and we illustrate here the power of this method on the 1D recursive convolution problem. The solution obtained is new, and runs twice faster than the previously known algorithms without any significant increase in area.

2. 1D CONVOLUTION

The 1D convolution problem is defined as follows (3): given the weighting coefficients $w_1, w_2, \ldots, w_h$ and the input sequence $\{x_i ; i>0\}$, we wish to compute the output sequence $\{y_i ; i>h\}$ defined by

$$y_i = w_1 \cdot x_{i-1} + w_2 \cdot x_{i-2} + \ldots + w_h \cdot x_{i-h}$$

In such a problem, the basic operation is a multiply-and-add performable by an inner-product-step (IPS) cell (9). Since the systolic structures we study are regular, the total area of a chip solving the 1D convolution problem can be approximated by the number of IPS cells (i.e. we neglect the interconnections and the delay-cells, and take as unit the area of one IPS cell). Moreover, the time performance of an array is measured by the efficiency e which is the average number of outputs y_i delivered every time cycle. KUNG (7) has proposed for such computations, two linear networks whose performances are summarized in the following table:

	number of IPS cells	efficiency
Design W1	h	1/2
Design W2	h	1

The reason why Design W1 of (7) is only of efficiency e = 1/2 is the following: in this design, variables x_j and y_i flow in opposite directions. It is easily checked that a variable y_i cannot meet two consecutive elements x_j and x_{j+1} if they are separated by a single cycle-time (see figure 1). As a consequence, any two consecutive variables in both flows are separated by two cycle times, which leads to the value e = 1/2.

However, we point out that design W1 can be used with an efficiency e = 1 to solve the 1D convolution problem when every second weigthing coefficient is zero: for instance, if

$$w_2 = w_4 = \ldots = w_{2i} = \ldots = 0$$

then a variable y_i needs to meet only x_{i-1}, x_{i-3}, x_{i-5}, ... This case is illustrated in figure 2: the cells corresponding to zero coefficients in Design W1 have been deleted and two consecutive variables are separated by a single cycle-time; the solution is now of efficiency e = 1. The design of figure 2 is the basic sub-array of the solution we propose here and is

denoted hereafter by W^*.

To solve the general problem with an efficiency e = 1, the divide-and-conquer approach can be applied by splitting the computation of every y_i into two sub-computations

$$y_i^1 = w_2 \cdot x_{i-2} + w_4 \cdot x_{i-4} + \cdots$$

and

$$y_i^2 = w_1 \cdot x_{i-1} + w_3 \cdot x_{i-3} + \cdots$$

Just as before, we can use a design $W1^*$ for the computation of the sequence $\{y_i^1;\ i \geq h\}$. We use $\lfloor h/2 \rfloor$ IPS cells for this first sub-computation. Similarly, we use a design $W2^*$ of $\lceil h/2 \rceil$ IPS cells for the computation of the sequence $\{y_i^2;\ i \geq h\}$. Finally, we use a special cell to perform the combination of the outputs y_i^1 and y_i^2 of these two networks (by adding them) into a solution y_i for the original problem: as shown in figure 3, this yields a design DAC with h+1 IPS cells ($\lfloor h/2 \rfloor$ cells for the sub-design $W1^*$, $\lceil h/2 \rceil$ cells for the sub-design $W2^*$, and the special cell) and which is of efficiency 1. Note that in figure 3 the special cell is the rightmost upper one.

This new design DAC is as fast as design W2 and, in the network, the data x_j and y_i flow in opposite directions as in design W1 of (7); this last property is very important because, if we feed any computed value y_i back into the network, then

- y_i can meet y_{i+2}^1, y_{i+4}^1, ... in the sub-design $W1^*$ (we just need to introduce a delay cell)
- y_i can meet y_{i+1}^2, y_{i+3}^2, ... in the sub-design $W2^*$ (we feed it back directly).

As shown in figure 4, this leads to a network RDAC of efficiency 1 with h IPS cells for the recursive computation

$$y_i = w_2 \cdot y_{i-2} + w_3 \cdot y_{i-3} + \cdots + w_h \cdot y_{i-h}$$

where $y_0, y_1, \ldots, y_{h-1}$ are given. We point out that we cannot use this method for the recursive computation

$$y_i = w_1 \cdot y_{i-1} + w_2 \cdot y_{i-2} + \cdots + w_h \cdot y_{i-h}$$

Indeed, if y_i is output from the network at time t, then y_{i+1}^2 performs the partial accumulation

$$y_{i+1}^2 := y_{i+1}^2 + w_1 \cdot y_i$$

at time $t' > t + 1$, and the special cell delivers

$$y_{i+1} = y^1_{i+1} + y^2_{i+1}$$

at time $t'' > t' + 1 > t + 2$. . Thus the efficiency cannot be equal to 1. We shall see how to solve the general one-dimensional recursive convolution problem in the next section.

3. 1D RECURSIVE CONVOLUTION

Let us now consider the general one-dimensional recursive convolution problem, which consists of computing

$$y_i = w_1 \cdot x_{i-1} + w_2 \cdot x_{i-2} + \ldots + w_h \cdot x_{i-h} + r_1 \cdot y_{i-1} + r_2 \cdot y_{i-2} + \ldots + r_k \cdot y_{i-k} \quad \text{for } i > \max(h,k)$$

where

$w_1, \ldots, w_h$; $r_1, \ldots, r_k$; $\{x_i ; i>0\}$ and $y_0, \ldots, y_{k-1}$ are given.

As shown in the previous section, we cannot derive a solution of efficiency $e = 1$ if y_i depends on y_{i-1} (i.e. $r_1 \neq 0$). Thus we rewrite this expression by replacing y_{i-1} by its definition formula

$$y_{i-1} = (w_1 \cdot x_{i-2} + \ldots + w_h \cdot x_{i-1-h}) + (r_1 \cdot y_{i-2} + \ldots + r_k \cdot y_{i-1-k})$$

which leads to

$$y_i = w_1 \cdot x_{i-1} + w_2 \cdot x_{i-2} + \ldots + w_h \cdot x_{i-h} + r_1 \cdot ((w_1 \cdot x_{i-2} + \ldots + w_h \cdot x_{i-1-h}) + (r_1 \cdot y_{i-2} + \ldots + r_k \cdot y_{i-1-k})) + r_2 \cdot y_{i-2} + \ldots + r_k \cdot y_{i-k}$$

We can now express y_i as

$$y_i = w'_1 \cdot x_{i-1} + w'_2 \cdot x_{i-2} + \ldots + w'_h \cdot x_{i-h} + w'_{h+1} \cdot x_{i-(h+1)} + r'_2 \cdot y_{i-2} + r'_3 \cdot y_{i-3} + \ldots + r'_k \cdot y_{i-k} + r'_{k+1} \cdot y_{i-(k+1)}$$

where (*) $w'_1 = w_1$; $w'_j = w_j + r_1 \cdot w_{j-1}$ for $2 \leq j \leq h$; $w'_{h+1} = r_1 \cdot w_h$

(**) $r'_j = r_j + r_1 \cdot r_{j-1}$ for $2 \leq j \leq k$; $r'_{k+1} = r_1 \cdot r_k$

As a consequence, as shown in figure 5, a combination of designs DAC and RDAC yields a network with h+k+1 IPS cells and of efficiency 1, for the solution of the general 1D recursive convolution problem . This design,

when compared with all the previously known networks, runs twice faster without any significant increase in area. Let us point out that our solution fits very well to the adaptative convolution problem: when the weights w_j and r_j are dynamically changed, formulae (*) and (**) induce a systolic solution for real-time performances (see figure 6 for the basic principle).

4. CONCLUDING REMARKS

In this paper, we have illustrated on the 1D convolution problem the power of the well-known divide-and-conquer approach for the design of efficient systolic architectures. Our new design is very suitable for applications where speed is the main criterion, since the computation time is significantly reduced.

The DAC approach has been used here to derive a new efficient array from a well-known systolic one. Other applications of this approach can be found in (11). From a methodological point of view, this technique may constitute the last step in the construction introduced by LEISERSON and SAXE (10): starting from a network with rippling combinational logic, use the Systolic Conversion Theorem of (10) to remove the rippling logic and obtain a systolic array; most often, this requires the introduction of a k-slowed version ($k > 2$) of the original array. The DAC approach is then a tool to recover an array of maximal efficiency.

REFERENCES

(1) AHMED H.M., DELOSME J.M., MORF M., Higly concurrent computing structures for matrix arithmetic and signal processing, Computer magazine, January 1982 p 65-82

(2) DOHI Y., FISHER A.L., KUNG H.T., MONIER L.M., WALKER H., Design of the PSC: A Programmable Systolic Chip, in Proc. of Third Caltech Conference on VLSI, R. BRYANT ed, Computer Science Press 1983

(3) FOSTER M.J., KUNG H.T., The design of special purpose VLSI chips, Computer magazine, January 1980 p 26-40

(4) HELLER B., IPSEN I., Systolic networks for orthogonal equivalence transformations and their applications, Proc. 1982 Conf. Advanced Research in VLSI, p 113-122, Massachussetts Institute of Technology, 1982

(5) HOROWITZ E., ZORAT A., Divide-and-Conquer for parallel processing, IEEE Trans. on Computers C32(6), June, 1983

(6) KULKARNI A.V., YEN D.W.L., Systolic processing and an implementation for signal and image processing, IEEE Trans. on Computers C31 N°10 (Oct.82) p 1000-1009

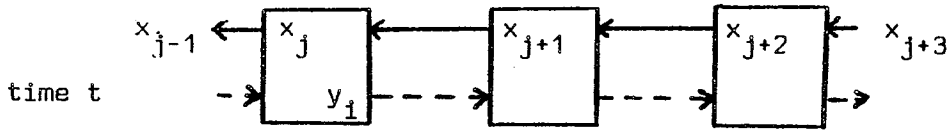
(7) KUNG H.T., Why systolic architectures, Computer Magazine , January 1982 p 37-46

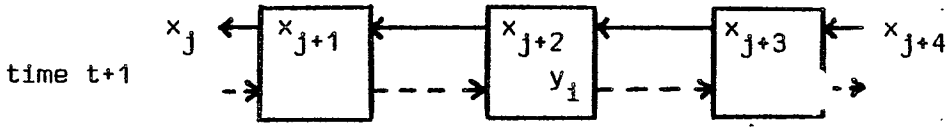
(8) KUNG H.T., Special-purpose device for signal and image processing, Proceedings of the SPIE, vol 241, Real time signal processing 3, July 1980, p 76-84

(9) KUNG H.T., LEISERSON C.E., Systolic arrays for (VLSI), Sparse Matrix Proc. 1978, I "UFF and G.W.STEWART eds., p 256-282, SIAM 1979

(10) LEISERSON C.E., SAXE J.B., Optimizing synchronous systems, 23-th Annual Symposium on Foundations of Computer Science, IEEE October 1981 p 23-36

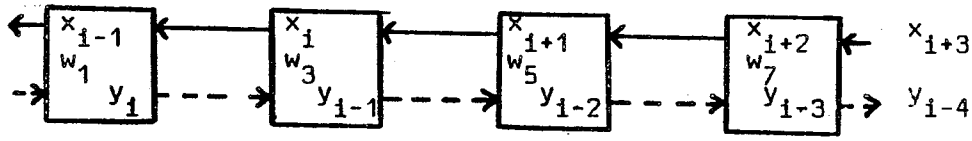
(11) ROBERT Y., TCHUENTE M., Réseaux systoliques pour des problèmes de mots, RAIRO Theoretical Computer Science, to appear in 1984





y_i does not meet x_{j+1}

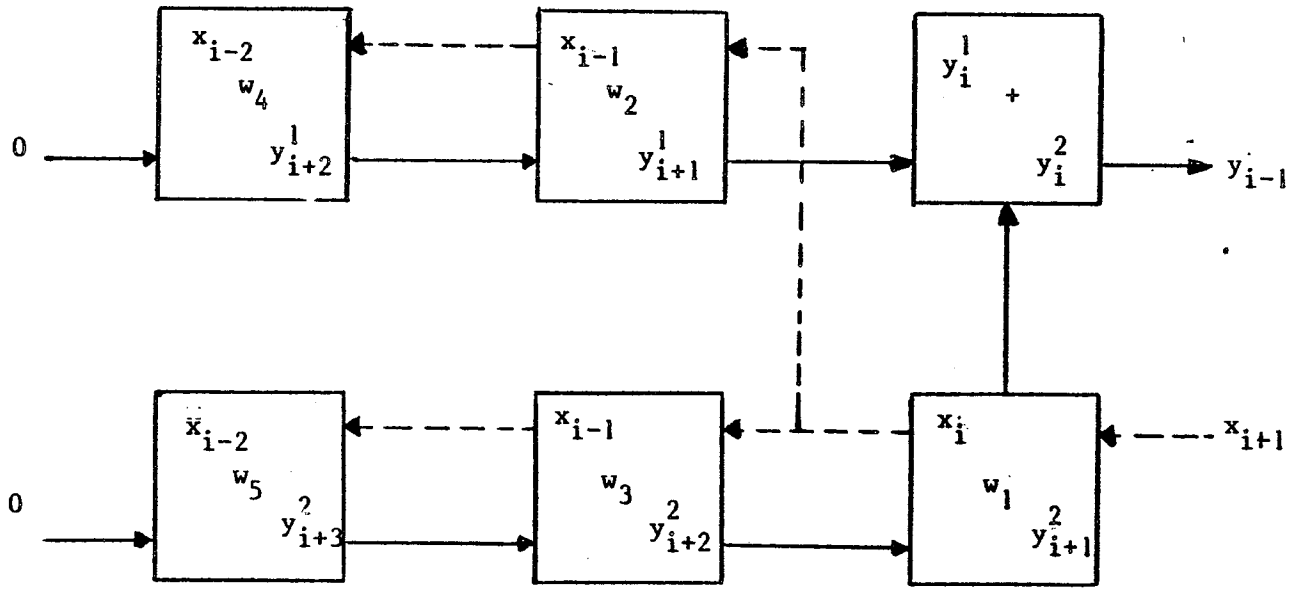
Figure 1



Computation of

$$y_i = \Sigma_{j=0}^{3} w_{2j+1} x_{i-(2j+1)}$$

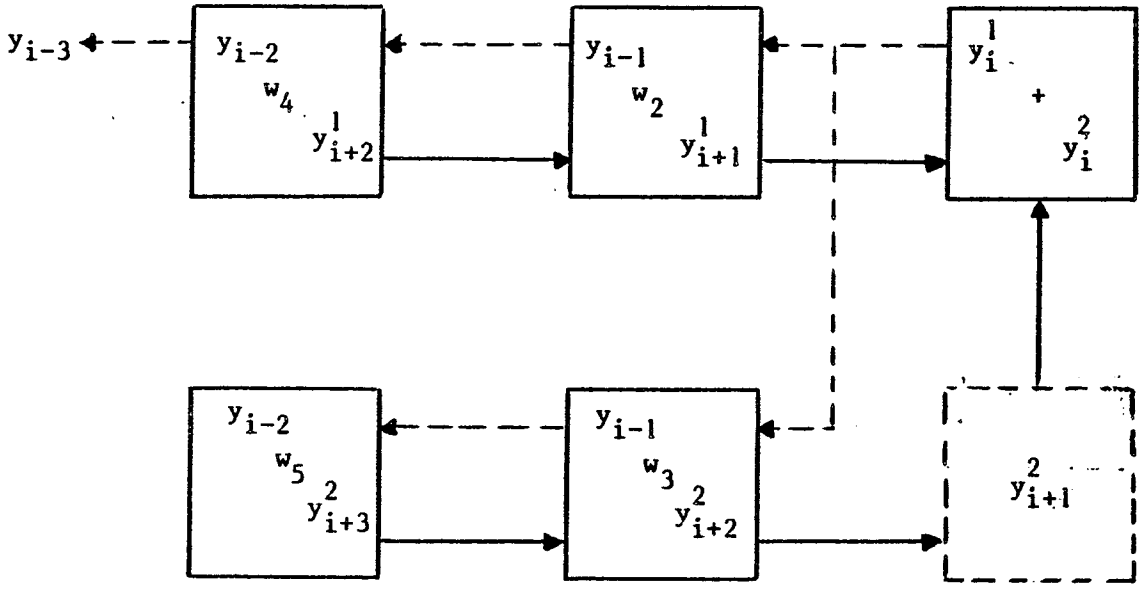
Figure 2



1D convolution with h = 5

$y_i = \Sigma_{j=1}^{5} w_j . x_{i-j}$

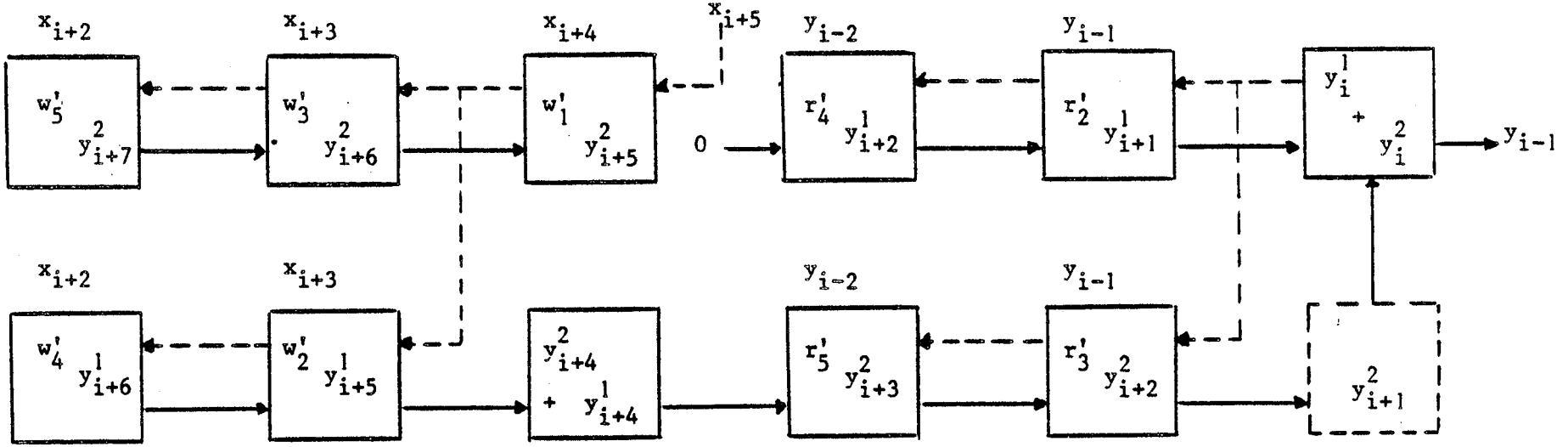
Figure 3



Computation of

$y_i = \Sigma_{j=2}^{5} w_j . y_{i-j}$ (h = 5)

Figure 4



1D recursive convolution with h = k = 4

$$y_i = \Sigma_{j=1}^{4} w_j . x_{i-j} + \Sigma_{j=1}^{4} r_j . y_{i-j}$$

Figure 5

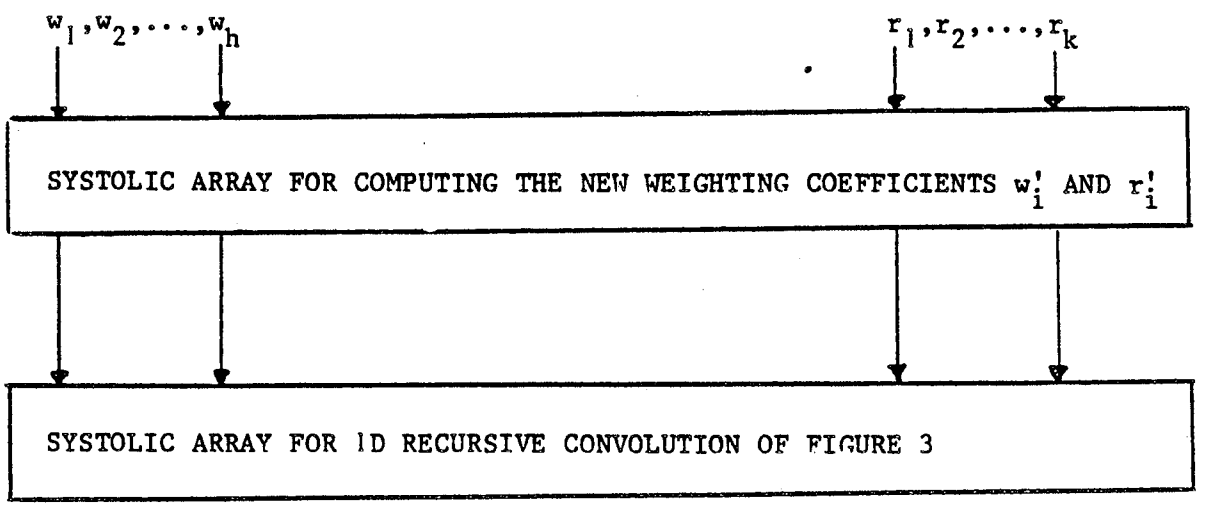


Figure 6

Adaptative convolution

SOUS-CHAPITRE 6.2.

R.A.I.R.O. Informatique théorique/Theoretical Informatics
(vol. 19, n° 2, 1985, p. 107 à 123)

RÉSEAUX SYSTOLIQUES POUR DES PROBLÈMES DE MOTS (*)

par Yves ROBERT (1) et Maurice TCHUENTE (1)
Communiqué par J. BERSTEL

Résumé. — *On présente des réseaux systoliques pour la résolution en temps réel de problèmes de convolution généralisée. On a en vue la définition d'architectures systoliques d'efficacité maximale à partir d'une approche de type « divide-and-conquer » (approche bien connue dans le cas séquentiel). On donne trois applications à des problèmes de mots :*

- *le calcul du nombre d'occurrences d'un mot dans une chaîne;*
- *la reconnaissance des palindromes;*
- *la reconnaissance des carrés.*

Cette technique « divide-and-conquer » apparaît comme le complément naturel de la méthodologie de systolisation des circuits développée par Leiserson et Saxe [11].

Abstract. — *We introduce systolic arrays for the real time solution of some general convolution problems. This paper is devoted to illustrate the power of the divide-and-conquer approach (which is well-known for sequential algorithms) for the design of efficient systolic architectures. We give three applications to words problems:*

- *the computation of the occurrences of a given word in a string;*
- *the real-time palindrome recognizer;*
- *the real-time square acceptor.*

This divide-and-conquer technique can be viewed as the last step of the methodology introduced by Leiserson and Saxe [11] *for the design of systolic systems.*

1. INTRODUCTION

Le modèle systolique a été introduit en 1978 par Kung et Leiserson [10] pour tirer parti des nouvelles technologies d'intégration à haute densité; en 5 ans, ce modèle s'est révélé être un outil puissant pour la conception de processeurs intégrés spécialisés.

Un circuit basé sur une architecture systolique comprend, pour l'essentiel, un réseau régulier de cellules élémentaires, localement interconnectées et globalement synchronisées, et réalise de hautes performances « en temps réel » par une utilisation simultanée et pipe-line des cellules.

(*) Reçu décembre 1983, révisé juin 1984.
(1) C.N.R.S., Laboratoire T.I.M. 3/I.M.A.G., B.P. n° 68, 38402, Saint-Martin d'Hères Cedex.

R.A.I.R.O. Informatique théorique/Theoretical Informatics
0399-0540/85/02 107 17/$ 3,40/

Kung décrit en détail dans [8] tout l'intérêt du modèle pour la réalisation de circuits spécialisés :

— Les architectures systoliques sont faciles à implémenter à cause de leur dessin simple et régulier. Tout algorithme systolique conduit directement à un schéma de réalisation sur silicium;

— ces architectures sont aisément reconfigurables en raison de leur modularité;

— elles permettent de nombreux calculs pour chaque accès-mémoire, ce qui accélère le traitement de problèmes coûteux en nombre d'opérations sans augmenter les exigences en matière d'entrées/sorties.

De fait, de telles architectures se sont avérées très performantes pour la résolution de nombreux problèmes issus d raitement du signal [7], de l'algèbre numérique linéaire [8, 10], ou encore des bases de données [9]. Plus récemment, des réseaux systoliques ont été proposés pour la reconnaissance des langages [3, 4, 5 et 14].

Une définition précise du terme systolique a été introduite en 1981 par Leiserson et Saxe [11]. De plus, ces deux auteurs présentent une démarche constructive pour définir une architecture systolique à partir d'une architecture comprenant de la logique combinatoire se propageant en cascade. Le seul inconvénient de leur méthodologie est qu'elle débouche le plus souvent sur des réseaux où chaque cellule n'est activée qu'un top d'horloge sur k, $k \geqq 2$, ce qui limite par un facteur k le degré de parallélisme obtenu et augmente le temps d'exécution.

Plusieurs solutions ont été proposées pour remédier à cette situation : Kung et Leiserson [10] proposent de fondre k cellules adjacentes en une seule, ce qui diminue la surface par un facteur k mais n'améliore pas le temps d'exécution. Une autre solution consisterait à entrelacer la résolution de k problèmes indépendants (de même nature) sur le réseau, mais ceci n'accélère pas la résolution de chaque problème.

Nous présentons ici une approche basée sur la technique classique « divide-and-conquer » qui peut s'énoncer ainsi [6] : étant donné un problème à résoudre, diviser ce problème en k sous-problèmes que l'on résoud séparément, puis combiner la solution de ces k sous-problèmes pour reconstruire une solution du problème initial. Cette technique est bien connue pour les algorithmes usuels, et nous allons illustrer son application aux architectures systoliques (pour augmenter la vitesse d'exécution) sur des exemples issus de problèmes de mots.

Nous commençons par rappeler brièvement la construction de Leiserson et Saxe [11].

2. UNE MÉTHODE DE SY[illegible]OLISATION

Un circuit est défini formellement comme un graphe orienté $G=(V, U)$, dont les sommets représentent les éléments fonctionnels du circuit. Un sommet particulier représente la structure hôte, qui permet au circuit de communiquer avec le monde extérieur.

Chaque sommet v de G est affecté d'un poids $d(v)$ positif qui est le temps de cycle de la cellule qu'il représente; chaque arc $e=(v, v')$ de U est affecté d'un poids entier $w(e)$ qui représente le nombre de registres que doit traverser une donnée pour aller de v à v'. Les circuits systoliques sont alors ceux pour lesquels chaque arc porte au moins un registre; leur synchronisation peut ainsi s'effectuer globalement au rythme d'une horloge dont le temps de cycle est $\operatorname{Max}_{v \in V} d(v)$.

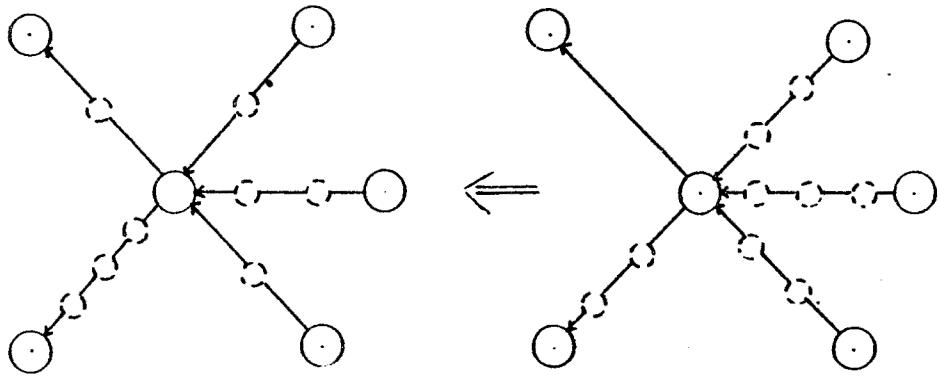


Figure 1. — Transformation élémentaire du graphe de communication.

Il est clair que la transformation qui consiste à retirer un registre sur chaque arc entrant dans une cellule donnée, et à en ajouter un sur chaque arc sortant de cette cellule, ne modifie en rien le comportement de la cellule vis-à-vis des éléments voisins (cf. *fig.* 1). Par ailleurs, on vérifie aisément que de telles transformations laissent invariant le nombre de registres sur tout circuit élémentaire; en conséquence, une condition nécessaire pour que de telles transformations conduisent à un circuit systolique, est que, sur chaque circuit élémentaire du graphe initial, le nombre de registres soit supérieur ou égal au nombre d'arcs. Leiserson et Saxe ont montré que cette condition nécessaire est aussi suffisante. Ceci les amène à proposer la méthodologie suivante pour la conception d'architectures systoliques :

— définir un réseau simple R (en général non systolique) où à chaque top d'horloge le résultat doit s'accumuler le long d'un chemin ne comportant pas de registres (logique combinatoire se propageant en cascade);

– déterminer le plus petit entier k tel que le réseau R_k obtenu à partir de R en multipliant par k le poids de tous les arcs soit systolisable; R_k a alors le même comportement externe que R mais sa vitesse est divisée par k;

– systoliser R_k, à l'aide des transformations précédentes (on peut définir des algorithmes polynomiaux performants pour cela).

Cette méthodologie conduit donc en général à un réseau qui délivre un résultat seulement une pulsation sur k. Nous proposons ici une approche purement algorithmique basée sur la stratégie classique du « divide-and-conquer » qui permet, à partir de R_k, d'obtenir un réseau R^* de même vitesse que R. Cette approche peut ainsi être considérée comme la dernière étape de la méthodologie de systolisation que nous venons de décrire brièvement.

3. CONVOLUTION GÉNÉRALISÉE

Soit E un ensemble non vide, $(B, *)$ un monoide commutatif, et $\top$ une loi de composition externe sur E à valeurs dans B :

$$E \times E \to B,$$
$$(e, e') \mapsto e \top e'.$$

Dans la suite, $\top$ est appelé opérateur de base.

La convolution généralisée se définit comme suit :

Données : – une suite de E, $(a_i; i \geqq 1)$; un élément a_i est appelé symbole d'entrée;

– une suite d'entiers $(n_i; i \geqq 1)$;

– une suite de vecteurs $W = (w^{(i)} \in E^{n_i}; i \geqq 1)$; un vecteur $w^{(i)}$ est appelé vecteur de référence.

Problème : Calcul de la suite $(y_i; i \geqq 1)$ définie par :

$$y_i = (a_i \top w_1^{(i)}) * (a_{i-1} \top w_2^{(i)}) * \ldots * (a_{i-n_i+1} \top w_{n_i}^{(i)}).$$

La convolution classique :

$$y_i = (a_i \top w_1) * (a_{i-1} \top w_2) * \ldots * (a_{i-k+1} \top w_k);$$
$$i \geqq k,$$

correspond au cas particulier où la suite W est constante et égale à $(w_1, w_2, \ldots, w_k)$.

L'approche « divide-and-conquer » consiste à séparer les calculs en deux parties :

$$y_i^1 = (a_i \mathsf{T} w_1^{(i)}) * (a_{i-2} \mathsf{T} w_3^{(i)}) * \ldots,$$
$$y_i^2 = (a_{i-1} \mathsf{T} w_2^{(i)}) * (a_{i-3} \mathsf{T} w_4^{(i)}) * \ldots$$

On construit alors un réseau R comportant :

- un sous-réseau $R1$ pour le calcul de y_i^1;
- un sous-réseau $R2$ pour le calcul de y_i^2;
- une cellule qui reçoit y_i^1, y_i^2 en entrée et délivre en sortie :

$$y_i = y_i^1 * y_i^2.$$

La raison de ce découpage est que, dans les réseaux systoliques proposés pour résoudre divers problèmes de convolution (au sens large où nous l'avons définie), les variables a_i et y_j circulent dans des directions opposées. Or on vérifie aisément (cf. *fig.* 2) que, dans un réseau linéaire, une variable y_j ne peut pas rencontrer deux éléments a_i, a_{i+1} qui se suivent. En conséquence, toute solution basée sur un réseau linéaire et où les variables a_i, y_j circulent dans des directions opposées, est d'efficacité 1/2, c'est-à-dire que deux variables consécutives y_j, y_{j+1} sont séparées par deux tops d'horloge [10] (cf. *fig.* 5).

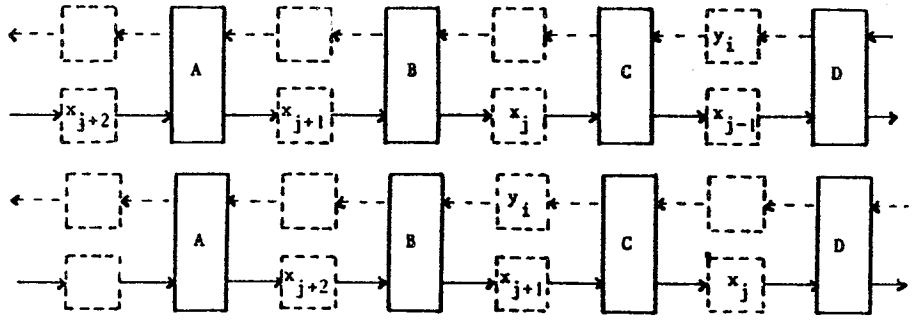


Figure 2. — Instant $t+1$: x_{j+2} et y_i sont des entrées de la cellule B.

L'approche « divide-and-conquer » permet de procéder ainsi :

— calcul de $y_i^1 = (a_i \mathsf{T} w_1^{(i)}) * (a_{i-2} \mathsf{T} w_3^{(i)}) * \ldots$;

sur un réseau $R1$ d'efficacité 1,

— calcul de $y_i^2 = (a_{i-1} \mathsf{T} w_2^{(i)}) * (a_{i-3} \mathsf{T} w_3^{(i)}) * \ldots$;

sur un réseau $R2$ d'efficacité 1,

— calcul de $y_i = y_i^1 * y_i^2$ avec une efficacité 1.

Dans la suite, nous allons étudier les cas suivants qui sont de difficulté croissante :

cas 1 : La suite de référence W est constante : $w^{(i)}=(w_1, w_2, \ldots, w_k)$ pour tout i;

cas 2 : Les vecteurs de référence s'allongent par la droite au cours du temps :

$$w^{(i)}=(w_1, w_2, \ldots, w_p); \qquad p=[i/2],$$

$w^{(i)}$ correspond aux p premiers termes d'une suite fixée $(w_j; j \geqq 1)$;

cas 3 : Les vecteurs de référence s'allongent par la gauche au cours du temps :

$$w^{(i)}=(w_p, w_{p-1}, \ldots, w_1); \qquad i=2p,$$

$w^{(i)}$ correspond aux p premiers termes, pris dans l'ordre inverse, d'une suite fixée $(w_j; j \geqq 1)$.

Chacun de ces cas sera illustré par un problème de mots.

4. NOMBRE D'OCCURRENCES D'UN MOT DANS UNE CHAÎNE

Le premier cas, celui de la convolution avec une référence constante, peut s'illustrer par le problème de détection des occurrences d'un mot dans une chaîne de caractères.

Soit donc $A=a_1 \ldots a_N$ et $W=w_n \ldots w_2 w_1$, $n<N$, deux mots sur un alphabet S. Une occurrence de W dans A est une sous-chaîne $a_i a_{i+1} \ldots a_{i+n-1}$ qui est égale à W. Le problème posé ici est celui de la détection des occurrences de W dans A. Par exemple, si $A=abababa$ et $W=aba$, alors $a_1 a_2 a_3$, $a_3 a_4 a_5$, $a_5 a_6 a_7$ sont des occurrences de W dans A.

Ce problème est équivalent au calcul des variables booléennes :

$$(1) \qquad y_i = \begin{cases} [a_i=w_1] \wedge [a_{i-1}=w_2] \wedge \ldots \wedge [a_{i-n+1}=w_n] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{pour} \quad i \geqq n, \\ 0, \qquad \text{pour} \quad i<n. \end{cases}$$

Une solution évidente (non systolique) est donnée par le schéma de la figure 3 [8]. Étant donnée la présence dans ce réseau de circuits comportant deux arcs et un seul registre, on commence par doubler le nombre de registres sur chaque arc, ce qui conduit au réseau deux fois plus lent de la figure 4. L'application de la méthode de [11] conduit alors au réseau de la figure 5.

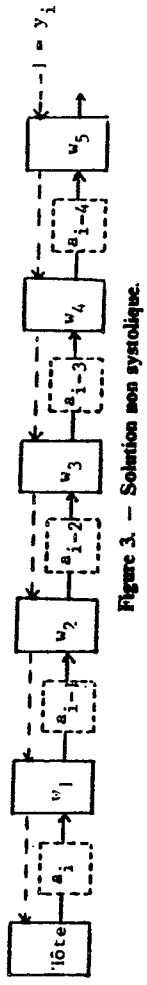


Figure 3. — Solution non systolique.

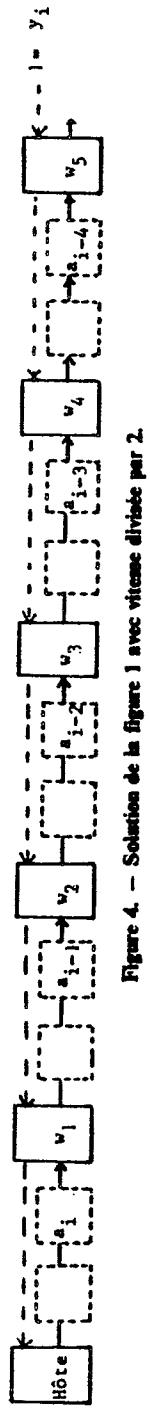


Figure 4. — Solution de la figure 1 avec vitesse divisée par 2.

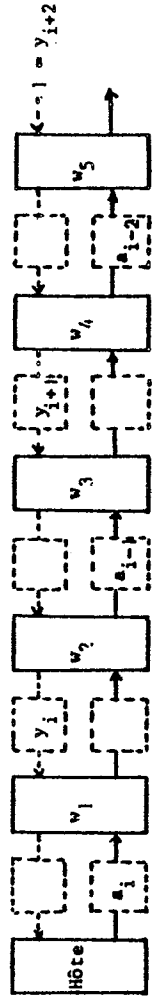


Figure 5. — Version systolisée du réseau de la figure 3.

Il s'agit bien d'un problème de convolution pour lequel :

- l'opérateur de base est l'égalité;
- $\wedge$ désigne la conjonction;
- la référence est constante et égale à $w_1 w_2 \ldots w_n$.

La technique « divide-and-conquer » peut s'appliquer ainsi :

- calcul dans un réseau $R1$ de :

$$y_i^1 = [a_i = w_1] \wedge [a_{i-2} = w_3] \wedge \ldots;$$

- calcul dans un réseau $R2$ de :

$$y_i^2 = [a_{i-1} = w_2] \wedge [a_{i-3} = w_4] \wedge \ldots;$$

- calcul de $y_i = y_i^1 \wedge y_i^2$ à la sortie de $R1$ et $R2$.

On obtient ainsi la solution présentée à la figure 6.

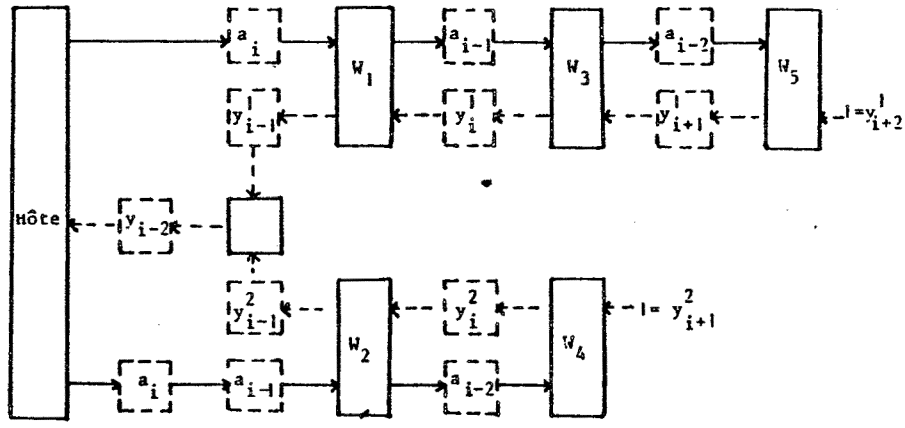


Figure 6. — Détection en temps réel des occurrences d'un mot.

Calcul du nombre d'occurences disjointes

Appelons occurences disjointes de W dans A deux occurences $a_i a_{i+1} \ldots a_{i+n-1}$, $a_j a_{j+1} \ldots a_{j+n-1}$ qui n'ont aucune position commune, c'est-à-dire telles que les intervalles $[i, i+n-1]$ et $[j, j+n-1]$ sont disjoints. Dans l'exemple examiné au début de ce paragraphe, avec $A = abababa$ et $B = aba$, le nombre d'occurences disjointes de B dans A est 2.

Le problème revient à calculer la suite $(y_i;\ i \geqq 1)$ définie par :

$$(2) \qquad y_i = \begin{cases} [a_i = w_1] \wedge \ldots \wedge [a_{i-n+1} = w_n] \\ \qquad \wedge \bar{y}_{i-1} \wedge \ldots \wedge \bar{y}_{i-n+1} \quad \text{pour} \quad i \geqq n, \\ 0, \quad \text{pour} \quad i < n, \end{cases}$$

où $\bar{y}$ désigne le complémentaire de y.

Pour résoudre ce problème à partir du réseau de la figure 6, il suffit que chaque variable de rang i, une fois calculée, rencontre celles de rang $i+k$, $1 \leqq k \leqq n-1$, afin de les faire basculer à 0 si elle-même vaut 1. Pour ce faire, on procède comme suit (cf. *fig.* 7) : chaque y_i, une fois calculée, est placée dans un registre d'entrée de la cellule qui calcule $y_i = y_i^1 \wedge y_i^2$, afin de rencontrer y_{i+1}, et est aussi renvoyée dans le réseau à la rencontre des y_{i+k}, $1 \leqq k \leqq n-1$; il est à noter que cette variable renvoyée dans le réseau ne doit pas atteindre toutes les cellules, car elle ferait alors basculer indûment à 0 des variables y_{i+m}, $m \geqq n$.

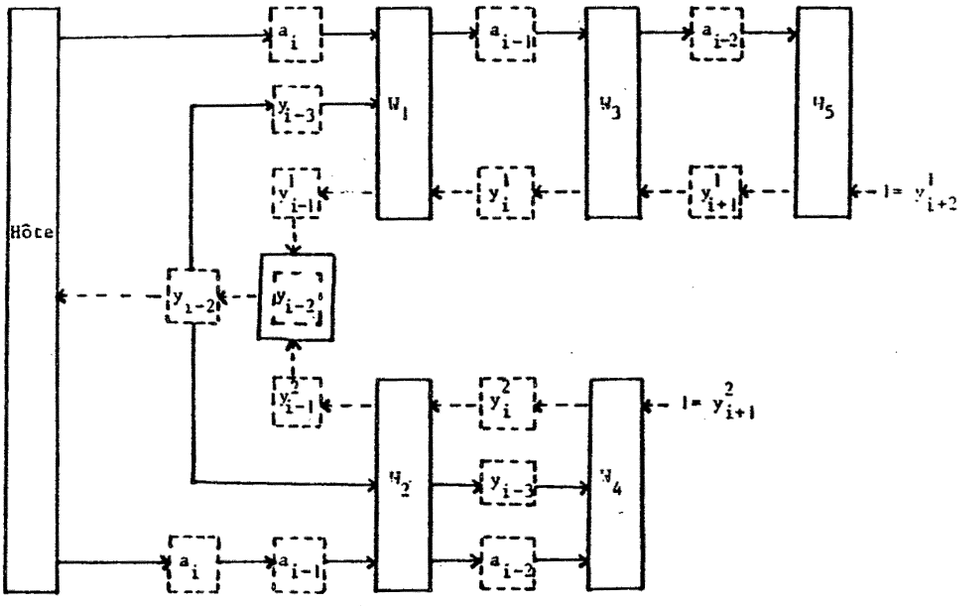


Figure 7. — Calcul en temps réel des occurrences disjointes.

Remarques 1. Le fait de placer y_i dans un registre d'entrée de la cellule qui calcule $y_i = y_i^1 \wedge y_i^2$ n'augmente pas le temps de cycle du réseau, car cette cellule est alors amenée à calculer :

$$y_i = y_i^1 \wedge y_i^2 \wedge \bar{r},$$

où r est la valeur du registre interne, et reste donc de complexité nettement inférieure aux autres cellules du réseau.

2. Le problème que nous venons de traiter est très semblable dans sa formulation à la convolution récursive :

$$y_i = w_1 \cdot x_i + \ldots + w_n \cdot x_{i-n+1} + r_1 \cdot y_{i-1} + \ldots + r_m \cdot y_{i-m},$$

w_j, r_j, x_i réels; $y_1, \ldots, y_m$ réels donnés.

Cependant la remarque 1 ne s'applique pas à la convolution récursive car la cellule qui calcule y_i aurait alors à effectuer une addition supplémentaire par rapport aux autres cellules; inversement, la technique exposée dans [12] et qui consiste à réécrire la convolution récursive sous la forme :

$$y_i = w_1^* \cdot x_i + \ldots + w_n^* \cdot x_{i-n+1} + r_2^* \cdot y_{i-2} + \ldots + r_{m+1}^* \cdot y_{i-m-1},$$

ne peut être utilisée ici car elle conduirait à des formules inexploitables.

5. RECONNAISSANCE DES PALINDROMES

Le deuxième cas que nous allons traiter est celui où la référence s'allonge par la droite au cours du temps. Ceci correspond à la situation où $w^{(i-1)}$ est préfixe de $w^{(i)}$, et peut s'illustrer à l'aide de l'exemple de la reconnaissance des palindromes.

Une chaîne $a_1 a_2 \ldots a_n$ est appelée un palindrome si elle est symétrique, c'est-à-dire si $a_i = a_{n-i+1}$ pour tout i. Les palindromes constituent un langage hors contexte très simple, et leur reconnaissance est un problème classique. Barzdin [1] a montré que la reconnaissance d'un palindrome de taille n sur une machine de Turing ayant une seule tête de lecture-écriture nécessite un temps de l'ordre de n^2. Plus tard, Smith [13] a exhibé un automate cellulaire uniforme de dimension un, capable de reco aître un palindrome de longueur n en temps linéaire $t_n \leqq n/2$; le déploiement du diagramme espace-temps de cet algorithme donne la solution proposée récemment par Culik *et al.* [4]. Cependant, le modèle de Smith n'est pas adapté au pipe-line et ne permet donc pas, pour une suite $(a_i;\ i \geqq 1)$ donnée, le calcul en temps réel de la suite booléenne $(y_i;\ i \geqq 1)$ définie par :

$$y_i = \begin{cases} 1 & \text{si } a_1 a_2 \ldots a_i \text{ est un palindrome,} \\ 0 & \text{sinon.} \end{cases}$$

Ce dernier problème a été résolu pour la première fois par Cole [2]; Leiserson et Saxe [11] ont proposé récemment une solution beaucoup plus simple mais dans laquelle deux symboles a_i consécutifs sont séparés par deux tops d'horloge, et ce pour les raisons que nous avons exposées au paragraphe précédent.

La formule de calcul d'une variable y_i est :

$$y_i = [a_i = a_1] \wedge [a_{i-1} = a_2] \wedge \ldots \wedge [a_{i-\lfloor i/2 \rfloor + 1} = a_{\lfloor i/2 \rfloor}],$$

où $[u]$ désigne le plus petit entier supérieur ou égal à u.

Il s'agit donc d'un problème de convolution pour lequel :

— l'opérateur de base est l'égalité;

— $\wedge$ désigne la conjonction;

— $w^{(i)} = a_1 a_2 \ldots a_{[i/2]}$ pour tout $i \geqq 1$.

Comme les mots de référence $w^{(i)}$, $i \geqq 1$, dépendent de la suite des symboles d'entrée $(a_i;\ i \geqq 1)$, le traitement d'un symbole a_i comporte trois phases :

— dans une première phase, tant que a_i rencontre y_j, $j \leqq 2(i-1)$, il joue le rôle de symbole d'entrée;

— dans une seconde phase, lorsqu'il rencontre y_{2i-1}, il sert simultanément de symbole d'entrée et de symbole de référence;

— enfin, dans la troisième phase, lorsqu'il rencontre y_j, $j \geqq 2i$, il joue le rôle de symbole de référence.

La solution au problème de la reconnaissance des palindromes en temps réel s'obtient à partir du réseau de reconnaissance des occurences d'un mot (cf. *fig.* 5), en adjoignant un dispositif permettant de générer dynamiquement les mots de référence $w^{(i)}$, $i \geqq 1$. On procède comme suit :

— a_{2j+1} ne sert de symbole de référence que dans le sous-réseau $R1$; en conséquence, on adjoint une variable booléenne Test_{2j+1}, qui vaut 1 dans la copie de a_{2j+1} envoyée dans $R1$, et 0 dans la copie envoyée dans $R2$. Un traitement analogue est appliqué aux variables a_{2j}, $j \geqq 1$;

— le problème important de détermination du début de la phase 3 pour un couple $(a_{2j}, 1)$ ou $(a_{2j+1}, 1)$ se résoud en remarquant tout simplement que celà correspond à la rencontre d'une cellule dont le registre interne est vide.

Ceci conduit à la solution de la figure 8.

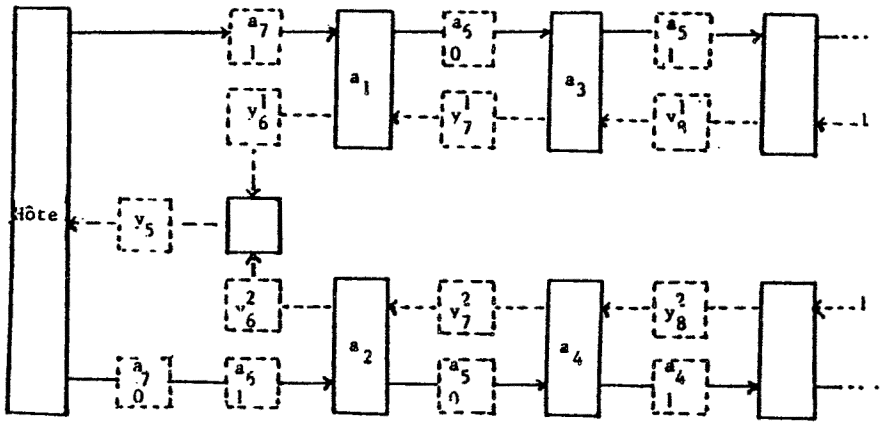


Figure 8. — Reconnaissance des palindromes en temps réel.

6. RECONNAISSANCE DES CARRÉS

Le troisième cas que nous étudions est celui où la référence s'allonge par la gauche au cours du temps. Ceci correspond à la situation où, pour tout $i \geqq 1$, $w^{(i-1)}$ est un suffixe de $w^{(i)}$, et peut s'illustrer par le problème de la reconnaissance des carrés.

Une chaîne $A = a_1 a_2 \ldots a_n$, $n = 2p$ entier pair, est appelée un carré si $a_i = a_{i+p}$ pour $1 \leqq i \leqq p$. Soit donc à évaluer, pour $(a_i; i \geqq 1)$ donnée, la suite booléenne définie par :

$$y_i = \begin{cases} 1 & \text{si } a_1 a_2 \ldots a_i \text{ est un carré,} \\ 0 & \text{sinon.} \end{cases}$$

La formule de calcul de y_i est :

$$y_i = \begin{cases} [a_i = a_p] \wedge [a_{i-1} = a_{p-1}] \wedge \ldots \\ \qquad \wedge [a_{p+1} = a_1] \quad \text{si } i = 2p, \\ 0 \quad \text{sinon.} \end{cases}$$

Il s'agit bien d'un problème de convolution pour lequel :

- l'opérateur de base est l'égalité;
- $\wedge$ désigne la conjonction;
- $w^{(i)} = a_p a_{p-1} \ldots a_1$ pour $i = 2p$; ($w^{(i)}$ est indéfini pour i impair).

Comme dans le cas du palindrome, chaque symbole a_i est amené à jouer deux rôles :

- lorsqu'il est comparé à a_j, $j < i$, pour le calcul d'une variable y_k, $k < 2i$, il sert de symbole d'entrée;
- lorsqu'il est comparé à a_j, $j > i$, pour le calcul d'une variable y_k, $k \geqq 2i$, il joue le rôle d'un caractère de référence.

Pour que le calcul s'effectue en temps réel, il est nécessaire qu'à tout instant $t = 2i$, le symbole d'entrée a_{2i}, le symbole de référence a_i et la variable y_{2i}^1 soient en entrée de la première cellule de $R1$, de manière à ce que l'accumulation :

$$y_i^1 := y_i^1 \wedge [a_{2i} = a_i],$$

puisse s'effectuer.

Pour des raisons de clarté, nous allons d'abord supposer que la structure hôte fournit les données de la manière suivante :

- à tout instant $t = 2i + 1$, elle délivre le symbole d'entrée a_{2i+1};
- à tout instant $t = 2i$, elle délivre le symbole d'entrées a_{2i} et le symbole de référence a_i.

Il faut alors définir un mécanisme qui, à partir des caractères d'entrée fournis par la structure hôte, génère dynamiquement les mots de référence :

$$w^{(i)} = a_p a_{p-1} \ldots a_1, \qquad i = 2p.$$

Considérons la copie du caractère de référence a_i qui a l'instant $t = 2i$ est envoyée dans le sous-réseau $R1$:

$$y_{2i} = [a_{2i} = a_i] \wedge [a_{2i-1} = a_{i-1}] \wedge \ldots \wedge [a_{i+1} = a_1],$$

elle joue le rôle de $w_1^{(2i)}$ et se trouve donc en entrée de la première cellule;

– comme les caractères de référence du sous-réseau $R1$ ne sont amenés à jouer les rôles de $w_k^{(j)}$ que pour k impair, a_i intervient dans la deuxième cellule de $R1$ pour jouer le rôle de $w_3^{(j)}$, ce qui correspond à $j = 2i+4$:

$$y_j = [a_{2i+4} = a_{i+2}] \wedge [a_{2i+3} = a_{i+1}] \wedge [a_{2i+2} = a_i] \wedge \ldots \wedge [a_{i+3} = a_1].$$

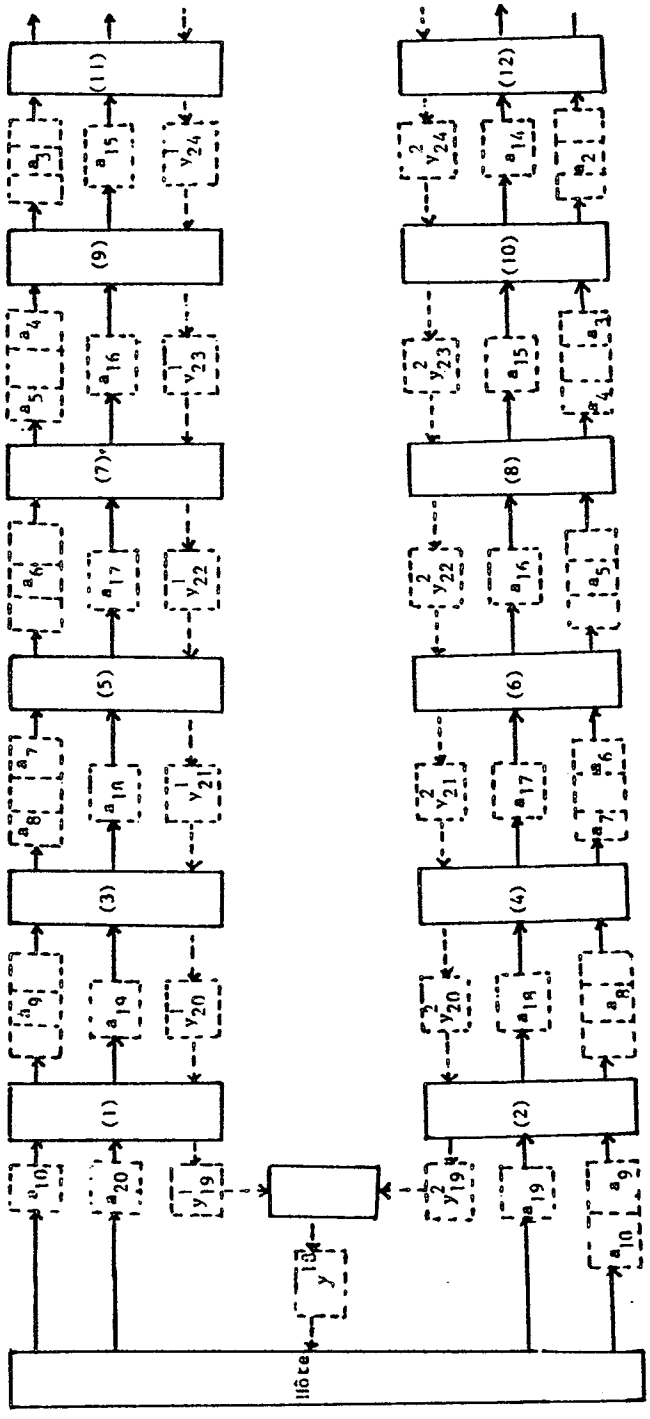
Comme l'accumulation :

$$y^1_{2i+4} := y^1_{2i+4} \wedge [a_{2i+2} = a_i],$$

s'effectue dans la deuxième cellule de $R1$ à l'instant $t = 2i+3$, on déduit que a_i doit traverser trois cellules de retard pour passer d'une cellule de $R1$ à la suivante.

Un raisonnement analogue s'applique à la copie du caractère de référence a_i qui est envoyée dans le sous-réseau $R2$. En outre, comme nous l'avons remarqué plus haut, la contrainte temps-réel impose que les caractères de référence soient espacés de deux tops d'horloge. On est ainsi conduit au réseau de la figure 9.

Intéressons-nous maintenant au réseau qui doit recevoir en entrée a_i à l'instant i et le délivrer à la première cellule à l'instant $2i$. Ce réseau est implicite dans la construction proposée par Cole [2] pour la reconnaissance des carrés en temps réel. Il suffit de considérer un réseau linéaire où chaque cellule comporte deux couloirs de circulation. Le principe de fonctionnement est le suivant : chaque symbole a_i circule dans le couloir du bas, en avançant à chaque top d'une cellule vers la droite, jusqu'à l'instant t_0 où dans la cellule voisine à sa droite les deux registres sont vides. A l'instant t_0+1 le symbole gagne le couloir du haut, et à partir de l'instant t_0+2 circule dans ce couloir vers la gauche, toujours en avançant d'une cellule à chaque top. Quelques pulsations consécutives sont représentées figure 10 pour illustrer la marche de ce dispositif.



Les tests $a_{i-s+1} = w_i^{(h)}$ s'effectuent dans la cellule (k).
Figure 9. — Reconnaissance des carrés en temps réel.

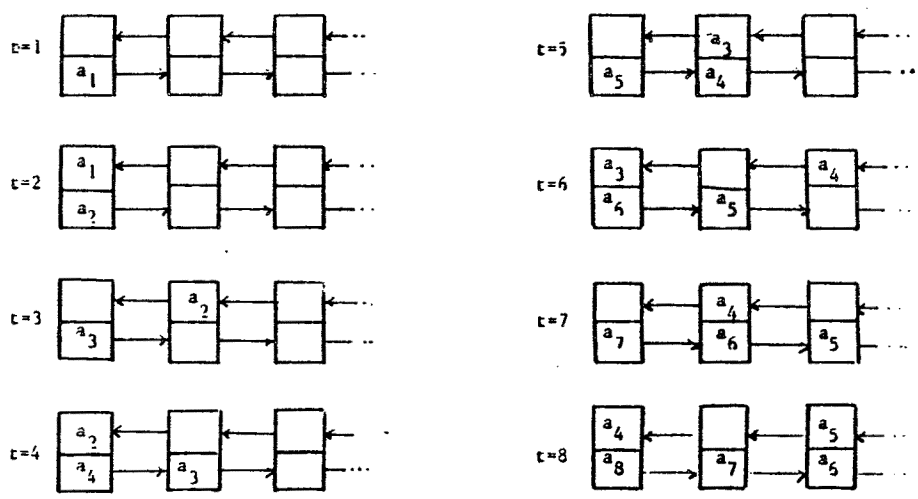


Figure 10

REMARQUE : Dans tous les exemples que nous avons traités, l'approche « divide-and-conquer » conduit à une solution où chaque variable d'entrée comporte deux copies dans le réseau. Lors d'une réalisation effective, cette duplication des entrées serait effectuée par une cellule spéciale à l'intérieur du circuit. Cependant, pour une raison de clarté, nous avons supposé que c'est la structure hôte qui effectue cette duplication des données.

7. CONCLUSION

La méthode de systolisation de Leiserson et Saxe [11], complétée par une transformation de type « divide-and-conquer » à son dernier stade, se révèle être un outil très efficace : ainsi pour le problème de convolution généralisée est-on conduit à une solution temps réel où le circuit reçoit une nouvelle donnée et délivre un nouveau résultat tous les tops d'horloge.

Des techniques de type « divide-and-conquer » avaient été introduites par les auteurs dans [12] pour améliorer les performances de réseaux systoliques pour la convolution récursive et le produit de matrices. De fait, cette approche peut être utilisée à partir de circuits déjà systoliques. Mais d'un point de vue méthodologique, elle s'avère constituer le complément naturel (la dernière étape) d'une démarche qui, partant d'une solution évidente mais non adaptée au calcul en temps réel, conduit à un circuit systolique de performance maximale.

D'autre part, l'introduction de la notion de convolution généralisée présente un intérêt en elle-même. En effet, comme nous l'avons montré, c'est un

contexte qui englobe des problèmes très variés sous un formalisme unique, et qui permet de résoudre ces problèmes en temps réel, en adjoignant au réseau de convolution classique un mécanisme de génération dynamique des vecteurs de référence. D'ailleurs, cette approche ne se limite pas aux problèmes de mots : ainsi la conception d'un réseau systolique d'efficacité maximale pour la multiplication d'une matrice A à structure bande par un vecteur ou encore pour la résolution d'un système triangulaire de matrice A :

$$A x = b,$$

rentrent directement dans ce formalisme et correspond au cas où les vecteurs de référence évoluent dynamiquement au cours du temps, mais tout en restant de longueur fixe.

Notons enfin que, pour la reconnaissance des palindromes et des carrés, on peut retrouver les réseaux proposés par Cole [2] à partir des réseaux présentés ici, en effectuant des regroupements entre des cellules voisines. Notre approche cumule ainsi deux avantages :

– elle évite les preuves compliquées de Cole [2], grâce à l'utilisation du formalisme de Leiserson et Saxe [11];

– elle conduit à une solution d'efficacité maximale, grâce à l'emploi d'une approche « divide-and-conquer », et remédie ainsi au manque de performance des solutions obtenues dans [11].

REMERCIEMENTS

Nous remercions le professeur Jean Berstel dont les suggestions nous ont permis de clarifier certains points délicats, ainsi que le rapporteur dont les remarques ont conduit à une amélioration de la présentation de cet article.

BIBLIOGRAPHIE

1. Y. M. Barzdin, *Complexity of Recognition of Symmetry in Turing Machines*, Problemy Kibernetiki, vol. 15, 1965.
2. S. N. Cole, *Real-time Computation by n-Dimensional Iterative Arrays of Finite-state Machines*, I.E.E.E. Trans. on Computers, vol. C 18, avril 1979, p. 349-365.
3. K. Culik II, J. Gruska et A. Salomaa, *On a Family of L. Languages Resulting from Systolic Tree Automata*, Research Report CS-81-36, University of Waterloo.
4. K. Culik II, J. Gruska et A. Salomaa, *Systolic Treillis Automata for (VLSI)*, Research Report CS-81-36, University of Waterloo.
5. M. J. Foster and H. T. Kung, *Recognize Regular Langages with Programmable Building-blocks, in the Proceedings of the VLSI*, 1981, Conference, Edinburgh.
6. E. Horowitz et A. Zora, *Divide-and-conquer for Parallel Processing*, I.E.E.E. Trans. on Computers, vol. C 32, (6), juin 1983.

7. A. V. KULKARNI et D. W. L. YEN, *Systolic Processing and an Implementation for Signal and Image Processing*, I.E.E.E. Trans. on Computers, vol. C 31, octobre 1982, p. 1000-1009.
8. H. T. KUNG, *Why Systolic Architectures*, I.E.E.E. Computer, vol. 15, n° 1, janvier 1982, p. 37-46.
9. H. T. KUNG et P. L. LEHMAN, *Systolic V.L.S.I. Arrays for Relational Database Operations*, Proceedings of the 1980 A.C.M./SIGMOD International Conference on Management of data, Los Angeles, U.S.A.
10. H. T. KUNG et C. E. LEISERSON, *Systolic Arrays for (VLSI)*, in the Proceedings of the Symposium on Sparse Matrix Computations and their Applications, Knoxville, 1978.
11. C. E. LEISERSON et J. D. SAXE, *Optimizing Synchronous Systems*, 22-th Annual Symposium on Foundations of Computer Science, I.E.E.E., octobre 1981, p. 23-36.
12. Y. ROBERT et M. TCHUENTE, *Designing Efficient Systolic Algorithms*, Rapport de Recherche 393, *Lab. I.M.A.G., Grenoble*, septembre 1983.
13. A. R. SMITH III, *Cellular Automata theory*, Technical Report 2, *Stanford Electronics Laboratories*, décembre 1969.
14. M. STEINBY, *Systolic Tree and Systolic Langage Recognition by Tree Automata*, Theoretical Computer Science, vol. 22, 1983, p. 219-232.

INFORMATIQUE THÉORIQUE. — Calcul en temps linéaire d'une plus longue sous-suite commune à deux chaînes sur une architecture systolique.

Note de **Yves Robert** et **Maurice Tchuente**, présentée par Jacques-Louis Lions.

Remise le 7 mai 1984.

Nous présentons une architecture parallèle spécialisée pour la détermination en temps $O(m+n)$ d'une plus longue sous-suite commune à deux chaînes de caractères A et B de longueurs respectives m et n.

COMPUTER SCIENCE. — Linear Time Computation for the LCS Problem on a Systolic Architecture.

We propose a special-purpose parallel architecture for the Longest Common Subsequence (LCS) problem. Given two strings A and B of respective lengths m and n, the computation of a LCS of A and B can be done in time $O(m+n)$.

1. INTRODUCTION. — La détermination d'une plus longue sous-suite commune (PLSC) à deux mots A et B sur un alphabet S est un problème que l'on rencontre dans de nombreux domaines d'application : par exemple en traitement de données pour identifier deux fichiers qui sont semblables, ou en génétique pour mesurer le degré de corrélation de deux molécules [7]. Tous les algorithmes connus exigent un temps de calcul proportionnel (pour l'essentiel) au produit des longueurs des mots, et ce, même s'il s'agit seulement de calculer la longueur d'une PLSC, et non pas d'exhiber une telle sous-suite.

Nous montrons ici que l'utilisation d'une architecture parallèle spécialisée permet de résoudre le problème en temps linéaire (c'est-à-dire proportionnel à la somme des longueurs des mots). Cette architecture est systolique [6], c'est-à-dire composée d'un réseau de processeurs identiques et localement interconnectés, qui au même rythme effectuent des opérations élémentaires sur les variables en provenance des processeurs voisins en entrée, puis transmettent un temps de cycle plus tard les résultats aux processeurs voisins en sortie.

2. DÉFINITIONS ET NOTATIONS. — Soit S un ensemble fini de symboles appelé alphabet. Une chaîne sur S est une suite finie quelconque d'éléments de S. On note S^+ l'ensemble de toutes les chaînes non vides. Soit A un élément de S^+, on représente A par $A = A(1)\,A(2)\ldots A(m)$, et $l(A) = m$ désigne la longueur de A. Une sous-suite de A est obtenue en supprimant certains éléments (non nécessairement consécutifs). En particulier, $A(1:i)$ désigne la sous-suite $A(1)\,A(2)\ldots A(i)$. Enfin, une PLSC de deux chaînes A et B est une sous-suite commune à A et B de longueur maximale. Par exemple si $A = abcdbb$ et $B = cbacbaab$, alors $A' = bcbb$ et $A'' = acbb$ sont les deux PLSC.

Dans toute la suite, m et n désignent les longueurs respectives des chaînes A et B; p représente la longueur d'une PLSC de A et B, et r le nombre total de sous-suites communes de longueur deux.

3. COMPLEXITÉ DU PROBLÈME. — Le tableau ci-dessous détaille les résultats connus (on suppose $m <= n$ sans perte de généralité, et la complexité désigne le temps de calcul dans le cas le moins favorable) *voir* tableau.

0249-6321/84/02990269 $ 2.00

TABLEAU

	Référence	Temps de calcul	Complexité
Algorithme 1	HUNT et coll. [5]	$O((r+n)\log n)$	$O(n*n*\log n)$
Algorithme 2	HIRSCHBERG [4]	$O(pn)$	$O(n*n)$
Algorithme 3	HIRSCHBERG [4]	$O(p(m+1-p)\log n)$	$O(n*n*\log n)$
Algorithme 4	NAKATSU et coll. [7]	$O((r+n)\log n)$	$O(n*n*\log n)$

Il est clair que les algorithmes 1 et 2 sont efficaces quand p est petit, alors que les algorithmes 3 et 4 sont adaptés au traitement de chaînes voisines. Aho et coll. [1] ont montré qu'en l'absence d'informations sur la faille de l'alphabet S, toute solution au problème n'utilisant que des comparaisons directes « égal/différent » nécessitent un temps $O(n^2)$. L'algorithme séquentiel le plus rapide pour le problème de la PLSC s'exécute en temps $O(n*n*\log\log n/\log n)$ [8], ce qui est essentiellement quadratique pour des valeurs usuelles de n.

Ce tour d'horizon établit tout l'intérêt d'un algorithme parallèle s'exécutant en temps linéaire $O(n)$. Quand seule la longueur d'une PLSC doit être calculée, nous utilisons un réseau linéaire de n processeurs élémentaires : notre algorithme parallèle s'exécute en temps $O(n+m)$, ce qui correspond à une efficacité maximale. Lorsqu'il faut déterminer une PLSC, le réseau précédent peut être étendu en une structure rectangulaire de surface $O(m*n)$, mais le temps d'exécution reste linéaire; notons que cette surface est du même ordre que la place mémoire maximale requise par les algorithmes séquentiels ([4], [7]).

4. CALCUL DE LA LONGUEUR D'UNE PLSC. — Notons $\mathrm{L\,max}(i, j)$ la longueur d'une PLSC de $\mathrm{A}(1:i)$ et $\mathrm{B}(1:j)$, $1<=i<=m$, $1<=j<=n$.

LEMME 1. — *Les relations suivantes sont vraies :*

$$\mathrm{L\,max}(i, 0)=0, \qquad 0<=i<=m,$$

$$\mathrm{L\,max}(0, j)=0, \qquad 0<=j<=n$$

$$\mathrm{L\,max}(i+1, j+1)=\mathrm{Max}\{\mathrm{L\,max}(i, j+1), \mathrm{L\,max}(i+1, j),$$

$$\mathrm{L\,max}(i, j)+[\mathrm{A}(i+1)=\mathrm{B}(j+1)]\},$$

où $[.=.]$ *vaut* 1 *si l'égalité est vérifiée, et* 0 *sinon.*

Ce lemme constitue la formulation récursive d'un algorithme séquentiel permettant le calcul en temps $O(m*n)$ de la longueur d'une PLSC de A et B. Pour l'implantation parallèle de cet algorithme, nous considérons un réseau de n processeurs élémentaires identiques, dont le programme de définition est détaillé sur la figure.

La chaîne A entre dans le réseau par la gauche $\mathrm{A}(1)$ au premier top d'horloge et un nouvel $\mathrm{A}(i)$ tous les tops suivants. Le registre b de la j-ième cellule à partir de la gauche contient $\mathrm{B}(j)$, le registre Tab de chaque cellule a pour contenu initial la valeur 0. La première cellule reçoit toujours la valeur 1 sur son port d'entrée Niv-in et la valeur 0 sur L max-in. Soit enfin c le temps de cycle d'un processeur élémentaire.

THÉORÈME. — *Soient* A *et* B *deux chaînes de longueurs respectives m et n. L'architecture parallèle spécialisée de n processeurs élémentaires décrite plus haut calcule la longueur d'une PLSC de* $\mathrm{A}(1:i)$ *et* B *en temps* $(i+n)*c$ *pour tout* $i<=m$. *En particulier, la longueur d'une PLSC de* A *et* B *est calculée en temps* $O(n+m)$.


```
Programme de définition :
si Niv-in <= Tab alors
début Niv-out:=Tab+1; Lmax-out:=Lmax-in;
fin
sinon si a=b alors
début Tab:=Niv-in; Niv-out:=Niv-in;
si Lmax-in<Tab alors Lmax-out:=Tab;
fin.
```

Fig.

5. DÉTERMINATION D'UNE PLSC. — Nous obtenons le résultat suivant :

PROPOSITION. — *Soient* A *et* B *deux chaînes de longueurs respectives m et n. La détermination d'une PLSC de* A *et* B *peut être faite en temps* $O(m+n)$ *sur une architecture parallèle spécialisée de surface totale* $O(m*n)$.

Schéma de démonstration. — Soit p la longueur d'une PLSC à A et B. $B(i_1)B(i_2)\ldots B(i_p)$ est une PLSC ssi il existe $j_1<j_2<\ldots<j_p$ tels que :

$$\text{Lmax}(B(1:i_r),\quad A(1:j_r))=r \qquad \text{pour} \quad 1<=r<=p.$$

L'idée est d'adjoindre à chaque processeur j une pile qui stocke le couple (i, Tab) chaque fois que Tab change de valeur à la suite d'un test d'égalité $[A(i)=B(j)]$. Les piles sont implantées sous forme de cellules élémentaires [3], et leur temps d'insertion/suppression est constant, ce qui assure que le temps de cycle de chaque processeur reste indépendant de m et n.

6. CONCLUSION. — Nous avons introduit un nouvel algorithme pour le calcul d'une PLSC de deux chaînes A et B. Utilisé séquentiellement, cet algorithme s'exécute en temps $O(n*m)$, ce qui est moins rapide que les meilleurs algorithmes connus. Mais il se prête à une parallélisation efficace, et peut s'implanter sur une architecture intégrée spécialisée à structure régulière. Comme bien souvent en calcul parallèle [2], les coûts en matière de communication priment sur le volume d'opérations effectuées.

RÉFÉRENCES BIBLIOGRAPHIQUES

[1] A. V. AHO et coll., *J.A.C.M.*, 23, n° 1, 1976, p. 1-12.
[2] M. J. FOSTER et H. T. KUNG, *Computer Magazine*, 13, n° 1, 1980, p. 26-40.
[3] L. J. GUIBAS et F. M. LIANG, *Proc. Conf. on Advanced Research*, 1982, M.I.T., p. 155-164.
[4] D. S. HIRSCHBERG, *J.A.C.M.*, 24, n° 4, 1977, p. 664-675.
[5] J. W. HUNT et T. G. SZYMANSKI, *C.A.C.M.*, 20, 1977, p. 350-353.
[6] H. T. KUNG et C. E. LEISERSON, *Sparse Matrix Proc.*, I. S. DUFF éd., 1978, p. 256-282.
[7] N. NAKATSU et coll., *Acta Informatica*, 18, 1982, p. 171-179.
[8] M. S. PATERSON, Manuscrit non publié, Univ. de Warvick, Coventry (G.B.), 1974.

Institut I.M.A.G., Tour des Mathématiques, B.P. n° 68,
38402 *Saint-Martin-d'Hères Cedex.*

Information Processing Letters 21 (1985) 191-198
North-Holland

7 October 1985

A SYSTOLIC ARRAY FOR THE LONGEST COMMON SUBSEQUENCE PROBLEM

Yves ROBERT and Maurice TCHUENTE

Centre National de la Recherche Scientifique, Laboratoire TIM3, Institut IMAG, BP 68, 38402 Saint Martin d'Hères Cedex, France

Communicated by L. Boasson
Received October 1984
Revised January 1985

We introduce a linear systolic array for the Longest Common Subsequence (LCS, for short) problem. We first present an array of m identical cells which computes the length of an LCS of two strings of length m and n, respectively, in linear time (i.e., in time proportional to m + n). Then we show that, by extending any cell with the systolic stack introduced by Guibas and Liang (1982), a new array can be designed to recover an LCS in linear time.

Keywords: Words, subsequence, systolic array

1. Introduction

Systolic arrays, as defined by Kung and Leiserson [11], are a useful tool for designing special-purpose VLSI chips. A chip based on a systolic design consists of essentially a few types of very simple cells which are mesh-interconnected in a regular and modular way, and achieves high performance through extensive concurrent and pipeline use of the cells.

Kung carefully explains in [10] how such architectures should result in cost-effective and high-performance special-purpose systems, and he gives three main arguments to justify the advantages of the systolic model:

- systolic systems are easy to implement because of their simple and regular design,
- they are easy to reconfigure because of their modularity,
- they permit multiple computations for each memory access, which speeds up the execution of compute-bound problems without increasing I/O requirements.

Indeed, such designs have been shown to be very suitable for the solution of compute-bound problems such as signal processing, numerical linear algebra, and raster graphics [9,11,14]. More recently, some systolic arrays [2,15] have been proposed for language recognition.

In this paper we introduce a systolic linear array of size m for computing the length p of a Longest Common Subsequence (LCS, for short) of two strings

$$A = A(1)\ldots A(n), \quad B = B(1)\ldots B(m), \quad m \leqslant n,$$

over an alphabet S. This systolic array can be extended to one of size $p * n$ at most in order to recover one LCS of A and B whenever necessary. As pointed out in [12], algorithms for the LCS problem have many applications: for instance, in data processing in order to identify files which are similar, and in genetic applications where they are used to measure the correlation between two molecules. All the previously known algorithms for the LCS problem have a worst-case time complexity essentially proportional to $m * n$ for practical values of m, n. As suggested in [12], parallel computa-

tion appears to be a possible way for obtaining faster algorithms. Here, we present a systolic array which works in a pipeline fashion and which solves the problem in linear time (i.e., in time proportional to m + n).

2. Basic definitions

Here we present some basic definitions and notations [12] that will be used throughout this paper.

Definition 1. Let S be any finite set of symbols, called an alphabet. A *string over* S is any finite sequence of elements from S. S^* means the set of all strings over S (including the empty string E). $S^* - \{E\}$ is denoted by S^+. $\ell(A)$ for A in S^+ denotes the length of A. (We set $\ell(E) = 0$.)

Definition 2. A string A in S is denoted by $A(1)A(2)\ldots A(n)$, where A(i) is in S for $1 \leqslant i \leqslant n = \ell(A)$.

$A' = A(i_1)A(i_2)\ldots A(i_k)$,

where $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$,

is called a *subsequence* of A. In other words, a subsequence of a string is obtained by deleting zero or more symbols from the string.

For a given string $A = A(1)\ldots A(n)$, $A(1:i)$ denotes the consecutive subsequence $A(1)A(2)\ldots A(i)$.

Example. For the string A = abcbcab, A′ = bbab is a subsequence of A and A(1 : 4) = abcb.

Definition 3. Let A and B be two strings over S. A string C is a *common subsequence* of A and B if C is a subsequence of both A and B. String C is a longest common subsequence (LCS, for short) if C is a common subsequence of A and B of maximum length. The problem of finding an LCS of two given strings is called the *LCS problem.*

Example. For the given two strings A = abcdbb and B = cbacbaaba, we see that A′ = bcbb and A″ = acbb are the two LCSs.

Throughout this paper, n and m denote the lengths of strings A and B respectively; p denotes the length of an LCS of A and B, and r is the total number of ordered pairs at which the two sequences match.

3. Previous results

Many algorithms for the LCS problem have appeared in the literature. In Table 1 some previous results are presented ($m \leqslant n$ is assumed in this section without loss of generality).

Clearly, Algorithms A and B are efficient when p is small, whereas Algorithms C and D are suitable for expected similar strings. Complexity results [1,7] have been demonstrated for the LCS problem. Aho et al. [1] have shown that unless the alphabet size is fixed, every solution to the problem by the 'equal–unequal' comparison will consume an amount of O(n • n) time. The only proven lower bound with an unrestricted alphabet is O(n • log n) [7], whereas the only known algorithm for the LCS problem with worst-case behaviour less than quadratic is due to Paterson [13] and is O(n • n • log log n/log n), which is essentially quadratic, however, for reasonable values of n.

This review shows the interest of an O(n) running time parallel algorithm. When only the length of an LCS is needed, we use an array of m elementary processors: our systolic algorithm has O(n + m) (which is O(n) since $m \leqslant n$ is assumed) running time and thus performs an optimal speed-up with respect to the usual efficiency measure [11]. When recovering an LCS is necessary, we exhibit a linear time systolic array which requires the same amount of storage as the known sequential algorithms [6].

Table 1

Algorithm		Running time	Worst-case behaviour
A	[8]	O(r + n) log n)	O(n • n • log n)
B	[6]	O(p • n)	O(n • n)
C	[6]	O(p(m + 1 − p) log n)	O(n • n • log n)
D	[12]	O(n(m − p))	O(n • n)

4. Computing the length of an LCS

In this section we present a linear array of m cells which computes the length p of an LCS of two strings $A = A(1)\dots A(n)$ and $B = B(1)\dots B(m)$, $m \leqslant n$, in time $\tau \cdot (n + 2m)$, where τ denotes the cycle-time of the basic cell of the array.

4.1. Basic result

Let LLCS(i, j) denote the length of an LCS of

$A(1:i) = A(1)\dots A(i)$ and

$B(1:j) = B(1)\dots B(j)$, $\quad 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant m.$

Thus, $p = LLCS(n, m)$. For notational convenience we shall write

$$LLCS(0, j) = LLCS(i, 0) = 0$$

for $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$.

The algorithm presented here is based on the following property.

Proposition ([5]). *For any* $i \geqslant 1$, $j \geqslant 1$,

$$LLCS(i, j) = \max\{LLCS(i-1, j),\ LLCS(i, j-1),\ E(i, j) + LLCS(i-1, j-1)\},$$

where

$$E(i, j) = \begin{cases} 1 & \text{if } A(i) = B(j), \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Since LLCS(i, j) is clearly greater than or equal to each of the three quantities

$LLCS(i-1, j)$, $LLCS(i, j-1)$ and

$E(i, j) + LLCS(i-1, j-1)$,

we first derive that

$$LLCS(i, j) \geqslant \max\{LLCS(i-1, j),\ LLCS(i, j-1),\ E(i, j) + LLCS(i-1, j-1)\}.$$

Let us now assume that

$$LLCS(i, j) > \max\{LLCS(i-1, j),\ LLCS(i, j-1)\}.$$

It is easily verified that

$$LLCS(i, j) = 1 + LLCS(i-1, j).$$

As a consequence, there exists a k, $1 \leqslant k \leqslant j$, such that

$A(i) = B(k)$ and

$LLCS(i-1, j) = LLCS(i-1, k-1)$.

If follows that

$$LLCS(i, k) = LLCS(i, j) = 1 + LLCS(i, j-1),$$

hence $k = j$.

We have therefore shown that, if

$$LLCS(i, j) > \max\{LLCS(i-1, j),\ LCCS(i, j-1)\},$$

then

$A(i) = B(j)$ and

$LLCS(i, j) = E(i, j) + LLCS(i-1, j-1)$. □

4.2. Systolic implementation

We first consider a linear array of m identical elementary processors as shown in Fig. 1.

A basic cell j contains a register B which stores B(j), and a register LL which stores LLCS(i, j) after the processing of A(i). Symbol A(i) is processed successively by cell 1, cell 2, ..., cell m;

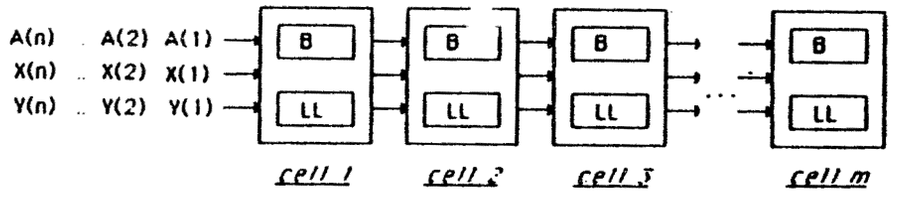


Fig. 1.

Before the processing of A(i) by cell j
B = B(j)
LL = LLCS(i-1,j)
X(i) = LLCS(i-1,j-1)
Y(i) = LLCS(i,j-1)

After the processing of A(i) by cell j
B = B(j)
LL = LLCS(i,j)
X(i) = LLCS(i-1,j)
Y(i) = LLCS(i,j)

Fig. 2.

as A(i) travels right, LLCS(i − 1, j) and LLCS(i, j) are computed, and it carries them along as X(i) and Y(i). As a consequence, A(i) is output at the rightmost cell with Y(i) equal to LLCS(i, m).

The processing of A(i) by cell j is summarized in Fig. 2. Cell j computes LLCS(i, j) as a function of A(i), B(j), LLCS(i − 1, j − 1), LLCS(i, j − 1), and LLCS(i − 1, j) according to the formula of the Proposition.

Let τ denote the cycle-time of the basic cell. It is obvious that A(i) is processed by an array of size m in time τm. Since the processing of A(1), A(2), ..., A(n) can be pipelined, it follows that an array of size m computes p = LLCS(n, m) in time $\tau(n + m)$ if, initially, the B- and LL-registers of any cell j, $1 \leq j \leq m$, contain respectively B(j) and 0.

So far, we have neglected the time necessary to store B(j) in cell j for j = 1, ..., m. We are now going to show how this initialization problem can be solved.

We allow the variables X, Y to assume a special value, say §. The symbols B(1), B(2), ..., B(m) are input sequentially to the leftmost cell with the

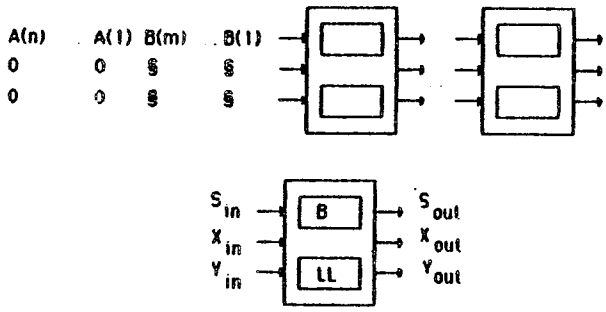


Fig. 3.

Program of the basic cell

```
begin  S_out := S_in ;
       if Y_in = § then
       begin (* S_in belongs to string B *)
           Y_out := Y_in ;
           if X_in = § then
           begin (* S_in must be stored in B *)
               B := S_in ;
               LL := 0 ;
               X_out := 0 ; end
           else (* B, LL remain unchanged *)
               X_out := X_in ; end
       else (* S_in belongs to string A *)
       begin  X_out := LL ;
              LL := Max [ LL , Y_in ] ;
              if S_in = B then
              LL := Max [ LL , 1 + X_in ] ; end
end
```

variables X and Y equal to §. When a cell receives S, X, Y as entries, the condition Y = § tells it that it is a B-symbol, and the condition X = § tells it that it must store S. The overall structure of the array is described in Fig. 3. Clearly, if τ denotes the cycle-time of the basic cell, then the array computes LLCS(n, m) in time $\tau(n + 2m)$. A detailed example is given in Fig. 4.

5. Recovering an LCS

In this section we show how the linear array described in the preceding section can be extended to a two-dimensional array which computes an LCS of

$A = A(1)\ldots A(n)$ and $B = B(1)\ldots B(m)$

in linear time.

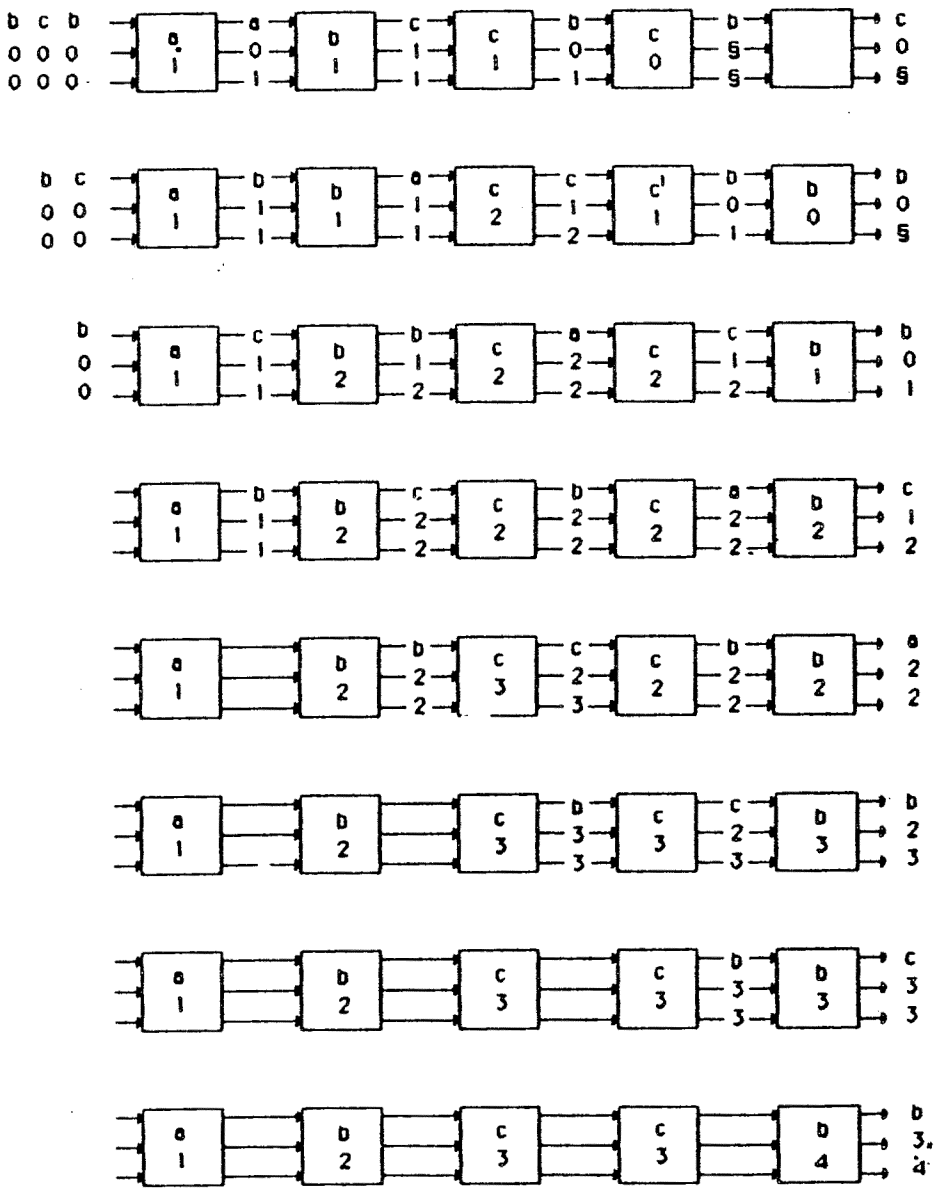


Fig. 4. Some consecutive pulsations for A = bcabcb and B = abccb.

Clearly, in order to recover an LCS of A and B, it is sufficient to compute the set

$$\mathcal{T} = \{(i, j, k) \mid A(i) = B(j) \text{ and } k = LLCS(i, j)\}.$$

However, for a reason which will become clear later, we shall only compute the subset $\mathcal{T}^*$ of $\mathcal{T}$ defined by

$$(i, j, k) \in \mathcal{T}^* \Leftrightarrow i = \min\{h \mid (h, j, k) \in \mathcal{T}\}.$$

The algorithm proceeds in two phases as follows.

Step 1: Computation of $\mathcal{T}^*$

$\mathcal{T}^*$ is computed by adjoining to any cell j of the array described in Section 4 a systolic stack [4], which stores all the couples (i, k) such that $(i, j, k) \in \mathcal{T}^*$. Since cell j processes the symbols of the string A in the order A(1), A(2),..., A(n), it follows that $(i, j, k) \in \mathcal{T}^*$ if and only if A(i) = B(j), k = LLCS(i, j) and the stack of column j does not contain at its top a couple of the form (i', k) (i.e., a couple whose second component is equal to k). As a consequence, the condition $(i, j, k) \in \mathcal{T}^*$ can be tested in constant time. Since the operation of insertion in a systolic stack can be done in constant time, it follows that Step 1 can be executed in time proportional to m + n.

Step 2: Computation of an LCS

At the end of Step 1, the *jth* column of the array contains B(j) and all the couples (i, k) such that $(i, j, k) \in \mathcal{T}^*$. Moreover, the length p of an LCS is known.

Clearly, there exists a j^*, $1 \leq j^* \leq m$, such that the stack associated with column j^* contains at its top a couple (i^*, p). For any such couple there exists an LCS

$$A(i_1) \ldots A(i_p) = B(j_1) \ldots B(j_p)$$

such that $i_p = i^*$ and $j_p = j^*$.

Let us now define a signal S(u, v), $u \geq 1$, $v \geq 1$, which will be processed by any cell j as explained below; we assume that S(u, v) is input to cell j at time t:

(i) if the couple (i, k) at the top of the stack is such that $i \geq u$ or $k \neq v$, then cell j transmits S(u, v) to its left neighbour and has nothing else to do;

(ii) otherwise, it has three things to do:
- first of all, i.e., at time t + 1, it sends to the left a signal dl(v) in order to delete in the stacks of the cells $h < j$ all the couples the form (s, v) (i.e., all the couples whose second component is equal to v). Note that any stack contains at most one such couple, and it is effectively deleted since it is at the top of the stack,
- simultaneously, i.e., at time t + 1, it sends to the right the content of its B-register, i.e., B(j),
- thirdly, one pulse later, i.e., at time t + 2, it sends to the left the signal S(i, v − 1).

Clearly, if S(m + 1, p) is input to cell m at the beginning of Step 2, then (i_p, j_p) is computed correctly and $B(j_p)$ will be output at cell m. Moreover, since (i_{p-1}, j_{p-1}) is characterized by

$$i_{p-1} < i_p, \quad j_{p-1} < j_p \quad \text{and}$$

$$(i_{p-1}, j_{p-1}, p-1) \in \mathcal{T}^*,$$

the signals dl(r) and $S(i_r, r-1)$ generated by the cells j_r, $r = p, p-1, \ldots, 1$ permit to recursively

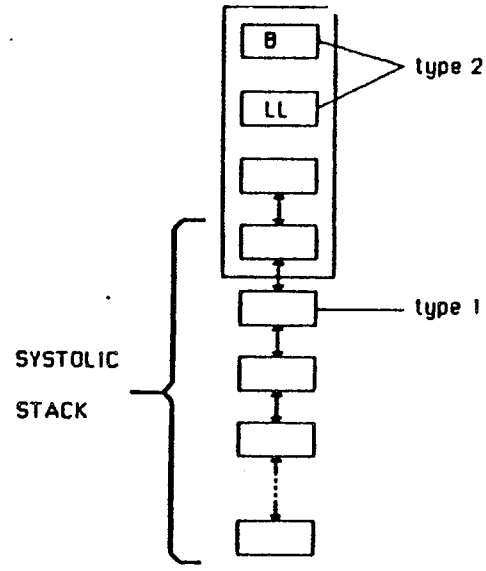


Fig. 5.

compute an LCS

$$A(i_1)\ldots A(i_p) = B(j_1)\ldots B(j_p)$$

such that $(i_r, j_r, r) \in \mathcal{T}^*$ for $r = 1, \ldots, p$.

It is thus possible to compute an LCS of A = A(1)...A(n) and B = B(1)...B(m) in time proportional to m + n on a two-dimensional systolic array containing two types of cells (see Fig. 5):

- the cells of type 1 are the elementary cells of a systolic stack [4],
- a cell of type 2 simulates an elementary cell of Section 4 together with the two top-cells of a systolic stack. This enables a cell of type 2 to perform all the operations which depend on the couple at the top of the systolic stack.

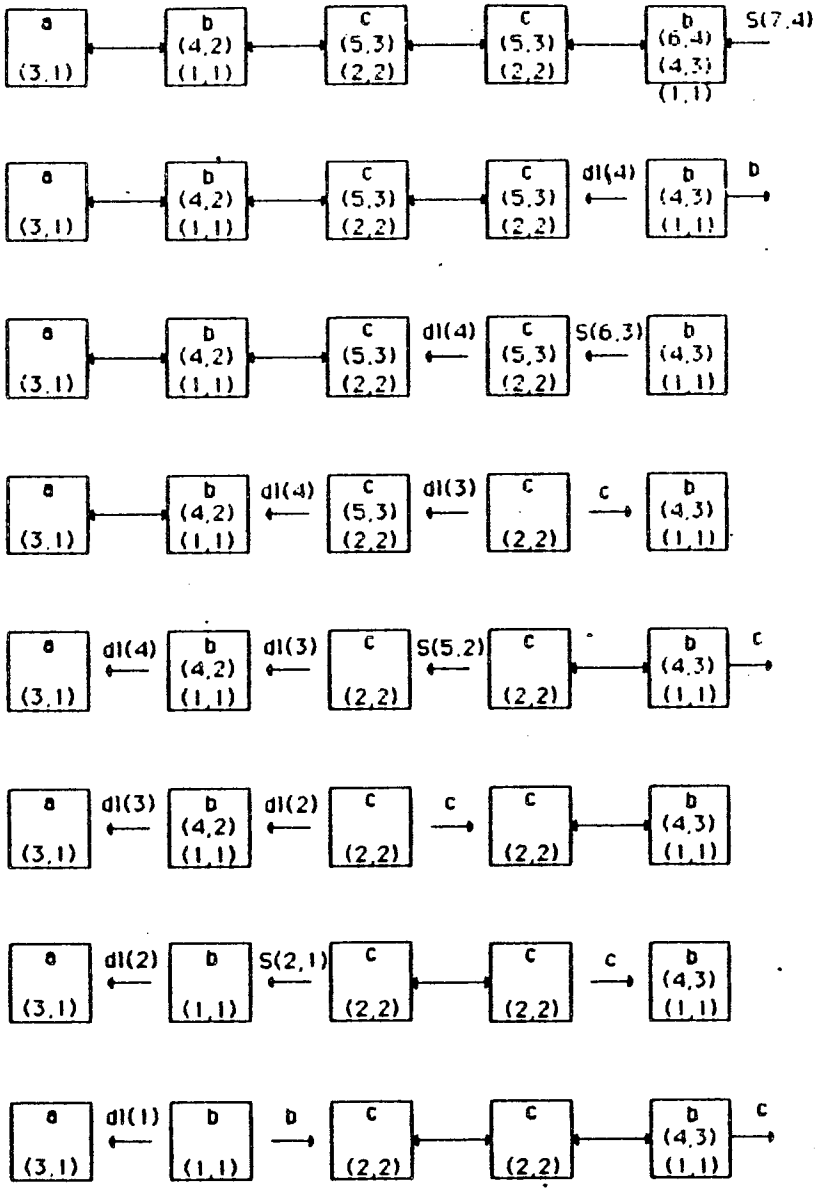


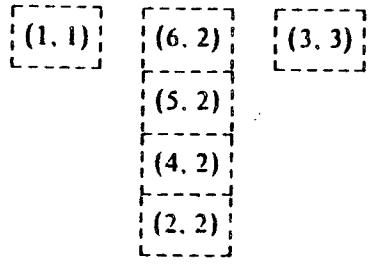
Fig. 6.

The critical instruction from the point of view of speed is the insertion/deletion in the systolic stacks. As pointed out in [4], this operation can be done nicely in constant time, so that the cycle time of our systolic processors is independent of m and n.

The recovering phase (i.e., Step 2) for the example of Fig. 4 is detailed in Fig. 6.

Comments

(i) As shown in the example below, an algorithm based on the computation of $\mathscr{T}$ would have been more complicated in the recovering phase: with A = abcbbb and $\mathring{B}$ = abc, the content of the stacks after Step 1 would be



(ii) Since $(i, j, k) \in \mathscr{T}^*$ and $(i', j', k') \in \mathscr{T}^* \Rightarrow k \neq k'$, it follows that any stack is of maximum depth p = LLCS(n, m). As a consequence, the algorithm requires an array of size proportional to p * m, and is suitable for applications where p is expected to be small. For similar text strings we would use a systolic implementation of the algorithm of Nakatsu et al. [12] (which can be similarly obtained) leading to an O(n * (m − p)) amount of storage.

6. Concluding remarks

We have introduced a systolic array for the LCS problem. The pipeline solution exhibited for computing the length of an LCS is very simple and may easily be implemented in VLSI. Unfortunately, when recovering an LCS is necessary, each cell of the array is extended with a systolic stack, and an interesting problem would be to look for alternate VLSI architectures yielding simpler solutions.

It is worthwhile to note that, when used sequentially, the algorithm we propose would have a quadratic running-time for any value of p, which is worse than the well-known algorithms such as those in [6,12]. This illustrates the fact that, as pointed out in [3], fast parallel algorithms are not those requiring minimal computation.

References

[1] A.V. Aho, D.S. Hirschberg and J.D. Ullman, Bounds on the complexity of the longest common subsequence problem, J. ACM 23 (1) (1976) 1–12.

[2] K. Culik II, J. Gruska and A. Salomaa, On a family of L languages resulting from systolic tree automata, Res. Rept. CS-81-36, Univ. of Waterloo, Canada, 1981.

[3] M.J. Foster and H.T. Kung, The design of special-purpose VLSI chips, IEEE Comput. 13 (1) (1980) 26–40.

[4] L.J. Guibas and F.M. Liang, Systolic stacks, queues and counters, in: Proc. Conf. on Advanced Research in VLSI, M.I.T., 1982.

[5] P.A.V. Hall and G.R. Dowling, Approximate string matching, Comput. Surveys 12 (4) (1980) 381–402.

[6] D.S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (4) (1977) 664–675.

[7] D.S. Hirschberg, An information-theoretic lower bound for the longest common subsequence problem, Inform. Process. Lett. 7 (1) (1978) 40–41.

[8] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences, Comm. ACM 20 (1977) 350–353.

[9] A.V. Kulkarni and D.W.L. Yen, Systolic processing and an implementation for signal and image processing, IEEE Trans. Comput. C-31 (10) (1982) 1000–1009.

[10] H.T. Kung, Why systolic architectures, IEEE Comput. 15 (1) (1982) 37–46.

[11] H.T. Kung and C.E. Leiserson, Systolic arrays for (VLSI), in: Proc. Symp. on Sparse Matrix and Their Applications, Knoxville, 1978.

[12] N. Nakatsu, Y. Kambayashi and S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Informatica 18 (1982) 171–179.

[13] M.S. Paterson, Unpublished manuscript, Univ. of Warwick, Coventry, U.K., 1974.

[14] Y. Robert and M. Tchuente, Résolution systolique de systèmes linéaires denses, Res. Rept. 420, IMAG, Grenoble, France, 1983; RAIRO Numer. Anal. 19 (3) (1985).

[15] M. Steinby, Systolic trees and systolic language recognition by tree automata (Note), Theoret. Comput. Sci. 22 (1983) 219–232.

INFORMATIQUE THÉORIQUE. — *Reconnaissance de langages en temps réel sur une architecture parallèle spécialisée.* Note de **Yves Robert** et **Maurice Tchuente,** présentée par Jacques-Louis Lions.

Soient $n \geq 2$ un entier fixé et A^+ l'ensemble des mots non vides sur un alphabet fini A. Soit f_i, $1 \leq i \leq n-1$, une famille de fonctions dont chacune est égale soit à l'identité soit à la fonction miroir. Nous présentons une architecture systolique périodique et unidimensionnelle pour la reconnaissance en temps réel du langage :

$$L = \{ f_{n-1}(w) f_{n-2}(w) \ldots f_1(w) w;\ w \in A^+ \},$$

et nous montrons que la période du réseau croît exponentiellement avec n.

COMPUTER SCIENCE. — Real-time language recognition on a parallel special-purpose architecture.

Let $n \geq 2$ be a fixed integer and A^+ the set of non-empty words over a finite alphabet A. Let f_i, $1 \leq i \leq n-1$, be a family of functions such that for any i, f_i is either the identity function Id ($\mathrm{Id}(w) = w$) *or the reverse function* Rev ($\mathrm{Rev}(w) = w^R$). *This paper is devoted to the real-time recognition of the language:*

$$L = \{ f_{n-1}(w) \ldots f_2(w) f_1(w) w;\ w \in A^+ \}.$$

We design a periodic one-dimensional systolic architecture to solve this problem, and show that the period of the array grows exponentially with n.

1. INTRODUCTION. — La reconnaissance des langages est un problème fondamental de l'informatique. Par exemple, la compilation qui consiste à traduire un langage évolué en langage machine, passe par une première phase au cours de laquelle le compilateur C vérifie si le programme source fourni par l'utilisateur est syntaxiquement correct, c'est-à-dire appartient au langage L(C) qui lui est associé.

Une manipulation performante des chaînes de caractères est à la base de telles opérations. Cependant, pour des problèmes de grande taille, les systèmes informatiques existants ne sont pas à même de traiter ce type de données de façon efficace, à cause du grand nombre d'accès à la mémoire centrale et de calculs à l'intérieur de boucles imbriquées que requiert ce genre de manipulations : pour améliorer la puissance et l'efficacité des opérations sur les chaînes de caractères, il faut recourir à des processeurs spécialisés pour une implantation sous forme de circuit intégré des algorithmes les plus utilisés. L'emploi d'architectures systoliques ([6], [8]) pour le traitement de chaînes de caractères s'est révélé être un outil adapté à la réalisation de tels processeurs spécialisés ([1], [3], [4], [5], [7], [9], [11]).

Dans cette Note, nous proposons des réseaux systoliques unidimensionnels pour la reconnaissance en temps réel des langages de la forme :

$$L = \{ f_{n-1}(w) f_{n-2}(w) \ldots f_1(w) w;\ w \in A^+ \},$$

où A est un alphabet fini, A^+ l'ensemble des mots non vides sur A, et pour tout i, $1 \leq i \leq n-1$, f_i est soit la fonction identité $f_i(w) = w$, soit la fonction miroir :

$$f_i(w) = f_i(w_1 w_2 \ldots w_n) = w_n w_{n-1} \ldots w_1 = w^R.$$

Ces réseaux généralisent ceux obtenus par Cole [2] pour la reconnaissance des palindromes ($n=2$, $f_1 = \mathrm{Rev}$) et des carrés ($n=2$, $f_1 = \mathrm{Id}$), et dont les auteurs ont proposé récemment des versions plus constructives en se basant sur le concept de convolution généralisée [8]. Notons que dans ce modèle de reconnaissance en temps réel introduit par Cole [2], on suppose que le réseau lit en entrée un nouveau symbole a_i tous les tops d'horloge; si le temps de cycle de la cellule de base est pris comme unité de temps, alors le réseau doit délivrer en sortie à tout instant $t = i + k$ (k indépendant de i), la réponse

0249-6321/85/03000363 $ 2.00

booléenne y_i correspondant au mot $a_1 a_2 \ldots a_i$, c'est-à-dire $y_i = 1$ si $a_1 a_2 \ldots a_i \in L$ et $y_i = 0$ sinon.

2. Position du problème. — Dans toute la suite, A est un alphabet fini, et A^+ est l'ensemble des mots non vides sur A; $n \geqq 2$ est un entier fixé, et $f_1, f_2, \ldots, f_{n-1}$ sont des fonctions données telles que $f_i \in \{\mathrm{Id}, \mathrm{Rev}\}$ pour tout i, où $\mathrm{Id}(w) = w$ et $\mathrm{Rev}(w) = w^R$ pour $w \in A^+$. Enfin, nous définissons le langage :

$$L = \{ f_{n-1}(w) \ldots f_2(w) f_1(w) w;\ w \in A^+ \}.$$

Étant donnée une suite $\{a_i;\ i \geqq 1\}$, nous devons calculer le signal booléen y_i correspondant à $a_1 a_2 \ldots a_i$, c'est-à-dire $y_i = 1$ si $i = pn\,(p \geqq 1)$ et $a_1 a_2 \ldots a_i \in L$, et $y_i = 0$ sinon. Soit $i = pn$; $y_i = y_{pn}$ peut s'écrire :

$$y_i = y_{i,1} \text{ et } y_{i,2} \text{ et} \ldots \text{et } y_{i,n-1},$$

où $y_{i,r}$, $1 \leqq r \leqq n-1$, vaut 1 si et seulement si :

$$f_r(a_{p(n-r-1)+1}\, a_{p(n-r-1)+2} \ldots a_{p(n-r)} \quad 1_{p(n-1)+1}\, a_{p(n-1)+2} \ldots a_{pn},$$

ce qui s'écrit encore :

$$y_{i,r} = y_{pn,r} = \pi_{0 \leqq k \leqq p-1} [a_{pn-k} = a_{p(n-r)-g(r,k,p)}],$$

où $[. = .]$ vaut 1 si l'égalité est vérifiée, et 0 sinon, avec :

$$g(r, k, p) = k \quad \text{si } f_r = \mathrm{Id} \qquad \text{et} \qquad g(r, k, p) = p - 1 - k \quad \text{si } f_r = \mathrm{Rev}.$$

3. Structure du réseau. — La contrainte de calcul en temps réel conduit à utiliser une approche de type « divide-and-conquer » [8] qui consiste à séparer le calcul de chaque $y_{i,r}$ en deux sous-produits :

$$y_{i,r,1} = y_{pn,r,1} = \pi_{0 \leqq 2k \leqq p-1} [a_{pn-2k} = a_{p(n-r)-g(r,2k,p)}],$$

$$y_{i,r,2} = y_{pn,r,2} = \pi_{1 \leqq 2k+1 \leqq p-1} [a_{pn-(2k+1)} = a_{p(n-r)-g(r,2k+1,p)}].$$

Pour le calcul des n sous-suites $y_{i,r,1}$, $0 \leqq r \leqq n-1$, on utilise un réseau linéaire R_1, où chaque cellule de calcul se compose de n couches numérotées de 0 à $n-1$; le flot des variables (y_i) circule sur cette même couche dans la direction opposée. Le calcul de la suite $(y_{i,r,1};\ i \geqq 1)$ s'obtient en éprouvant l'égalité des contenus des couches 0 et r dans chaque cellule : à tout instant $i = pn$, la couche r doit contenir les symboles a_i, $p(n-r-1)+1 \leqq i \leqq p(n-r)$, rangés dans l'ordre croissant ou décroissant selon que f_r est l'identité ou la fonction miroir. Les contraintes de génération des flots de variable dans les différentes couches amènent à introduire entre les cellules de calcul des cellules de retard, dont nous discutons ci-après la répartition. Le calcul de la suite $(y_{i,r,2};\ i \geqq 1)$ s'effectue sur un réseau R_2 analogue à R_1. La structure générale du réseau est schématisée sur la figure.

3.1. *Opérations de pré-traitement.* — Notons P_r la couche numéro r du réseau de pré-traitement P. Si $f_r = \mathrm{Id}$, P_r transforme la sous-suite $(a_i;\ i \geqq n-r)$ en insérant r délais après chaque bloc de $n-r$ données; en d'autres termes, toute variable a_i telle que :

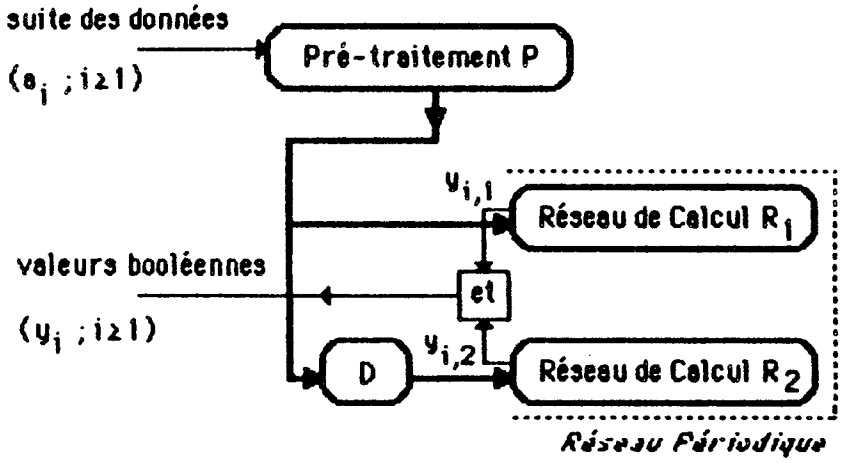
$$i = p(n-r)+q, \qquad 0 \leqq q < n-r, \quad p \geqq 1,$$

est en entrée de P_r au temps $t = i$, et est délivrée en sortie au temps :

$$t' = pn + q = i + pr = i + r.[i/(n-r)].$$

Nous utilisons un réseau linéaire de cellules élémentaires, composées seulement de registres et de compteurs de taille fixée, pour réaliser ce pré-traitement.

D'autre part, l'opération de D est la suivante : tout symbole de la couche 0 doit être retardé d'un temps de cycle, et tout symbole de la couche r, $r \geqq 1$ avec $f_r = \mathrm{Id}$, suivi par r



Structure générale du Réseau

TABLEAU I

Cellule.	0	1	...	k	...	$[(p-1)/2]$
couche 0.	a_{pn}	a_{pn-2}	...	a_{pn-2k}	...	$a_{pn-2[(p-1)/2]}$
couche r.	$a_{p(n-r)}$	$a_{p(n-r)-2}$	...	$a_{p(n-r)-2k}$	...	$a_{p(n-r)-2[(p-1)/2]}$
temps.	pn	$pn-1$	...	$pn-k$	...	$pn-[(p-1)/2]$

TABLEAU II

cellule.	0	1	...	k	...	$[(p-1)/2]$
couche 0.	a_{pn}	a_{pn-2}	...	a_{pn-2k}	...	$a_{pn-2[(p-1)/2]}$
couche r.	$a_{p(n-r)-p+1}$	$a_{p(n-r)-p+3}$	...	$a_{p(n-r)-p+1+2k}$	...	$a_{p(n-r)-r(p)}$
temps.	pn	$pn-1$	...	$pn-k$	...	$pn-[(p-1)/2]$

symboles blancs consécutifs, doit être retardé de $r+1$ temps de cycle. Cette opération élémentaire peut être réalisée par une même cellule pour chacune des couches.

Nous renvoyons à [10] pour une construction détaillée de P_r et de D.

3.2. *Réseau de calcul.* — Le réseau R_1 (et sa copie R_2) constituent le cœur de notre architecture spécialisée. Numérotons les cellules 0, 1, 2, ... à partir de la gauche. Sur la couche 0, a_i est présenté en entrée de la cellule 0 au temps i, et circule de gauche à droite à la vitesse 1. y_i, $i=pn$, est initialisé à 1 et rencontre a_{i-2k} dans la cellule k. Pour $r \geqq 1$, la structure de la couche r dépend de la valeur de f_r. Dans la suite, nous notons $\partial(x)$ et $\mu(x)$ les deux fonctions définies pour tout entier x par :

$$\partial(x)=2x \text{ si } x \text{ est pair et } \partial(x)=x \text{ si } x \text{ est impair,}$$

$$\mu(x)=x/2 \text{ si } x \text{ est pair et } \mu(x)=x \text{ si } x \text{ est impair.}$$

Cas 1 : $f_r=\mathrm{Id}$.

Nous devons organiser le flot des variables sur la couche r pour permettre les tests d'égalité indiqués au tableau 1. Une façon naturelle de procéder est d'insérer des cellules de retard entre les cellules de calcul. On vérifie que le nombre $R(r, k)$ de cellules de retard entre les cellules de calcul $k-1$ et k sur la couche r est donné par la formule :

$$R(r, k)=r^*([2k/(n-r)]-[2(k-1)/(n-r)]),$$

ce qui permet d'établir la :

PROPOSITION 1. — *Pour tout $r \geqq 1$ tel que $f_r=\mathrm{Id}$, la couche de R_1 numérotée r est une structure périodique unidimensionnelle, de période $T_r=\mu(n-r)$.*

Cas 2 : $f_r=\mathrm{Rev}$

Le flot des variables sur la couche r est ici généré par la suite des données $(a_i;\ i \geqq 1)$ de la couche 0, et circule de droite à gauche afin de permettre les tests d'égalité indiqués au tableau 2, où $e(p)=p-1 \bmod 2$.

Soit $k \geqq 0$. Lorsqu'un élément $a_{(2k+t)(n-r)-t+1}$, $1 \leqq t \leqq \partial(n-r)$, est présenté en entrée de la cellule k (couche 0), on doit en stocker une copie dans la couche r de cette cellule jusqu'au temps $(2k+t)n-k$. Pour ce faire, on adjoint à chaque cellule un ensemble de n registres et un réseau combinatoire de contrôle permettant d'accéder à chaque élément stocké en temps constant (nous utilisons ici notre connaissance *a priori* du nombre d'éléments devant être stockés). Quant à la répartition des cellules de retard sur la couche r, nous obtenons la :

PROPOSITION 2. — *Pour tout $r \geqq 1$ tel que $f_r=\mathrm{Rev}$, la couche de R_1 numérotée r est une structure périodique unidimensionnelle, de période $T_r=\mu(n-r-1)$.*

Cas général. — Nous pouvons énoncer notre principal résultat :

THÉORÈME. — *Pour tout $n \geqq 2$, pour tout alphabet fini A et pour tout ensemble de n fonctions $f_1, f_2, \ldots, f_{n-1}$ égales soit à l'identité* Id *soit à la fonction miroir* Rev, *le langage :*

$$L=\{f_{n-1}(w)\ldots f_2(w)\,f_1(w)\,w;\ w \in A^+\}$$

peut être reconnu en temps réel par un réseau systolique périodique unidimensionnel, dont la période croit exponentiellement avec n.

Plus précisément, soit :

$$J=\{r \mid 1 \leqq r \leqq n-1 \text{ et } f_r=\mathrm{Id}\} \quad \text{et} \quad R=\{r \mid 1 \leqq r \leqq n-1 \text{ et } f_r=\mathrm{Rev}\};$$

la période T du réseau est :

$$T=\mathrm{ppcm}\{\mathrm{ppcm}(\mu(n-r);\ r \in J),\ \mathrm{ppcm}(\mu(n-r-1);\ r \in R)\}.$$

Il est important de noter que la périodicité n'est relative qu'à la configuration des cellules de retard : les cellules de calcul sont toutes identiques, et le réseau peut être réalisé physiquement en dupliquant la structure d'une cellule de calcul élémentaire. Qui plus est, cette cellule de calcul peut elle-même s'obtenir en dupliquant deux réseaux combinatoires, à savoir le réseau de comparaison de symboles et le mécanisme de stockage dans les cellules; ceci permet d'envisager le dessin de cellules programmables, ce qui nous permettrait de traiter une large classe de problèmes de reconnaissance de langages.

Commentaire. — Pour certains éléments de la famille de langages étudiés ici, par exemple dans le cas particulier $L=\{w^n,\ w\in A^+\}$, il est possible d'exhiber des solutions plus simples. Mais la portée des réseaux présentés est en fait beaucoup plus générale; pour $a=a_1 a_2 \ldots a_{nk}$, notons $f_0=\mathrm{Id}$ et :

$$b=b_1 b_2 \ldots b_{nk}=f_{n-1}(a_1 \ldots a_k)\ f_{n-2}(a_{k+1} \ldots a_{2k}) \ldots f_0(a_{(n-1)k+1} \ldots a_{nk});$$

l'architecture systolique que nous présentons permet d'organiser le cheminement des données nécessaires au calcul en temps réel de toute suite de la forme :

$$y_i=\begin{cases} 0 & \text{si } i\not\equiv 0 \bmod n, \\ \pi_{1\leqq i\leqq k}\,\varphi_{k-j}(b_j, b_{j+k}, \ldots, b_{j+(n-1)k}) & \text{si } i\equiv nk, \end{cases}$$

où les φ_j, $1\leqq j\leqq k$, sont des fonctions arbitraires.

4. Conclusion. — Nous avons montré dans cette Note que certaines manipulations non triviales de chaînes de caractères peuvent être réalisées par des architectures intégrées spécialisées. Le réseau présenté ici constitue un pas vers un traitement efficace de telles chaînes dans un environnement informatique performant.

Remise le 3 décembre 1984, acceptée le 14 janvier 1985.

Références bibliographiques

[1] A. Apostolico et A. Negro, I.E.E.E., *Trans. Comp. C* 33, 4, 1984, p. 361-364.
[2] S. N. Cole, *I.E.E.E. Trans. Comp. C* 18, 4, 1969, p. 349-365.
[3] K. Culik II et coll., *R.A.I.R.O. Informatique théorique*, 18, n° 1, 1984, p. 53-69.
[4] T. W. Curry et coll., *Proc. VLSI* 83, F. Anceau éd., Elsevier Sc. Pub, 1983, p. 327-336.
[5] M. J. Foster et H. T. Kung, *I.E.E.E. Computer*, 13, n° 1, 1980, p. 26-40.
[6] H. T. Kung, *I.E.E.E. Comp r*, 15, n 1, 1982, p. 37-46.
[7] C. E. Leiserson et J. B. Saxe, Proc. 22 Symp. Foundations of Comp. Sc., *I.E.E.E.*, 1981, p. 23-36.
[8] Y. Robert et M. Tchuente, *R.A.I.R.O. Informatique Théorique* (à paraître).
[9] Y. Robert et M. Tchuente, *Comptes rendus*, 299, série I, 1984, p. 269-271.
[10] Y. Robert et M. Tchuente, R.R. 455, I.M.A.G., Grenoble, 1984.
[11] M. Steinby, *Theoretical Computer Science*, 22, 1983, p. 219-232.

C.N.R.S., Laboratoire T.I.M. 3-I.M.A.G., B.P. 68,
38402 *Saint-Martin-d'Hères Cedex.*

Techniques de l'Informatique, des Mathématiques, de la Microélectronique et de la Microscopie quantitative.
Unité associée au C.N.R.S. n° 397

BP 68
38402 Saint Martin d'Hères cedex
France
tél. (76) 51.46.00
secrétariat : poste 5257

Titre : **Reconnaissance en temps réel du langage $L = \{ f_{n-1}(w)f_{n-2}(w)...f_1(w)w \; ; \; w \in A^+ \}$ sur une architecture systolique**

Auteurs : Yves ROBERT et Maurice TCHUENTE
CNRS - Laboratoire TIM3/IMAG
BP 68 - 38402 Saint Martin d'Hères Cedex

Résumé : Soit $n \geq 2$ un entier fixé et A^+ l'ensemble des mots non vides sur un alphabet fini A. Soit f_i, $1 \leq i \leq n-1$, une famille de fonctions dont chacune est égale soit à l'identité soit à la fonctioin miroir. Nous présentons une architecture systolique périodique et unidimensionnelle pour la reconnaissance en temps réel du langage

$$L = \{ f_{n-1}(w)f_{n-2}(w)...f_1(w)w \; ; \; w \in A^+ \},$$

et nous montrons que la période du réseau croit exponentiellement avec n.

Mots-clés : reconnaissance de langages, architecture systolique, calcul en temps réel, parallélisme, palindrômes, carrés

REAL-TIME RECOGNITION OF THE LANGUAGE
$L = \{ f_{n-1}(w)...f_2(w)f_1(w)w; \; w \in A^+ \}$
ON A SPECIAL-PURPOSE ARCHITECTURE

RR n° 474

Yves ROBERT
Maurice TCHUENTE Novembre 1984

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Title :

Real-time Recognition of the Language $L = \{ f_{n-1}(w)...f_2(w)f_1(w)w \; ; \; w \in A^+ \}$ on a Special-Purpose Architecture

Authors :

Yves ROBERT and Maurice TCHUENTE
CNRS - Laboratoire TIM3 / IMAG
BP 68
38402 Saint Martin d'Hères Cedex
FRANCE

Abstract :

Let $n \geq 2$ be a fixed integer and A^+ the set of non-empty words over a finite alphabet A. Let f_i, $1 \leq i \leq n-1$, be a family of functions such that for any i, f_i is either the identity function Id ($Id(w) = w$) or the reverse function Rev ($Rev(w) = w^R$). This paper is devoted to the real-time recognition of the language

$$L = \{ f_{n-1}(w)...f_2(w)f_1(w)w \; ; \; w \in A^+ \}$$

We design a periodic one-dimensional systolic architecture to solve this problem, and show that the period of the array grows exponentially with n

Key-words :

language recognition, systolic array, real-time computation, palindrome recognizer, square acceptor

1. INTRODUCTION

The character string is currently a commonly encountered data type, and all indications regarding the development of fifth-generation computers are that its use will be growing even more. Strings are obviously central in words processing systems, which provide a variety of capabilitites for the manipulation of text; text strings can be quite large, and efficient algorithms play an important role in manipulating them. More generally, string manipulations play an important role in most computing environments and recent developments underscore the importance of human readable character strings as a natural data type [3].

However, existing computer architectures do not provide tools for being able to handle this data type efficiently. Larus [8] shows that any software algorithm which manipulates strings must perform considerable looping and memory accessing, with the associated time penalties of address decoding and inst. ction fetching [3]. To increase the power and efficiency of string manipulations, special-purpose systolic architectures can be proposed for a hardware implementation of some frequently used algorithms.

Systolic arrays, as defined by Kung and Leiserson [7], are a useful tool for designing special-purpose VLSI chips. A chip based on a systolic design consists of essentially a few types of very simple cells which are mesh-interconnected in a regular and modular way, and achieves high performance through extensive concurrent and pipeline use of the cells.

KUNG carefully explains in [6] how such architectures should result in cost-effective and high-performance special-purpose systems, and he gives three main reasons to justify the advantages of the systolic model:
- systolic systems are easy to implement because of their simple and regular design
- they are easy to reconfigure because of their modularity
- they permit multiple computations for each memory access, which speeds up the execution of compute-bound problems without increasing I/O requirements

The validity of the systolic model to string matching and related problems was first pointed out by Foster and Kung [4], who designed a systolic pattern matching VLSI chip. More recently, some systolic arrays [1] [5] [9] [10] [11] [12] have been proposed for language recognition problems.

In fact, the systolic model is strongly related to the notion of real-time computation by one-dimensional finite-state machines, which was first introduced by Cole [2]. In this model of computation, one assumes the existence of an infinite linearly connected collection of identical cells. The next state of cell $i \in Z$ is a function of its present state and the present state of its neighbours $i-1$, $i+1$. The array contains a particular cell which receives external inputs (from the host) and delivers external outputs (to the host). The constraint of real-time computation

means that a new symbol a_i is read every time-step, and for any $m \geq 1$, the accepting or rejecting signal corresponding to $a_1 a_2 \ldots a_m$ is delivered at time m+k, where k is a fixed integer independent of m. Cole [2] has illustrated the power of this method by exhibiting a palindrome recognizer and a square acceptor. Leiserson and Saxe [9] have reformulated Cole's palindrome recognizer and proposed a methodology for the design of cellular arrays (systolic arrays) for real-time computation; their approach leads generally to networks where any cell is active only every p-th step, $p \geq 2$, and Robert and Tchuente [11] have shown that this drawback can be eliminated by a divide-and-conquer approach.

In this paper, we are interested in the problem of designing a systolic array for the real-time recognition of the language

$$L = \{ f_{n-1}(w) \ldots f_2(w) f_1(w) w ;\ w \in A^+ \}$$

where $n \geq 2$ is a fixed integer, A^+ is the set of non-empty words over a finite alphabet A, and any function f_i, $1 \leq i \leq n$, is either the identity function Id ($f_i(w) = w$) or the reverse function Rev ($f_i(w) = w^R$). This problem can thus be viewed as a direct generalization of either the square acceptor (n=2 and f_1=Id) or the palindrome recognizer (n=2 and f_1=Rev). We point out that if all the functions f_i are equal to Id, the language we want to recognize is

$$L = \{ w^n ;\ w \in A^+ \}$$

As we shall see, the complexity of such an array grows exponentially with n. More precisely, the number of delay cells that must be inserted between the processing cells is a periodic sequence whose period grows exponentially with n (for instance, if all the f_i are equal to Id, this period is half the least common multiple of {2,3,...,n-1}). As a consequence, it is probable that such a solution cannot be obtained either by a direct generalization of the uniform array of Cole [2] or by applying the methodology of Leiserson and Saxe [9]. However, it is important to note that the periodicity applies only to the configuration of delay cells: thus the array may be fabricated easily by duplicating the structure of the basic processing cell.

2. GENERAL STRUCTURE OF THE ARRAY

2.1. Setting of the problem

Throughout the paper, A is a finite set of symbols called an alphabet, and A^+ denotes the set of all non-empty strings over A; $n \geq 2$ is a fixed integer, and $f_1, f_2, \ldots, f_{n-1}$ are given functions such that $f_i \in \{ Id, Rev \}$ for all i, where $Id(w) = w$ and $Rev(w) = w^R$ for $w \in A^+$. Finally, we define the language $L = \{ f_{n-1}(w)\ldots f_2(w) f_1(w) w ; w \in A^+ \}$.

Given an input sequence $(a_i ; i \geq 1)$, we have to compute the accepting or rejecting boolean symbol y_i corresponding to $a_1 a_2 \ldots a_i$, that is $y_i = 1$ if $i = pn$ $(p \geq 1)$ and $a_1 a_2 \ldots a_i \in L$, and $y_i = 0$ otherwise.

Let $i = pn$; $y_i = y_{pn}$ can be written $y_i = y_{i,1}$ and $y_{i,2}$ and ... and $y_{i,n-1}$, where for $1 \leq r \leq n-1$,

$$y_{i,r} = \begin{cases} 1 \text{ if } f_r = Id \text{ and} \\ \quad a_{p(n-r-1)+1} a_{p(n-r-1)+2} \ldots a_{p(n-r)} = a_{p(n-1)+1} a_{p(n-1)+2} \ldots a_{pn} \\ \text{or } f_r = Rev \text{ and} \\ \quad a_{p(n-r)} a_{p(n-r)-1} \ldots a_{p(n-r-1)+1} = a_{p(n-1)+1} a_{p(n-1)+2} \ldots a_{pn} \\ 0 \text{ otherwise} \end{cases}$$

For instance, assume that $n=5$ and $f_1 = f_3 = f_4 = Id$, $f_2 = Rev$; in order to compute $y_{20} = y_{20,1}$ and $y_{20,2}$ and $y_{20,3}$ and $y_{20,4}$, we have to compare $a_{17}a_{18}a_{19}a_{20}$ with $a_{13}a_{14}a_{15}a_{16}$, $a_{12}a_{11}a_{10}a_9$, $a_5a_6a_7a_8$, $a_1a_2a_3a_4$ for the computation of $y_{20,1}$, $y_{20,2}$, $y_{20,3}$ and $y_{20,4}$ respectively.

This can be formalized as follows: letting $i = pn$, $y_i = y_{pn}$ is equal to 1 if and only if any two elements whose indices belong to the same of the following p subsets $P_{i,k}$, $0 \leq k \leq p-1$, are identical:

$$P_{i,k} = \{ pn-k, p(n-1)-g_1(k,p), \ldots, p(n-r)-g_r(k,p), \ldots, p-g_{n-1}(k) \}$$

where $g_r(k,p) = k$ if $f_r = Id$ and $g_r(k,p) = p-1-k$ if $f_r = Rev$

For sake of convenience, we number the elements of any $P_{i,k}$ from 0 to n-1. For instance, assume that n= 5 and all the f_i are equal to Id. To compute y_{20}, we have to match the elements of the following sets:

$$P_{20,0} = \{ 20, 16, 12, 8, 4 \}$$
$$P_{20,1} = \{ 19, 15, 11, 7, 3 \}$$
$$P_{20,2} = \{ 18, 14, 10, 6, 2 \}$$
$$P_{20,3} = \{ 17, 13, 9, 5, 1 \}$$

Now, if f_1 = Rev, we get

$$P_{20,0} = (20, 13, 12, 8, 4)$$
$$P_{20,1} = (19, 14, 11, 7, 3)$$
$$P_{20,2} = (18, 15, 10, 6, 2)$$
$$P_{20,3} = (17, 16, 9, 5, 1)$$

and so on when reversing several components.

2.2. Using a linear processing array

Given an input sequence $(a_i; i \geq 1)$, we want to compute in real-time the output sequence $(y_i; i \geq 1)$ defined above. A natural approach to derive a solution consists of introducing a linear array of locally interconnected cells (see figure 1)

Each cell is composed of n layers numbered from top to bottom 0, 1, ..., n-1, and includes some latches to synchronize the flow on each layer. The sequence (a_i) moves from left to right at speed 1 on the first layer (numbered 0), and a_i is an input to cell 0 at time 2i-1. On the other hand, the sequence (y_i) moves at speed 1 in the opposite direction on the same layer. The computation of the sequence $(y_{i,r}; r \geq 1)$ is obtained by matching the contents of layer 0 and layer r. Every y_i is initialized to 1 if i is a multiple of p and to 0 otherwise; y_i meets a_{i-k} in cell k at time 2i-1-k. The flow on the r-th layer, $r \geq 1$, will be defined below; let us only assume that before entering the array the input sequence (a_i) has been duplicated n-1 times, so that a copy can be used as input for each layer $r \geq 1$ (in fact we will need such a copy only for the layers r such that f_r = Id).

The key-difficulty is to manage the movement of data in these layers so that, whenever i=pn, y_i meets the other elements of $P_{i,k}$ together with a_{i-k} in cell k in order to perform the partial accumulation

$$y_i := y_i \text{ and (the elements of } P_{i,k} \text{ are all equal to } a_{i-k})$$

Thus we need to be able to store at time 2pn-1-k in the r-th layer ($r \geq 1$) of cell k the r-th element of $P_{pn,k}$, for the partial accumulation

$$y_{i,r} := y_{i,r} \text{ and } (a_{p(n-r)-g_r(k,p)} = a_{i-k})$$

In order to do that, we could feed after some preprocessing (depending whether f_r is equal to Id or Rev) one of the copies of (a_i) in each layer $r \geq 1$. To derive a solution (whenever it exists), it would remain to determine in detail the flow of data on each layer separately, including among others the eventual insertion of additional delay cells between the processing cells.

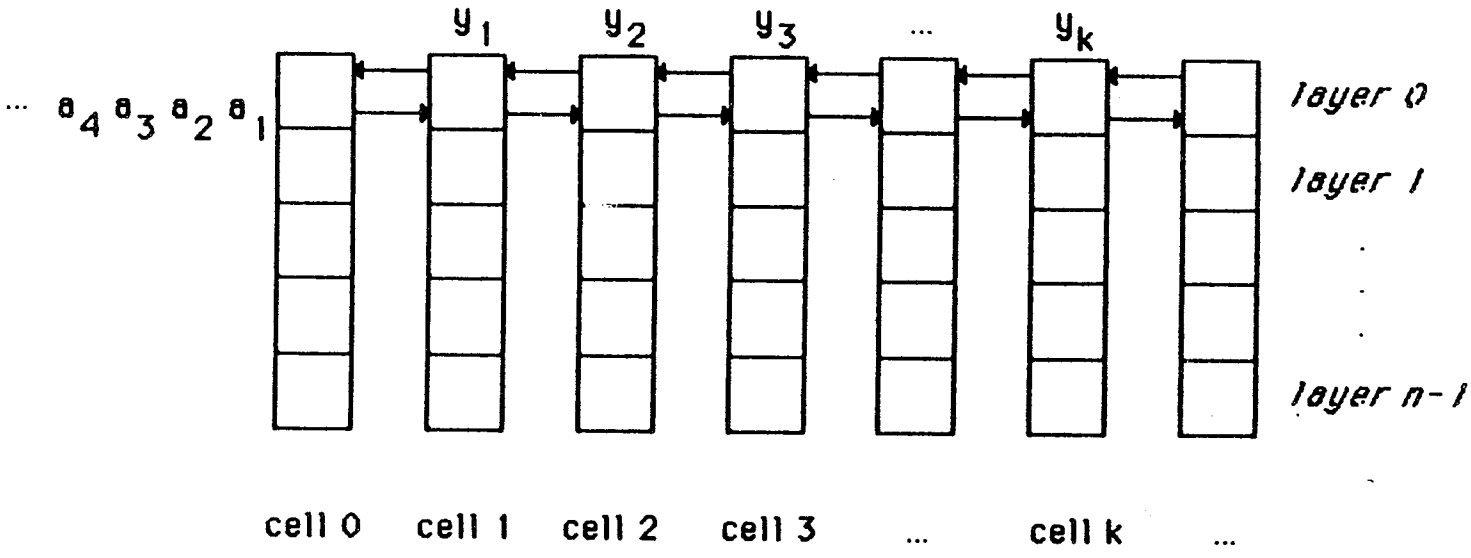
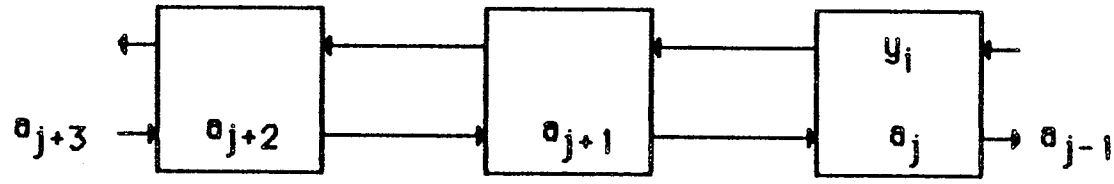
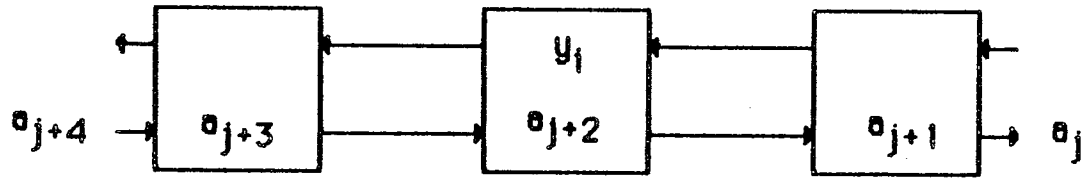

y1
y2
y3
...
yk
... a4 a3 a2 a1
layer 0
layer 1
layer n-1
cell 0
cell 1
cell 2
cell 3
...
cell k
...

Figure 1

time t



time t+1



y_i does not meet x_{j+1}

Figure 2

However, we do not discuss further such a solution since it is not of maximal efficiency: if the time-performance of the array is measured by the average number e of outputs y_i delivered every cycle time, then we have here e=1/2, since a new y_i is ouput every second time step. It is easily checked that a variable y_i cannot meet two consecutive elements a_j and a_{j+1} if they are separated by a single time step (see figure 2). As a consequence, two consecutive elements in both flows on layer 0 are separated by two cycle times, leading to the value e=1/2.

2.3 Structure of the solution

In this paper we design an array of maximal efficiency e=1 to solve our language recognition problem. We use a divide-and-conquer approach which consist of using two linear arrays PA_1 and PA_2 (PA stands for Processing Array) similar to the one of figure 1, but where two consecutive variables in any flow on layer 0 are separated by a single unit of time. Instead of computing $y_i = y_{pn}$ by accumulating on the same array the result of the matchings of the elements of $P_{i,k}$ for $0 \le k \le p-1$, we compute

- the result u_i of the matchings of the elements of $P_{i,2k}$ $(2k \le p-1)$ on the first array PA_1
- the result v_i of the matchings of the elements of $P_{i,2k+1}$ $(2k+1 \le p-1)$ on the second array PA_2

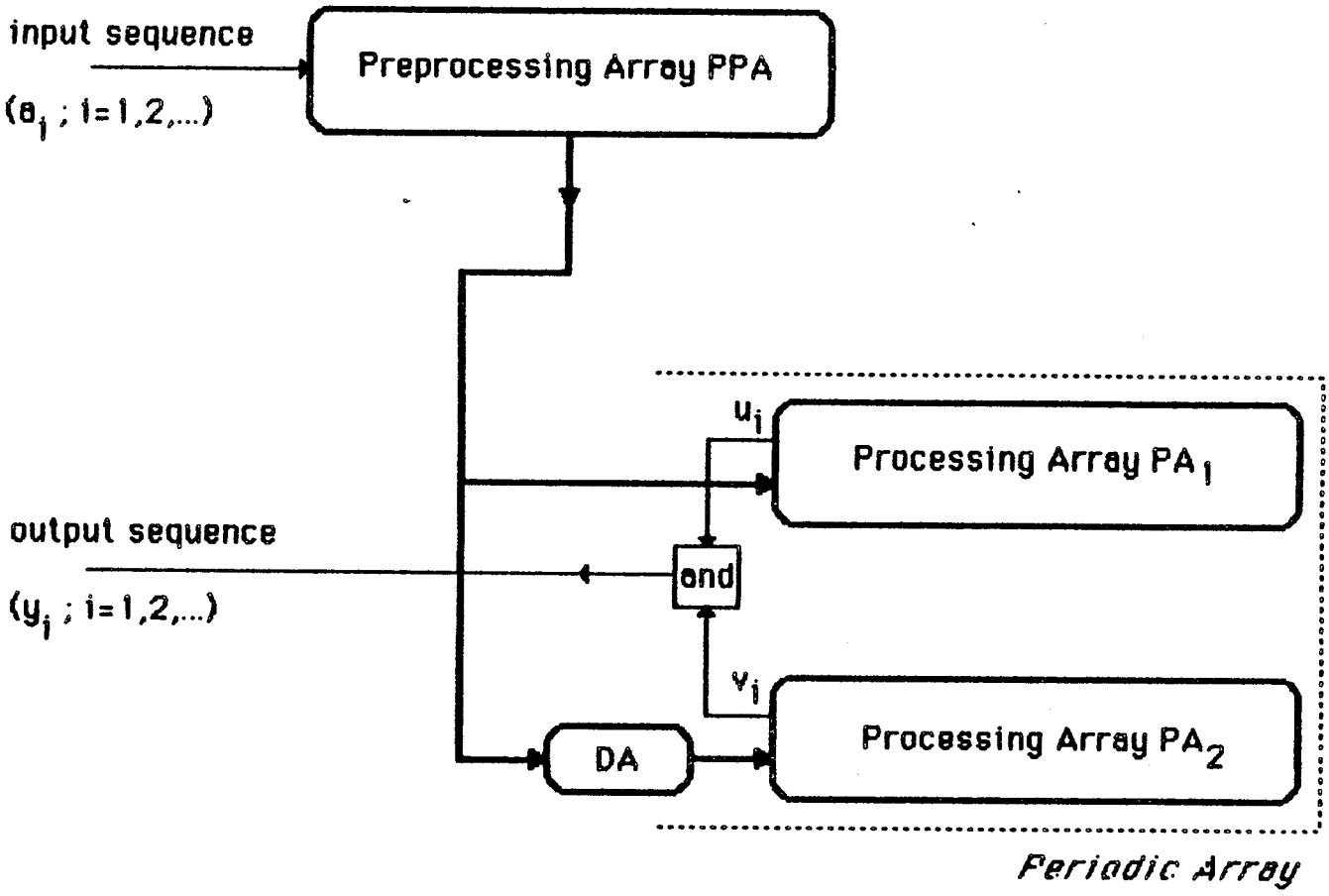
and we obtain y_i as output of a special cell which performs the computation

$$y_i := u_i \text{ and } v_i$$

Now u_i meets only every second variable a_{i-k} on PA_1, since any two consecutive elements on lay(0 are separated by a single unit of time, but this does not matter since v_i accumulates the results of the matchings that u_i has missed.

If we reconsider our previous example with n=5, the results of the matchings of the elements of $P_{20,0}$, $P_{20,2}$ and $P_{20,4}$ will be accumulated by u_{20} on PA_1 whereas those concerning $P_{20,1}$ and $P_{20,3}$ will be computed on PA_2. The computation of y_{20} is achieved at time 21 in the special cell.

As depicted in figure 3, the whole array is composed of several distinct sub-blocks. First, a preprocessing array PPA receives the input sequence $(a_i;\ i \ge 1)$ and generates the input streams for the r-th layers $(r \ge 1)$ of the two processing sub-arrays PA_1 and PA_2: if f_r = Id then PPA generates some extra delays between the elements of (a_i), as described in



Global structure of the array

Figure 3

the following paragraph.; if f_r = Rev then PPA does nothing, the processing arrays PA_1 and PA_2 generate the flow of data needed by the r-th layer.

Then the output of PPA is duplicated, one copy is directly fed into PA_1 whereas the other copy is delayed by a simple array DA before entering PA_2. The function of DA is only to delay by one cycle time the variables on each layer r≥1 at the output of PPA before they enter PA_2. That is to say, PA_2 operates exactly as PA_1, but with a delay of one cycle-time. We will detail in section 4 the structure of PA_1, which is the main block of the array. We briefly describe the operation of PPA and DA in the next section, and refer to the appendix for their construction.

3. The operation of the preprocessing arrays

3.1. Operation of PPA

The operation of PPA is depicted figure 4 in the case where all the functions f_i are equal to Id. In this case, the input sequence (a_i) is duplicated n-1 times, one copy being fed in each sub-array PPA_r, $r \geq 1$ (PPA_r denotes the r-th layer of PPA). To handle the general case we simply suppress the sub-arrays PPA_r for the r such that f_r = Rev (and duplicate the initial sequence only T times, where T is the number of f_i equal to Id).

If f_r = Id, the operation of PPA_r, which generates the flow of the r-th layer of the input stream of PA_1 (and PA_2 after the delay introduced by DA), consists of tranforming the subsequence $(a_i ; i \geq n-r)$ by inserting r delays after any block of n-r consecutive inputs. In other words, any variable a_i such that

$$i = p(n-r) + q, \quad 0 \leq q < n-r, \ p \geq 1$$

is input at time t = i and output at time

$$t' = pn + q = i + pr = i + r\lfloor i/(n-r) \rfloor$$

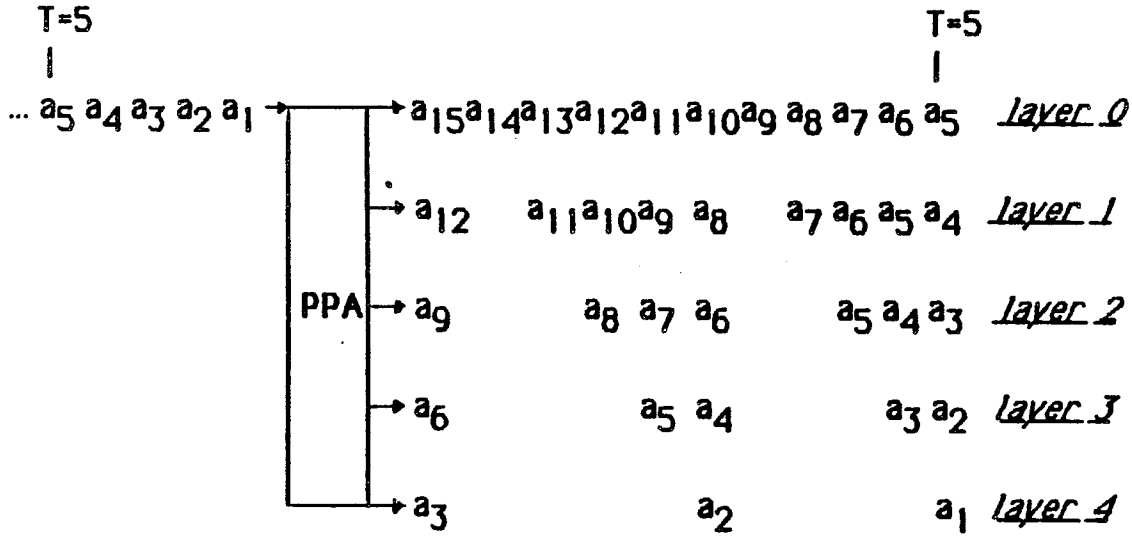
The reason to perform this transformation will become clear in the next section.

A linearly connected array of elementary cells (involving only registers and counters of fixed size) can be used for this preprocessing. We refer to the appendix for its construction, but we point out that the size of the elementary cells of PPA_r does not depend on p (the size of the input sequence), but only on n (the size of the problem).

3.2. Operation of DA

We also need a preprocessing array DA which performs the transformation illustrated in figure 5 (suppre ; the layers r correponding to the functions f_r = Rev).

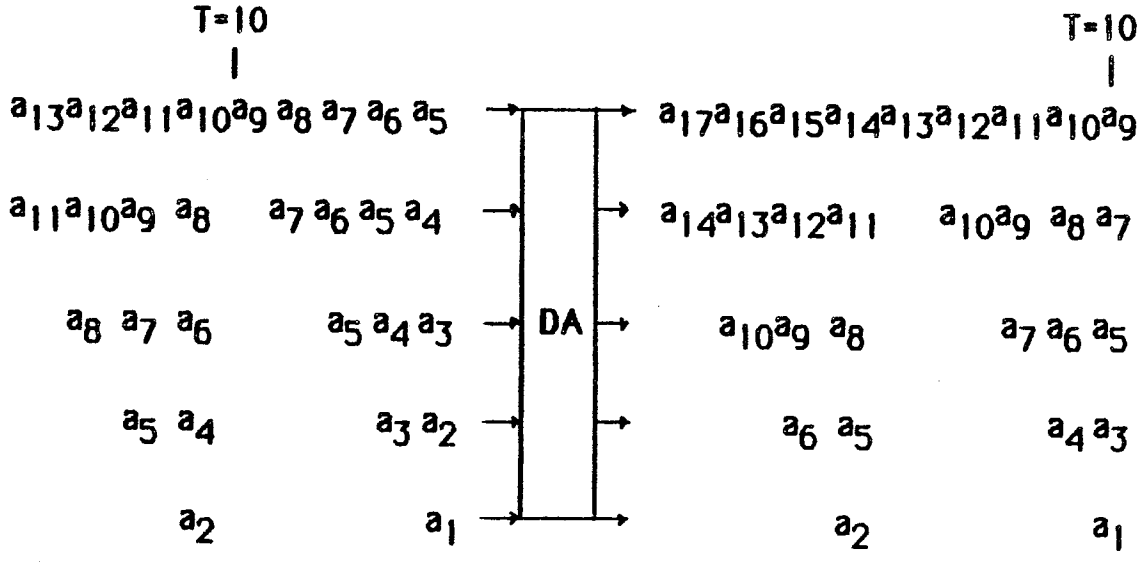
In other words, any symbol a in layer 0 must be delayed by one step and any symbol a in layer r, $r \geq 1$, which is followed by r blank symbols, must be delayed by r steps. This transformation can be performed on each layer by the same simple cell depicted in figure 6.



Operation of the preprocessing array PPA
with n=5 and $f_1 = f_2 = f_3 = f_4 = Id$:
the input stream $(a_i;\ i \geq 1)$ is duplicated four times

* the original flow is input to layer 0 without being delayed
* for $1 \leq r \leq 4$, we insert in the subsequence $(a_i;\ i \geq n-r)$ r delays after any block of n-r consecutive inputs

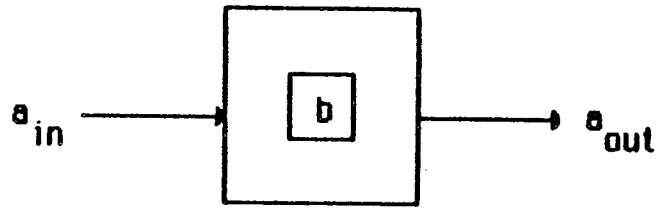
Figure 4



Operation of the preprocessing array DA
with n=5 and $f_1 = f_2 = f_3 = f_4 = id$:

* any symbol in layer 0 is delayed by one time-step
* for $1 \le r \le 4$, any symbol followed by r blank symbols is delayed by r+1 time-steps

Figure 5




```
if a in := nil then a out := nil
else
begin a out := b
      b := a in
end
```

(nil means no symbol)

Figure 6

4. THE PROCESSING ARRAY

We derive now the structure of PA_f. We recall that the flow on layer 0 is defined as follows:

* a_i is an input to cell 0 at time i and and moves from left to right at speed 1. Thus it is an input of cell k, $k \geq 1$, at step i+k
* u_i is initialized to 1 and moves from right to left at speed 1. It meets a_i in cell 0 at time i to perform its last accumulation. Thus it meets a_{i-2k} in cell k at time i-k.

In what follows, r is fixed, with $1 \leq r \leq n-1$. We detail here the design of the r-th layer of the processing array PA_f.

4.1 Case 1 : f_r = Id

This case generalizes the square acceptor, which corresponds to n=2 and f_1 = Id. We organize the flow on layer r in order to permit the matchings of the following table:

	cell 0	cell 1	...	cell k	...	cell $\lfloor (p-1)/2 \rfloor$
layer 0	a_{pn}	a_{pn-2}	...	a_{pn-2k}	...	$a_{pn-2\lfloor (p-1)/2 \rfloor}$
layer r	$a_{p(n-r)}$	$a_{p(n-r)-2}$	...	$a_{p(n-r)-2k}$	...	$a_{p(n-r)-2\lfloor (p-1)/2 \rfloor}$
time	pn	pn-1	...	pn-k	...	pn-$\lfloor (p-1)/2 \rfloor$

A natural way to construct such a flow is to insert in layer r some additional delay cells between the processsing cells.

The first constraint we have to meet is that $a_{p(n-r)}$ and a_{pn} must be input simultaneously, to layer 0 and layer r respectively. It is the reason why the preprocessing array PPA_r generates on layer r the element $a_{p(n-r)+j}$ at time pn+j for j = 0, 1, ..., n-r-1. This leads to the input format (nil means no symbol)

layer 0	$a_{(p+1)n}$	...	a_{pn+n-r}	$a_{pn+n-r-1}$	...	a_{pn+1}	a_{pn}
layer r	$a_{(p+1)(n-r)}$	...	nil	$a_{p(n-r)+n-r-1}$	...	$a_{p(n-r)+1}$	$a_{p(n-r)}$
time	(p+1)n	...	pn+n-r	pn+r-1	...	pn+1	pn

Now, we have to match in the k-th cell at time T = pn-k the element a_{pn-2k} (layer 0) with $a_{p(n-r)-2k}$ (layer r) for any p,k with $1 \le p$ and $k \le \lfloor (p-1)/2 \rfloor$:

* on layer 0, a_{pn-2k} is an input to cell 0 at time pn-2k and is an input of cell k at time (pn-2k) + k = T.

* on layer r, we have to determine at what time $a_{p(n-r)-2k}$ is an input to cell 0; we can express

$$i = p(n-r) - 2k$$

as

$$i = h(n-r) + q, \text{ with } 0 \le q < n-r$$

where $h = p - \lceil 2k/(n-r) \rceil$.

Thus, from the construction of PPA_r, $a_{p(n-r)-2k}$ is input to layer r of PA_1 at time

$$\begin{aligned} i + hr &= p(n-r) - 2k + (p - \lceil 2k/(n-r) \rceil)r \\ &= pn - 2k - r\lceil 2k/(n-r) \rceil \end{aligned}$$

If follows that between cell 0 and cell k, for all $k \ge 1$, the layer numbered r must contain $r\lceil 2k/(n-r) \rceil$ more delay cells than the layer numbered 0. Since we have assumed there are no delay cells between the processing cells on layer 0, we have to insert $r\lceil 2k/(n-r) \rceil$ delay cells between cell 0 and cell k on layer r, for all $k \ge 1$. As a consequence, the number f(r,k) of delay cells between cell k-1 and cell k on layer r is equal to

$$f(r,k) = r * (\lceil 2k/(n-r) \rceil - \lceil (2(k-1)/(n-r) \rceil$$

for all $k \ge 1$, with $1 \le r \le n-1$.

We can now derive the structure of the r-th layer of the processing array PA_1:

Proposition 1 : For any $r \ge 1$ such that $f_r = Id$, the layer of PA_1 numbered r is a periodic one-dimensional structure , of period

$$T_r = \begin{cases} (n-r) / 2 & \text{if } n-r \text{ is even} \\ n-r & \text{if } n-r \text{ is odd} \end{cases}$$

Proof : We compute the period T of f(r,k) with respect to k, that is the smallest non-zero integer such that

$$f(r,k+T) = f(r,k) \text{ for any } k \ge 1$$

and show that T has the value T_r stated in proposition 1:

<u>case 1</u> : n-r is even

We write n-r = 2q:

$\lceil 2(k+T)/2q \rceil - \lceil (2(k+T)-2)/2q \rceil = \lceil 2k/2q \rceil - \lceil (2k-2)/2q \rceil$ for any $k \ge 1$.

Taking k=1, it follows that $\lceil (T+1)/q \rceil - \lceil T/q \rceil = \lceil 1/q \rceil - 0 = 1$

which implies that T is a multiple of q = (n-r)/2

case 2 : n-r is odd

We write n-r = 2q + 1:

$\lceil 2(k+T)/(2q+1) \rceil - \lceil (2(k+T)-2)/(2q+1) \rceil = \lceil 2k/(2q+1) \rceil - \lceil (2k-2)/(2q+1) \rceil$

for any $k \geq 1$.

subcase 2a : q=0

Then T is an arbitrary non-zero integer; we choose T = 1.

subcase 2b : $q \geq 1$

Taking k=1, it follows that

$\lceil (2T+2)/(2q+1) \rceil - \lceil 2T/(2q+1) \rceil = \lceil 2/(2q+1) \rceil = 1$,

which implies that 2T or 2T+1 is a multiple of 2q+1 (*)

Taking k=q+1, it follows that $\lceil (2T+1)/(2q+1) \rceil - \lceil (2T-1)/(2q+1) \rceil = 1$

which implies that 2T-1 or 2T is a multiple of 2q+1 (**)

(*) and (**) together show that 2T is a multiple of 2q+1.

Hence, since 2q+1 is odd, T is a multiple of 2q+1 = n-r.

We point out that the periodicity is relative only to the disposition of delay cells: the processing cells are all identical, which is a very important feature for any physical realization.

Proposition 1 enables us to derive the complete structure of PA_1 when all the functions f_r, $1 \leq r \leq n-1$, are equal to Id:

Theorem 1 : For any $n \geq 2$ and for any finite alphabet A, the language

$$L = \{ w^n ; w \in A^+ \}$$

can be recognized in real-time by a periodic one-dimensional systolic array whose period is half the least common multiple of (2,3,...,n-1).

Proof : The preprocessing array is of period 1. In the processing array, the layer numbered 0 is of period 1 and any layer numbered r , $1 \leq r \leq n-1$, is of period T_r according to proposition 1, where

$$T_r = \begin{cases} (n-r) / 2 & \text{if } n-r \text{ is even} \\ n-r & \text{if } n-r \text{ is odd} \end{cases}$$

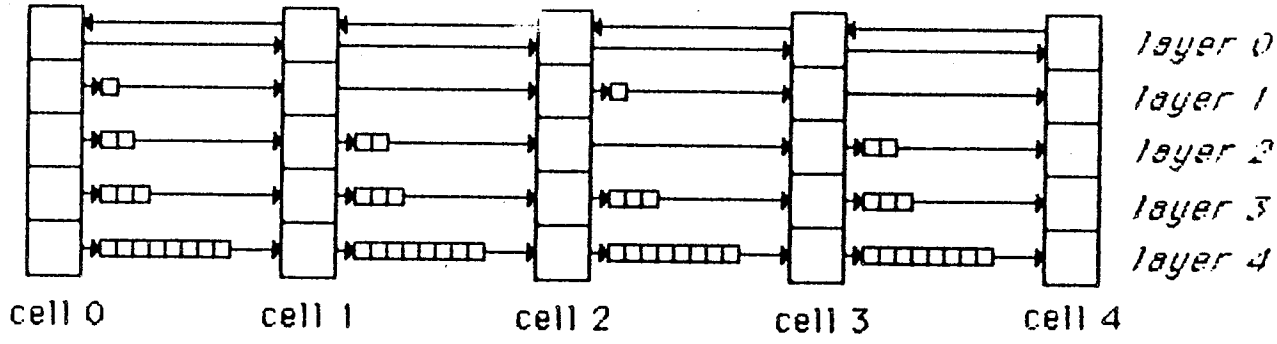
As a consequence, the period T of the array is the least commmon multiple of the periods T_r, $1 \leq r \leq n-1$, that is

$$T = \text{lcm}\{1,2,3,.....n-1\} / 2$$

which is the value stated above.

The structure of PA_1 in the case n = 5 and f_r = Id for $1 \leq r \leq 4$, is illustrated in figure 7.

A direct consequence of theorem 1 is that the complexity of the array grows exponentially with n in this case. To prove this, it is sufficient to see that the period T is greater than the product of the prime numbers greater than n/2 and smaller than n-1, whose number is



Structure of the Processing Array PA$_1$
for $n = 5$ and $f_r = Id$, $1 \leq r \leq 4$

* the repartition of delay cells on layer 1 is 1,0,1,0,1,0,...
* the repartition of delay cells on layer 2 is 2,2,0,2,2,0,...
* the repartition of delay cells on layer 3 is 3,3,3,3,3,3,...
* the repartition of delay cells on layer 4 is 8,8,8,8,8,8,...

The period of the array is 6

Figure 7

asymptotically equal to $n/(2\log n)$. Hence:

$$T \geq (n/2)^{n/(2\log n)}$$

4.2 Case 2 : f_r = Rev

This case generalizes the palindrome recognizer, which corresponds to n=2 and f_1 = Rev. We organize the flow on layer r in such a way that the input sequence will be generated by the flow $(a_i;\ i\geq 1)$ of layer 0, and once generated will move from right to left: this will permit to match for $i = pn$, $p\geq 1$, the following elements

	cell 0	cell 1	...	cell k	...	cell $\lfloor (p-1)/2 \rfloor$
layer 0	a_{pn}	a_{pn-2}	...	a_{pn-2k}	...	$a_{pn-2\lfloor (p-1)/2 \rfloor}$
layer r	$a_{p(n-r)-p+1}$	$a_{p(n-r)-p+3}$	...	$a_{p(n-r)-p+1+2k}$	...	$a_{p(n-r)-e(p)}$
time	pn	pn-1	...	pn-k	...	$pn-\lfloor (p-1)/2 \rfloor$

where $e(p) = p-1 \bmod 2$.

To derive the structure of layer r, let us discuss the operation of cell k, $k\geq 0$. The first accepting or rejecting symbol u_i that performs an accumulation in cell k is $u_{(2k+1)n}$. For $p\geq 2k+1$, cell k operates for the computation of u_{pn}; since a_{pn} enters the array on layer 0 at time pn, this operation takes place at time pn-k., and corresponds to the matching of a_{pn-2k} of layer 0 with the symbol $a_{p(n-r)-p+1+2k}$ of layer r. Expressing p as $p = 2k + t$, $t\geq 1$, we have the following situation:

(*) { cell k operates every n-th time-step, beginning at time $(2k+1)n-k$;
for the computation of $u_{(2k+t)n}$, $t\geq 1$, it matches $a_{(2k+t)n-2k}$ of layer 0 with $a_{(2k+t)(n-r)-t+1}$ of layer r at time $(2k+t)n-k$.

These results are completed by the following proposition, which we state after some definitions

- we say that an element of layer r is used in a cell k if it is matched with an element of layer 0 for the computation of some u_{pn}, $p\geq 1$. When an element is used in cell k, this cell is said to be activated
- we let $\partial(x)$ and $\mu(x)$ be the two functions defined for any integer x by

$\partial(x) = 2x$ if x is even and $\partial(x) = x$ if x is odd

$\mu(x) = x/2$ if x is even and $\mu(x) = x$ if x is odd

Proposition 2 : Let $r \geq 1$ such that f_r = Rev, and $k \geq 0$. The elements of layer r which are used in cell k are those of index $(2k+t)(n-r)-t+1$ for $t \geq 1$; moreover:
- for $1 \leq t \leq \partial(n-r)$, $a_{(2k+t)(n-r)-t+1}$ is used for the first time in cell k
- for $t > \partial(n-r)$, $a_{(2k+t)(n-r)-t+1}$ is used in cell $k+\mu(n-r-1)$ before being used in cell k, and it is not used between these two cells.
Finally, the number of delay cells which have to be inserted on layer r between cell k and cell $k+\mu(n-r-1)$ is 2n if n-r is even and n otherwise.

Proof : Assume that $a_{(2k+t)(n-r)-t+1}$ is used in cell k+s for some $s \geq 1$. Then from (*), its index $(2k+t)(n-r)-t+1$ can be also expressed as

$$(2k+t)(n-r)-t+1 = (2(k+s)+t')(n-r)-t'+1$$

for some $t' \geq 1$.
Thus: (**) $(n-r-1)(t-t') = 2s(n-r)$
Since n-r-1 and n-r are relatively prime, this implies that n-r-1 divides 2s: there exists an integer q such that $2s = q(n-r-1)$, and reporting in (**) we get $t-t' = q(n-r)$.

The smallest possible value for s is obtained when q is minimum: we can choose q=1 if n-r is odd (then s=(n-r-1)/2) and q=2 if n-r is even (then s=n-r-1). Hence with our notation, the miminum value of s is $\mu(n-r-1)$. Let us consider two different cases according to the parity of n-r:

<u>i) n-r is even</u>

the minimum value of q is q=2. Since $t' \geq 1$, we have $t \geq 2(n-r)+1$. Thus if $t \leq 2(n-r)$, then $a_{(2k+t)(n-r)-t+1}$ is not used in any cell h such that h>k. On the other hand, for $t \geq 2(n-r)+1$, let T_1 be the time at which it is used in cell k and T_2 be the time at which it is used in cell $k+(n-r-1)$. From (*) we know that $T_1 = (2k + t)n - k$

Expressing $(2k+t)(n-r)-t+1$ as $(2(k+s)+t')(n-r)-t'+1$ where s=n-r-1 and $t-t'=2(n-r)$ leads in a similar way to $T_2 = (2(k+s) + t')n - (k+s)$, that is

$$T_2 = (2(k+(n-r-1)) + t-2(n-r))n - (k+(n-r-1))$$

Computing the difference $T_1 - T_2$, we get

$$T_1 - T_2 = 2n + (n-r-1)$$

This proves that $a_{(2k+t)(n-r)-t+1}$ is used in cell k <u>after</u> being used in cell $k+(n-r-1)$. Morover (see the first part of the proof), this shows that the elements $a_{(2k+t)(n-r)-t+1}$ with $1 \leq t \leq 2(n-r)$ are used for the first time in cell k, and then they move from right to left on layer r: they are used for the second time in cell k-(n-r-1), for the third time in cell k-2(n-r-1), and so on until they reach the leftmost cell of the array. Finally, we can deduce from the computation of $T_1 - T_2$ that for any $k \geq 0$, there must be 2n additional delay cells inserted between cell k and cell k + (n-r-1)

ii) n-r is odd

the minimum value of q is q=1. Since t'≥1, we have t≥(n-r)+1. Thus if t≤(n-r), then $a_{(2k+t)(n-r)-t+1}$ is not used in any cell h susc that h>k. On the other hand, if t≥(n-r)+1, let T_1 be the time at which it is used in cell k and T_2 be the time at which it is used in cell k + (n-r-1)/2. Clearly,

$$T_1 = (2k+t)n - k$$

Expressing (2k+t)(n-r)-t+1 as (2(k+s)+t')(n-r)-t'+1 where s=(n-r-1)/2 and t-t'=n-r leads to $T_2 = (2(k+s) + t').n - (k+s)$, that is

$$T_2 = (2(k + (n-r-1)/2) + t-(n-r)).n - (k + (n-r-1)/2)$$

Computing the difference $T_1 - T_2$, we get

$$T_1 - T_2 = n + (n-r-1)/2$$

This proves that $a_{(2k+t)(n-r)-t+1}$ takes n + (n-r-1)/2 time-steps to go from cell k + (n-r-1)/2 to cell k.; it follows that n delay cells must be added between cell k and cell k + (n-r-1)/2 for any k≥0.

For 1≤t≤∂(n-r), $a_{(2k+t)(n-r)-t+1}$ is an input to layer 0 of cell k at time (2k+t)(n-r)-t+1+k; according to proposition 2, we need to design a mecanism permitting to store a copy of it in the r-th layer of cell k until it is used for $u_{(2k+t)n}$, that is at time (2k+t)n-k. Later $a_{(2k+t)(n-r)-t+1}$ must be sent back to the left, in order to be used by the cells k-μ(n-r-1), k-2μ(n-r-1), ..., until it reaches the leftmost cell. Note that the original element goes on rightwards on layer 0.

Storing ∂(n-r) elements in cell k is not difficult. First we see that cell k stores its first element at time (2k+1)(n-r), and then stores a new one every (n-r-1)-th step. A simple way to realize this is to deliver at time n-r as input to cell 0 (layer r) a special boolean signal Sto which moves from left to right, goes through 2(n-r)-1 delay cells before reaching a new processing cell and hence reaches cell k at time

$$n-r + k + (2(n-r)-1)k = (2k+1)(n-r)$$

Sto is the signal which activates the storage in any cell of layer r. Now each processing cell k includes two counters:

- a counter C_{store} initialized to 0, which is activated when the cell receives the storing signal Sto. C_{store} is incremented every time-step modulo (n-r-1): thus a new element can be stored when $C_{store} = 0$.
- a counter $C_{\#(el.store)}$ initialized to 0 which is incremented when a new element is stored, that is when $C_{store} = 0$. This counter $C_{\#(el.store)}$ counts the number of the elements stored in the cell, and its maximum value is ∂(n-r). When $C_{\#(el.store)} = ∂(n-r)$, the storage mechanism is stopped.

Similarly, we design an algorithm to send back to the left of the processing cell k the elements which have been stored in it. For $1 \le t \le \partial(n-r)$, the element $a_{(2k+t)(n-r)}$ must be sent back at time $(2k+t)n-k$, after being used for $u_{(2k+t)n}$. Just as before, a simple way to realize this is to deliver at time n as input to cell 0 (layer r) a special boolean signal Send which moves from left to right, goes through 2n-2 delay cells before reaching a new processing cell and hence reaches cell k at time

$$n + k + (2n-2)k = (2k+1)n-k$$

Send is the signal which activates the sending to the left of the stored elements in any cell of layer r. Thus each processing cell k includes two other counters:

- a counter C_{send} initialized to 0, which is activated when the signal Send reaches the cell and which is incremented every time-step modulo n: a new element can be sent to the left when $C_{send} = 0$.
- a counter $C_{\#(el.send)}$ initialized to 0, which is incremented when a new element is sent to the left, that is when $C_{send} = 0$. This counter $C_{\#(el.send)}$ counts the number of the elements sent to the left, and its maximum value $\partial(n-r)$: then it stops the activation processus of cell k, whose operation in layer r after this moment only consists of matching the elements coming from the right and transmitting them to the left.

Finally, to be able to store $\partial(n-r)$ elements in layer r of cell k for all $k \ge 0$, we adjoin to each cell a set of $\partial(n-r)$ registers and some combinatorial circuitry permitting to access any stored element in constant time. Extending each cell with a systolic stack [6] whose operation of insertion/deletion can be nicely done in constant time could be another solution, but we can use here our knowledge "a priori" of the number of elements to be stored, and design a faster solution.

It is important to note that all the elements of the subsequence $(a_i; i \ge n-r)$ are stored in at most one cell:

- if n-r is even, then 2(n-r) and n-r-1 are relatively prime, and for any integer $i \ge n-r$, we can write in a unique way

$$i - 1 = 2(n-r)k + (n-r-1)t, \text{ with } 1 \le t \le 2(n-r)$$

Thus $i = (2k+t)(n-r)-t+1$ will be stored in cell k.

- if n-r is odd, n-r and (n-r-1)/2 are relatively prime and for any odd integer $i=2j+1 \ge n-r$, we can write in a unique way

$$j = (n-r)k + ((n-r-1)/2)t, \text{ with } 1 \le t \le n-r$$

Thus $i = (2k+t)(n-r)-t+1$ will be stored in cell k.

The storage and "send-to-the left" mechanisms described above permit to generate and control the flow on layer r. Our last construction to complete the design of layer r is to determine the repartition of the additional delay cells that must be inserted between the processing cells. We already know that for any $k \ge 0$:

- if n-r is even, 2n delay cells must be inserted between cell k and cell k+(n-r-1)
- if n-r is odd, n delay cells must be inserted between cell k and cell k+(n-r-1)/2.

The constraint we have to meet for this repartition is the following: if an element a_i moves from right to left and is an input to a cell k at time T in which it is not used, then cell k must not be activated by performing another matching at time T.

So let us consider an element a_i, $i = (2k+t)(n-r)-t+1$ with $1 \le t \le \partial(n-r)$, which is sent leftwards by cell k to the cells k-m, $1 \le m \le k$. We have shown that:
- any cell k-m is activated at time-steps T such that T = -(k-m) modulo n
- a_i is sent leftwards at time (2k+t)n-k
- a_i is active in cells k-m such that m = 0 modulo $\mu(n-r-1)$.

As a consequence, the number $\Delta(k,m)$ of delay cells inserted between cell k and cell k-m must satisfy

$$(***) \quad \begin{cases} -k + m + \Delta(k,1) \equiv -(k-m) \text{ modulo } n \text{ for } 0 \le m < \mu(n-r-1) \\ \Delta(k,k-\mu(n-r-1)) = \begin{cases} 2n \text{ if } n-r \text{ is even} \\ n \text{ otherwise} \end{cases} \end{cases}$$

Moreover, the repartition of delay cells is periodic, of period $\mu(n-r-1)$.

We can proceed as follows: we let all the $\Delta(k,1)$ be equal to 1 except one every $\mu(n-r-1)$ cells; for instance:
- if $k \not\equiv 0$ modulo $\mu(n-r-1)$, insert one delay cell between cell k and cell k+1
- otherwise, that is if k=0, n-r-1, 2(n-r-1), 3(n-r-1), ..., insert between cell k and cell k+1

either $2n - (n-r-1) + 1$ delay cells if n-r is even
or $n - (n-r-1)/2 + 1$ delay cells if n-r is odd

Clearly, the number of delay cells inserted between cell k and cell $k+\mu(n-r-1)$ for all $k \ge \mu(n-r-1)$ is the desired value 2n or n, depending wether n-r is even or odd.

The case r = n-1 is worth noting: then $\partial(n-r) = 1$, thus for any $k \ge 0$, cell k stores only one element, namely a_{2k+1}; since $\mu(n-r-1) = 0$, this element a_{2k+1} stays indefinitely in cell k. Hence there is no flow on layer n-1.

Since the storage and "send-to-the-left" mechanisms are of period 1, we have proven the following proposition:

Proposition 3 : For any $r \geq 1$ such that $f_r = Rev$, the layer of PA_1 numbered r is a periodic one-dimensional structure , of period

$$T_r = \begin{cases} n-r-1 & \text{if } n-r \text{ is even} \\ (n-r-1)/2 & \text{if } n-r \text{ is odd} \end{cases}$$

The structure of PA_1 in the case $n = 5$ and $f_r = Rev$ for $1 \leq r \leq 4$, is illustrated in figure 8. We omit the storage and "send-to-the-left" mechanisms to emphasize the repartition of delay cells.

We are now ready to prove our main result on the structure of the array in the general case:

Theorem 2 : For any $n \geq 2$, for any finite alphabet A and for any set of $n-1$ functions $f_1, f_2, \ldots, f_{n-1}$ equal either to the identity function Id or to the reverse function Rev, the language

$$L = \{ f_{n-1}(w) \ldots f_2(w) f_1(w) w \; ; \; w \in A^+ \}$$

can be recognized in real-time by a periodic one-dimensional systolic array whose period grows exponentially with n.

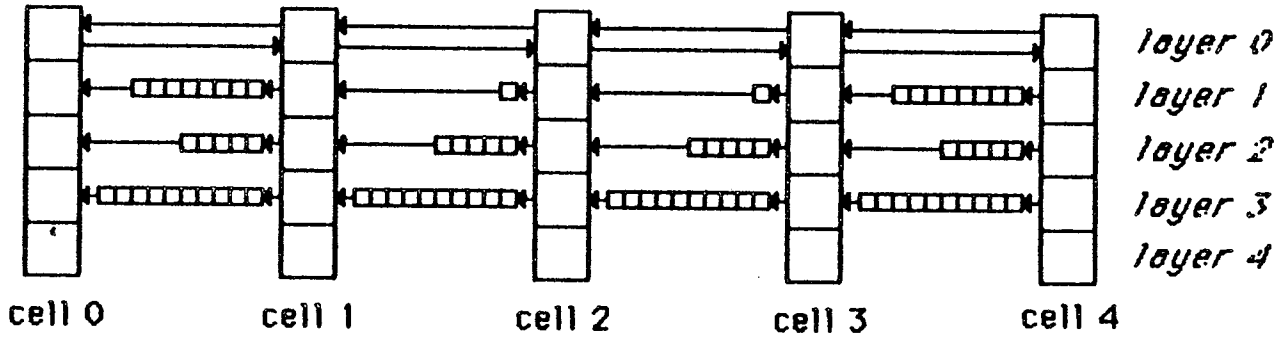
More precisely, let $I = \{ r \; / \; 1 \leq r \leq n-1 \text{ and } f_r = Id \}$ and $R = \{ r \; / \; 1 \leq r \leq n-1 \text{ and } f_r = Rev \}$; the period T of the array is

$$T = lcm \{ lcm (\mu(n-r) ; r \in I) , lcm(\mu(n-r-1) ; r \in R) \} \qquad (\Omega)$$

where $\mu(x)$ is equal to $x/2$ if x is even and to x otherwise.

Proof : The value T of the period of the array stated in relation (Ω) follows directly from the propositions 1 and 3, since the preprocessing array as well as the strorage and "send-to-the-left" mechanisms are of period 1. Similar arguments to those developed after the proof of theorem 1 show that T grows exponentially with n.

It is important to note that the periodicity applies only to the configuration of delay cells: the processing cells are all the same, and the array may be fabricated easily by duplicating the structure of the basic processing cell. Also, we see that the design of this basic processing cell depends heavily on the nature of the function f_i, $1 \leq i \leq n-1$: designing layers r such that $f_r = Rev$ requires much more hardware (due to the necessity of storage) than designing layers r such that $f_r = Id$ (for which only a matching network is needed).



Structure of the Processing Array PA_1
for n = 5 and f_r = Rev, $1 \le r \le 4$

* the repartition of delay cells on layer 1 is 8,1,1,8,1,1,...
* the repartition of delay cells on layer 2 is 5,5,5,5,5,5,...
* the repartition of delay cells on layer 3 is 10,10,10,10,...
* there is no flow on layer 4

The period of the array is 3

Figure 8

5. CONCLUSION

Systolic arrays for string manipulations or language recognition problems have been considered by several authors: see for instance [1] [3] [4] [5] [10] [11] [12] [13]. The network presented here for the real-time recognition of the language

$$L = \{ f_{n-1}(w)...f_2(w)f_1(w)w \; ; \; w \in A^+ \}$$

$n \geq 2$, A finite alphabet, f_r = Id or Rev

is a periodic systolic structure. The periodicity of the array may of course be eliminated by coalescing a sufficient number of elementary cells into a single cell, but this is not realistic because the size of the resulting cell will grow exponentially with n (theorem 2). We have already pointed out that the periodicity applies only to the configuration of delay cells, which would facilitate any physical realization : it is sufficient to duplicate the structure of the basic processing cell. Moreover, the basic cell itself can be fabricated by duplicating the structure of two elementary combinatorial networks, namely the matching and storage mechanisms: by this way, we would obtain programmable cells, since the stucture of each layer r could be chosen according to the value of f_r, and this would enable us to handle a large class of language recognition problems.

We have shown in this paper that some non trivial string manipulations can be casted into a systolic VLSI architecture, in a way that appears both neat and suitable for realization. The array presented here can be viewed as a new step towards efficiently handling character strings in a powerful computer environment.

References

[1] A. Apostolico, A. Negro, Systolic algorithms for string manipulations, IEEE Trans. on Computers C33, 4 (1984), p 361-364

[2] S.N. Cole, Real-time computation by n-dimensional iterative arrays of finite-state machines, IEEE Trans. on Computers C18, 4 (1969), p 349-365

[3] K. Culik II, Systolic tree acceptors, RAIRO Theoretical Computer Science 18, 1 (1984), p 53-69

[4] T.W. Curry, A. Mukhopadhyay, Realization of efficient non-numeric operations through VLSI, in Proc. of VLSI 83, F. Anceau and E.J. Aas eds, Elsevier Science Publishers B.V. (North Holland), 1983, p 327-336

[5] M.J. Foster, H.T. Kung, The design of special-purpose VLSI chips, IEEE Computer 13, 1 (1980), p 26-40

[6] L.J. Guibas, F.M. Liang, Systolic stacks, queues and counters, in Proc. of the 1982 Conference of Advanced Research in VLSI, M.I.T. Boston

[7] H.T. Kung, Why systolic architectures, IEEE Computer 15, 1 (1982), p 37-46

[8] H.T. Kung, C.E. Leiserson, Systolic arrays for (VLSI), in Proc. of the Symposium on Sparse Matrices Computations, I.S. Duff and G.W. Stewart eds, Knoxville, Tenn., 1978, p 256-282

[9] J. Larus, A comparison of microcode, assembly code and high-level languages on the VAX11 and RISC1, Computer Architecture News 10, 5 (1982)

[10] C.E. Leiserson, J.B. Saxe, Optimizing synchronous systems, in Proc. of the 22-th Annual Symposium on Foundations of Computer Science, IEEE, October 1981, p 23-36

[11] Y. Robert, M. Tchuente, Réseaux systoliques pour des problèmes de mots, RAIRO Theoretical Computer Science, to appear 1985

[12] Y. Robert, M. Tchuente, A systolic array for the longest common subsequence problem, in Proc. of the International Workshop on High-Level Computer Architecture, Los Angeles, June 1984

[13] M. Steinby, Systolic trees and systolic language recognition by tree automata, Theoretical Computer Science 22 (1983), p 219-232

APPENDIX

We detail here the design of the preprocessing array PPA when all the functions f_r, $1 \le r \le n-1$, are equal to Id. We determine the structure of each sub-array PPA_r, which generates the input stream of the r-th layer of the processing array PA_1 (and PA_2 after the delay introduced by DA): any variable a_i such that $i = p(n-r) + q$, with $0 \le q < n-r$ and $p \ge 1$, is an input to PPA_r at time i and is output from PPA_r at time $i + pr$.

A.1. Structure of PPA_r for $r \ge 2$

The design of PPA_r, $r \ge 2$, can be done as follows: we prepare an (infinite) linearly connected array of elementary cells. Each cell j is composed of two parts (see figure 9a):
- a lower part consisting of a counter C and a register L(j) (the register L(1) of the first cell is the input register of the array PPA_r)
- an upper part composed of r-1 registers U(j,1), U(j,2), ..., U(j,r-1)

Every counter is initialized to 0 and can be incremented up to the maximal value Max = n-r, except for the first cell for which Max = 2(n-r)-1.

In the lower part of the array, the data flow from left to right, according to the following rules : assume that at step t, the symbol a is a left input of the lower register of cell j :

if C < Max then C is incremented (C := C+1), and a is transmitted to its rightmost upper register
else a is transmitted to the lower register of cell j+1.

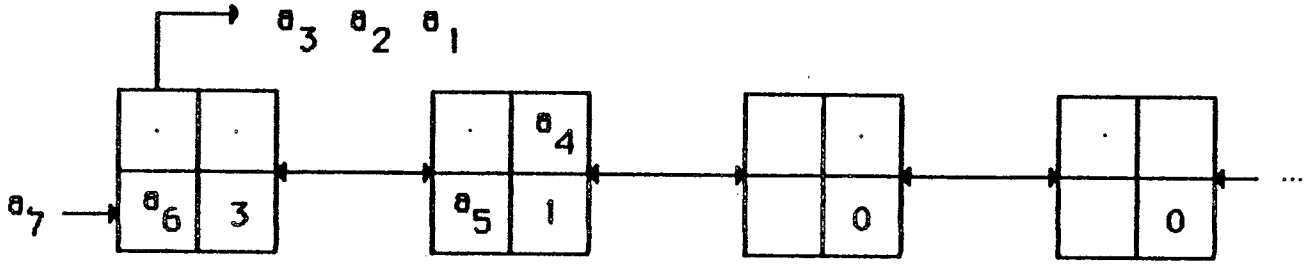
Once a variable has reached the upper part of PPA_r, it moves from right to left at speed 1 until it is output from the leftmost cell.

Let us now show that PPA_r works correctly. If $i = p(n-r) + q$, where $p \ge 1$ and $0 \le q < n-r$, then as a_i is processed by PPA_r, it goes through the registers L(1), L(2), ..., L(p) and then through the registers U(p,r-1), U(p,r-2), ..., U(p,1), U(p-1, r-1), U(p-1, r-2), ..., U(p-1,1), ..., U(1,r-1), U(1,1). Thus a_i is delayed by $p + p(r-1)$ time-steps, and is output from PPa_r at time $i + pr$, which is the value we stated above.

A.2. Structure of PPA_1

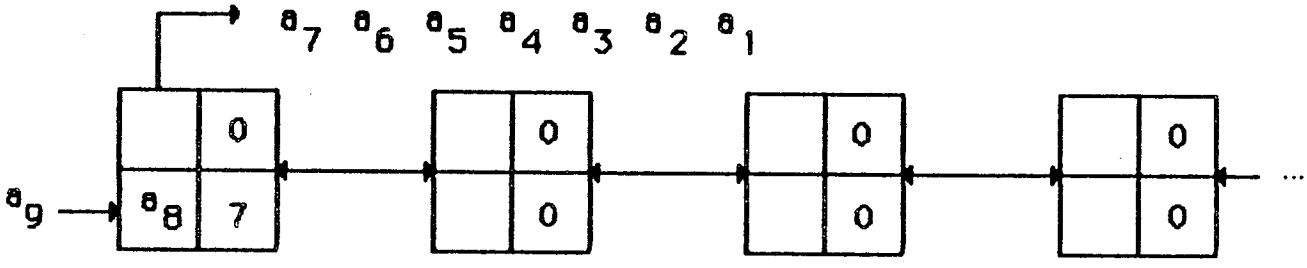
The design of PPA_1 is a little bit different. A basic cell of PPA_1 consists of two parts, each involving a counter and a register (see figure 9b). Every counter can be incremented up to a maximal value Max = n-1, except for the lower counter of the leftmost cell for which Max = 2n-3. The operation of an elementary cell is the following : assume that a symbol a is a left input to cell j at step t :

Struture of the Preprocessing Array



Structure of PPA 3 for n=5 at time T=6

Figure 9a



Structure of PPA 1 for n=5 at time T=8

Figure 9b

if $C_{down} < Max$ then C_{down} is incremented ($C_{down} := C_{down} + 1$) and a is transmitted to the upper register U(j-1) of its left neighbor j-1 (or output from the array if j = 1)
else
if $C_{up} < Max$ then C_{up} is incremented ($C_{up} := C_{up} + 1$) and a is transmitted to its upper register U(j)
else a is transmitted to the lower register L(j+1) of its right neighbour j+1

Just as before, once a variable has reached the upper part of the array, it moves from right to left at speed 1 until it is output from the leftmost cell.

To show that PPA_1 works correctly, assume that $i = p(n-r) + q$, where $p \geq 1$ and $0 \leq q < n-r$. As a_i is processed by PPA_1, it goes through the registers L(1), L(2), ..., L(s) and U(1), U(2),..., U(s) if $p = 2s$, or through the registers L(1), L(2), ..., L(s), L(s+1), and U(1), U(2), ..., U(s) if $p = 2s+1$. Thus in both cases a_i is delayed by p time-steps and is output from PPA_1 at time $i+p$, the desired value.

Techniques de l'Informatique, des Mathématiques, de la Microélectronique et de la Microscopie quantitative.
Unité associée au C.N.R.S. n° 397

BP 68
38402 Saint Martin d'Hères cedex
France
tél. (76) 51.46.00
secrétariat : poste 5257

UN RESEAU SYSTOLIQUE ORTHOGONAL POUR LE PROBLEME DU CHEMIN ALGEBRIQUE

Yves ROBERT & Denis TRYSTRAM

Résumé : Le concept de chemin algébrique permet la formulation abstraite et unifiée d'une vaste classe de problèmes, englobant l'inversion d'une matrice, le calcul des plus courtes distances dans un graphe pondéré, et la détermination de la fermeture réflexive et transitive d'une relation binaire. Nous présentons un réseau systolique orthogonal de $(n+1)^2$ processeurs élémentaires qui peut résoudre toute instance de ce problème du chemin algébrique en seulement 5n-2 unités de temps, contre 7n-2 pour le réseau hexagonal de Rote [10] qui comporte le même nombre de processeurs.

UN RESEAU SYSTOLIQUE ORTHOGONAL POUR
LE PROBLEME DU CHEMIN ALGEBRIQUE

RR n° 553 - Juillet 1985
Yves ROBERT & Denis TRYSTRAM

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

AN ORTHOGONAL SYSTOLIC ARRAY FOR THE ALGEBRAIC PATH PROBLEM

Yves ROBERT & Denis TRYSTRAM

Abstract - Zusammenfassung

An Orthogonal Systolic Array for the Algebraic Path Problem. This paper is devoted to the design of an orthogonal systolic array of $(n+1)^2$ elementary processors which can solve any instance of the Algebraic Path Problem within only 5n-2 time steps, to be compared with the 7n-2 time steps of the hexagonal systolic array of Rote [9].

AMS subject classifications: 68A05, (05C35, 05C38, 16A78, 65F05, 68E10).

CR categories and subjet descriptors: C.1.2 [processor architectures]: multiple data stream architectures (multiprocessors) - systolic arrays; G.1.0 [numerical analysis]: general - parallel algorithms; G.1.3 [numerical analysis]: numerical linear algebra - matrix inversion; G.2.2 [discrete mathematics]: graph theory - path problems; B.6.1 [logic design]: design styles - cellular arrays; B.7.1 [integrated circuits]: types and design styles - algorithms implemented in harware; VLSI (very large scale integration).

General terms: algorithms, design, performance

Ein Orthogonal Systolisch Feld für das Algebraische Wegproblem. Es wird dargestellt, wie ein orthogonallisches systolischen Feld, das algebraische Wegproblem in linearer Zeit von (5n-2) ausführen kann. Dieses Feld ist mit dem (7n-2) hexagonallischen Feld von Rote [9] verglichen.

1. INTRODUCTION

In a recent paper [9], Rote introduces the Gauss-Jordan elimination algorithm for the Algebraic Path Problem (APP for short), a general framework which unifies several algorithms arising from various fields of computer science, such as the determination of the inverse of a real matrix, the shortest distances in a weighted graph, and the transitive and reflexive closure of a binary relation. Rote [9] presents an hexagonally connected systolic array of $(n+1)^2$ processors which can solve any instance of size n of the APP in 7n-2 time steps, each time being the time necessary to achieve a multiply-and-add in the underlying algebra.

The main result of this paper is the design of an orthogonally connected array of $(n+1)^2$ processors which solves the same problem within only 5n-2 time steps.

Section 2 briefly reviews the APP algorithm, as introduced by Rote [9] The new systolic array is presented in section 3, and its performance are compared with Rote's array and others from the literature in section 4.

Using massive parallelism and pipelining, the systolic array concept allows a system implementor to design extremely efficient machines for specific computations. The reader not familiar with systolic arrays is referred to Kung [5]. All definitions from algebra or graph theory are consistent with Rote [9].

2. THE ALGEBRAIC PATH PROBLEM

2.1. The algorithm

The Algebraic Path Problem is defined as follows [10]: given a weighted graph G = (V,E,w) where V is a finite vertex set, E an arc set, an application w : E -> H with weights from a semiring (H,(+),(x)) with zero (0) and unity (1), find for all pairs of vertices (i,j) the quantities

$$d_{ij} = \bigoplus_{p \in M_{ij}} w(p),$$

where M_{ij} denotes the set of all paths from i to j.

With the weighted graph (V,E,w) we associate as in [9] the n by n weigth matrix $A = (a_{ij})$, where $a_{ij} = w(i,j)$ if $(i,j) \in E$ and $a_{ij} = 0$ otherwise. We denote $M_{ij}^{(k)}$ the set of all paths from i to j which contain only vertices x with $1 \leq x \leq k$ as intermediate vertices. In practice,

$$a_{ij}^{(k)} = \bigoplus_{p \in M_{ij}^{(k)}} w(p),$$

egals the successive values of a_{ij} which we want to compute, starting from the intial value $a_{ij}^{(0)} = a_{ij}$ up to $a_{ij}^{(n)} = d_{ij}$.

Rote's algorithm [9] for solving any particular instance of the APP using only the semiring operations (+), (x) and * is detailed below:

```
{ phase 1 : initialization of the network and
computation of the a_ij^(j) if i>j and a_ij^(i-1) if i≤j }
for i := 1 to n
for j := 1 to n
begin
    for k := 1 to min(i,j)-1
            a_ij^(k) := a_ij^(k-1) (+) a_ik^(k) (x) a_kj^(k-1) ;
    if i=j then a_ij^(i) := (a_ii^(i-1))* ;
    if i>j then a_ij^(i) := a_ij^(j-1) (x) a_jj^(j) ;
end ;

{ phase 2 : computation of the a_ij^(k)
such that k=j if i≤j and k=i-1 if i>j }
for i := 1 to n
for j := 1 to n
begin
    if i<j then a_ij^(i) := a_ii^(i) (x) a_ij^(i-1) ;
    for k := min(i,j)+1 to max(i,j)-1
            a_ij^(k) := a_ij^(k-1) (+) a_ik^(k) (x) a_kj^(k-1) ;
    if i<j then a_ij^(j) := a_ij^(j-1) (x) a_jj^(j) ;
end ;

{ phase 3: computation of the a_ij^(n) }
for i := 1 to n
for j := 1 to n
begin
    if i>j then a_ij^(i) := a_ii^(i) (x) a_ij^(i-1) ;
    for k := max(i,j)+1 to n
            a_ij^(k) := a_ij^(k-1) (+) a_ik^(k) (x) a_kj^(k-1) ;
end ;
```

Applications of the APP are obtained by specializing the operations (+), (x), and * in the appropriate semirings. Rote [9] describes three of them in detail: the determination of the inverse of a real matrix, the shortest distances in a weighted graph, and the transitive and reflexive closure of a binary relation. We shall concentrate on the determination of the inverse of a real matrix, and we introduce some notations for the general APP problem which are derived from its interpretation in numerical linear algebra.

2.2 Matrix inversion

We have a very important application of the APP algorithm by letting (+) and (x) be the usual operations on real numbers, and defining the *-operation as follows:

if $c \neq 1$ **then** $c := 1/(1-c)$

The APP algorithm will then compute the inverse $(I-A)^{-1}$ of the matrix $(I-A)$. In fact, the APP algorithm corresponds in this case to the well-known Gauss-Jordan elimination algorithm. Even more, we have a natural interpretation of the three phases of the algorithm: let us denote

- $L = (l_{ij})$ the strictly lower triangular matrix with $l_{ij} = a_{ij}^{(j)}$ if $i>j$
- $U = (u_{ij})$ the strictly upper triangular matrix with $u_{ij} = a_{ij}^{(i-1)}$ if $i<j$
- $D = (d_{ij})$ the diagonal matrix with $d_{ij} = a_{ii}^{(i)}$ if $i=j$

We have the Gaussian decomposition

$$I - A = (I - L) \cdot (D^{-1} - U) = (I - L) \cdot D^{-1} \cdot (I - DU)$$

so that

$$(I - A)^{-1} = (D^{-1} - U)^{-1} \cdot (I - L)^{-1} = (I - DU)^{-1} \cdot D \cdot (I - L)^{-1}$$

Phase 1 of the algorithm computes L, U and D, whereas phase 2 computes the two inverses

$$(I - L)^{-1} = \begin{cases} a_{ij}^{(i-1)} & \text{if } i>j \\ 1 & \text{if } i=j \\ 0 & \text{if } i<j \end{cases} \qquad \text{(i)}$$

and

$$(D^{-1} - U)^{-1} = (I - DU)^{-1} \cdot D = \begin{cases} a_{ij}^{(j)} & \text{if } i \leq j \\ 0 & \text{if } i>j \end{cases} \qquad \text{(ii)}$$

Finally, phase 3 computes the product

$$(D^{-1} - U)^{-1} \cdot (I - L)^{-1} = (I - A)^{-1}$$

Rote [9] shows how very simple cosmetic transformations permit to compute A^{-1} rather than $(I-A)^{-1}$. Also, these transformations could be made in a straightforward manner for the systolic array that we propose. An other way to proceed is to input directly the matrix $(I-A)$ to the array to get A^{-1} without modifications in the algorithm. We shall adopt this

solution in our design , since it has the advantage to respect the general formalism of the APP.

Notation : letting $\mathfrak{L} = I-L$ and $\mathfrak{V} = D - U$, we denote $\mathfrak{L}^{-1}$ and $\mathfrak{W}$ the matrices respectively defined by the relations (i) and (ii). We adopt the notation $\mathfrak{L}^{-1}$ for the general instance of the APP, although it is only significant for the particular case of matrix inversion (where $(I-A)^{-1} = \mathfrak{W}.\mathfrak{L}^{-1}$)

3. THE SYSTOLIC ARRAY

3.1. Formal description

We use a bidimensional array of orthogonally connected processors (see figure 1). The array is composed of n rows, each row k including n+1 processors numbered from left to right $P_{k1}, ..., P_{k,n+1}$.

The matrix A, followed by I, the identity matrix of order n, is fed into the array row by row. More specifically, row k of the n by 2n matrix (A,I) is input to processor P_{1k}, one new element each time-step, beginning at time t=k. This input format is depicted in the figure 1.

The operation of each processor is detailed in the figure 2. There are three types of processors :

- circle processors are devoted to the computation of the * operations.
- square processors have first their current register initialized ; then they act as multiply-and-add cells (when (+) and (x) are the standard operations on real numbers, they become classical IPS cells [5] [7] [8]).
- double square processors actually operate as square processors, with the exception that their current register is not initialized in the same way.

We conceptually divide the array into two triangular sub-blocks, namely the left block and the right block, as illustrated in the figure 3.

3.2. Operation of the array

To explain the mapping of the APP algorithm on the systolic architecture, we describe sequentially the execution of each of the three phases of the algorithm, although they would be pipelined in an actual execution. This sequential description is used in the figure 4, which offers a synthetic view of the whole execution of the APP algorithm.

Phase 1

We may consider here that the matrix A is input to the left triangular block of the array: we do not deal neither with the identity matrix that follows A, nor with the right part of the array. Moreover, the output of the

left block after phase 1 will be taken to be the input of the right block for phase 2.

In phase 1, the array acts essentially as the triangularization array of Ahmed et al [1]. At step 1, processor P_{11} computes $a_{11}^{(1)}$ and then operates as a one-step delay cell. At step 2, P_{12} has his current register initialized with $a_{21}^{(1)}$ and then operates as a multiplicaty-and-add cell: indeed at step 3, it updates $a_{22}^{(0)}$ into

$$a_{22}^{(1)} := a_{22}^{(0)} (+) a_{21}^{(1)} (x) a_{12}^{(0)},$$

and so on with $a_{23}, a_{24}, \ldots$

The following table shows for the operation of the first row of processors during phase 1 (we choose n=4 in this example):

	P_{11}	P_{12}	P_{13}	P_{14}
time-step 1	$a_{11}^{(1)}$	-	-	-
2	-	$a_{21}^{(1)}$	-	-
3	-	$a_{22}^{(1)}$	$a_{31}^{(1)}$	-
4	-	$a_{23}^{(1)}$	$a_{32}^{(1)}$	$a_{41}^{(1)}$
5	-	$a_{24}^{(1)}$	$a_{33}^{(1)}$	$a_{42}^{(1)}$

Figure 4 shows the content of the registers of the left block processors as well as the output of the array after the execution of the first phase.

Phase 2

The left block processors are now initialized and operate as multiply-and-add cells, transforming the identity matrix into $\mathfrak{L}^{-1}$. Indeed, the array realizes a linear transformation [2]. Since it transforms A into $\mathfrak{V}$, it applies the transformation $\mathfrak{L}^{-1}$ to any matrix. If the identity is fed in the initialized array, the output is $\mathfrak{L}^{-1}$.

In parallel, the right block of the array operates in a similar way as the left one during the previous phase. Firstly the right block processors are initialized, then they operate as multiply-and-add cells. See the figure 5 for a global look of the array after the execution of phase 2.

Phase 3

Left processors are id'. Right block processors continue to operate as multiply-and-add cells, and they compute the product $\mathfrak{W}.\mathfrak{L}^{-1}$ whose last element is output at time 5n-2.

In an actual execution, these three phases are pipelined, according to the following schedule:
- phase 1 begins at time 1 and ends at time 3n-2
- phase 2 begins at time n+1 and ends at time 4n-2
- phase 3 begins at time 2n+1 and ends at time 5n-2

Hence we can state the following theorem:

theorem : the orthogonal systolic array of $(n+1)^2$ processors can solve any instance of the APP in 5n-2 time-steps, each step being the time necessary to achieve a multiply-and-add in the underlying algebra.

4. CONCLUSION

We have introduced a new systolic array involving the same number of processors as that of Rote [10] which solves the APP within 5n-2 time-steps, instead of 7n-2. As pointed out in [10], this value is optimal within one step, provided that each of the n^2 data elements is used only once in each step. Moreover, the solution of a new instance of the APP can begin every 2n steps.

Rote [10] presents a unidirectional version, called GJE-UNI, which solves the APP in 5n-2 steps too. However, this array requires $5n^2$ processors (instead of n^2), and its drawbacks are that the data flow is not as regular, and processors actions are not as uniform. In the contrary, the data moves in a regular and systematic way in our array. The importance of designing arrays without feedback cycles is emphasized in [6]: acyclic implementations usually exhibit more favorable characteristics with respect to fault-tolerance, two-level pipelining, and problem decomposition in general.

Moreover, we point out that the array can be micro-programmed to solve various instances of the APP by selecting the appropriate semiring operations, contrarily to the array of Guibas et al [4] which can only compute the transitive closure of a binary relation.

Finally, the array that we have presented here is very similar to the array introduced by Robert and Tchuente [9] for solving dense linear systems of equations using the Jordan diagonalization method. However, the array of [9] is more specialized: division is used as *-operation (**if** $c \neq 0$ **then** $c^* := 1/c$), (x) is the multiplication on real numbers, and these two operation commute, leading to another ordering in the APP algorithm.

REFERENCES

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

[2] J.M. DELOSME, Algorithms for finite shift-rank processes, Ph.D. Thesis, Technical Report M735-22, Sept. 1982, Stanford Electronics Laboratories

[3] A.L. FISHER, H.T. KUNG, K. SAROCKY, Experience with the CMU programmable systolic chip, Proc. SPIE Symp., vol. 495, Real-Time Signal Processing *VII* (1984)

[4] L.J. GUIBAS, H.T. KUNG, C.D. THOMPSON, Direct VLSI implementation of combinatorial algorithms, Proc. Caltech Conf. on VLSI: Architecture, design, fabrication, California Institute of technology, Pasadena (1979)

[5] H.T. KUNG, Why systolic architectures, IEEE Computer 15, 1 (1982), p 37-46

[6] H.T. KUNG, M.S. LAM, Fault-tolerance and two-level pipelining in VLSI systolic arrays, Technical Report, Carnegie Mellon University, November 1983

[7] H.T. KUNG, C.E. LEISERSON, Systolic arrays for (VLSI), in Proc. of the Symposium on Sparse Matrices Computations, I.S. Duff and G.W. Stewart eds, Knoxville, Tenn., 1978, p 256-282

[8] Y. ROBERT, Block LU decomposition of a band matrix on a systolic array, Int. J. Computer Math., to appear 1985

[9] G. ROTE, A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion), Computing 34, 191-219 (1985)

[10] U. ZIMMERMANN, Linear and combinatorial optimization in ordered algebraic structures, Ann. Discrete Math. 10, 1, 1-380 (1981)

Yves ROBERT,
CNRS, Laboratoire TIM3-INPG
BP 68, 38402 Saint Martin d'Hères Cedex, France

Denis TRYSTRAM,
Ecole Centrale Paris,
92295 Chatenay Malabry Cedex, France

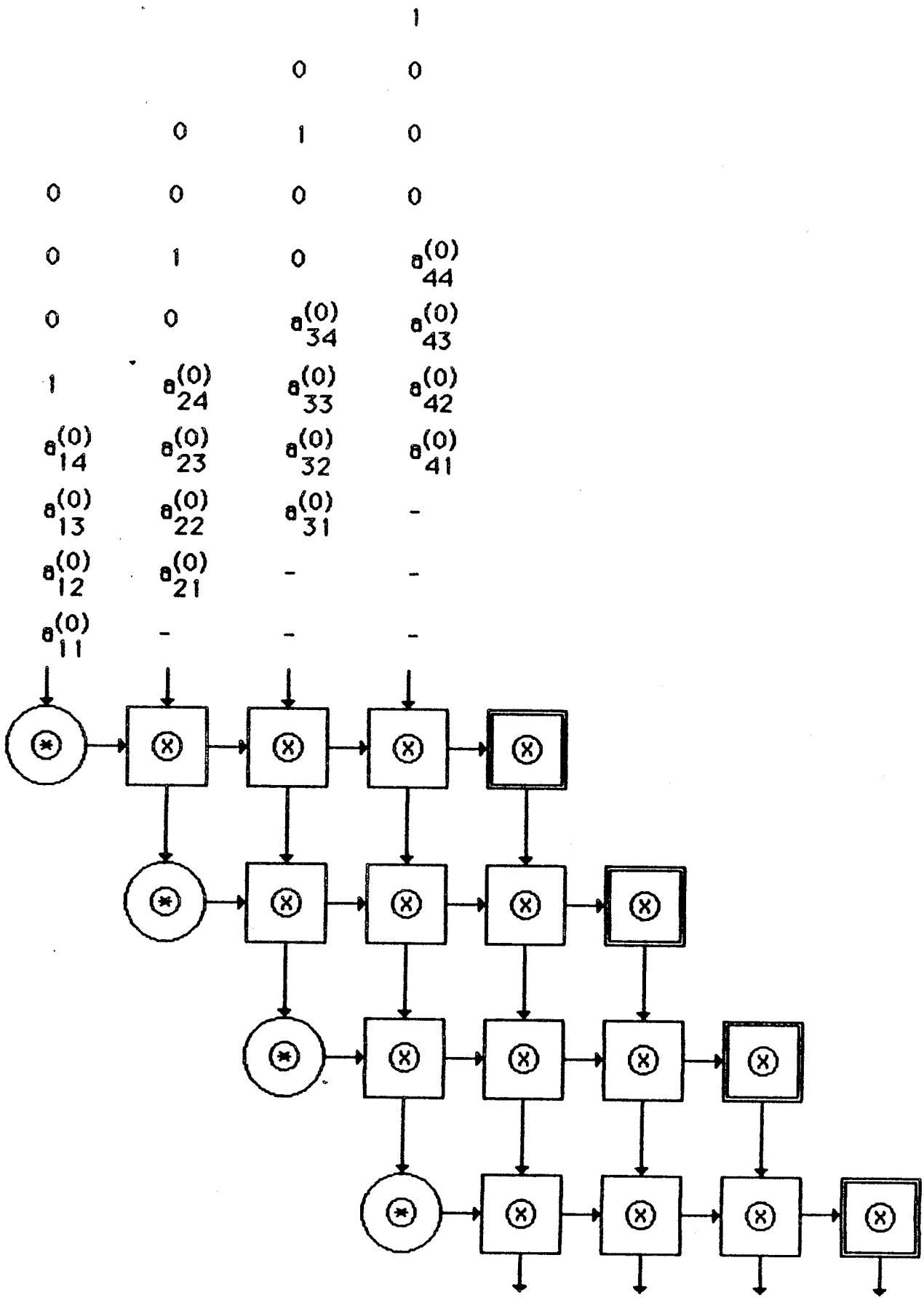
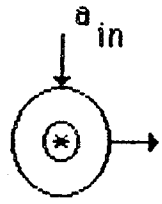
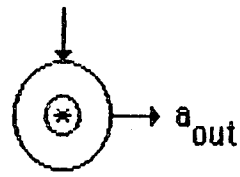


Figure 1 : Input of the systolic array

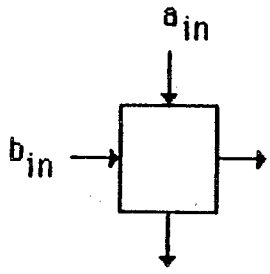


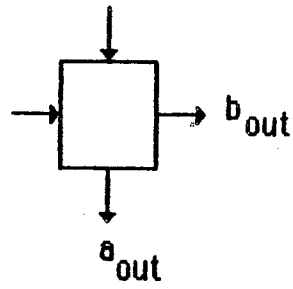


Step t

Step t+1

```
if bool = true
then { perform *-operation }
     begin a_out := a_in* ;
           bool := false ;
     end
else { transfer data }
     a_out := a_in
```





Step t

Step t+1

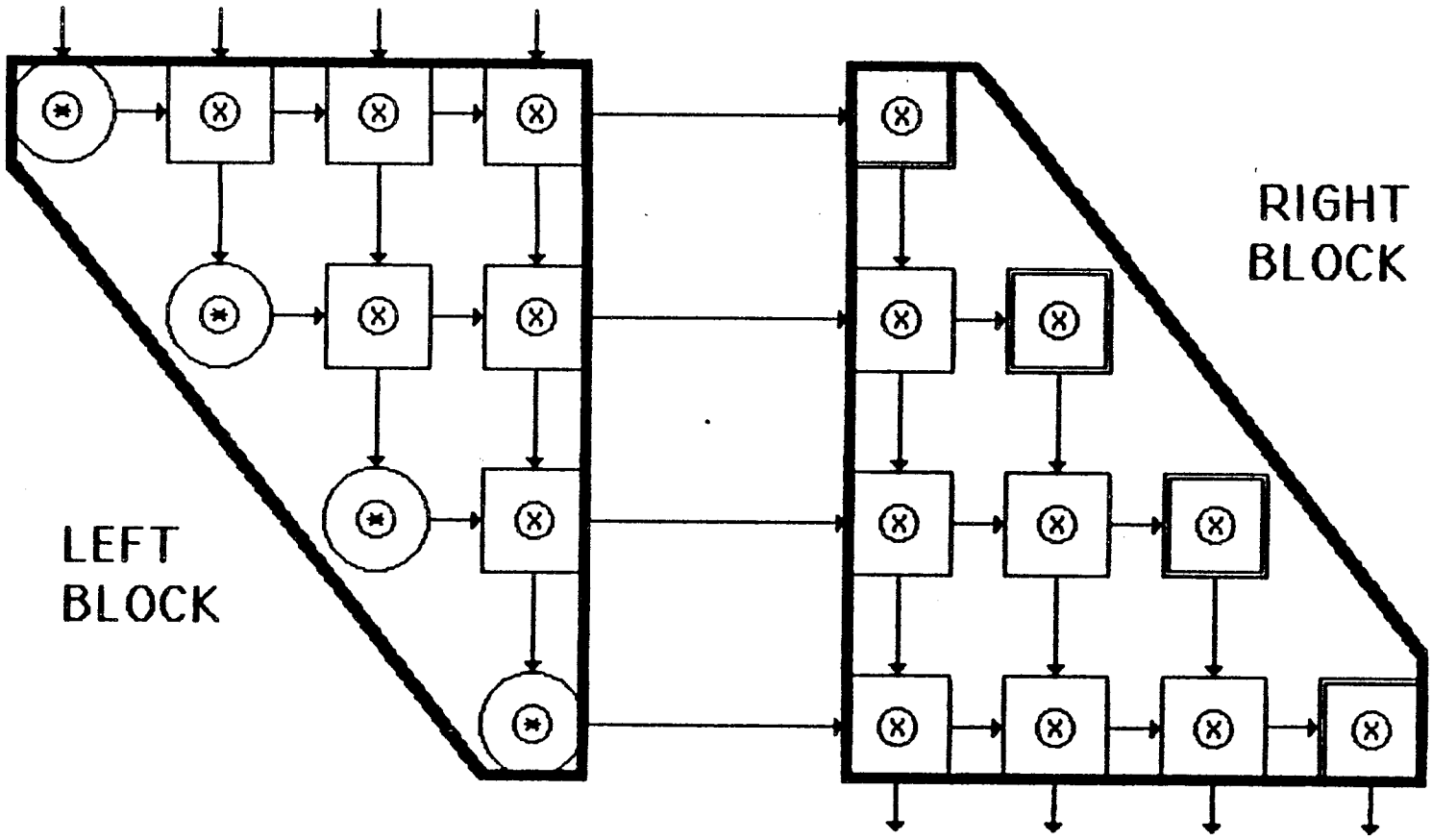
```
if bool = true
then { initialize current register }
     begin r := a_in (x) b_in ;
           bool := false ;
           b_out := b_in
     end
else { update a_in }
     begin a_out := a_in (+) r (x) b_in ;
           b_out := b_in
     end
```

Figure 2 : Operation of the processors

Step t Step t+1

```
if bool = true
then ( initialize current register )
     begin r := b_in ;
           bool := false ;
     end
else ( update b_in )
     a_out := r (x) b_in
```

Figure 2 (continued)



Conceptual partition of the array into two sub-blocks

Figure 3

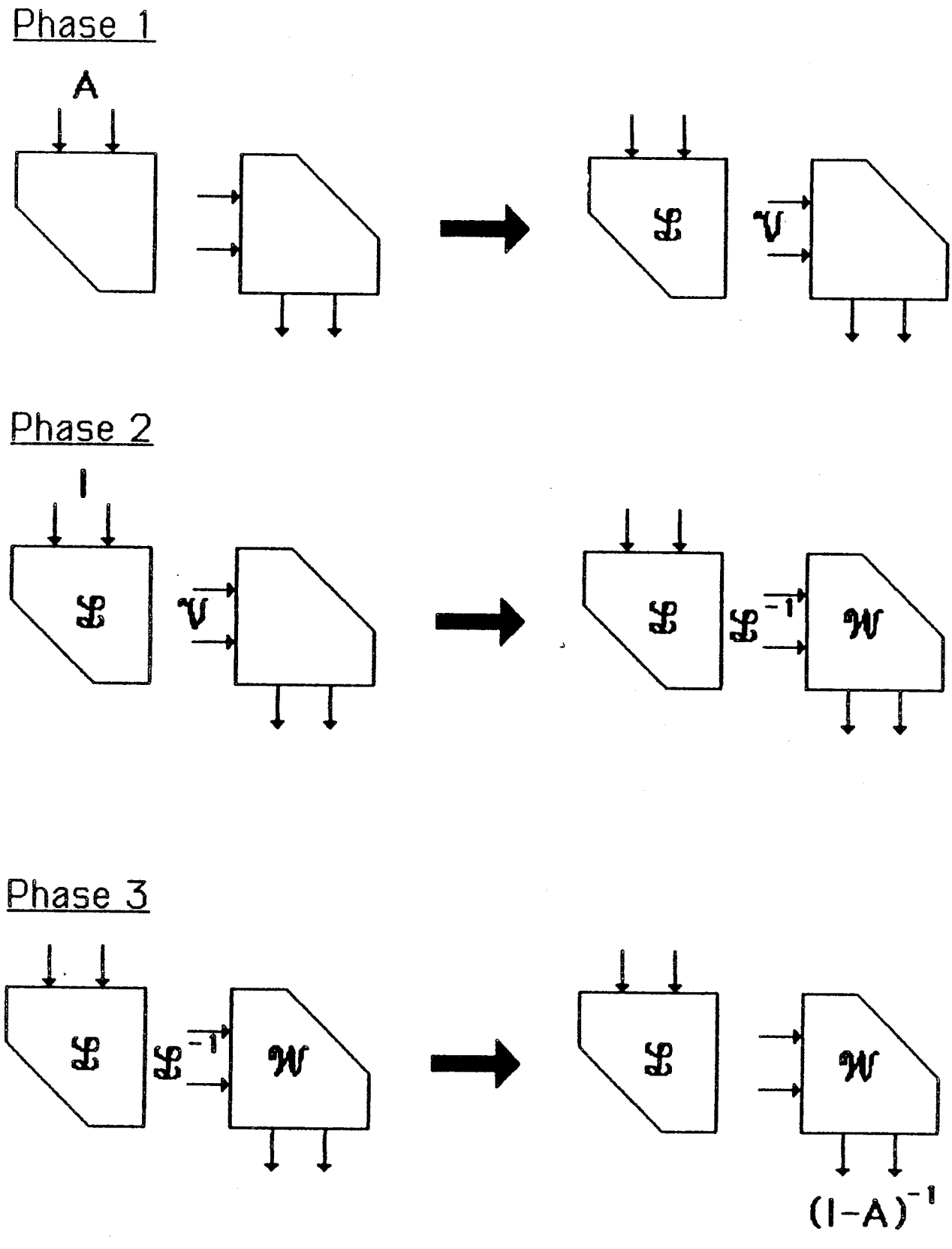


Figure 4 : Symbolic execution of the algorithm

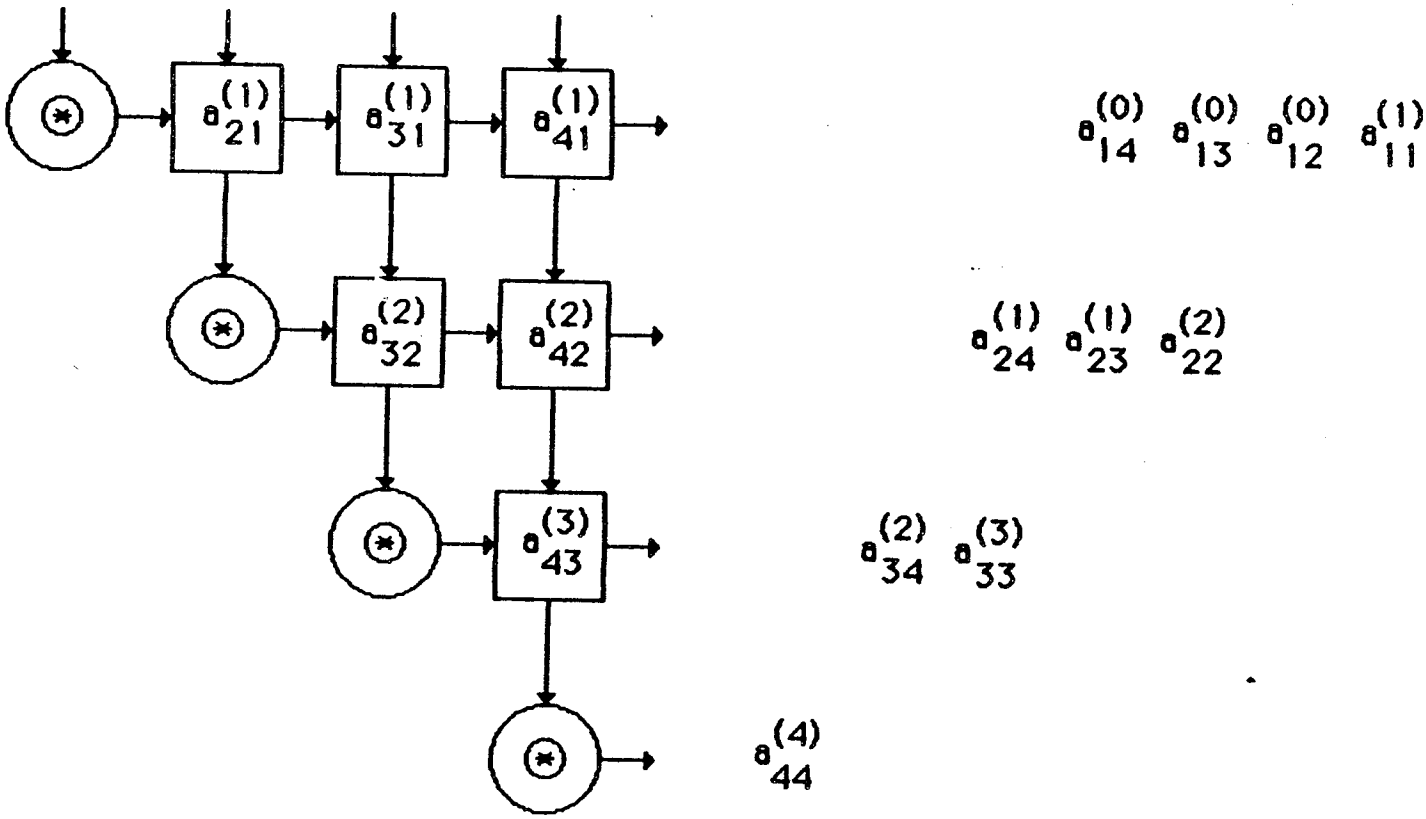


Figure 5 : Phase 1 of the algorithm

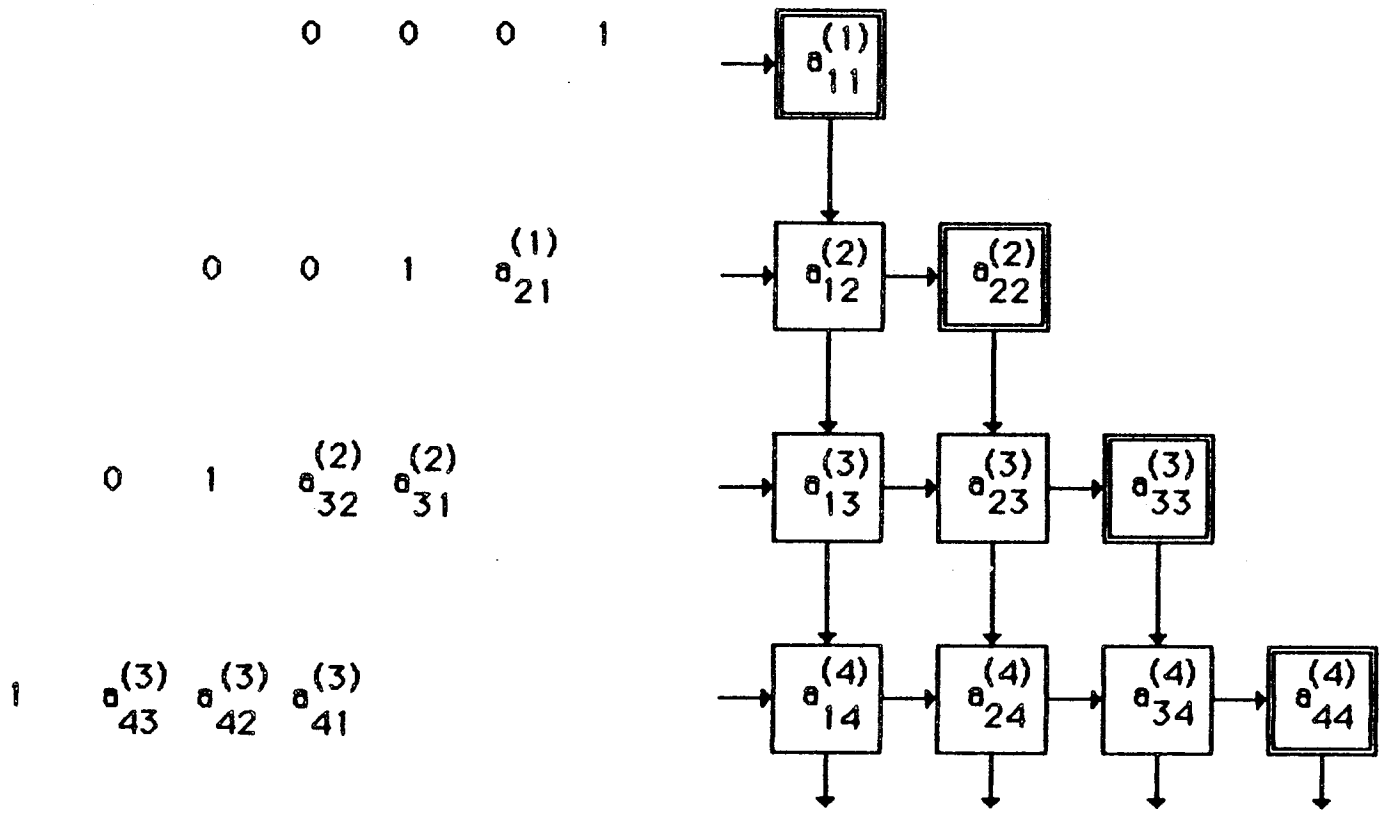


Figure 6 : Phase 2 of the algorithm

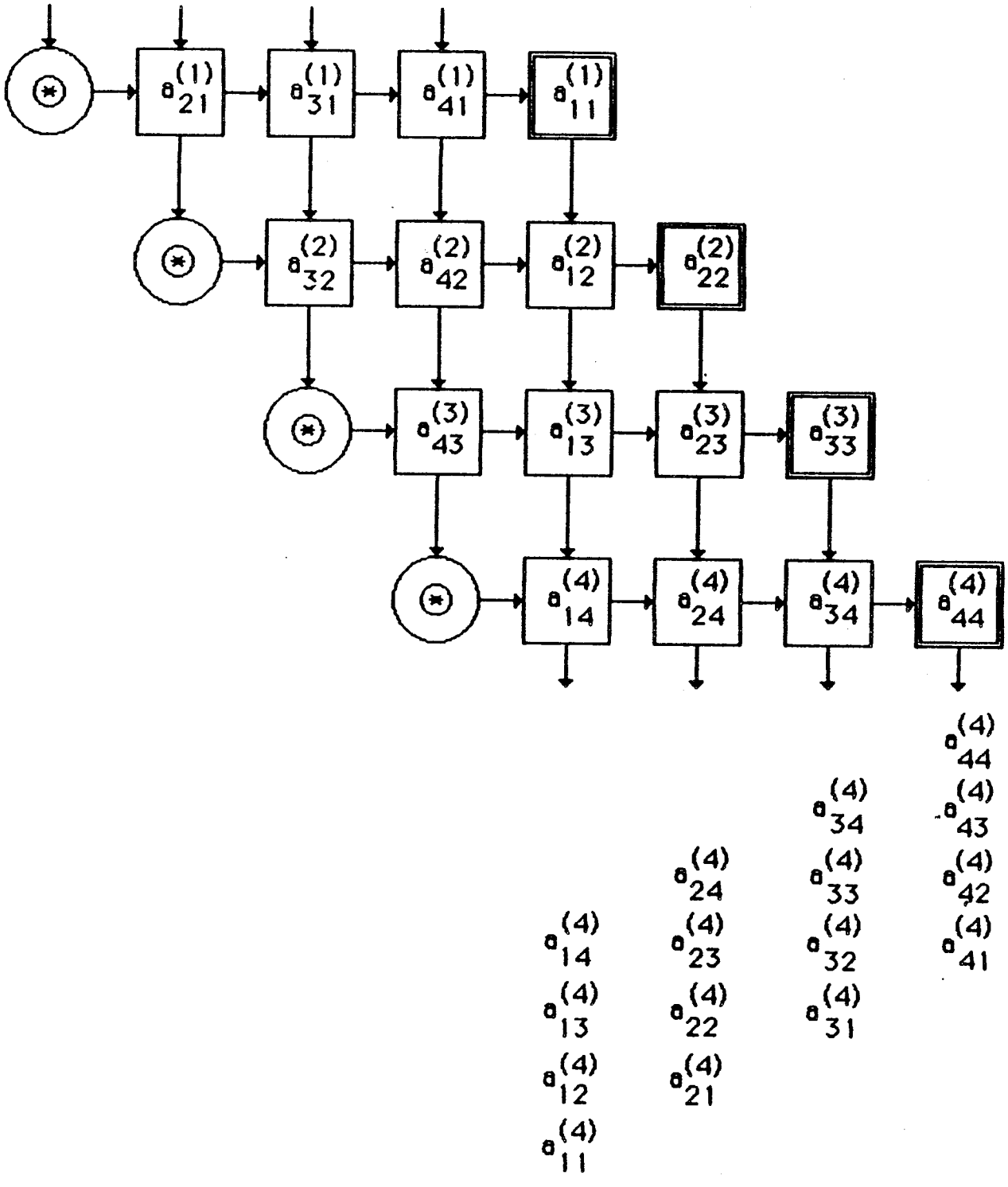


Figure 7 : Phase 3 of the algorithm

SOUS-CHAPITRE 6.3.

Implementation VLSI d'algorithmes modulaires issus du Calcul Formel

James H. Davenport[+] & Yves ROBERT[++]

Actes du Séminaire "Techniques du Calcul Formel pour l'Automatique, l'Analyse des Systèmes et le Traitement du Signal", Journées ATP du CNRS, 19-20 Décembre 1983, Grenoble, P. CHENIN éditeur (à paraître aux éditions du CNRS)

[+] University of Bath, School of Mathematics,
Claverton Down, Bath BA 7AY, England

[++] CNRS, Laboratoire TIM3/IMAG
BP 68, 38402 Saint Martin d'Hères Cedex

IMPLEMENTATION VLSI D'ALGORITHMES MODULAIRES ISSUS DU CALCUL FORMEL

James H.DAVENPORT + et Yves ROBERT *

RESUME :

En CALCUL FORMEL, les expressions manipulées sont complexes, et, de ce fait, les algorithmes usuels souffrent d'une relative lenteur. Si l'on envisage leur intégration pour gagner en vitesse de calcul, le recours à des méthodes modulaires s'avère indispensable pour éliminer les problèmes de croissance des résultats intermédiaires. On étudie deux exemples: le produit de deux polynômes, et le calcul de leur PGCD.

ABSTRACT :

In COMPUTER ALGEBRA systems, the expressions manipulated are large and complicated, and hence the algorithms suffer, in general, from a relative slowness. In order to increase the speed of these systems by implementing certain basic functions in hardware, we need to use a modular approach which solves the problem of "intermediate expression swell". Two examples are discussed: the product of two polynomials, and the calculation of their GCD.

+ University of Bath, School of Mathematics, Claverton Down, BATH BA2 7AY, ENGLAND

* Analyse Numérique, Laboratoire IMAG, BP 68, 38402 SAINT MARTIN D'HERES CEDEX

1. INTRODUCTION

Dans les systèmes de CALCUL FORMEL (Macsyma, Reduce), les expressions manipulées sont complexes - des fractions rationnelles à plusieurs indéterminées, par exemple - et, de ce fait, les algorithmes de traitement souffrent généralement d'une relative lenteur, même quand ils sont implantés sur les plus puissantes unités de calcul.

Peut-on espérer augmenter le rendement ou la capacité de ces systèmes en "cablant" certaines fonctions de base? Il ne s'agit pas de transformer totalement les systèmes de CALCUL FORMEL, mais de leur adjoindre des processeurs intégrés spécialisés performants afin d'augmenter la vitesse de traitement (compte tenu de la nette supériorité du matériel sur le logiciel). Les progrès de la technologie VLSI (intégration à très haute densité de composants) permettent de concevoir dès maintenant de telles réalisations.

2. MULTIPLICATION SYSTOLIQUE DE DEUX POLYNOMES

Prenons comme premier exemple le problème simple suivant: étant donnés deux polynômes à coefficients entiers

$$A(x) = \Sigma_{i=0}^{m-1} a_i \cdot x^i \text{ et } B(x) = \Sigma_{i=0}^{n-1} b_i \cdot x^i \text{ , } m < n$$

calculer leur produit (de convolution)

$$C(x) = \Sigma_{i=0}^{m+n-2} c_i \cdot x^i \text{ , où } c_i = \Sigma_{k=0}^{m} a_k \cdot b_{i-k}$$

Un algorithme séquentiel s'exécutera en un temps $O(mn).\tau$, si τ est le temps nécessaire pour effectuer une multiplication suivie d'une addition. Peut-on améliorer ce temps de calcul si l'on dispose de plusieurs processeurs? Une parallélisation idéale serait de déterminer tous les coefficients de C en $O(n)\tau$ unités de temps sur un réseau comportant m processeurs.

2.1 Approche systolique

Le modèle systolique introduit par H.T.KUNG(11) conduit effectivement à ce degré de parallélisme. Pour calculer les

coefficients du polynôme produit, KUNG utilise dans (12) un réseau linéaire de m processeurs élémentaires identiques, dont les spécifications sont données figure 1. La synchronisation du réseau est assurée de manière globale, par deux horloges alternées et non recouvrantes. Après une phase d'initialisation, le réseau fonctionne de manière pipe-line, et délivre en sortie une nouvelle valeur c_i toutes les deux pulsations: il est important de noter que la valeur d'une pulsation - un top d'horloge - est égale au temps de cycle d'un seul processeur: on retrouve la valeur τ précédente. Quelques pulsations consécutives du réseau sont illustrées figure 2. On vérifie bien que le temps de calcul est égal à $(m+2n).\tau$.

2.2 Une parenthèse "divide and conquer" (16)

Dans le réseau précédent, chaque cellule n'est activée qu'une pulsation sur deux. Pour aller deux fois plus vite, il faudrait que les données b_i rentrent dans le réseau tous les tops d'horloge, et que les valeurs des c_i soient accumulées à ce rythme. Pour ce faire, il suffit de calculer séparément sur deux copies d'un réseau de taille moitié du précédent les valeurs

$c_i^1 = \Sigma_{k=0,\ k\ \text{pair}}^{m-1} a_k \cdot b_{i-k}$ et $c_i^2 = \Sigma_{k=0,\ k\ \text{impair}}^{m-1} a_k \cdot b_{i-k}$

puis de les ajouter. Le réseau présenté figure 3 délivre les coefficients c_i en temps $(\lceil m/2 \rceil + 2+n).\tau$.

L'intérêt de l'approche systolique réside surtout dans le fait que toute architecture systolique conduit directement à une implémentation VLSI. Les deux réseaux proposés sont agencés de façon régulière et modulaire, et il ne faudrait guère de temps à un concepteur disposant d'un exemplaire du processeur élémentaire de la figure 1 en bibliothèque pour dessiner les masques d'un circuit solution.

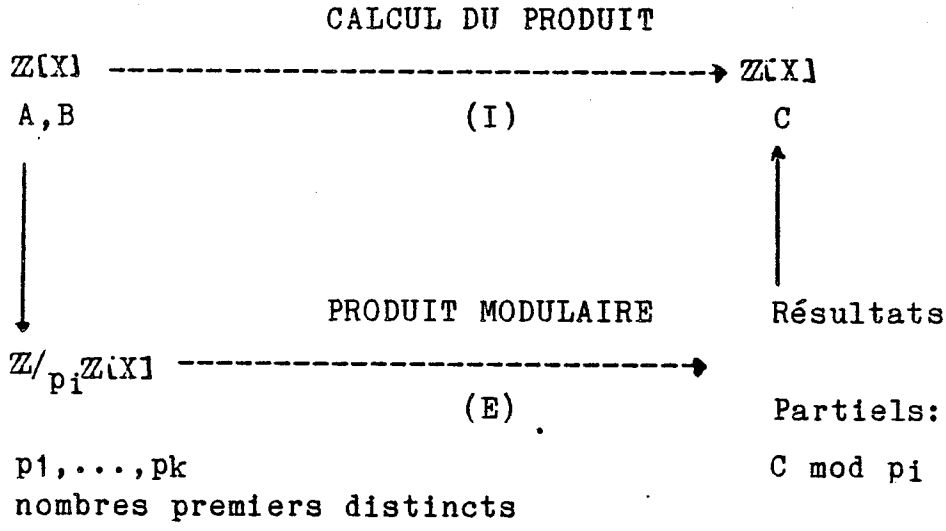
Hélas! Autant la solution systolique est séduisante pour des convolutions appliquées au traitement du signal, où les données sont échantillonnées et de taille connue a priori, autant elle est inefficace dans notre contexte: les polynômes

que l'on rencontre en CALCUL FORMEL sont souvent lacunaires, et leurs coefficients ne sont pas du même ordre de grandeur; la taille des calculs pouvant être menés sur un circuit étant par nature limitée, il faudra "choisir" entre implémenter des multiplieurs (de deux entiers)
- de petite taille ... mais insuffisants
- de grande taille - en faisant l'hypothèse (peu réaliste) de pouvoir borner a priori la taille des nombres qui vont apparaitre - ... mais presque jamais utilisés dans toute leur capacité, perdant ainsi l'avantage de la vitesse.

Pour surmonter cette difficulté, il faut recourir à des méthodes plus élaborées.

2.3 Les méthodes modulaires

L'approche est la suivante: pour résoudre le problème initial (I), on donne une solution VLSI de l'étape (E), alors que réduction et reconstruction sont conduites classiquement :



Dans cette perspective, il est nécessaire d'effectuer plusieurs calculs distincts avec des choix de p différents, et de déduire la valeur dans $\mathbb{Z}$ du produit C des deux polynômes des différents produits modulaires obtenus, à partir du théorème des Restes Chinois. Plus précisément, introduisons l'opérateur M_p de réduction modulo p qui applique $\mathbb{Z}$ sur $\mathbb{Z}/p\,\mathbb{Z} = \mathbb{Z}_p$ et $\mathbb{Z}[X]$ sur $\mathbb{Z}_p[X]$. Si p ne divise pas les coefficients

directeurs de A et de B, alors $M_p(C) = M_p(A) \cdot M_p(B)$. On donne ci-dessous la procedure du calcul modulaire de C, basée sur le résultat suivant (1): soient $p_0, p_1, \ldots, p_k$ des entiers premiers entre eux deux à deux et p leur produit; soit $u_i = u \bmod p_i$; alors $u \leftrightarrow (u_0, \ldots, u_k)$ est une relation bijective entre les entiers u, $0 \leq u < p$ et les k-uplets $(u_0, \ldots, u_k)$, $0 \leq u_i < p_i$; on suppose A et B de coefficient directeur égal à 1 pour simplifier.

```
PROCEDURE PRODUIT(A,B)
Taille := 2 * deg(A) * Max(|coeff A|) * Max(|coeff B|)
Base := Un nombre premier
Jusqu'ici := PRODUITmod( A , B , Base )
Tant que Base < Taille faire {
  p:= un nombre premier
  Temp := PRODUITmod( A , B , p )
  Jusqu'ici := Thm.Chinois( Jusqu'ici , Temp , Base )
  Base := Base * p  }
```

2.4 Coût du calcul modulaire

Il s'agit donc mener tous les calculs modulo un nombre premier p; le problème est relativement simple ici, puisque nous n'avons besoin que d'additions et de multiplications:

- pour l'addition: le calcul de a + b mod p exige le calcul de a + b et celui de a + b - p, le signe de cette dernière quantité permettant de choisir. Le calcul modulaire est ici deux fois plus coûteux que le calcul dans $\mathbb{Z}$

- pour la multiplication: le calcul de a * b mod p semble exiger une multiplication suivie d'une division; on peut cependant modifier l'algorithme usuel de la multiplication pour éviter toute division:

Algorithme IMA (Integer Multiplication Algorithm (3))

```
( calcul du produit A * B, où A et B sont < 2**H  )
Lire A, B, H ;
```

```
W <-- 0 ; S <-- H -1 ;
Tant que S > 0 faire
            Si B(S) = 1 alors W <-- W + A
            W <-- 2 * W
            S <-- S - 1
W est le produit cherché
```

Algorithme MMA (Modular Multiplication Algorithm (3))

```
( calcul du produit A * B mod p, où A, B et p sont < 2**H )
Lire A, B, p, H
W <-- 0 ; S <-- H - 1 ;
Tant que S > 0 faire
            Si B(S) = 1 alors W <-- W + A mod p
            W <-- W + W mod p
            S <-- S - 1
W est le produit modulaire cherché
```

L'algorithme IMA conduit classiquement (10) à l' implantation d'un multiplieur cellulaire itératif. La même architecture peut être reprise pour l'implantation du multiplieur MMA. ici encore, le calcul modulaire s'avère deux fois plus coûteux. Il est vrai que le choix de $p = 2^n \pm 1$ simplifierait grandement les calculs (15), mais nous devons traiter le problème pour p arbitraire.

2.5 Le processeur systolique "revisited"

Nous voilà en mesure de préciser la structure du processeur élémentaire (P) - voir figure 4 -. Si p est codé sur H bits, le temps de cycle de (P) est environ le double que précédemment lorsqu'on traitait dans $\mathbb{Z}$ des nombres codés sur H bits. Mais bien que coûteuse au niveau miscroscopique, l'approche modulaire conduit à une solution systolique effective et performante (quant à la parallélisation) du problème initial.

3. UN CIRCUIT POUR LE CALCUL DU PGCD DE DEUX POLYNOMES

L'exemple du calcul du PGCD de deux polynômes A et B

est encore plus éloquent que celui de leur produit de convolution: choisissant avec BROWN(5)

$$A = x^8+x^6-3x^4-3x^3+8x^2+2x+5 \text{ et } B = 3x^6+5x^4-4x^2-9x+21$$

on trouve, selon la méthode utilisée :

- 1259333879550074310093114199218750O (algorithme d'EUCLIDE)
- 869224534119352216237500 (méthode réduite)
- 23552630096885020 (méthode des sous-résultants)

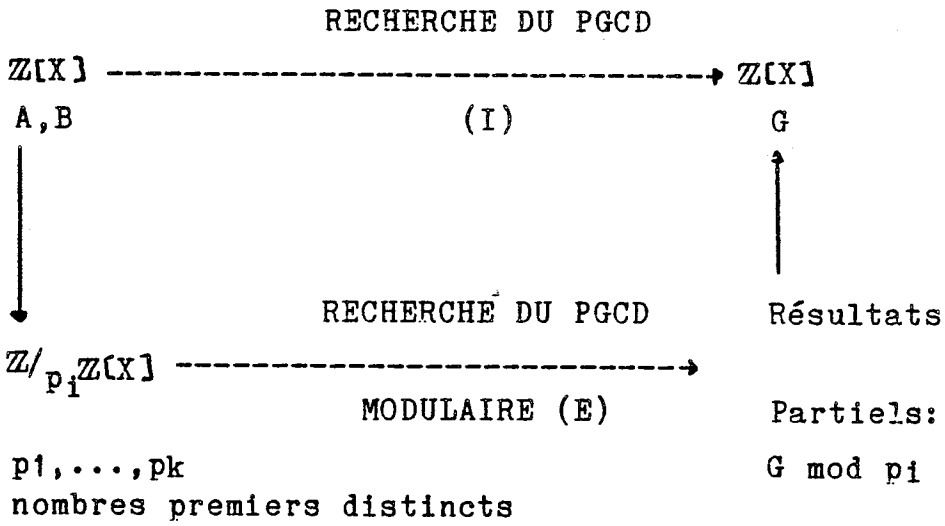
On a commencé avec des entiers de deux chiffres, on finit avec des nombres de 35,24 ou 17 chiffres, bien que la vraie réponse soit que les polynômes sont premiers entre eux.

Pourtant, cet algorithme du PGCD est essentiel en CALCUL FORMEL (ainsi l'addition de deux fractions rationnelles exige 2 calculs de PGCD).

Comme pour le problème de la multiplication, il faudra recourir à des méthodes plus sophistiquées.

3.1 PGCD modulaire

Le contexte est le même qu'en 2.3:



Toutefois, la situation se complique un peu. Si p ne divise pas les coefficients directeurs $c_p(A)$ et $c_p(B)$ de A et B, alors $M_p(PGCD(A,B))$ divise $PGCD(M_p(A),M_p(B))$. On dira que p est fortuné si ces derniers polynômes sont égaux; il n'existe qu'un nombre fini de p infortunés - ceux qui divisent le résultant Res(A/G , B/G), où G est le PGCD de A et B dans ℤ

(3) -. On détaille ci-dessous la procedure PGCD; elle est très proche de celle donnée dans (3), mais on utilise une borne qui provient de l'inégalité de LANDAU (13):

```
        PROCEDURE PGCD(A,B)
        Taille := 2.Min( Max( | coeff A | ), Max( | coeff B | ))
L1      Base := Un nombre premier
        Jusqu'ici := PGCDmod( A , B , Base )
L2      degré := deg( Sofar )
        Si degré = 0 alors retour 1
        Tant que Base<2^degré * Taille faire {
L3      { p := Un nombre premier
          Temp := PGCDmod( A , B , p ) }
          Si deg( Temp )  degré alors aller à L3
          Si deg( Temp )  degré alors
              { Base := p ; Jusqu'ici := Temp ; aller à L2 }
        Jusqu'ici := Thm.Chinois( Jusqu'ici , Temp , Base )
        Base := Base * p }
        Si Jusqu'ici ne divise pas A alors aller à L1
        Si Jusqu'ici ne divise pas B alors aller à L1
        retour Jusqu'ici
```

On trouvera dans (9) une étude détaillée de la complexité de cette procédure. On établit en particulier le coût relatif de l'étape (E), la seule à devoir être intégrée. Tous les calculs modulaires de PGCD pourront être réalisés en temps O(n) si l'on dispose d'autant d'exemplaires du circuit (programmable avec p) que de nombres premiers nécessaires à la reconstruction. Dans la suite, nous allons détailler l'implémentation de cette étape (E), réglant ainsi les problèmes de taille évoqués: comme on l'a vu plus haut, l'algorithme du PGCD est très fréquemment utilisé (jusqu'à 95% du temps total (6)), et cette procédure limite la vitesse de traitement de tout le système: ce qui motive la définition d'un processeur spécialisé pour ce calcul.

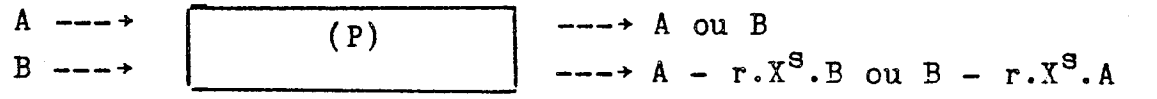
3.2 Algorithme et spécifications du circuit

Pour le calcul du PGCD de deux polynômes A et B, on

utilise la méthode de l'abaissement du degré:

tant que $\delta(A).\delta(B) \neq 0$ faire
si $\delta(A) > \delta(B)$ alors $A \leftarrow A - B.X^{\delta(A)-\delta(B)}.a_{\delta(A)}/b_{\delta(B)}$
sinon $B \leftarrow B - A.X^{\delta(B)-\delta(A)}.b_{\delta(B)}/a_{\delta(A)}$
si $A = 0$ alors PGCD $\leftarrow$ B sinon PGCD $\leftarrow$ A

Chaque étape de l'algorithme correspond donc à une élimination partielle; sous cette forme, il est clair que la méthode proposée - au contraire de l'algorithme d'EUCLIDE - se prête à un certain parallélisme: l'exécution peut être sérialisée en opérations élémentaires effectuées par des processeurs identiques:



Bien mieux, on peut éviter la multiplication par une puissance X^s (qui consiste à "décaler" de s positions les coefficients de l'un des deux polynômes): si à la suite d'une élimination précédente, on a abaissé le degré de plus d'une unité, on convient de sortir un zéro comme coefficient dominant du polynôme obtenu: en ce cas, le rôle du processeur (P) suivant sera simplement:
- d'éliminer ce coefficient nul
- d'abaisser d'une unité le degré du polynôme
- de transmettre sans modification l'autre polynôme au processeur suivant.

On obtient ainsi la réalisation systolique recherchée, en disposant d'un réseau de processeurs identiques, dont les spécifications sont les suivantes (voir figure 5):
- (i) si n ou m = 0, transmettre sans modification A et B
- sinon (ii), si a_n ou $b_m = 0$, décaler l'un des polynômes et transmettre l'autre - a_n et b_m ne sont jamais simultanément nuls - (par exemple si $a_n = 0$, supprimer a_n, faire $n \leftarrow n-1$, transmettre B sans modification)
- sinon (iii), faire une élimination partielle (si $n > m$, faire $a_i \leftarrow a_i - r.b_{i+m-n}$ $0 \leq i \leq n-1$ où $r = a_n / b_m$)

Nous simplifions en remarquant que (ii) s'interprète comme (iii) avec r=0; de même nous poserons r=0 dans le cas (i).

3.3 Utilisation du calcul modulaire

On doit calculer des expressions du type $a - \alpha / \beta * b$ La principale difficulté réside dans la division. En général, on choisit d' utiliser une table d'inverses modulo p.

Mais quitte à employer une table, il est possible de donner une solution élégante et rapide pour le calcul des 4 opérations usuelles +,-,x,/ modulo p .

Un entier modulo p est habituellement représenté par un élément de l'ensemble $E_p = \{0,1, \ldots , p-1 \}$. Convenant de fixer une racine primitive K de p (ie un élément engendrant le groupe multiplicatif $\mathbb{Z}p - \{0\}$), on peut écrire tout élément non nul de E_p (de façon unique) comme une puissance de K comprise entre 0 et p-2; attachant à 0 la représentation symbolique *, on notera simplement i l'élément K^i de E_p, $0 \leq i \leq p-2$. La multiplication et la division sont respectivement transformées en addition et soustraction, selon les règles:

$$i \bar{x} j = \begin{cases} * \text{ si } i \text{ ou } j = * \\ i + j \bmod p-1 \text{ sinon} \end{cases}$$

$$i \bar{/} j = \begin{cases} * \text{ si } i = * \text{ (non défini si } j = *) \\ i - j \bmod p-1 \text{ sinon} \end{cases}$$

Pour l'addition (et la soustraction), il faut recourir aux "logarithmes de ZECH" (17): on définit Z(u) par

$K^{Z(u)} = 1 + K^u$ pour $u \in \{ 0,1, \ldots ,p-2\} - \{\frac{p-1}{2}\}$

$Z(\frac{p-1}{2}) = *$ et $Z(*) = 0$ (ces deux valeurs ne seront pas utilisées)

On obtient alors pour l'addition les règles suivantes:

$$i \bar{+} j = \begin{cases} i \text{ si } j = * \\ j \text{ si } i = * \\ * \text{ si } j-i = + \frac{p-1}{2} \\ i + Z(j-i) \bmod p-1 \text{ si } j-i \neq + \frac{p-1}{2} \end{cases}$$

et des règles similaires pour la soustraction .

La représentation modulo un nombre premier p de n bits se fera sur n+1 bits, un bit spécial étant réservé à *. Grâce à l'utilisation d'une table unidimensionnelle fournissant les logarithmes de ZECH, on réussit à éliminer les opérateurs à deux entrées " multiplication et division ". Le tableau ci-dessous, donnant en unités λ (MEAD/CONWAY(14)) la surface standard des différents éléments de calcul, établit clairement tout l'intérêt de la solution proposée pour des $p < 2^8$, ce qui couvre largement le champ d'application qui nous intérese.

Unité	Surface en λ^2
Multiplieur n-bits	$212 \cdot n^2$ (M)
Additionneur n-bits	$212 \cdot n$ (A)
Table n-bits	$3n \cdot 2^{n+6}$ (T)

On convient que la surface d'un diviseur est sensiblement celle d'un multiplieur, et que la surface d'une unité arithmétique mod p est voisine du double de celle de l'unité dans $\mathbb{Z}$.

Représentation conventionnelle (C):

4M + A (un multiplieur, un diviseur et un additionneur mod p)

Représentation de ZECH (Z):

T + 6A (une table et 3 additionnneurs mod p)

$$R = \frac{(Z)}{(C)} = 3 \cdot \frac{2^n + 2^7}{n.2^8 + 2^7}$$ fonction croissante de n

Notons que R = 9/17 pour n = 8, ce qui signifie que la représentation de Zech utilise une surface deux fois plus petite.

2.4 Architecture d'une cellule

La figure 6 détaille l'organisation architecturale d'une cellule. La réalisation du circuit est entreprise en

collaboration avec Y.CLAUZEL à l'aide du langage de conception LUCIE (2).

4. CONCLUSION

Dans (4), BRENT et KUNG présentent un réseau systolique pour le calcul du PGCD de deux polynômes. Au niveau macroscopique, l'architecture cellulaire de leur circuit est très voisine de celle proposée au paragraphe 3.2. Mais l'intérêt de notre approche modulaire est double:
- c'est la seule qui règle les problèmes de croissance des résultats intermédiaires, condition "sine qua non" (7) pour des problèmes issus du CALCUL FORMEL.
- elle permet, au niveau microscopique, une réalisation originale et concurrentielle de l'architecture interne de chaque cellule.

Les descrriptions algorithmiques que nous avont faites démontrent la possibilité de réaliser , au vu des performances technologiques actuelles, deux unités VLSI spécialisées pour le CALCUL FORMEL. C'est bien sûr un premier pas: quels autres algorithmes est-il possible d'implémenter? Nous savons déjà - cf (8) - que l'on peut étendre l'algorithme modulaire du PGCD au calcul des coefficients U et V de l'algorithme d' Euclide étendu (tels que A.U + B.V = G).

REFERENCES

(1) AHO A.V., HOPCROFT J.E., ULLMAN J.D., The design and analysis of computer algorithms, Addison Wesley 1974

(2) ANCEAU F., GUYOT A., RAYMOND J., Présentation du projet CMP, Rapport Equipe de recherche en architectures d'ordinateurs IMAG Grenoble 1981

(3) BLAKLEY G.R., A Computer Algorithm for Calculating the Product AB modulc M, IEEE Trans. on Computers vol C-32 Mai 1983 p 497-500

(4) BRENT R.P., KUNG H.T., Systolic VLSI arrays for linear-

time gcd computations, in "VLSI 83", F.ANCEAU et E.J.AAS eds, Elsevier Science Publishers, 1983

(5) BROWN W.S, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, J. ACM 18(1971) pp 478-504

(6) DAVENPORT J.H, On the integration of Algebraic Functions, Springer Lectures Notes in Computer Science n°102

(7) DAVENPORT J.H, VLSI et Calcul Formel, RR IMAG Grenoble n°357 Mars 1983

(8) DAVENPORT J.H, ROBERT Y., PGCD et VLSI, RR IMAG Grenoble n°358 Mars 1983

(9) DAVENPORT J.H., ROBERT Y., VLSI and Computer Algebra: the GCD Example, à paraitre dans "Comportement d'Automates et applications", Luminy, Septembre 1983, J.DEMONGEOT et al. eds, Springer Verlag 1984

(10) HWANG K, Computer Arithmetic: Principles, Architecture and Design, John Wiley and Sons, 1979

(11) KUNG H.T., Why systolic architectures, Computer Magazine 15(1):37-46, January, 1982

(12) KUNG H.T., Use of VLSI in algebraic computation: some suggestions, Proceedings of the 1981 ACM symposium on symbolic and algebraic comput·tion, Wang P.S.ed, ACM SIGSAM, August 1981 p 218-222

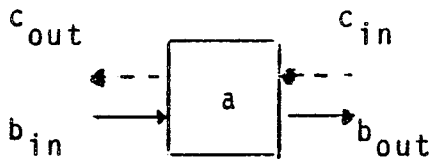
(13) MEAD C., CONWAY L., Introduction to VLSI systems, Addison Wesley 1980

(14) MIGNOTTE M., Some Useful Bounds, in "Computer Algebra Symbolic and Algebraic Computation", B.BUCHBERGER et al. eds, Springer Verlag, Vienna 1982

(15) NUSSBAUMER H.J., Fast Fourier Transform and Convolution Algorithms, Springer Series in Information Sciences, Springer Verlag 1980

(16) ROBERT Y, TCHUENTE M, Une approche "Divide and Conquer" pour des algorithmes systoliques, RR n°393 IMAG Grenoble 1983

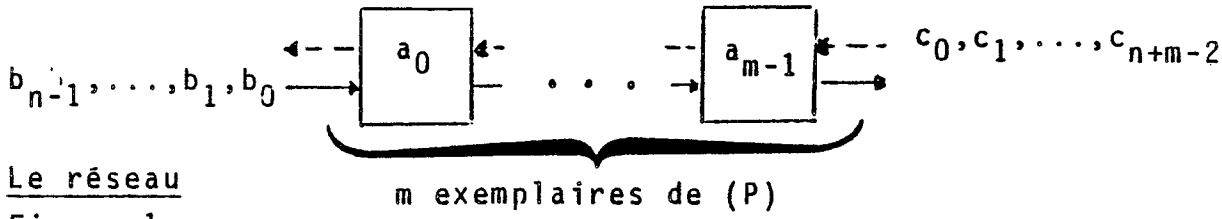
(17) ZIMMER H.G, "Computational Methods in NUMBER Theory", Springer Lecture Notes in Mathematics n°272



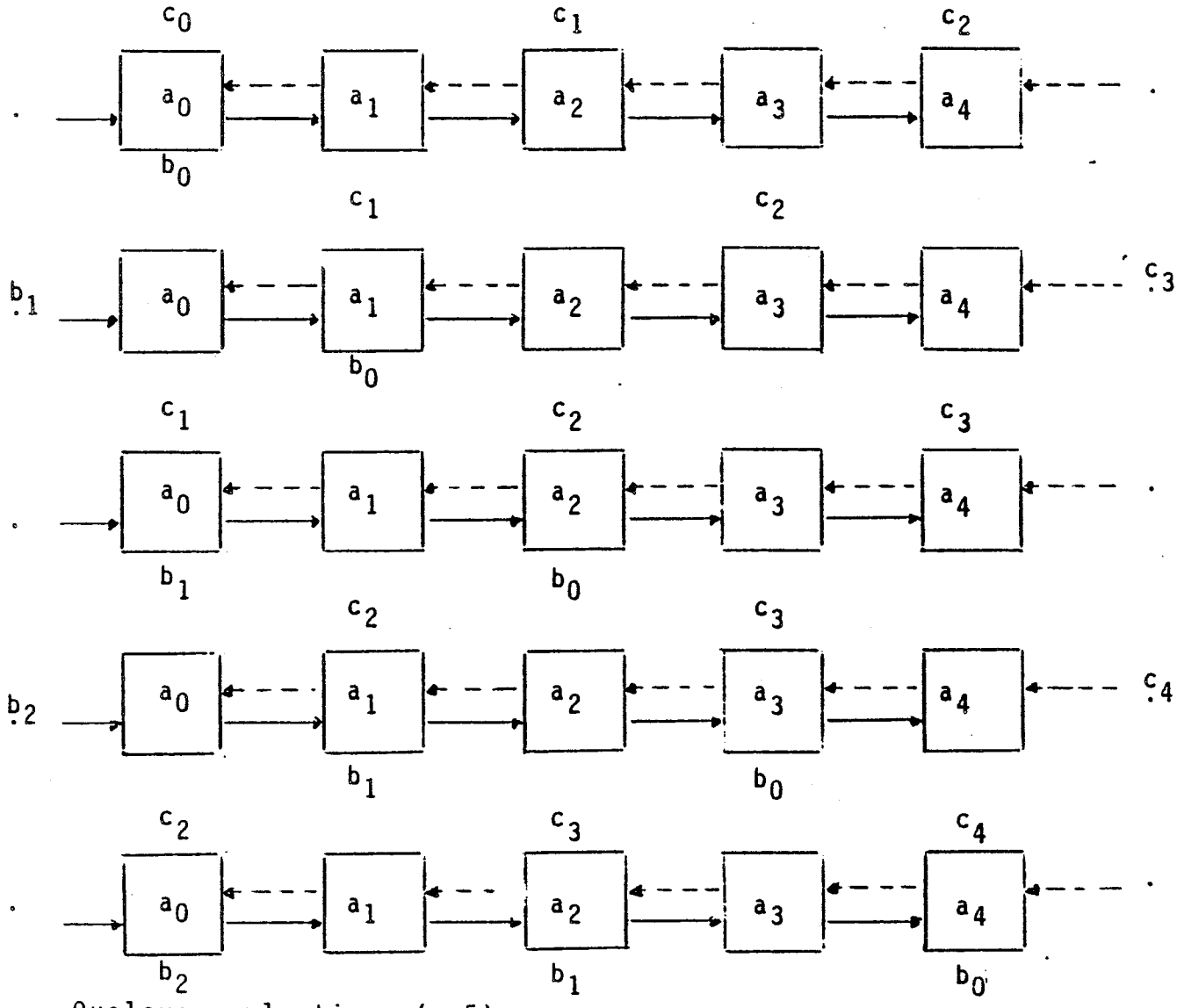
$b_{out} \leftarrow b_{in}$

$c_{out} \leftarrow c_{in} + a * b_{in}$

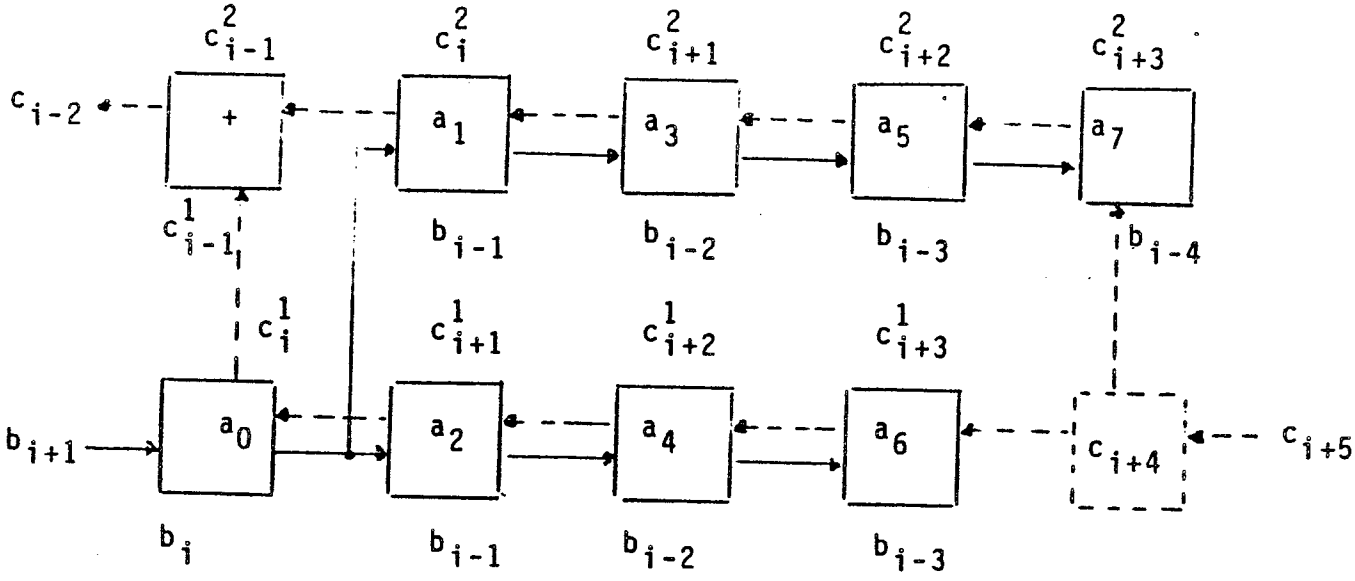
Le processeur élémentaire (P)



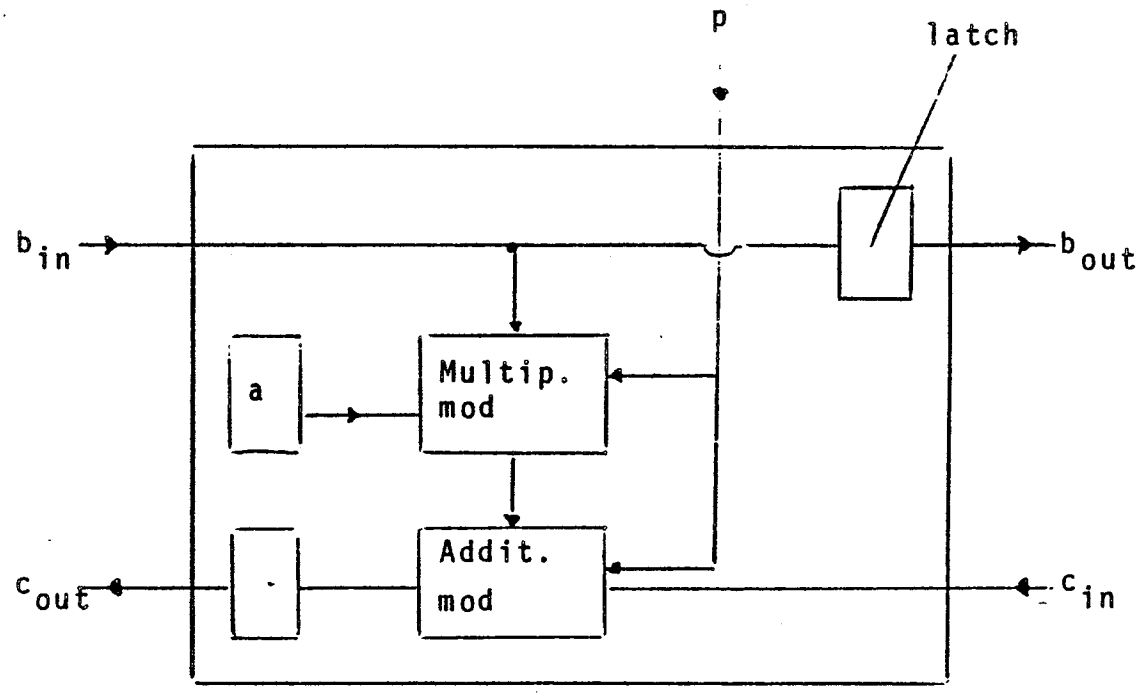
Le réseau
Figure 1



Quelques pulsations (m=5)
Figure 2



Le nouveau réseau (m=8)
Figure 3



Processeur modulaire: scnéma de fonctionnement
Figure 4

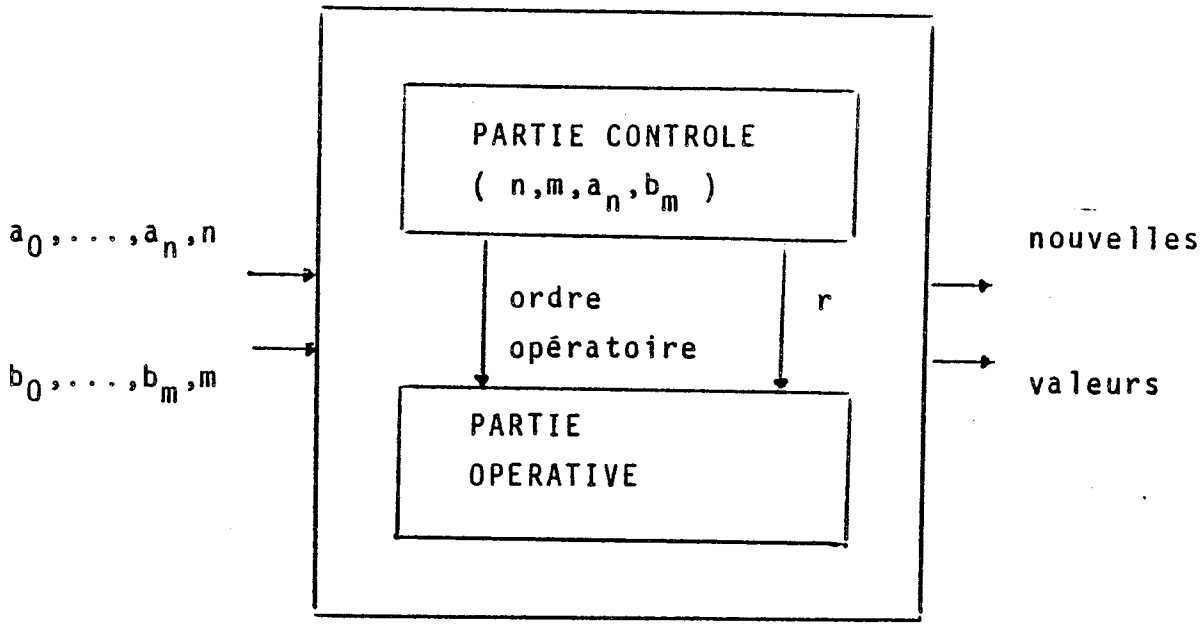
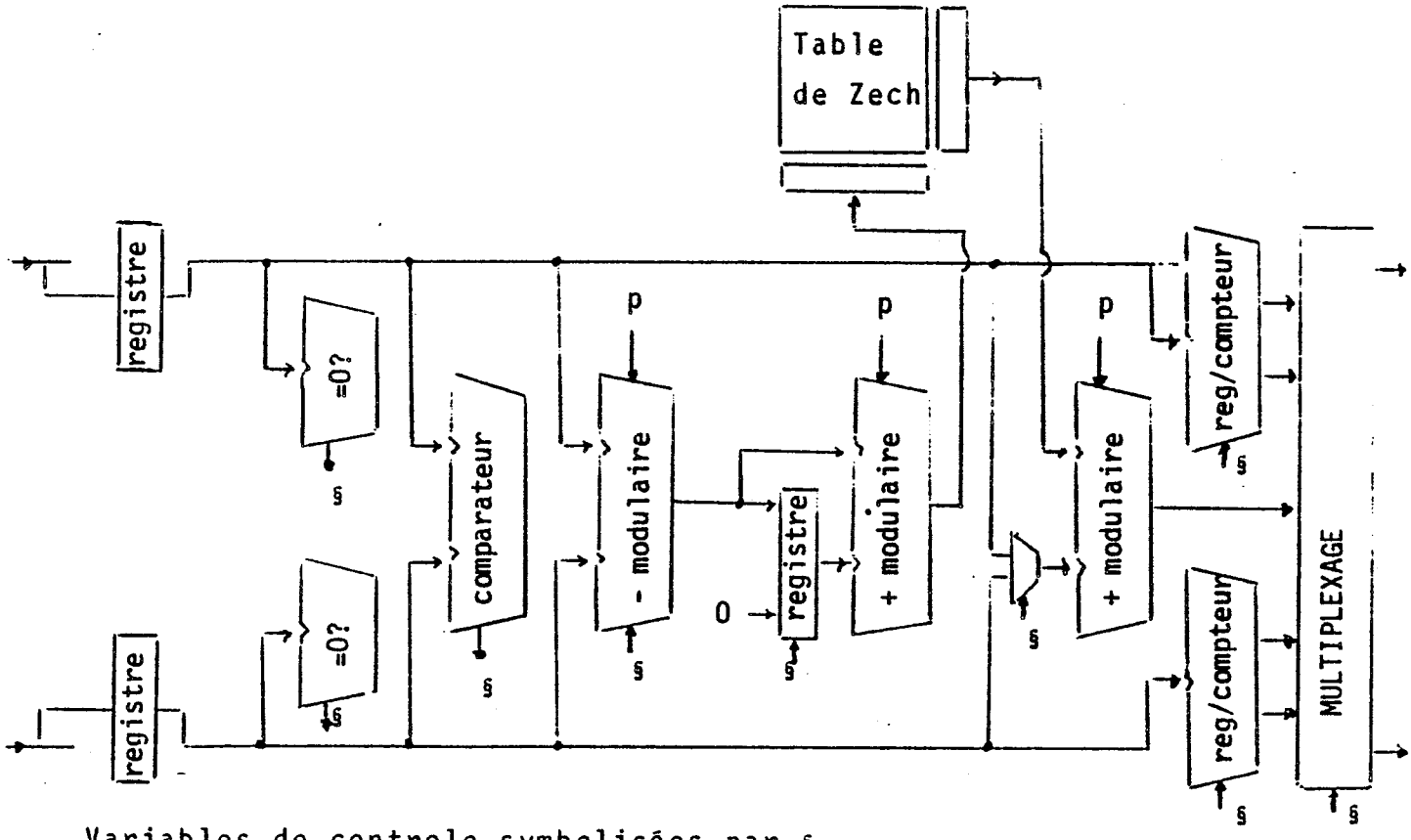


Schéma de principe
Figure 5



Variables de controle symbolisées par §
Architecture de la partie opérative
Figure 6

VLSI and Computer Algebra : The G.C.D. Example

James Davenport* & Yves Robert

Analyse Numérique, Laboratoire I.M.A.G.,

B.P. 68, 38402 SAINT MARTIN D'HERES CEDEX, France

1. INTRODUCTION

In computer algebra systems (such as REDUCE and MACSYMA), the expressions manipulated are large and complicated, for example rational functions in several variables, and hence the algorithms suffer, in general, from a relative slowness (even when they are implemented on the most powerful computers).

Could one hope to increase the capacity or speed (or both) of these systems by implementing certain basic functions in hardware? We are not talking here about completely re-writing a computer algebra system, truly a major task, but adding certain specialised processors to increase the speed of critical algorithms. The progress of VLSI technology means that such implementations are now possible.

2. GENERAL REMARKS

Consider, for the moment, the example of multiplication of two polynomials P and Q from $Z[x]$. The systolic approach introduced by Kung [8], appears attractive, and indeed he proposed a systolic array for this calculation [9]. But the polynomials found in computer algebra are by no means any old polynomials: they are often sparse, and their coefficients are not of the same size. This

---

* Present address: School of Mathematics, University of Bath, Bath, England, BA2 7AY. This work was performed while the first author was a Research Fellow of Emmanuel College, Cambridge.

DYNAMICAL SYSTEMS
AND CELLULAR AUTOMATA

ISBN 0-12-209060-8

last is a major drawback, since the size of calculation possible on a circuit is inherently limited; it is necessary to choose between two implementations of integer multiplication:
- small integers, occasionally causing overflows;
- large integers*, almost never completely used, and therefore losing the speed advantage.
In order to achieve a high-performance implementation, it seems that the circuit must use small integers, but that the algorithm must not cause overflow.

The calculation of g.c.d.s is an even more eloquent example: if we choose, after Brown [3], $P(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x + 5$ and $Q(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$, we find, depending on the exact algorithm chosen:
- 1259333879550074310093114199218750 0 (direct Euclid's algorithm);
- 869224534119352216237500 ("reduced" algorithm);
- 23552630096885020 (sub-resultant algorithm).
We started with two-digit numbers, and finished with 35, 24 or 17 digits, even though the true answer is that the polynomials are relatively prime.

To make matters worse, g.c.d. calculation is a corner-stone of much computer algebra, from the manipulation of fractions to the calculation of integrals. We could cite other examples, from integration [5] or elsewhere, but it is well-known that, in nearly all algorithms of computer algebra, "intermediate expression swell" occurs, and this complicates the search for a hardware implementation as much as it does the existing software. If we examine the methods already developed for use on conventional computers (classical algorithms, modular algorithms and p-adic algorithms), we see that a computer algebra system based on specialised circuits will have to be based on non-classical algorithms, even more than a purely software implementation has to be. For real multi-variate problems, one probably wants to adopt a sparse algorithm, such as [15].

3. A CIRCUIT TO CALCULATE THE G.C.D. OF TWO POLYNOMIALS.

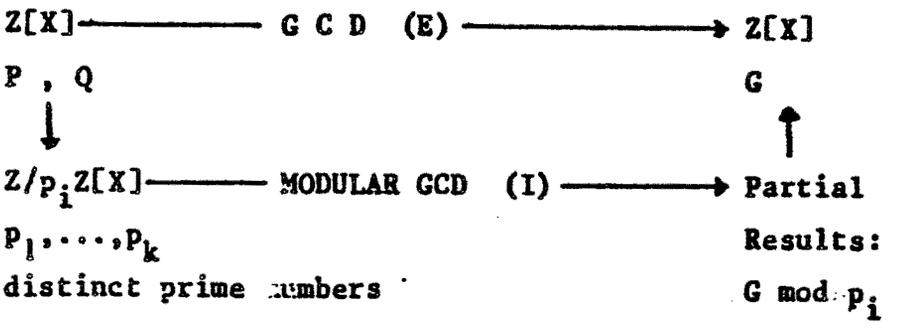
In the rest of this paper, we will describe the implementation of an algorithm for the modular calculation of the g.c.d., solving in the process the problems of size mentioned above. Our choice of this problem is not accidental: this procedure is very often invoked, and can take over 95% of the total time of a calculation [4].

---

* Making the unrealistic assumption that we can bound the size of integers appearing in a calculation before configuring the hardware.

3.1. MODULAR G.C.D.

We are working in the following context: with respect to the original problem (I), we will produce a hardware implementation of step (E), leaving the reduction and reconstruction to be performed classically.



In this approach, it is necessary to do perform several g.c.d. calculations with different choices of p, and to deduce the value of the g.c.d. over the integers from the various modular values by means of the Chinese Remainder Theorem. To explain this, we introduce the operator M_p, which performs reduction modulo p, taking Z to $Z/pZ = Z_p$ and $Z[X]$ to $Z_p[X]$. If p does not divide the leading coefficients $lc(P)$ and $lc(Q)$ of P and Q, then $M_p(\gcd(P,Q))$ divides $\gcd(M_p(P),M_p(Q))$. We say that p is *lucky* if these two polynomials are equal. There are only a finite number of unlucky primes - those that divide the resultant $\mathrm{Res}(P/G,Q/G)$, where G is the g.c.d. of P and Q over the integers [3, 10]. We give below the procedure GCD for this calculation - it is very similar to that of Brown [3] except that we use a bound based on Landau's inequality [12, 13].

```
      Procedure GCD(P,Q)
      Size:=2*Min(Max(|Coeff(P)|),Max(|Coeff(Q)|))
L1:   Base:=a prime number
      SoFar:=ModularGcd(P,Q,Base)
L2:   Degree:=deg(SoFar)
      if degree=0 then return 1
      While Base<2**Degree*Size do
L3:    p:=a prime number
       Temp:=ModularGcd(P,Q,p)
       If deg(Temp)>Degree then go to L3
       If deg(Temp)<Degree then
        Base:=p
        SoFar:=Temp
        go to L2
       SoFar:=ChineseRemainder(SoFar,Temp,Base,p)
       Base:=Base*p
      If SoFar does not divide P then go to L1
```

```
If SoFar does not divide Q then go to L1
return SoFar
```

3.2. COMPLEXITY

The aim of the complexity analysis which follows is
a) to specify the total cost of the modular g.c.d. algorithm
b) to discuss the relative cost of step (E), the only step not to be performed on the conventional computer.

Notation

We will suppose that all the prime numbers used are larger than, but close to, 2^β ($\beta = 7$ for example). We write:

$$n_P = \deg(P) + 1,\ n_Q = \deg(Q) + 1,$$

$$l_P = \max \log (P_i),\ l_Q = \max \log (Q_i),$$

$$n = \max(n_P,n_Q),\ l = \max(l_P,l_Q),\ d = \deg(\gcd(P,Q)).$$

We suppose for simplicity that P and Q are monic. All logarithms are to base 2.

Asymptotic Formulae

The following table summarises the various bounds that are relevant

unlucky primes	N_i	$2n(\frac{1}{2} \log 2n + l)/\beta$
lucky primes necessary	N_f	$[(d-1)(\frac{1}{2} \log n + l) + 1]/\beta$
Chinese Remainder cost	C_{CR}	$d\beta N_f O(\log^{3+\varepsilon} N_f + \log^{2+\varepsilon} \beta \log N_f)$

N_i comes from bounding the size of the resultant [10, 13], and improves on that found in [3]. N_f is deduced from theorem 4 of [13] (see also theorem 2 of [12]). The cost of the Chinese Remainder process is deduced assuming a tree-based method (though we wrote a linear method in the program above) - see [1] section 8.11 or [5] chapter 2.

Total cost

We suppose that all calculation modulo p takes a fixed amount of time (we have chosen $\beta = 7$, and most computers use fixed constant time for 16 or 32 bits). This gives, for the three stages:
- reduction modulo p: $(N_f + N_i)(n_P l_P + n_Q l_Q)$
- modular g.c.d.: not performed on the main computer;
- Chinese Remainder Theorem: $d\beta N_f O$(logarithms).

Furthermore, step (E) takes time $O(n)$ (assuming enough hardware is available), and so, before the last reductions are finished, the first inputs will be ready for the Chinese Remainder process. The reduction is bounded by $O(n^2 \log nl, n^2 l^2)/\beta$, and the re-construction by $d^2 lO$(logarithms), and hence we have an overall bound of $O(n^2 l^2, n^2 l$ logarithms).

Some Pragmatic Remarks

The reduction is the most expensive step, and the factor of $1/\beta$ means that progress in VLSI technology, by permitting the treatment of larger numbers on one chip, will diminish N_f and N_i, leading directly to a cheaper implementation.

In practice, unlucky primes are rare, and the bounds given for the coefficients, and hence N_f, are pessimistic. Brown [3], proposes a "heuristic" bound of l, and one could well stop at this point and see if one indeed had a true g.c.d. Of course, l may be larger than the true bound we compute, and a better heuristic bound might be $\min(l_P, l_Q)$.

Finally, we must emphasise that the possibility of unlucky primes, small though it may be, means that we can not pre-determine the primes to use, and hence can not pre-calculate the inverses in the tree-structure for the Chinese Remainder Theorem.

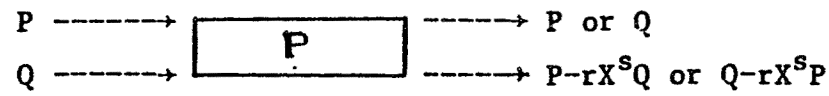
3.3. ALGORITHM AND CIRCUIT SPECIFICATION

For the g.c.d. calculation, we use the method of degree reduction, related to Euclid's algorithm as subtraction is to division:

```
while d(P)*d(Q) > 0 do
  if d(A) ≥ d(B)
       then A := A - B * x**(d(A)-d(B))*(lc(A)/lc(B))
       else B := B - A * x**(d(B)-d(A))*(lc(B)/lc(A))
if A = 0 then GCD:=B else GCD:=A
```

Each step of the algorithm corresponds to a partial elimination, and it is clear from this form that the algorithm lends itself to a certain parallelism. The execution can be divided into elementary operations performed by identical

processors:



Better still, we can avoid the multiplication by a power x^s (which consists of "shifting" the coefficients of one of the polynomials by s possibilities): if after an elimination step, we have decreased the degree by more than one, we decide to output a zero as the leading coefficient of that polynomial. In this case, the role of the box P consists simply in:
- eliminating the zero coefficient;
- decreasing the degree by one;
- transmitting the other polynomial unchanged.

We thus obtain the desired pipe-line implementation in using a vector of identical processors, whose operation is outlined in figure 1.

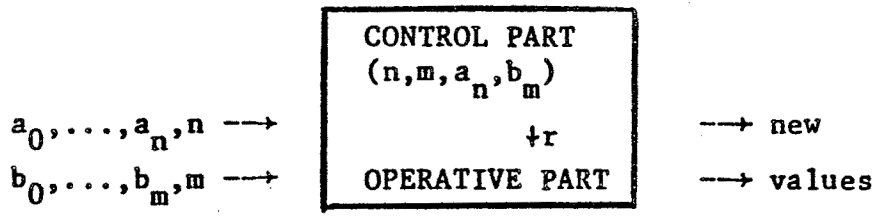


Figure 1

1) If n or $m = 0$, transmit A and B without modification.
2) Otherwise, if a_n or $b_m = 0$, shift one of the polynomials and transmit the other. A and B are never simultaneously zero. For example, if $a_n = 0$, we suppress a_n, set n to $n-1$, and transmit B unchanged.
3) Otherwise, perform a partial elimination, e.g. if $n \geq m$, $a_i := a_i - r*b_{i+m-n}$, where $r = a_n/b_m$.

We can simplify the presentation somewhat, in remarking that (2) is the same step as (3), but with $r = 0$ similarly, we write $r = 0$ in case 1.

3.4. USE OF MODULAR CALCULATION.

We have seen that calculation modulo a prime number p is indispensable for avoiding the problems of intermediate expression swell. All the calculations are then conducted modulo p, where p is a known *fixed* (the architecture is greatly simplified if p is known when the circuit is designed) prime number. Let

us make two preliminary remarks.
i) Modular calculation is about two times as expensive as integer calculation (e.g. the calculation of $a + b$ (modulo p) requires the computation of $a + b$ and $a + b-p$). It is true that the choice $p = 2^n \pm 1$ greatly simplifies the calculations, but we have to be able to treat arbitrary p.
ii) The principal difficulty is in the division. In general one either uses the extended Euclidean algorithm in Z_p, or one uses a table of inverses.

If we are prepared to use a table, though, we can give a quick and elegant solution for the computation of the four elementary operations +,-,*,/ modulo p.

An integer modulo p is usually represented as an element of the set $E_p = \{0,1,..,p + 1\}$. Let us fix a primitive root K of p (i.e. a generator of the multiplicative group $Z_p-\{0\}$). Then every non-zero element of E_p can be written uniquely in the form K^i $(0 \le i \le p-2)$. If we attach the special representation * to 0, we can represent the element K^i of E_p by i. Multiplication and division are then transformed into addition and division, according to the rules :

$$\bar{i*j} = \begin{cases} * \text{ if } i \text{ or } j = * \\ i + j \bmod p-1 \text{ otherwise} \end{cases}$$

$$\bar{i/j} = \begin{cases} * \text{ if } i = * \\ \text{undefined if } j = * \\ i-j \bmod p-1 \text{ otherwise} \end{cases}$$

For addition (and subtraction) we use *Zech logarithms* [14]: define $Z(u)$ by

$$K^{Z(u)} = 1 + \lambda \text{ for } u \in \{0,1,..,p-2\}-\{\frac{p-1}{2}\},$$

$$Z(\frac{p-1}{2}) = * \text{ and } Z(*) = 0.$$

One deduces the following rules for addition:

$$\overline{i+j} = \begin{cases} j \text{ if } i = * \\ i \text{ if } j = * \\ * \text{ if} j-i = \pm(p-1)/2 \\ i + Z(j-i) \bmod p-1 \text{ otherwise} \end{cases}$$

and similar rules for subtraction.

These representations modulo a prime number p of n bits have $n + 1$ bits, since a special bit is reserved for *. The use of this one-dimensional table of Zech logarithms enables us to eliminate all the bivariate operations. In the units λ^2 [11], we have the following costs for the various components:

Component	name	surface
n-bit multiplier	M	$2^{12}n^2$
n-bit adder	A	$2^{12}n$
n-bit table	T	$3n2^{n+6}$

If we assume that division is about the same surface as multiplication, and that mod p units are twice the size of integer ones, we have that the cost of a conventional circuit (multiplier, divisor, adder) is $C(n) = 4M + 2A$, while the cost for a Zech-based solution is $Z(n) = T + 6A$. The ratio is

$$R(n) = \frac{Z(n)}{C(n)} = 3\frac{2^n + 2^7}{n2^8 + 2^7}.$$

This is an increasing function of n, but takes on the value 9/17 at $n = 8$, meaning that the Zech representation uses just over half the area.

4. CONCLUSION

In [2], Brent and Kung present systolic architectures for the calculation of greatest common divisors of polynomials. The high-level architecture of their circuits is very close to those we present in section 3.3. But our modular approach has a double interest:
-it is the only proposal which solves the problem of "intermediate expression swell", a *sine qua non* of computer algebra algorithms;
-it permits, at the lowest level, an original and parallel implementation of the architecture of each cell.
References [6] and [7] complete this study.

The algorithmic description which we have given shows that it is possible, with today's technology, to build VLSI units specialised for the modular gcd algorithm. Obviously this is a first step - what other VLSI algorithms can be designed for computer algebra? We already know, after [7], that this method can be extended to compute not only the g.c.d. G of P and Q, but also polynomials A and B such that $AP + BQ = G$ (analogous to the extended Euclidean algorithm).

5. REFERENCES

[1] Aho,A.V., Hopcroft,J.E. & UllmanJ.D., "The Design and Analysis of Computer Algorithms". Addison-Wesley, Reading, Mass., 1974.

[2] Brent,R.P. & Kung,H.T., Systolic VLSI arrays for linear-time gcd computation. *In* "VLSI 83" (Eds. F. Anceau & E.J. Aas) pp. 145-154. Elsevier Science Publishers, 1983.

[3] Brown,W.S., On Euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM 18*(1971) pp. 478-404.

[4] Davenport,J.H., "On the Integration of Algebraic Functions" (Lecture Notes in Computer Science 102). Springer-Verlag, Berlin-Heidelberg-New York, 1981.
[5] Davenport,J.H., "Intégration Formelle", RR IMAG Grenoble 375, June 1983.
[6] Davenport,J.H., "VLSI et Calcul Formel", RR IMAG Grenoble 357, March 1983.
[7] Davenport,J.H.& Robert,Y., "PGCD et VLSI", RR IMAG Grenoble 358, March 1983.
[8] Kung,H.T., "Why systolic architectures", TR Carnegie-Mellon University, 1981.
[9] Kung,H.T., Use of VLSI in algebraic computation: some suggestions. *In* "Proc. SYMSAC 81" (Ed. P.S. Wang) pp. 218-222. ACM, New York, 1981.
[10] Loos,R., Generalized polynomial remainder sequences. *In* "Computer Algebra Symbolic and Algebraic Computation" (Ed. B. Buchberger et al.) pp. 115-137. Springer-Ve .g, Vienna, 1982.
[11] Mead,C. & Conway,L., "Introduction to VLSI Systems". Addison-Wesley, Reading, Mass., 1980.
[12] Mignotte, M., Some inequalities about univariate polynomials.
[13] Mignotte, M., Some useful bounds. *In* "Computer Algebra Symbolic and Algebraic Computation" (Ed. B. Buchberger et al.) pp. 259-263. Springer-Verlag, Vienna, 1982.
[14] Zimmer,H.G., "Computational Methods in Number Theory" (Lecture Notes in Mathematics 272) Springer-Verlag, Berlin-Heidelberg-New York, 1972.
[15] Zippel,R.E., Newton's iteration and the sparse Hensel algorithm. *In* "Proc. SYMSAC 81" (Ed. P.S. V ng) pp. 68-72. ACM, New York, 1981.

Using a silicon compiler for Computer Algebra

Robert Jamier, Ahmed Jerraya & Yves ROBERT
Laboratoire TIM3/INPG
46 avenue Félix Viallet, 38031 Grenoble Cedex

Actes du Congrès "Le Calcul ... Demain",
2-6 Décembre 1985, Grenoble
Masson éditeur

USING A SILICON COMPILER FOR COMPUTER ALGEBRA

Robert JAMIER, Ahmed JERRAYA, Yves ROBERT
Laboratoire TIM3 - INPG
46 avenue Félix Viallet, 38031 GRENOBLE CEDEX

Résumé

On présente la réalisation VLSI, à l'aide du compilateur de silicium CAPRI, de la partie opérative d'un circuit calculant le PGCD modulaire de deux polynômes à coefficients entiers

Abstract

We demonstrate the use of the CAPRI silicon compiler for the design of a VLSI chip for the modular calculation of the GCD of two polynomials over Z.

The emergent needs of Scientific computation make it imperious to increase dramatically the capacity or speed (or both) of Computer Algebra systems. The progress of VLSI technology means that it is now possible to implement the critical algorithms by adding certain specialized processors to the systems (Kun). However, it is well-known that "intermediate expressio swell" occurs in nearly all algorithms for Computer Algebra, and this complicates the search for a hardware implementation as much as it does the existing software (Dav).

GCD calculation is a corner-stone of much computer algebra, from the manipulation of fractions to the calculation of integrals (Dav). In this paper, we describe the use of a silicon compiler to implement an algorithm for the modular calculation of the GCD of two polynomials, solving in the process the problems of size mentioned above.

The algorithm we design matches the requirements of the silicon compiler (standard library, variable types are limited to registers, ...) without sacrifying the performances.

1. BRIEF DESCRIPTION OF THE ALGORITHM

We are working in the context of figure 1. Step (1) is the only one not to be performed on the conventional computer. A detailed complexity analysis is give in (DaR) to demonstrate the full interest of such a modular approach.

For the GCD calculation, we use the method of degree reduction, related to Euclid's algorithm as division is to soustraction. Each step of the algorithm corresponds to a partial elimination, and it is clear from this form that the algorithm lends itself to a certain parallelism. The execution is divided into elementary operations performed by identical processors, whose operation is depicted in the figure 2.

Calculation modulo a prime nomber p is a "sine-qua-non" for eliminating the intermediate expression swell. Moreover, we have to treat arbitrary p (i.e. not only $p = 2^n - 1$ whose choice greatly simplifies the calculations). The principal difficulty is the division. In general, one either uses the extended Euclid algorithm over Z_p, or one uses a table of inverses. Here, we give a quick and elegant solution for the four elementary operations, using only a one

-dimensional table and eliminating all the bivariate operations. We refer to the figure 3 for details on our implementation (see (Dar)), which is based upon Zech's logarithms (Zim). Such a design is both performant (it uses about half the area of that of a conventional approach (DaR)) and well-suited to using the CAPRI silicon compiler

2. THE CAPRI SILICON COMPILER

The CAPRI silicon compiler produces circuit layouts from an IRENE description (Anc). IRENE is a register transfer description langage. CAPRI deals with microprocessor-like circuits. It uses a template architecture, the circuits generated are composed from a data path and a control part (see figure (4)).

The compiler is in progress, it generates the whole data path layout, and the control section description. The circuit is finished by a direct design using a PLA generator and a silicon assembler. Most of the design time is spent for the circuit description validation.

The data path (JaJ) allows parallel execution of the basic actions. The user defines the degree of parallelism when specifying the description. This permits to control the speed of the circuit produced by the compiler: more parallelism increases the circuit speed (decreases the number of machine cycles needed for each instruction) but increases also the chip area (e.g. the circuit must have two adders to perform in parallel two additions).

The basic machine cycle (register transfer time) is enforced by the compiler libray ,it is about 400 ns. The data path can perform three types of operations:

- simple transfers,
- I/O operations,
- logic and arithmetic operations.

The compiler has an arithmetic and logic operator library. It does not include multiplication and division, but these area-consuming operators have been eliminated from our algorithm.

We use the compiler to generate two versions of the GCD circuit data path. Version (a) requires a single ALU, it is slower than version (b) which requires three ALUs and therefore can execute three

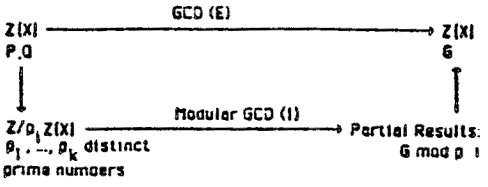
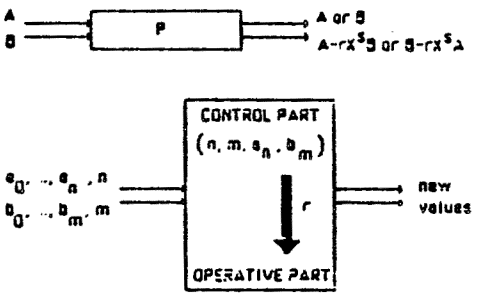


Figure 1

while d(A)≠d(B) > 0 do
  if d(A) ≥ d(B)
    then A := A - B * x**(d(A)-d(B)) * (lc(A)/lc(B))
    else B := B - A * x**(d(B)-d(A)) * (lc(B)/lc(A))
If A = 0 then GCD := B else GCD := A

Degree reduction algorithm



1) if n or m = 0, transmit A and B without modification
2) otherwise, if a_n or b_m = 0, shift one of the polynomials and transmit the other. A and B are never simultaneously zero. For example, if a_n = 0, we suppress a_n, set n to n-1 and transmit B unchanged
3) otherwise, perform a partial elimination, e.g. if n≥m, $a_i := a_i - r*b_{i+m-n}$, where $r = a_n/b_m$

Figure 2

USE OF MODULAR CALCULATION

An integer modulo p is usually represented by an element of the set
$E_p = \{0, 1, \ldots, p-1\}$.

We fix a primitive root K of p (i.e. a generator of the multiplicative group $Z_p^*=\{0\}$). Then every non-zero element of E_p can be written uniquely in the form K^j, $0 \le j \le p-2$. If we attach the special representation * to 0, we can represent the element K^j of E_p by j. Multiplication and division are then transformed into addition and substraction, according to the rules:

$$i \tilde{*} j = \begin{cases} * \text{ if } i \text{ or } j = * \\ i+j \bmod p-1 \text{ otherwise} \end{cases}$$

$$i \tilde{/} j = \begin{cases} * \text{ if } i = * \\ \text{undefined if } j = * \\ i-j \bmod p \text{ otherwise} \end{cases}$$

For addition (and substraction) we use Zech logarithms: define Z(u) by

$$K^{Z(u)} = 1 + K^u \text{ for } u \in \{0, 1, \ldots, p-2\} \setminus \{(p-1)/2\}$$
$$Z((p-1)/2) = * \text{ and } Z(*) = 0$$

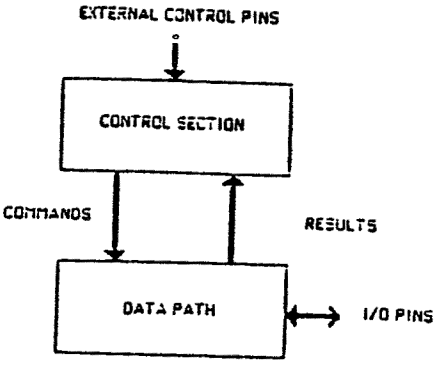
One deduces the following rules for addition

$$i \tilde{+} j = \begin{cases} j \text{ if } i = * \\ i \text{ if } j = * \\ * \text{ if } j-i = (p-1)/2 \\ i + Z(j-i) \text{ otherwise} \end{cases}$$

and similar rules for substraction.

These representations modulo a prime number p of n bits have n+1 bits, since a special bit is reserved for *. The use of this one-dimensional table of Zech logarithms enables us to eliminate all the bivariate operations.

Figure 3



Architecture Template
Figure 4

logic or arithmetic operations in parallel. The behavioral descriptions of the two versions have been produced by the compiler, leading to the data path structures shown in the figure 5. Finally, the layout of version (a) is presented in the figure 6.

REFERENCES

(Anc) F. ANCEAU, A design methodology and a silcon compiler for circuits specified by algorithmes, Proc. Third CALTECH Conf. on VLSI, R. Bryant ed.n Computer Sc. Press 1983.

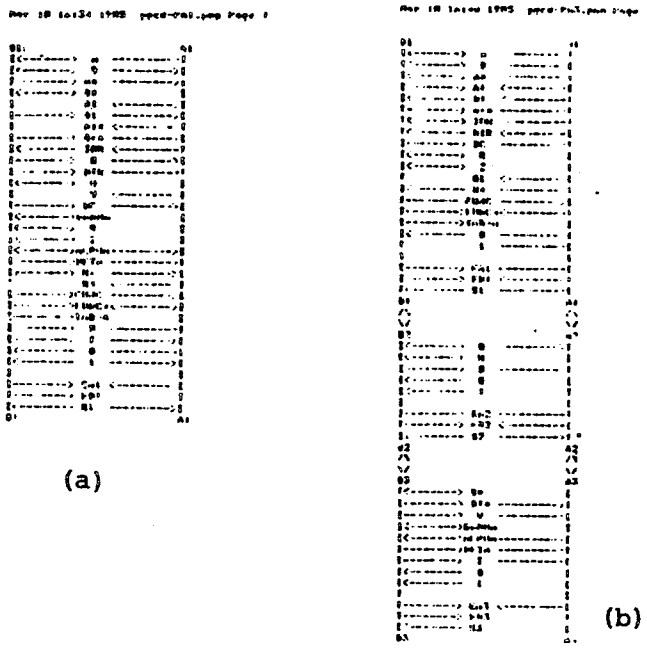
(DaR) J.H. DAVENPORT, Y. ROBERT, VLSI and computer algebra : the GCD example, Dynamical systems and Cellular Automata, J. Demongeot et al. eds, Academic Press, 1985.

(Dav) J.H. DAVENPORT, VLSI et calcul formel, RR IMAG 357, March 1983.

(JaJ) R.JAMIER, A.A. JERRAYA, The Apollon data-path silicon compiler, Proc ICCD 85, New-York, Oct. 1985

(Kun) H.T. KUNG, Use of VLSI in algebraic computation: some suggestions, in Proc. SYMSAC 81, P.S. Wang ed., ACM, NewYork, 1981.

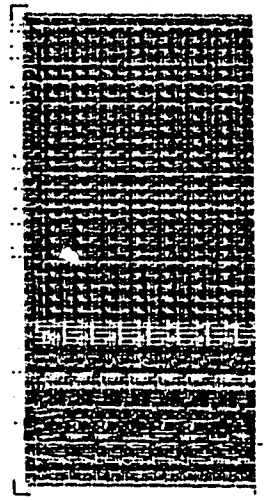
(Zim) H.G. ZIMMER, Computational methods in number theory, Lect. Notes in Math. 272, Springer Verlag, 1972.



(a) (b)

Data path structures

Figure 5



Layout of version (a)

Figure 6

SOUS-CHAPITRE 6.4.

Calcul en parallèle sur des réseaux systoliques

Some Systolic Algorithms for Parallel Computation

Y. Robert
(IMAG, Grenoble)

M. Tchuente
(IMAG, Grenoble)

1 INTRODUCTION

Le modèle systolique introduit par H.T.KUNG /2/ permet de donner aux algorithmes usuels une structure à fort degré de parallélisme, et adaptée aux exigences de la technologie VLSI /3/ .

La première étape dans la conception d'un algorithme systolique pour le calcul d'une fonction

$$F: x=(x_1,\dots,x_m) \longrightarrow (f_1(x),\dots,f_n(x))$$

consiste à mettre chaque f_i sous la forme

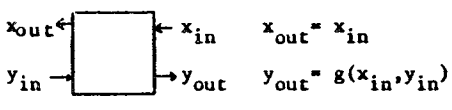
$$f_i(x) = b_i + \Sigma_{(j_1,\dots,j_r)\in V_i} g_{i,j_1,\dots,j_r}(x_{j_1},\dots,x_{j_r})$$

On cherche alors un réseau où chaque cellule a r+1 entrées et r+1 sorties :
- toutes les variables-données x_i circulent d'une cellule à l'autre sans changer de valeur
- chaque variable-résultat y_i, initialisée à b_i, circule de manière à rencontrer tous les r-uplets dont elle a besoin, et, à la rencontre de $(x_{j_1},\dots,x_{j_r})$, on a

$$y_i := y_i + g_{i,j_1,\dots,j_r}(x_{j_1},\dots,x_{j_r})$$

y_i se calcule donc par accumulations successives

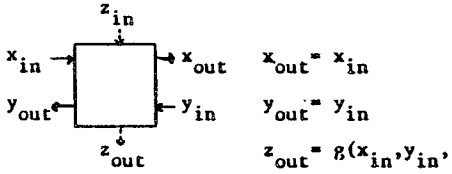
Ainsi, pour calculer la fonction F,on la décompose en opérations élémentaires indépendantes pouvant être effectuées simultanément . Nous distinguerons deux types de cellules élémentaires dans la suite, selon que nous utiliserons un seul (r=1) ou deux (r=2) flots de variables-données :



$x_{out} = x_{in}$

$y_{out} = g(x_{in},y_{in})$

type 1



$x_{out} = x_{in}$

$y_{out} = y_{in}$

$z_{out} = g(x_{in},y_{in},z_{in})$

type 2

Soit (P) un problème donné . Notons t(k) le temps nécessaire à la résolution de (P) sur un réseau comportant k processeurs . Nous mesurons l'efficacité du parallélisme par la quantité

$$e = \frac{t(1)}{k.t(k)}$$

Comme on peut toujours simuler sur un seul processeur en un temps k.t(k) un algorithme à k processeurs, on a $e \leq 1$.

Nous présentons ici deux algorithmes systoliques
- le premier, consacré à l'évaluation de récurrences vectorielles (généralisation du cas scalaire traité dans /2/) se décompose naturellement en opérations élémentaires . La difficulté de conception réside dans l'organisation du réseau et la discussion de la complexité (bornes sur e).
- le second modélise le calcul de la médiane d'une suite unidimensionnelle . La non-linéarité impose cette fois la recherche préalable d'une décomposition en opérations élémentaires .

2 RECURRENCES VECTORIELLES /4/

Soient A et B deux matrices d'ordre n . On s'intéresse
- au calcul de la suite vectorielle $(y^i)_{i\geq 1}$ définie par

$$y^i = Ax^i + Bx^{i-1}$$

à partir d'une suite $(x^i)_{i\geq 0}$ de vecteurs à n composantes
- à l'évaluation de la récurrence

$$x^i = Ax^{i-1} + Bx^{i-2} + b^i$$

$b^0 = x^0$, $b^1 = x^1$, et $(b^i)_{i\geq 2}$ étant donnés .

La linéarité du problème conduit à choisir des cellules élémentaires de type 1, n'effectuant que des opérations scalaires :

$$g(x_{in},y_{in}) = y_{in} + \lambda_g . x_{in} .$$

Le nombre de ces cellules sera nécessairement $2n^2$.

2.1 Suites vectorielles

La figure 1 explicite un réseau solution pour n = 3 . Son efficacité est e = 1/2 .

2.2 Récurrences linéaires

Chaque x^i_j va d'abord jouer le rôle de variable-résultat , puis ayant accumulé sa valeur définitive, celui de variable-donnée pour le calcul des

x_k^{i+m} , m = 1,2 et 1 ≤ k ≤ n .

Supposons sans perte de généralité que le réseau comporte n lignes de 2n cellules, et que chaque x_j^i reste sur la ligne j au cours de son calcul (mais on ne fera aucune hypothèse sur la trajectoire de x_j^i une fois qu'il est calculé) .

Lemme

(i) x_k^h se calcule après tous x_j^i , 1 ≤ j ≤ n , i ≤ h-1

(ii) l'écart d_j (dans le temps) entre x_j^i et x_j^{i-1}, qui est aussi l'écart entre b_j^i et b_j^{i-1} puisque les sorties sont délivrées au même rythme que les entrées, est indépendant de j .

Proposition

(i) tout réseau-solution a une efficacité e ≤ 1/n+1.

(ii) tout algorithme où les b_j^i , 1 ≤ j ≤ n, entrent en même temps dans le réseau, a une efficacité
$$e \leq 1/(n+\lfloor n/2 \rfloor)$$
et cette borne peut être atteinte .

(iii) un mode d'approche analytique assure d'une solution régulière d'efficacité e = 1/2n . Une telle solution est représentée figure 2 pour n = 3 .

2.3 Remarques

Ces résultats se généralisent immédiatement au calcul de

$$x^i = A^{(1)}x^{i-1} + A^{(2)}x^{i-2} + \ldots + A^{(p)}x^{i-p} + b^i$$

D'autre part une application intéressante est la modélisation d'évolutions discrètes issues de l'automatique /5/.

3 LISSAGE PAR MEDIANE

Le lissage par médiane est une technique de filtrage (utilisée en traitement du signal) qui consiste à remplacer chaque point de l'image par la valeur médiane prise à l'intérieur d'un certain voisinage local . Ce problème a été résolu par A.L. FISHER /1/. Nous proposons ici un algorithme dans le cas unidimensionnel qui n'utilise que des cellules de structure élémentaire . La méthode peut s'étendre au calcul de formes homogènes dans un anneau commutatif .

La médiane de 2k-1 valeurs a_1 , ... , a_{2k-1} est par définition le k-ième élément obtenu quand on ordonne totalement $\{a_1,\ldots,a_{2k-1}\}$. L'idée est d'exprimer cette médiane par la formule

$$\Sigma_{i_1<\ldots<i_k}\ a_{i_1}.a_{i_2}\ldots a_{i_k}$$
où a + b = max (a,b)
a . b = min (a,b)

Plus généralement, on introduit les fonctions

$$F_{n,k}(a_1,\ldots,a_k) = \Sigma_{i_1<\ldots<i_n}\ a_{i_1}.a_{i_2}\ldots a_{i_n} \quad (n\leq k)$$

Pour le calcul de $F_{k,2k-1}$, on cherche alors une construction récursive à partir des fonctions $F_{i,j}$ où i<k et j<2k-1 : si on dispose de réseaux modélisant le calcul de ces fonctions, on saura exhiber un réseau-solution pour le calcul de $F_{k,2k-1}$ moyennant l'utilisation de processeurs de retard (dont on cherchera à minimiser le nombre) .

3.1 Médiane à trois

$$F_{2,3}(a_1,a_2,a_3) = F_{2,2}(a_1,a_2)+F_{2,2}(a_1,a_3)+F_{2,2}(a_2,a_3)$$

Cette écriture invite à choisir des cellules élémentaires de type 1 avec

$$g(x_{in},y_{in}) = x_{in} \cdot y_{in}$$
ou
$$g(x_{in},y_{in}) = x_{in} + y_{in}$$

et conduit au réseau de la figure 3 .

3.2 Médiane à cinq

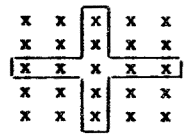
$$F_{3,5}(a_1,a_2,a_3,a_4,a_5) = a_1 \cdot F_{2,4}(a_2,a_3,a_4,a_5) + a_2 \cdot F_{2,3}(a_3,a_4,a_5) + a_3 \cdot F_{2,2}(a_4,a_5)$$

Cette écriture invite à choisir des cellules élémentaires analogues aux précédentes; mais cette fois, nous aurons besoin aussi de cellules de type 2 avec

$$g(x_{in},y_{in},z_{in}) = z_{in} + x_{in}.y_{in}$$
ou
$$g(x_{in},y_{in},z_{in}) = z_{in} \cdot x_{in} \cdot y_{in}$$

Nous proposons figure 4 un réseau-solution, dont nous donnons – vu sa complexité – une preuve de validité .

Cet algorithme s'applique immédiatement au lissage par médiane d'un tableau rectangulaire en utilisant le voisinage de VON NEUMANN :



4 CONCLUSION

Comme le montrent les divers exemples traités, les réseaux systoliques constituent un modèle qui combine les deux types classiques de parallélisme :
- le parallélisme dans l'espace: pour calculer une fonction F, on la décompose en opérations élémentaires indépendantes pouvant être effectuées simultanément .
- le parallélisme dans le temps (Pipe-line): lorsqu'on doit calculer $F(x_t)$, t=0,1,... on peut commencer le calcul de $F(x_t)$ avant d'avoir terminé celui de $F(x_{t-1})$.

Ces deux caractéristiques permettent dans la plupart des cas d'obtenir des algorithmes en temps réel (c'est à dire où les sorties sont délivrées au même rythme que les entrées). De plus, la régularité et la modularité des réseaux systoliques permettent d'envisager l'implémentation sur silicium de configurations comportant un très

grand nombre de cellules .

REFERENCES

/1/ A.L. FISHER : "Systolic algorithms for running order statistics in signal and image processing", Technical Report (1981) C.M.U.

/2/ H.T. KUNG : "The structure of parallel algorithms", Advances in Computers 19 (1980), pp. 65-112.

/3/ H.T. KUNG : "Why systolic architechtures", Technical Report (1981) C.M.U.

/4/ Y.ROBERT et M.TCHUENTE : "Organisation du calcul en parallele de récurrences vectorielles", Rapport de Recherche (1981) I.M.A.G. Grenoble.

/5/ Y.ROBERT et M.TCHUENTE : "Calcul en parallèle sur des réseaux systoliques", S.A.N.G. n°368 (1981) I.M.A.G. Grenoble.

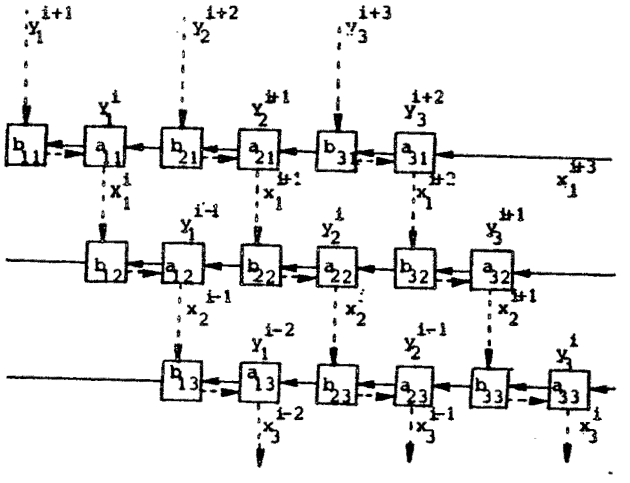


Figure 1 : $Y^i = AX^i + BX^{i-1}$

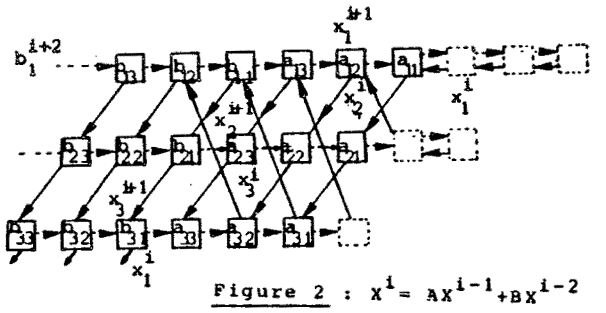


Figure 2 : $X^i = AX^{i-1} + BX^{i-2}$

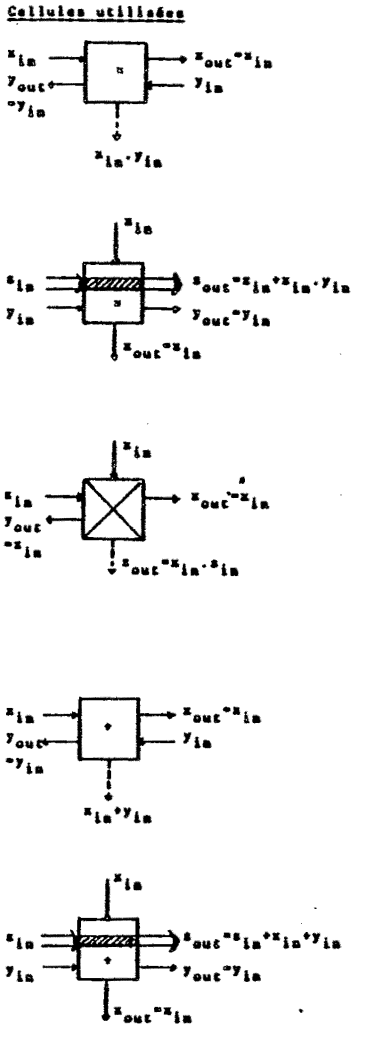
Commentaires pour la figure 1

Les cellules sont de type 1. Les variables-données circulent horizontalement (en trait plein) et les variables-résultats circulent selon les traits pointillés de manière à accumuler leur valeur définitive .

Commentaires pour la figure 2

Les variables-résultat, notées x_j^i , circulent horizontalement sur la ligne j; quand elles ont accumulé leur valeur définitive, elles se transforment en variables-données en rebondissant sur les processeurs de retard (en pointillé) pour circuler alors selon le trait plein (notation x_j^i)

Cellules utilisées



Une entrée manquante est supposée constante, et égale à -∞ ou +∞ suivant le cas

Commentaires pour la figure 3

A l'étage du haut, les a_i circulent horizontalement, et croisent a_{i-1} et a_{i-2}; les produits crées sont accumulés à l'étage du bas.

Commentaires pour la figure 4

Le premier étage permet de former les produits $a_i.a_j$; les autres étages ont un double rôle:
- effectuer des produits

$$(a_i.a_j).(a_i.a_k) = a_i.a_j.a_k$$

- accumuler ces produits dans la colonne centrale

Preuve de validité de la médiane à cinq

Pour s'assurer que le réseau délivre en sortie le résultat annoncé, une justification formelle consiste à expliciter les valeurs des différentes variables du réseau en se basant sur l'invariance par translation .

$a = a_{i+4}.a_{i+5}$

$b = a$

$b' = a_{i+4}.a_{i+6}$

$c = (a_{i+3}.a_{i+4}.a_{i+5})+(a_{i+3}.a_{i+4}.a_{i+6})$

$c' = (a_{i+2}.a_{i+3}.a_{i+4})+(a_{i+2}.a_{i+3}.a_{i+5})$
$+(a_{i+2}.a_{i+3}.a_{i+6})$

$d = a_{i+3}.a_{i+6}$

$d' = a_{i+3}.a_{i+4}.a_{i+5}$

$d'' = d$

$e = d$

$e' = a_{i+3}.a_{i+5}$

$f = a_{i+2}.a_{i+6}$

$f' = a_{i+2}.a_{i+4}.a_{i+5}$

$f'' = a_{i+2}.a_{i+4}$

$g = e'$

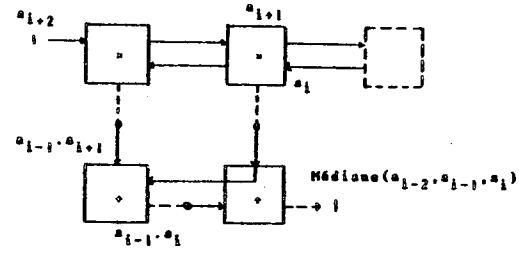
$h = d'$

$h' = (a_{i+1}.a_{i+2}.a_{i+3})+(a_{i+1}.a_{i+2}.a_{i+4})$
$+(a_{i+1}.a_{i+2}.a_{i+5})+(a_{i+1}.a_{i+3}.a_{i+4})$
$+(a_{i+1}.a_{i+3}.a_{i+5})+(a_{i+2}.a_{i+3}.a_{i+4})$
$+(a_{i+2}.a_{i+3}.a_{i+5})+(a_{i+2}.a_{i+4}.a_{i+5})$

$h'' = a_{i+1}.a_{i+4}.a_{i+5}$

$k = a_{i+2}.a_{i+5}$

Remerciements Nous remercions Monsieur le Professeur C.BELLISSANT, de l'I.M.A.G. Grenoble, qui est à l'origine d'une partie de ce travail .



Médiane à trois

Figure 3

Figure 4 Médiane à cinq

AUTHORS: Y.CLAUZEL Computer Architecture Group
Y.ROBERT and M.TCHUENTE, Numerical Analysis Group
Laboratory "IMAG"
BP 68 / F 38402 SAINT MARTIN D'HERES CEDEX
Tel +33 76 548145 ext 407,541 or 547

TITLE: A CASE STUDY FOR THE DESIGN AND LAYOUT OF SYSTOLIC ARCHITECTURES

Abstract: We present systolic networks for median filtering meeting the "systolic schema" developped by KUNG/LEISERSON(9); then we propose a specimen layout. Finally we discuss the important notion of efficiency (ie, degree of parallelism) in systolic architectures .

Résumé: On présente une architecture systolique (formulation théorique et dessin des masques) pour le lissage par médiane. On discute ensuite le concept d'efficacité dans les réseaux systoliques .

A CASE STUDY FOR THE
DESIGN AND LAYOUT OF
SYSTOLIC ARCHITECTURES

..R. N° 369

Y. CLAUZEL
Y. ROBERT
M. TCHUENTE

mars 1983

Introduction

We deal with the systolic model which has been introduced and developped by KUNG(6). We refer to (6) for the performances of such a model.

This paper is divided into three parts: the first one is devoted to a recursive algorithmic approach to median filtering, whereas the second one briefly presents a specimen layout for such computations. The third part discusses an evaluation of the degree of parallelism which a systolic network can achieve .

1) Systolic networks for Median Smoothing

Median smoothing is a filtering technique widely used in digital signal and image processing, which consists of replacing each sample value in a grid by the median of the sample values within some local neighbourhood. As pointed out in (5), median filtering - as opposed to convolution - does not introduce intermed te values not found in the original pattern .

This problem has already been solved by FISHER(5), but:
- the given network delivers the sequence completely ordered (that is, much more than asked)
- the basic cell of the array is rather complicated and needs a significant control part .

We propose here another solution, based on the careful and regular arrangement of a simple basic cell, meeting thus the "systolic schema" developped by KUNG/LEISERSON(9). Our idea consists of
-i) expressing the median of 2k-1 values $a_1,\ldots,a_{2k-1}$ by the well-known formula $F_{2k-1,k}(a_1,\ldots,a_{2k-1})$ where, denoting by + (resp.) the maximum (resp:minimum) operation, we have

$$F_{n,k}(a_1,\ldots,a_n) = \sum_{i_1<i_2<\ldots<i_k} a_{i_1}.a_{i_2}.\ \ldots\ a_{i_k}$$

for any integers n.k with n>k>1

-ii) computing it recursively according to the scheme

$$F_{2k-1.k}(a_1,\ldots,a_{2k-1}) = \sum_{1\le i<k-1} F_{k,i}(a_1,\ldots,a_k)\cdot F_{k-1.k-i}(a_{k+1},\ldots,a_{2k-1}) + F_{k,k}(a_1,\ldots,a_k)$$

To demonstrate in a relatively simple notation the usefulness of this approach to 2D-filtering. let us choose k=2 as in figure 1 and suppress the indexes: for evaluating the median M of

$$\begin{array}{lll} a_1 & a_2 & a_3 \quad \text{row a} \\ b_1 & b_2 & b_3 \quad \text{row b} \\ c_1 & c_2 & c_3 \quad \text{row c} \end{array}$$

- compute for each row x the minimum $X_1=F_{3,1}(x_1,x_2,x_3)$, the median $X_2=F_{3,2}(x_1,x_2,x_3)$ and the maximum $X_3=F_{3,3}(x_1,x_2,x_3)$

- express M as $\sum_{i+j+k=5} A_i.B_j.C_k$, or, more clearly, under the symmetric form

$$F_{3,2}(U,V,W) = UV + VW + WU \text{ , where}$$

$U = F_{3,3}(A_1,B_1,C_1) = A_1 + B_1 + C_1$

$V = F_{3,2}(A_2,B_2,C_2) = A_2B_2 + B_2C_2 + C_2A_2$

$W = F_{3,1}(A_1,B_1,C_1) = A_3.B_3.C_3$

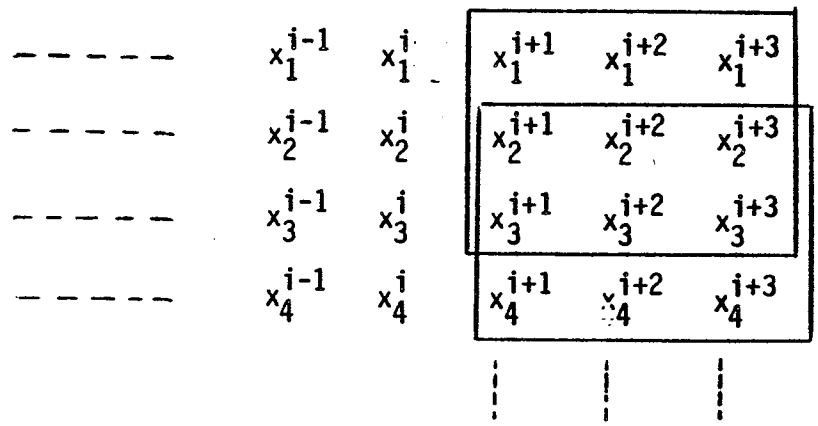
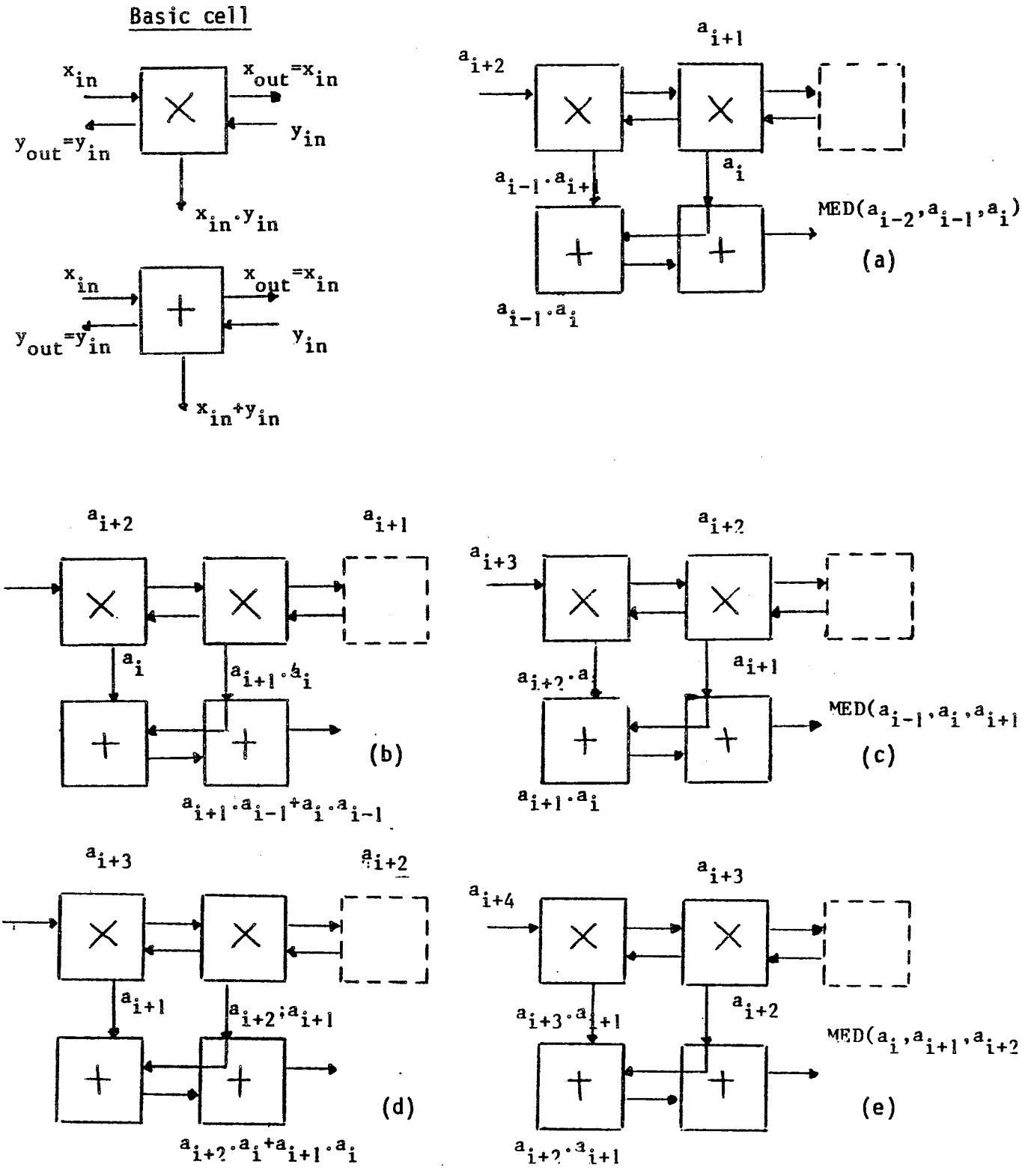


Figure 1



A network for computing $F_{3,2}$
with five successive steps

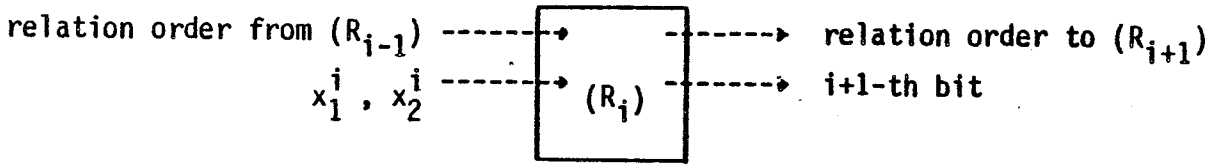
Figure 2

The basic cell is clearly chosen to compute the maximum and/or the minimum of two binary inputs, and will be described in the next section. We present in figure 2 a network with efficiency e=1/2 for the computation of $F_{3,2}$, establishing the possibility of performing general median filtering with that degree of parallelism. Further details may be found in reference (12).The fact that the given solution has an efficiency e=1/2 (couldn't we design a network with e=1 ?) will lead us to an interesting discussion on the efficiency of systolic networks in section 3 .

2) Layout of a systolic network for the computation of $F_{3,2}$

2.1) detail of the basic cell

Let (R) be a combinational network receiving two n-bit integers x_1 and x_2, and delivering their minimum: for small n we choose a serial network; writing $x_k = (x_k^0,\ldots,x_k^{n-1}) = \Sigma_{i=0}^{n-1} x_k^{n-1-i}.2^i$ (k=1,2) , the computation of the i+1-th bit will be



where $E_1^i = (x_1(i) > x_2(i))$, $E_2^i = (x_2(i) > x_1(i))$, where $x_k(i)$ is the truncation of x_k to its i most significant bits. For instance, $E_1^{i+1} = E_1.(\bar{E_2^i} + x_1^i + \bar{x_2^i})$.

For large integers, we could choose a network similar to the BRENT/KUNG adder(3) and compute in parallel the comparison orders E^i, owing to the associativity of the operator o

$$(u',v',w') \text{ o } (u,v,w) = (u'.(\bar{v}+w),v'.(\bar{u}+w),w)$$

where $u = E_1^i$, $v = E_2^i$ and $w = (x_1^i > x_2^i)$

2.2) a network for $F_{3,2}$

Figure 3 shows the organization required for the layout: the data pass through several shift-registers, whose outputs feed combinational networks similar to (R); the global synchronization (and control) is ensured by two non overlapping clocks .

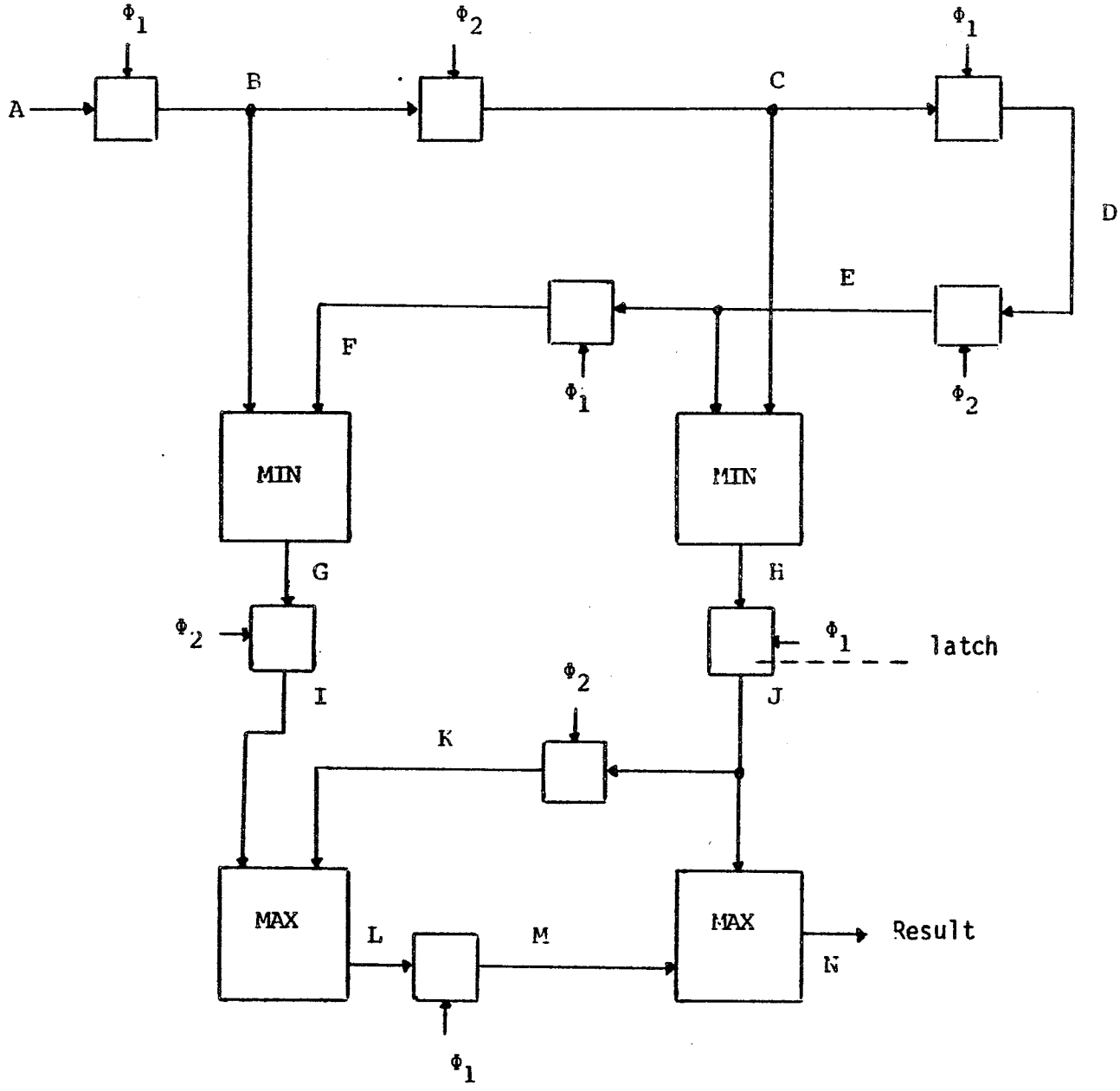

Φ1
Φ2
Φ1
A
B
C
D
E
F
Φ1
Φ2
MIN
MIN
G
H
Φ2
Φ1
latch
I
J
Φ2
K
MAX
L
M
MAX
Result
N
Φ1

Figure 3

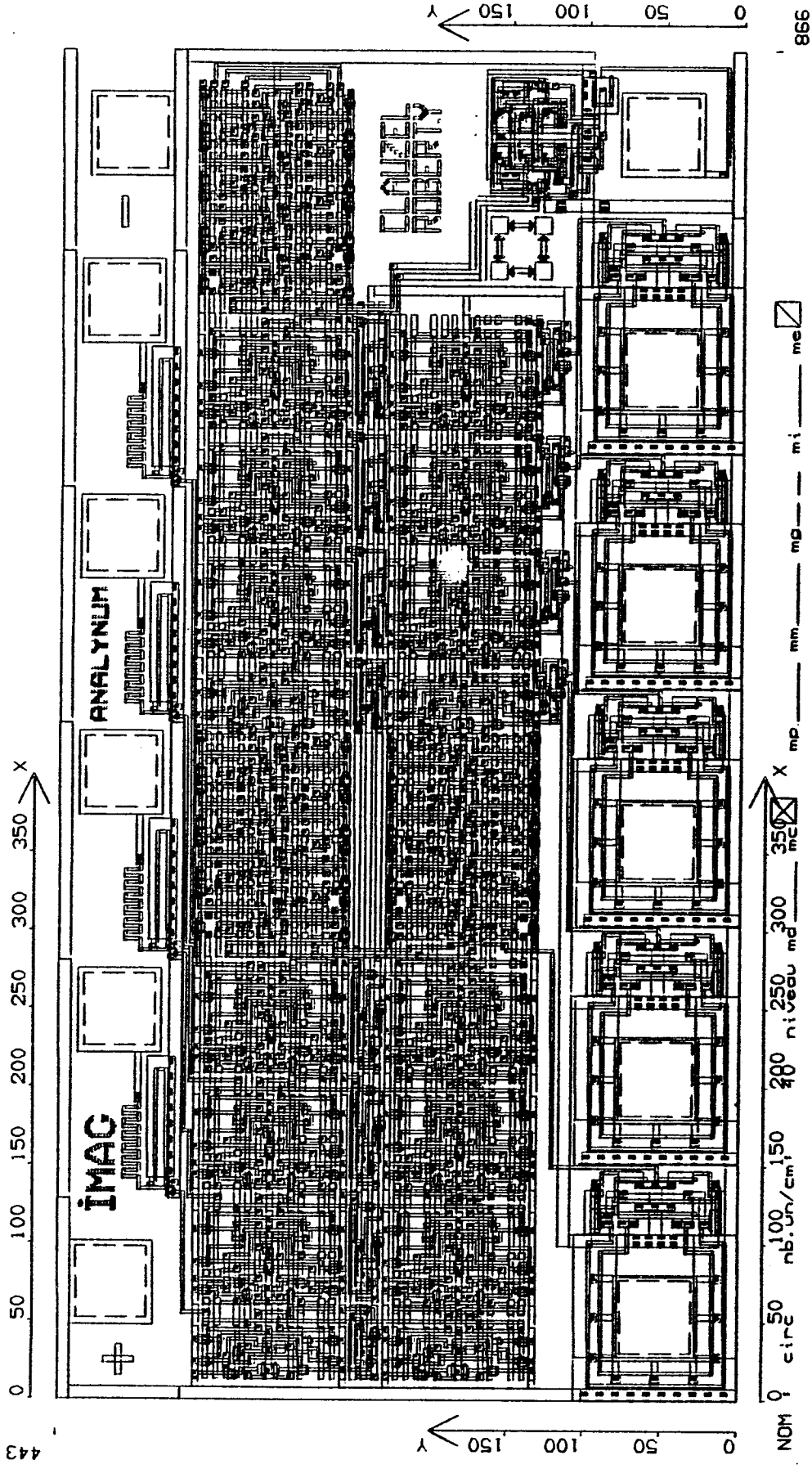
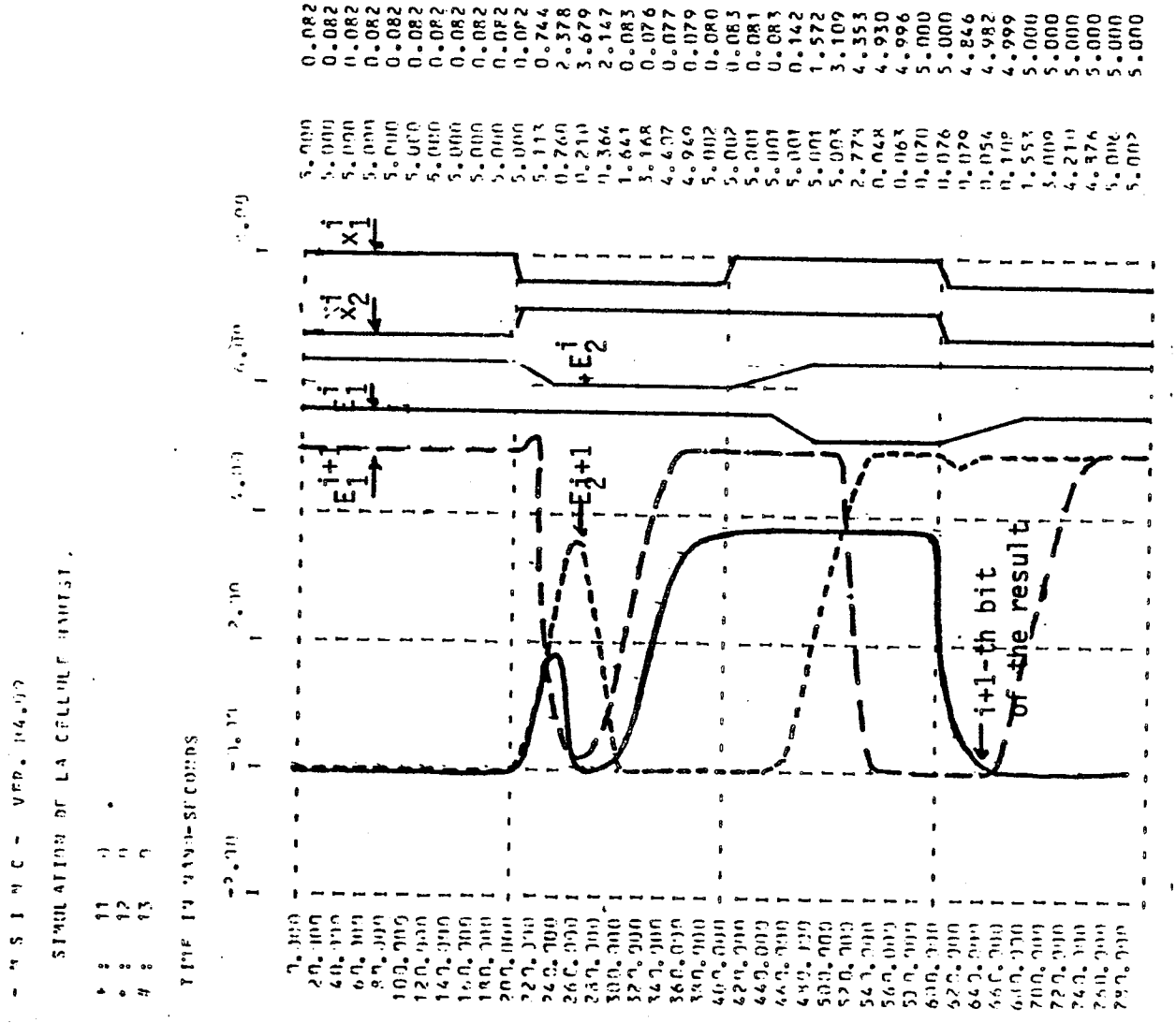

IMAG
ANALYNUM
X
Y
0
50
100
150
200
250
300
350
443
866
NOM
cire
nb.Un/cm²
niveau
mp
mm
mg
mi
me
md

Figure 4



We describe here an electrical simulation of (R_i). The four upper curves represent the inputs: x^i and y^i. which are imposed from the exterior of the chip and therefore supposed perfect regarding the rise and fall times; E_1^i and E_2^i whose delay, rise and fall times are provided by a previous simulation of (R_{i-1}). The chronogram can be divided into four phases. each of them being characterized by new values for x^i and y^i. and a new comparison order, as described now:

- phase 1: "at this point, x=y, thus compare x^i and y^i, compute a new order and output MIN(x^i,y^i)"
- phase 2: "x_1 x_2, thus output x_2 and propagate order"
- phase 3: "x_2 x_1, thus output x_1 and propagate order"
- phase 4: same order as phase 1

Such a simulation makes some trouble spots appear clearly, and is helpful for backtracking the size of some devices. For example, phase 2 shows a critical race where E_1^i and E_2^i change although they are not supposed to; as a consequence, since the output is computed from them, its setup time is more important than in other phases .

Figure 5

Design cost has been reduced by the use of programmable strip-organized cells (2). A 4-bit prototype (physical realization via the French MPC 1983 (1)) is presented in figure 4, and an example of a critical race (electrical simulation) is given in figure 5 .

3) On the efficiency of systolic networks

If a network of k cells solves a problem (P) in time t_k, we say its efficiency is $e = t_1 / k.t_k$ where t_1 is the time neccessary to solve it using a single cell. Clarly, $e \leq 1$ and a parallel algorithm is of maximal efficiency if $e = 1$. Note however that with a repetitive use of the network to solve (P) many times, the initialization delay (the time needed by the first inputs to pass trough the network) should be neglected to evaluate the performances, leading to the asymptotic efficiency (AE) of the network .

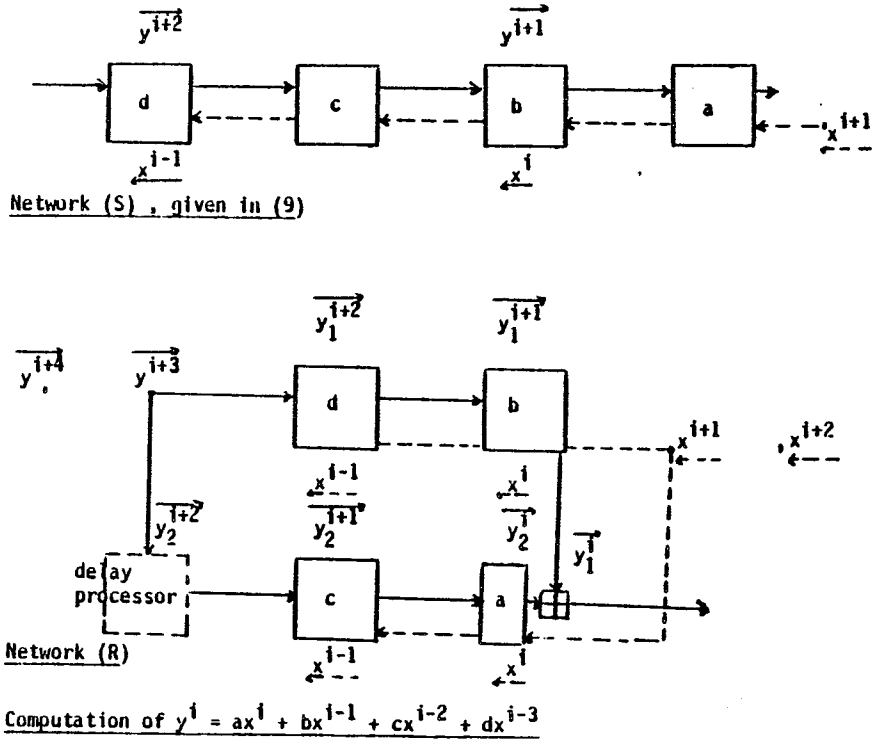


Figure 6

Consider the two networks of figure 6: (R) seems to be twice as good as (S). but we can always combine two distincts computations on (S) to get a maximal degree of parallelism...Moreover, we shall see that inefficient networks can lead to better solutions when using a divide-and-conquer approach .

Let (P_n) be the problem of computing the product C = A.B of two dense matrices A.B of size n. Assume n is divisible by k: computing C means computing k^3 products of size n/k and performing then some additions. Given a network N solving $(P_{n/k})$ with efficiency e=1/k, concatenating k copies of N, interleaving the computation of k distincts sub-blocks, repeating the whole operation k times seems to be a good strategy, as is proved by the following table:

NETWORK	Area	Time	e	AE	$AT, AT^2, AP, AP^2/n^3$
1 DAVIS/WEISER(4)	$3n^2$	3n	1	n	9 27 3 3
2 KUNG/LEHMAN(7)	$2n^2$	3n	1/2	2n	6 18 4 8
3 KUNG/LEISERSON(9)	$3n^2+ O(n)$	5n	1/3	3n	15 75 9 27
2* (2 parallel)	n^2	3n	1	2n	3 9 2 4
3* (3 parallel)	n^2	5n	1	3n	5 25 3 9
5 KUNG/GUIBAS(7)	n^2	2n	1	2n	2 4 2 4
VUILLEMIN(11)	(but with a greater complexity of the cells)				

We can point out the good performances of *), small and fast, despite the efficiency e=1/2 of (2) .

Conclusion

Our design for median filtering enjoy regular structure and localized connections, and thus is well-suited for layout, as shown in part 2. In many systolic architectures, the cells in the network are alternately idle; KUNG/LEISERSON(9) have proposed coalescing pairs of adjacent processors in order to obtain an efficiency e = 1: we suggest rather (as discussed in part 3) decomposing the original problem into independant sub-computations which can be performed in parallel, leading to the maximal degree of parallelism .

REFERENCES

(1) ANCEAU F., GUYOT A. RAYMOND J. Présentation du projet CMP Rapport Equipe de recherche en architectures d'ordinateurs IMAG Grenoble 1981

(2) ANCEAU F. REIS R., Complex integrated circuits design strategy RR n° 267 IMAG Grenoble 1981

(3) BRENT.R.P,KUNG.H.T, A regular layout for parallel adders, TR, CMU 1979

(4) DAVIS A.L.,WEISER U. Mathematical representation for VLSI arrays. T R University of Utah 1980

(5) FISHER A.L. Systolic algorithms for running order statistics in signal and image processing, T R CMU 1981

(6) KUNG H.T. Why systolic architectures, T R, CMU 1981

(7) KUNG H T GUIBAS L J., THOMPSON C.D., Direct VLSI implementation of combinatorial algorithms, Caltech Confernce on VLSI, January 1979 p 509-525

(8) KUNG H.T.. LEHMAN P.L., Systolic (VLSI) arrays for relational database operations, T R CMU 1979

(9) KUNG H.T. LEISERSON C.E. Systolic arrays for (VLSI), in (10) p 271-292

(10) MEAD C., CONWAY L. Introduction to VLSI systems, Addison Wesley 1980

(11) PREPARATA F.P VUILLEMIN J Area-time optimal VLSI networks for multiplying matrices, Information processings letters vol 11 (1980) n°2 p 77-80

(12) ROBERT Y. TCHUENTE M. Calcul en parallele sur des réseaux systoliques Conference "Méthodes vectorielles et parallèles en calcul scientifique", GAMNI Paris, Mars 1983

Acknowledgments: Thanks are due to A.GUYOT and A.A.JERRAYA for helpful suggestions concerning part 2

CHAPITRE 7

INFORMATIQUE THÉORIQUE. — *Complexité de la factorisation QR en parallèle.* Note (*) de **Michel Cosnard** et **Yves Robert**, présentée par Jacques-Louis Lions.

Nous présentons un algorithme optimal pour la décomposition QR en parallèle d'une matrice pleine de taille N. Nous en déduisons que la complexité de cette décomposition est asymptotiquement 2N avec un nombre illimité de processeurs.

COMPUTER SCIENCE. — Complexity of Parallel QR Factorization.

We propose an optimal algorithm to perform the parallel QR decomposition of a dense matrix of size N. We deduce that the complexity of such a decomposition is asymptotically 2N when an unlimited number of processors is avalaible.

1. INTRODUCTION. — La résolution en parallèle de problèmes linéaires denses a fait l'objet de nombreuses études (Heller [3], Sameh [6] et les références qu'ils citent; *voir* aussi Gentleman et Kung [1], Kung et Leiserson [4] pour des réseaux systoliques). La plupart des algorithmes parallèles proposés sont des adaptations des méthodes séquentielles directes. Parmi celles-ci, il est communément admis que la plus performante est la méthode de décomposition QR utilisant les rotations de Givens sans racines carrées [2] pour des raisons de simplicité et de stabilité numérique.

Sameh et Kuck présentent dans [5] un processus parallèle de calcul réalisant une telle décomposition. Ils considèrent un ordinateur de type SIMD (Single Instruction Multiple Data) avec [N/2] (¹) processeurs, où N est la taille de la matrice à décomposer. Leur algorithme nécessite $2N-3$ unités de temps, chaque unité correspondant à la réalisation d'une rotation élémentaire de Givens. Le nombre total de rotations étant $N(N-1)/2$, leur algorithme est aussi stable que la version séquentielle. Une légère modification conduit à un algorithme utilisant le même nombre de processeurs, mais dont le temps de calcul n'est que $2(N-1)-[\log N]$ (²). Cette valeur semble être la meilleure borne supérieure connue pour la complexité de la décomposition QR d'une matrice dense.

Dans la lignée de ce travail, nous présentons un algorithme optimal (lequel n'est pas unique) utilisant aussi [N/2] processeurs. Nous montrons alors que la complexité T_N de la décomposition QR d'une matrice de taille N vérifie $T_N = 2N + o(N)$.

2. UN ALGORITHME OPTIMAL. — A est une matrice carrée réelle de taille N. On veut obtenir une décomposition $A = Q.R$, où Q est orthogonale et R triangulaire supérieure. Une transformation de Givens sans racine carrée est une rotation plane combinant deux lignes de A pour annuler un élément placé sous la diagonale. Dans la version séquentielle de l'algorithme, les éléments sont annulés depuis le bas de chaque colonne, un à chaque étape. L'idée-clé pour paralléliser cette réduction est d'annuler plusieurs éléments à chaque étape en combinant des lignes de A, sans détruire les zéros précédemment introduits. Nous supposons qu'un processeur peut réaliser une transformation à chaque étape, mais nous n'autorisons pas les duplications de lignes, si bien que le nombre maximal de processeurs pouvant être utilisés est [N/2].

Nous représentons un processus de calcul par un tableau triangulaire inférieur T tel que $t(i, j)$ représente le dernier instant où l'élément d'indice (i, j), $i > j$, de A est annulé. On note $(i, j) \# (k, l)$ si $t(i, j) \leqq t(k, l)$. Soient T_1 et T_2 deux processus de calcul, on dira que T_1 est meilleur que T_2 si $t_1(N, N-1) \leqq t_2(N, N-1)$.

LEMME 1. — *Il existe un algorithme optimal annulant chaque élément une seule fois et vérifiant* $(i, j) \# (k, l)$ *si* $[(j < l)$ *ou* $(j = l$ *et* $i \leqq k)]$.

Nous allons décrire un algorithme optimal T satisfaisant aux conditions du lemme 1, et n'utilisant donc que $N(N-1)/2$ rotations. Pour ce faire, nous adoptons la notation suivante : la k-ième colonne de T est représentée par $\alpha_{1,k}^{n_{1,k}}\alpha_{2,k}^{n_{2,k}}\ldots\alpha_{q,k}^{n_{q,k}}$ avec :

$$n_{i,k}\geqq 1; \qquad n_{1,k}+\ldots+n_{q,k}=N-k$$

où :

$$\alpha_{1,k}=t(j,k) \qquad \text{si} \quad N-n_{1,k}<j\leqq N,$$

$$\alpha_{i,k}=t(j,k) \qquad \text{si} \quad N-(n_{1,k}+\ldots+n_{i,k})<j\leqq N-(n_{1,k}+\ldots+n_{i-1,k}), \qquad i\geqq 2.$$

Nous construisons les colonnes de T par récurrence :

— *première colonne :*

(i) définition de $\alpha_{1,1}$ et de $n_{1,1}$:

$$\alpha_{1,1}=1 \qquad \text{et} \qquad n_{1,1}=[N/2];$$

(ii) si $n_{1,1}+\ldots+n_{k,1}<N-1$, on pose :

$$\alpha_{k+1,1}=\alpha_{k,1}+1 \qquad \text{et} \qquad n_{k+1,1}=[(N-(n_{1,1}+\ldots+n_{k,1}))/2];$$

— *de la colonne k à la colonne $k+1$:*

(i) définition de $\alpha_{1,k+1}$ et de $n_{1,k+1}$:

si $n_{1,k}\geqq 2$ alors :

$$\alpha_{1,k+1}=\alpha_{1,k}+1 \qquad \text{et} \qquad n_{1,k+1}=[n_{1,k}/2]$$

sinon :

$$\alpha_{1,k+1}=\alpha_{1,k}+2 \qquad \text{et} \qquad n_{1,k+1}=[(n_{1,k}+n_{2,k})/2]$$

(ii) si $n_{1,k+1}+\ldots+n_{j,k+1}<N-(k+1)$, on pose :

$$\alpha_{j+1,k+1}=\alpha_{j,k+1}+1 \qquad \text{et} \qquad n_{j+1,k+1}=[(N-(n_{1,k+1}+\ldots+n_{j,k+1}))/2].$$

PROPOSITION 2. — *La construction décrite conduit effectivement à un algorithme satisfaisant aux conditions du lemme 1. Cet algorithme est optimal.*

3. COMPLEXITÉ ASYMPTOTIQUE. — Appelons T_N la complexité de la décomposition QR en parallèle d'une matrice de taille N, c'est-à-dire le temps de calcul d'un algorithme optimal réalisant cette décomposition.

LEMME 2. — $T_2=1$ *et pour tout* $N\geqq 2$, $T_N+1\leqq T_{N+1}\leqq T_N+2$.

Il résulte du lemme 2 que $N-1\leqq T_N\leqq 2N-3$. Le tableau ci-dessous montre que la borne supérieure ainsi obtenue est réaliste.

N	4	8	16	32	64	128	256	512	1024	2048	4096
T_N	4	11	26	56	118	243	495	1000	2015	4051	8129

THÉORÈME 3. — $T_N=2N+o(N)$.

Schéma de démonstration. — D'après ce qui précède, il suffit d'établir, en utilisant l'algorithme optimal que nous avons défini, que :

$$\liminf_{N\to+\infty} T_N/2N=1.$$

Aux formules de récurrence définies plus haut, nous associons des formules similaires en éliminant les parties entières. Celles-ci nous permettent d'approcher la valeur des entiers $\alpha_{i,k}$ à l'aide de la formule de Stirling et d'en déduire le résultat annoncé.

4. Avec un nombre limité de processeurs. — Si l'on ne dispose que de p processeurs, on modifie l'algorithme précédent comme suit : si m éléments doivent être annulés au temps t, où $ip+1 \leqq m \leqq (i+1)p$, nous les annulerons avec p processeurs en $i+1$ unités de temps. L'algorithme nécessite alors un temps de calcul $t_{N,p}$ vérifiant :

$$t_{N,p} \leqq 4p\left(\sum_{i=1}^{\lfloor N/2p \rfloor} i\right).$$

Il en résulte que la complexité $T_{N,N/q}$ de la décomposition QR en parallèle d'une matrice de taille N avec N/q processeurs vérifie :

$$T_{N,N/q} \leqq t_{N,N/q} \leqq N(1+q/2).$$

5. Efficacité. — Lorsque la résolution d'un problème (P) de taille N avec p processeurs exige un temps de calcul $t_{N,p}$, une bonne mesure du degré de parallélisme est donnée par la quantité $e_{N,p} = t_1/pt_p$. Nous obtenons ici :

$$\lim_{N \to +\infty} e_{N,N/2} = 1/2 \qquad \text{et} \qquad e_{N,N/q} \leqq q/(q+2).$$

Remarquons que la parallélisation présentée est d'une bonne efficacité.

6. Conclusion. — La factorisation QR est un des algorithmes les plus communément utilisés pour la résolution de problèmes linéaires. Sa complexité $O(N^2)$ motive sa parallélisation. Nous avons montré dans cette Note que l'algorithme présenté par Kuck et Sameh [5] est le meilleur possible, des deux points de vue suivants :

— en pratique : il est impossible d'obtenir une amélioration du temps de calcul sans accroître de manière importante les difficultés de programmation;

— asymptotiquement : il est optimal.

De plus, il conduit à un algorithme de calcul très performant lorsque seul un nombre limité de processeurs est disponible.

(1) $[u]$ est le plus grand entier inférieur ou égal à u.

(2) Tous les logarithmes considérés sont en base 2.

(*) Remise le 4 juillet 1983.

[1] M. Gentleman et H. T. Kung, *Proc. SPIE.*, 298, 1981.
[2] S. Hammerling, *J. Inst. Math. Appl.*, 13, 1974, p. 215.
[3] D. Heller, *Siam Review*, 20, 1978, p. 740.
[4] H. T. Kung et C. E. Leiserson, *Sparse Matrix Proc.*, Duff, 1978.
[5] A. Sameh et D. Kuck, *J. A.C.M.*, 25, n° 1, 1978, p. 81.
[6] A. Sameh, *Bull. E.D.F., C*, 1, 1983, p. 129.

Laboratoire IMAG, Université de Grenoble,
B.P. n° 68, 38402 *Saint-Martin-d'Hères.*

Numer. Math. , – (1985)

Numerische Mathematik

Parallel QR Decomposition of a Rectangular Matrix

M. Cosnard, J.-M. Müller and Y. Robert

CNRS, Laboratoire TIM3, Institut IMAG, BP 68, F-38402 Saint Martin d'Heres Cedex, France

Summary. We show that the greedy algorithm introduced in [1] and [5] to perform the parallel QR decomposition of a dense rectangular matrix of size $m \times n$ is optimal. Then we assume that m/n^2 tends to zero as m and n go to infinity, and prove that the complexity of such a decomposition is asymptotically $2n$, when an unlimited number of processors is available.

Subject Classifications: AMS(MOS): 65F05; CR: G1.3.

1. Introduction

Let A be a dense rectangular matrix of size $m \times n$, $m \geqq n$. The Givens transformations algorithm produces an orthogonal decomposition of A which is used in several problems of numerical analysis, including linear least squares problems. This algorithm requires $mn - n(n+1)/2$ steps on a sequential machine, each step being the time necessary to achieve a Givens rotation.

Parallel versions for SIMD or MIMD computers (Flynn [2]) of the Givens transformations algorithm have been introduced in the literature: see Lord et al. [4], Sameh and Kuck [6] and the survey papers of Heller [3] or Sameh [7]. Assuming that enough processors are available to perform at each step up to $\lfloor m/2 \rfloor$ independent rotations simultaneously (where $\lfloor \cdot \rfloor$ denotes the floor function), Sameh and Kuck [6] propose a scheme whose number of steps is $m + n - 2$ if $m > n$ and $2n - 3$ if $m = n$. Recently, Modi and Clarke [5] have introduced and partially analyzed a greedy algorithm to compute the QR decomposition of an m by n matrix. Their numerical experiments show that this new algorithm takes appreciably fewer stages than Sameh and Kuck's scheme, which is confirmed by an asymptotic theoretical analysis: Modi and Clarke obtain the approximation

$$\log_2 m + (n-1)\log_2 \log_2 m$$

which is valid if m goes to infinity, n fixed. In their paper, Modi and Clarke introduce another class of algorithms, namely the Fibonacci schemes, and

Ms. No. 880 Author Cosnard Ms. 1-14 Pages 1-11

Springer-Verlag, Heidelberg / H. Stürtz AG, Würzburg

Provisorische Seitenzahlen / Provisional page numbers 1. Korr. Date 14.19.85

discuss in detail their performances. However, they observe that these schemes seem to be less efficient than the greedy algorithm.

To our knowledge, no result has been published so far on the determination of an optimal algorithm for the QR decomposition by Givens transformations of a rectangular matrix, and the evaluation of its performance. Moreover, all the algorithms presented by Sameh and Kuck [6] and Modi and Clarke [5] are Givens sequences, that is sequences of Givens rotations in which zeros once created are preserved. The question whether temporarily annihilating elements and introducing zeros that will be destroyed later on can lead to any additional parallelism has never been discussed.

In this paper (Sect. 2) we prove that the greedy algorithm is always optimal, not only in the class of the Givens sequences, but more generally in the class of all possible parallel algorithms. This result is true for any rectangular matrix, and gives a negative answer to the Modi and Clarke's question concerning the existence of m by n matrices with a very large value of m and smallish n for which the Fibonacci schemes would be more efficient than the greedy algorithm.

Furthermore, Modi and Clarke's approximate analysis can now be viewed as a result on the asymptotic complexity of the problem of computing the QR decomposition by Givens transformations of an m by n matrix in the case m goes to infinity, n fixed. In Sect. 3, we derive a similar result for the case $m = o(n^2)$, $m \geqq n$ (recall that $f(x) = o(x)$ means that $f(x)/x$ tends to zero as x goes to infinity). More precisely, let $\mathrm{Opt}(m, n)$ denote the number of parallel steps in the greedy algorithm for an m by n matrix, and consider a function $f(n)$ satisfying to the following properties:

- $\lim f(n) = 0$ (as n goes to infinity)
- $f(n) \geqq 1/n$ for all n.

Then $\mathrm{Opt}(nf(n), n) = 2n + o(n)$.

In particular, this includes the case where m and n are proportional, which is of great interest for the linear least squares problem. We give experimental results to illustrate this case at the end of Sect. 3.

Throughout the paper, A is an m by n rectangular matrix. We assume that $[m/2]$ Givens rotations can be performed simultaneously. We refer to Modi and Clarke [5] for a straightforward modification of the algorithms if there are not enough processors, as well for discussing error bounds. Finally, we denote $\lim f(x)$ the limit of a function f as x goes to infinity.

2. An Optimal Algorithm

In this section, we show that the greedy algorithm is optimal for decomposing any rectangular matrix A of size $m \times n$, $m \geqq n$. We first reduce the problem to a particular class of parallel algorithms, namely the Givens sequences.

Moreover we prove that the elements can be annihilated from left to right and from bottom to top: such an algorithm which reduces A to upper triangular form is called a Standard Parallel Givens Sequence.

Then we introduce the Greedy Standard Parallel Givens Sequence and show that this algorithm is optimal.

Definitions and Notations

$R(i,j,k)$, $i \neq j$, $1 \leqq i,j \leqq m$ and $1 \leqq k \leqq n$, denotes the rotation in plane (i,j) which annihilates the element $A(i,k)$. $\mathrm{Id}(i,j)$ and $\mathrm{Perm}(i,j)$ denote the rotations in plane (i,j) which correspond respectively to the identity and to the permutation of rows i and j. An algorithm M of length T is represented by $M = (M(1), \ldots, M(T))$ where for $1 \leqq t \leqq T$, $M(t)$ is a group of $r(t)$ disjoint rotations, which can be performed simultaneously. M is used in order to construct a sequence $A(t)$ such that: $A(0) = A$ and for $1 \leqq t \leqq T$, $A(t)$ is obtained by applying in parallel the rotations in $M(t)$ to $A(t-1)$. For short we use (M, T, R) where $R = r(1) + \ldots + r(T)$.

Using these notations, we can state more precisely the following definitions:

Definitions. (M, T, R) is a Givens sequence [5] if zeroes once created are preserved, i.e. for any couple of distinct rotations $R(i,j,k)$ and $R(i',j',k')$ in M we have $(i,k) \neq (i',k')$.

(M, T, R) is a Standard Parallel Givens Sequence (SPGS for short) if it is a Givens sequence which reduces A to upper triangular form and annihilates elements from left to right and from bottom to top, i.e. if $R(i,j,k) \in M(t)$ and $R(i',j',k') \in M(t')$ where $t \leqq t'$, then $k < k'$ or $(k = k'$ and $i \geqq i')$.

Theorem 1. *Let (M, T, R) be an algorithm which reduces A to upper triangular form. There exists a Standard Parallel Givens Sequence of length T.*

Proof. In the first step, we construct (M', T, R) which annihilates the elements in any row from left to right.

Let us call $p(1,t), \ldots, p(m,t)$ the number of annihilated elements of $A(t)$ in rows $1, \ldots, m$.

We shall construct M' ar the sequence $A'(t)$ (recall that $A'(0) = A'$ and for $1 \leqq t \leqq T$, $A'(t)$ is obtained by applying in parallel the rotations in $M'(t)$ to $A'(t-1)$) by induction on t so that $A'(t)$ has $p'(1,t), \ldots, p'(m,t)$ zeros with the following properties:

- $p'(i,t) \geqq p(i,t)$; $i = 1, \ldots, m$
- the first $p'(i,t)$ elements of row i in $A'(t)$ are zeros.

At time 1, if $R(i,j,k)$ belongs to $M(1)$, let $R(i,j,1)$ belong to $M'(1)$. The preceding induction hypothesis are clearly satisfied for $A'(1)$.

We assume that $M'(t)$ has been constructed so that $A'(t)$ has the required properties. Because the rotations in $M(t+1)$ act on disjoint pairs of rows, we can consider each pair separately. Let $R(i,j,k)$ belong to $M(t+1)$. We must construct an operation in $M'(t+1)$ that preserves the induction properties. There are two cases:

- if a new zero is required in row i of $A'(t)$, we use a rotation to introduce it or use a permutation to bring it up from row j.
- If row i of $A'(t)$ already has enough zeros we do nothing (identity). What is actually done is determined by an analysis of cases, to which we now proceed.

Assume that the induction hypothesis is valid at time t. If $R(i,j,k)$ belongs to $M(t+1)$, discuss the following cases:

a) the positions of the zeros in rows i and j of $A(t)$ are the same. Thus $p(i,t) = p(j,t)$.

a1) $p'(i,t)=p'(j,t)=p(i,t)$
Replace $R(i,j,k)$ in $M(t+1)$ by $R(i,j,p'(i,t)+1)$ in $M'(t+1)$.
Set $p'(i,1)=p'(i,t)+1$ and $p'(j,t+1)=p'(j,t)$.

a2) $p'(i,t)>p(i,t)$
Replace $R(i,j,k)$ in $M(t+1)$ by $\mathrm{Id}(i,j)$ in $M'(t+1)$.
Set $p'(i,t+1)=p'(i,t)$ and $p'(j,t+1)=p'(j,t)$.

a3) $p'(i,t)=p(i,t)$ and $p'(j,t)>p(j,t)$
Replace $R(i,j,k)$ in $M(t+1)$ by $\mathrm{Perm}(i,j)$ in $M'(t+1)$.
Set $p'(i,t+1)=p'(j,t)$ and $p'(j,t+1)=p'(i,t)$.

b) the positions of the zeros in rows i and j of $A(t)$ are not the same. Thus $p(j,t+1)\leqq p(j,t)$.

b1) $p(i,t+1)\leqq p(i,t)$
Replace $R(i,j,k)$ in $M(t+1)$ by $\mathrm{Id}(i,j)$ in $M'(t+1)$.
Set $p'(i,t+1)=p'(i,t)$ and $p'(j,t+1)=p'(j,t)$.

b2) $p(i,t+1)=p(i,t)+1$
Thus the set of the indices of the zeros in row i of $A(t)$ is strictly included into the one of row j, which implies

$$p(j,t+1)=p(i,t)<p(j,t)$$

- (i) $p'(i,t)\geqq p(i,t)+1$
  Replace $R(i,j,k)$ in $M(t+1)$ by $\mathrm{Id}(i,j)$ in $M'(t+1)$.
  Set $p'(i,t+1)=p'(i,t)$ and $p'(j,t+1)=p'(j,t)$.

- (ii) $p'(i,t)=p(i,t)$
  Replace $R(i,j,k)$ in $M(t+1)$ by $\mathrm{Perm}(i,j)$ in $M'(t+1)$.
  Set $p'(i,t+1)=p'(j,t)$ and $p'(j,t+1)=p'(i,t)$.

From this construction we can deduce that the induction hypothesis is satisfied at time $t+1$. Since M reduces A to upper triangular form, M' reduces A to the same form.

We now proceed to the second part of the proof which consists in constructing a Givens sequence M'' of length T, which triangularizes A. M'' contains the same rotations as M' but those which reduce to the identity or a permutation are suppressed: instead of permuting the rows of A we permute the indices of the rows. A rotation which strictly increases the number of zeros (hence which differs from the identity or a permutation) is called efficient.

We use $\mathrm{trans}(i,j)$ in order to represent the permutation which exchanges the integers i and j. We construct now the following permutation s acting on $\{1,\ldots,m\}$:

- $s(0)$ is the identity.
- $s(t+1)=s(t)$ if there is no Perm in $M'(t+1)$.
- $s(t+1)=s(t)0\,\mathrm{trans}(i_1,j_1)0\,\mathrm{trans}(i_2,j_2)0\ldots$ if $\mathrm{Perm}(i_1,j_1)$, $\mathrm{Perm}(i_2,j_2),\ldots$ belong to $M'(t+1)$ (the composition of the permutations in $M'(t+1)$ are commutative since they act on disjoint elements).
- $S=S(T)$.

Fig. 1. A detailed example

t	$M(t)$	Case in proof	$M'(t)$	$M''(t)$
1	$R(3,1,2)$	a1	$R(3,1,1)$	$R(4,1,1)$
	$R(4,2,2)$	a1	$R(4,2,1)$	$R(3,2,1)$
2	$R(3,4,1)$	a1	$R(3,4,2)$	$R(4,3,2)$
	$R(2,1,3)$	a1	$R(2,1,1)$	$R(2,1,1)$
3	$R(1,2,2)$	b2(ii)	Perm(1, 2)	–
	$R(4,3,3)$	b2(ii)	Perm(4, 3)	–
4	$R(3,1,1)$	a1	$R(3,1,2)$	$R(3,2,2)$
5	$R(4,3,4)$	a1	$R(4,3,3)$	$R(4,3,3)$
	$R(2,1,1)$	b2(ii)	Perm(2, 1)	–
6	$R(3,1,1)$	b1	Id(3, 1)	–
7	$R(3,2,2)$	b2(i)	Id(3, 2)	–
	$R(4,1,1)$	b1	Id(4, 1)	–
8	$R(4,2,2)$	a2	Id(4, 2)	–
9	$R(4,3,3)$	a2	Id(4, 3)	–

$$s(0)=s(1)=s(2)=\begin{pmatrix}1&2&3&4\\ \downarrow&\downarrow&\downarrow&\downarrow\\ 1&2&3&4\end{pmatrix} \quad s(3)=\begin{pmatrix}1&2&3&4\\ \downarrow&\downarrow&\downarrow&\downarrow\\ 2&1&4&3\end{pmatrix}$$

$$s(5)=s(6)=s(7)=s(8)=s(9)=s=s^{-1}=\begin{pmatrix}1&2&3&4\\ \downarrow&\downarrow&\downarrow&\downarrow\\ 1&2&4&3\end{pmatrix}$$

We derive a Givens sequence M'' by letting:

$$R(i,j,k)\in M'(t) \text{ is efficient} \Leftrightarrow R(s^{-1}(s(t))(i), s^{-1}(s(t))(j), k)\in M''(t),\ 1\leqq t\leqq T.$$

M'' is a Givens sequence which reduces A to upper triangular form using $mn-n(n+1)/2$ rotations (since each rotation is efficient) and annihilating the elements from left to right. Clearly, the length of M'' is T.

From this Givens sequence, the last part of the proof consists in constructing a Standard Parallel Givens Sequence of the same length: this can be done by constructing a new permutation of the indices of the rows in a very straightforward manner, we omit a detailed description of this step.

The example given in Fig. 1 illustrates the proof of Theorem 1 on a 4×4 matrix.

In order to describe a SPGS, let us call $r(i,t)$ the number of zeros in column i introduced at time t and $s(i,t)=r(i,1)+\ldots+r(i,t)$. A simple proof (see [1]) leads to:

Lemma 1. *Let M be an algorithm which introduces $r(i,t)$ zeros in column i at time t. M is a SPGS if and only if:*

$$s(0,t)=m \quad \textit{for } t\geqq 0,$$
$$s(i,0)=0 \quad \textit{for } 1\leqq i\leqq m,$$
$$s(i,t)\leqq s(i,t-1)+[(s(i-1,t-1)-s(i,t-1))/2] \quad \textit{for } i,t\geqq 1.$$

It is clear that the length of a SPGS is equal to:

$$T=\min\{t/s(n,t)\geqq m-n\}.$$

The greedy Standard Parallel Givens Sequence has been introduced independently by Modi and Clarke [5] and by Cosnard and Robert [1]. We use $rg(i,t)$ and $sg(i,t)$ in order to specify the number $r(i,t)$ and $s(i,t)$ associated to the greedy SPGS. It is defined by letting:

$$\begin{aligned}
&rg(0,0)=m,\\
&rg(0,t)=0 \quad \text{for } t\geqq 1,\\
&rg(i,0)=0 \quad \text{for } 1\leqq i\leqq m,\\
&rg(i,t)=\lfloor(sg(i-1,t-1)-sg(i,t-1))/2\rfloor \quad \text{for } i,t\geqq 1.
\end{aligned}$$

Hence we have:

$$sg(i,t)=sg(i,t-1)+\lfloor(sg(i-1,t-1)-sg(i,t-1))/2\rfloor \quad \text{for } i,t\geqq 1.$$

Theorem 2. *The greedy SPGS is an optimal algorithm.*

Proof. Theorem 1 implies that we have only to show the optimality of the greedy SPGS among the SPGS. This optimality will be deduced from the following inequality:

$$s(i,t)\leqq sg(i,t) \quad \text{for } i,t\geqq 1.$$

We prove it by induction on i and t. For i or t equal to 0, we have:

$$s(0,0)=sg(0,0)=m,\ s(0,t)=sg(0,t)=m \text{ for } t\geqq 1,\ s(i,0)=sg(i,0)=0 \text{ for } i\geqq 1.$$

Assume that for j and u such that $j<i$ or ($j=i$ and $u<t$) we have:

$$s(j,u)\leqq sg(j,u).$$

Then from Lemma 1:

$$\begin{aligned}
s(i,t)&\leqq s(i,t-1)+\lfloor(s(i-1,t-1)-s(i,t-1))/2\rfloor\\
&\leqq sg(i,t-1)+\lfloor(sg(i-1,t-1)-sg(i,t-1))/2\rfloor\\
&\leqq sg(i,t).
\end{aligned}$$

This ends the proof of Theorem 2.

The optimality of the greedy SPGS could appear to be intuitively evident. However the proof is long and tedious. As can be seen on Fig. 2, the greedy algorithm performs $[m/2]$ rotations during the first steps. Hence it works with efficiency 1 at the beginning. The smaller the number of elements to be annihilated per column, the smaller the number of simultaneous rotations which are performed. We could then imagine that an algorithm beginning slowly would be as good as the greedy SPGS. Besides, an algorithm which even annihilates elements that will be destroyed later on, can be optimal, as is shown in Fig. 2.

						•								•
					•								4	2(4)
				•								5(7)	3	1 •
			•										3	5 7 •
		•											2	4 6 8 •
	•												1	3 5 7 9 •
•													1	2 4 6 8 10 •
3	•													
2	5	•												
2	4	7	•											
1	3	6	8	•										
1	3	5	7	9	•									
1	2	4	6	8	10	•								

Greedy SPGS (*M*, 10, 23)

The numbers denote the time at which the zeros are introduced. In parenthesis, appears the time at which a zero is destroyed. It can be seen that M is also optimal ($T=10$) although it performs 2 unnecessary rotations. Below a he rotations performed by both algorithms.

	Greedy SPGS	(*M*, 10, 23)
Step 1	$R(7,4,1)$ & $R(6,3,1)$ & $R(5,2,1)$	$R(7,5,1)$ & $R(6,4,1)$ & $R(3,2,2)$
Step 2	$R(4,2,1)$ & $R(3,1,1)$ & $R(7,6,2)$	$R(5,4,1)$ & $R(7,6,2)$ & $R(2,1,2)$
Step 3	$R(2,1,1)$ & $R(6,4,2)$ & $R(5,3,2)$	$R(4,2,1)$ & $R(3,1,1)$ & $R(6,5,2)$
Step 4	$R(4,3,2)$ & $R(7,6,3)$	$R(2,1,1)$ & $R(5,4,2)$ & $R(7,6,3)$
Step 5	$R(3,2,2)$ & $R(6,5,3)$	$R(4,2,2)$ & $R(6,5,3)$ & $R(3,2,5)$
Step 6	$R(5,4,3)$ & $R(7,5,4)$	$R(5,4,3)$ & $R(7,6,4)$
Step 7	$R(4,3,3)$ & $R(6,5,4)$	$R(4,3,3)$ & $R(6,5,4)$
Step 8	$R(5,4,4)$ & $R(7,6,5)$	$R(5,4,4)$ & $R(7,6,5)$
Step 9	$R(6,5,5)$	$R(6,5,5)$
Step 10	$R(7,6,6)$	$R(7,6,6)$

Note that

- a_{22} is annihilated at step 2 and destroyed at step 4
- a_{35} is annihilated at step 5 and destroyed at step 7

Fig. 2

3. A Result of Complexity

We concentrate now on the evaluation of the complexity of the greedy SPGS. We let $\mathrm{Opt}(m,n)$ denote the number of parallel steps in any optimal algorithm (e.g. the greedy algorithm) for an m by n matrix. We do not succeed in obtaining an exact formula, but we can state the following theorem which express the asymptotic value of $\mathrm{Opt}(m,n)$ for a wide class of matrices:

Theorem 3. *Let $m(n)=n^2 f(n)$, where $f(n)$ is a function such that*

- $f(n)\geqq 1/n$ *for all* $n\geqq 1$
- $\lim f(n)=0$.

Then

$$\mathrm{Opt}(m(n),n)=2n+o(n)$$

or, equivalently,

$$\lim \mathrm{Opt}(m(n),n)/2n=1.$$

Proof. Divided into two parts

- *we first show that* $\liminf \mathrm{Opt}(m(n),n)/2n\geqq 1$.

Consider the relations defining the partial sums $sg(i,t)$ for the greedy algorithm in the previous section. Evaluating these partial sums is difficult, due to the presence of the floor function. So we define the real sums

$$tg(0,t)=m \text{ for } t\geqq 0; \quad tg(i,0)=0 \text{ for } 1\leqq i\leqq m,$$
$$tg(i,t)=tg(i,t-1)+(tg(i-1,t-1)-tg(i,t-1))/2 \text{ for } i,t\geqq 1.$$

Clearly, $sg(i,t)\leqq tg(i,t)$ for all i,t. It is shown in Modi and Clarke [5] that

$$tg(i,t)=m\cdot\left(1-2^{-t}\sum_{k=0}^{n-1}\binom{t}{k}\right).$$

We know that $\mathrm{Opt}(m(n),n)=\min\{t/sg(n,t)\geqq m(n)-n\}$. Let e be such that $0<e<1$. We are going to show that

$$\lim tg(n,\lfloor(1+e)n\rfloor)=0.$$

The result will follow, since we deduce that $sg(n,\lfloor(1+e)n\rfloor)$ tends to zero too, which proves that

$$\mathrm{Opt}(m(n),n)\geqq\lfloor(1+e)n\rfloor \quad \text{for } n \text{ large enough,}$$

or, equivalently

$$\liminf \mathrm{Opt}(m(n),n)/2n\geqq(1+e)/2 \quad \text{for any } e<1.$$

Finally, this establishes our claim:

$$\liminf \mathrm{Opt}(m(n),n)/2n\geqq 1.$$

Lemma 2. *For any* $0<e<1$, $\lim tg(n,\lfloor(1+e)n\rfloor)=0$.

Proof. We set $(1+a)n=\lfloor(1+e)n\rfloor$, thus $a\leqq e\leqq a+1/n$.

We have

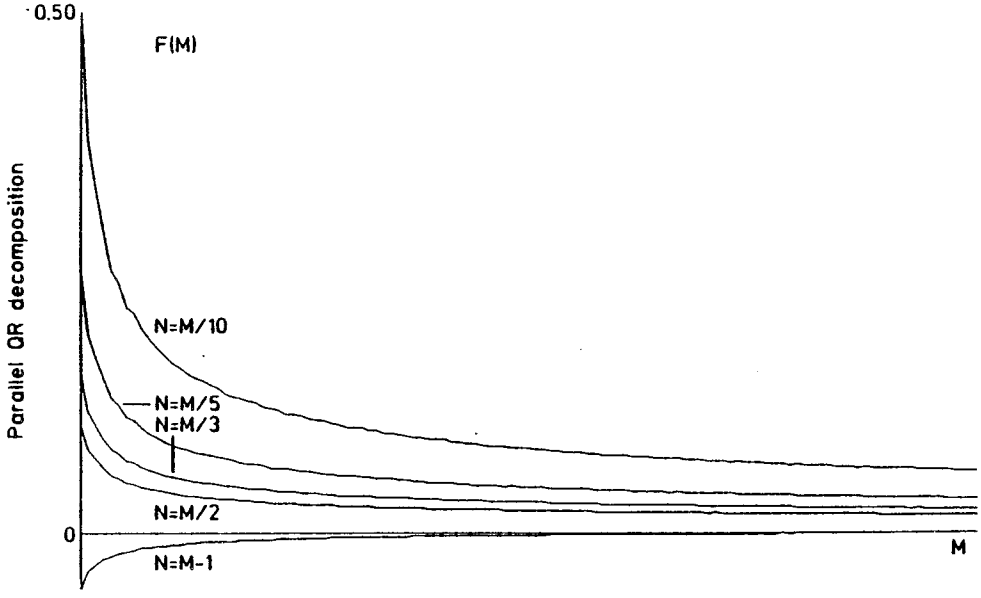
$$\begin{aligned} tg(n,(1+a)n)&=m\cdot\left(1-2^{-(1+a)n}\sum_{k=0}^{n-1}\binom{(1+a)n}{k}\right)\\ &=m\cdot 2^{-(1+a)n}\sum_{k=0}^{an}\binom{(1+a)n}{k}\\ &\leqq m\cdot(an+1)\cdot 2^{-(1+a)n}\end{aligned}$$

(for the last inequality, since $a<1$, we have $an\leqq(1+a)n/2$, and the binomial coefficients are increasing from $k=0$ up to $k=an$).

Now, using Strirling's formula, it follows that

$$2^{-(1-a)n}\binom{(1+a)n}{an}=Cn^{1/2}\cdot L(a)^n\cdot(1+g(n))$$

where C is some constant, $\lim g(n)=0$ and $L(a)=((1+a)/2)^{1+a}\cdot a^{-a}$. Using logarithms, one checks easily that $L(a)<1$ for $a<1$. Moreover, $m(n)$



F(M)=*Opt*(M,N)/2N-1 for $60 \leqq M \leqq 7200$

Fig. 3.

$=n^2 f(n) \leqq n^2$ for n large enough. Thus, there exists $K(a)$ with $L(a)<K(a)<1$ such that for n large enough

$$tg(n,(1+a)n) \leqq (K(a))^n.$$

Hence if $e<1$ then $tg(n,\lfloor(1+e)n\rfloor)$ tends to zero as n goes to infinity, which ends the proof of the lemma.

• *Consider now the Fibonacci scheme of order 1* introduced by Modi and Clarke: its number of steps is $2n+q$ is the lowest integer such that

$$q(q+1)/2 \geqq m(n)-n.$$

Clearly, $q \leqq (2m)^{1/2}$, and since the greedy SPGS is optimal, we deduce that

$$\text{Opt}\,(m(n),n) \leqq 2n+q \leqq 2n+(2m)^{1/2}.$$

We have $m(n)=n^2 f(n)$, hence $\lim m(n)^{1/2}/n=0$.

As a consequence, $\limsup \text{Opt}\,(m(n),n)/2n \leqq 1$, thereby establishing the theorem.

The case where m and n are proportional, i.e. $m(n)=k\cdot n$, $k \geqq 1$, is worth noting. It corresponds to the choice $f(n)=k/n$. Sameh and Kuck's scheme requires $(k+1)n-2$ steps, and is asymptotically optimal only in the case $k=1$, which corresponds to square matrices. The Fibonacci scheme of order 1 of Modi and Clarke is thus better suited to handle matrices where m and n are proportional, since it is always asymptotically optimal. The curves in Fig. 3

show the value of

$$F(m) = \mathrm{Opt}(m(n), n)/2n - 1$$

for different values of k ($k = 1/2$, $1/3$, $1/5$ and $1/10$), m varying from 60 to 7,200.

It appears that the smaller k, the more accurate our asymptotic expression is. We have also plotted the curve corresponding to square matrices: $F(m) = \mathrm{Opt}(m, m-1)/2m - 1$ is then negative. In fact, it can be observed that given m, $\mathrm{Opt}(m, n)/2n$ decreases (and becomes negative) as n increases up to $m-1$, the value n for which it equals 1 tending to m as m and n go to infinity.

Finally, we point out that Modi and Clarke's approximate analysis of the greedy algorithm in the case m goes to infinity and n is fixed, can now be viewed as a result on the asymptotic complexity of the problem of computing the QR decomposition by Givens transformations, according to Theorem 2:

Corollary 1. *Let m go to infinity and n be fixed. Then*

$$\mathrm{Opt}(m, n) = \log_2 m + (n-1)\log_2 \log_2 m + o(\log_2 \log_2 m).$$

4. Efficiency

When the resolution of a problem of size $m \times n$ with p processors requires a computation time $T_{m,n}(p)$, a good measure of the degree of the parallelism is given by the efficiency [6]

$$e_{m,n}(p) = T_{m,n}(1)/(p \cdot T_{m,n}(p)).$$

Clearly, $e_{m,n}(p) \leqq 1$, and a parallel algorithm is of good efficiency if $e_{m,n}(p)$ is constant.

From the preceding results, we derive the following expression for the asymptotic efficiency of the QR decomposition of an $m \times n$ matrix:

Corollary 2

1. *n fixed, $m \to \infty$*

$$e_{m,n}(\lfloor m/2 \rfloor) = 2n/\log_2 m + o(1/\log_2 m)$$

2. *$m = k \cdot n$, k fixed, $m \to \infty$*

$$e_{m,n}(\lfloor m/2 \rfloor) = (2k-1)/2k + o(1/m).$$

Proof. In case of a single processor, we have $T_{m,n}(1) = mn - n(n+1)/2$. The results follow directly.

5. Concluding Remarks

We have shown that the greedy algorithm introduced independently by Modi and Clarke [5] and Cosnard and Robert [1] is always optimal for computing the QR decomposition of a rectangular matrix using Givens rotations. This

result includes all parallel algorithms using Givens rotations, and not only Givens sequences. Furthermore, whenever the greedy algorithm would be too difficult to implement on a parallel computer, the asymptotic expression we give in the case $m=o(n^2)$ permits to select among the well-known algorithms the ones which are asymptotically optimal. In particular if m and n are proportional, the Fibonacci scheme of order 1 of Modi and Clarke [5] is to be preferred to Sameh and Kuck's scheme [6] unless the matrix is square.

Acknowledgment. We are greatly imdebted to the (anonymous) referree whose comments have greatly improved the quality of the final version.

References

1. Cosnard, M., Robert, Y.: Complexité de la factorisation QR en parallèle. C.R. Acad. Sci. 297, (A) 549–552 (1983)
2. Flynn, M.J.: Very high-speed computing systems. Proc. IEEE 54, 1901–1909 (1966)
3. Heller, D.: A survey of parallel algorithms in numerical linear algebra. SIAM Rev. 20, 740–777 (1978)
4. Lord, R.E., Kowalik, J.S., Kumar, S.P.: Solving linear algebraic equations on an MIMD computer. J. ACM 30, (1) 103–117 (1983)
5. Modi, J.J., Clarke, M.R.B.: An alternative Givens ordering. Numer. Math. 43, 83–90 (1984)
6. Sameh, A., Kuck, D.: On stable parallel linear system solvers. J. ACM 25, (1) 81–91 (1978)
7. Sameh, A.: An overview of parallel algorithms. Bull. EDF C1, 129–134 (1983)

Received October 12, 1984 / July 29, 1985

COMPLEXITY OF THE PARALLEL GIVENS FACTORIZATION FOR LINEAR LEAST SQUARES PROBLEMS

Michel COSNARD and Yves ROBERT
CNRS, Laboratoire TIM3, Institut IMAG, BP 68
38402 Saint Martin d'Heres Cedex, FRANCE

Abstract

We show that the greedy algorithm introduced in [1] and [13] to perform the parallel Givens factorization of a dense m by n rectangular matrix is optimal, assuming that an unlimited number of processors is available. In this case, we establish asymptotic complexity results, and we use them to select the most performant algorithms among those which have been introduced in the literature, given the values of m and n.. Then we discuss the case where the maximum number of Givens rotations that can be performed simultaneously is $p < \lfloor m/2 \rfloor$.

To appear in the Proceedings of the Symposium "Vector and Parallel Processors for Scientific Computation", Academia Nazionale dei Lincei and IBM Italia, Roma, May 27-29, 1985.

1. INTRODUCTION

Supercomputers with vector and/or parallel computing facilities are becoming available to an increasing extent for scientific computation. The concept of parallel processing is a departure from the trend of achieving increases in speed by performing single operations faster. Parallel processing achieves increases in speed by performing several operations in parallel [10].

The unconstrained linear least squares problem arises in numerous scientific computations. It can be formulated as follows:

Let A be a dense m by n rectangular matrix, with $m \geq n$, and let b be a m-vector. Obtain a n-vector x such that $\|Ax-b\|$ is minimized, where $\| \ \|$ denotes the Euclidean norm

There are two cases: (i) rank(A)=n and (ii) rank(A)<n. The sequential algorithms for both cases are well-known [7] [11] and proceed as follows:

- first stage (common to both cases):
  compute an orthogonal factorization of A, $QA = (R,0)^T$, either via plane rotations (Givens factorization) or via elementary reflectors (Householder reduction). Compute also $Qb = c$.
- second stage:
  - case (i): solve for x the triangular system $Rx = (c_1, ..., c_n)^T$ with a residual of norm $\|r\| = \|(c_{n+1}, ... , c_m)\|$.
  - case (ii): orthogonalize the rows of R to obtain the singular value decompostion $QAU = (\Sigma,0)^T$, where $\Sigma = diag(\sigma_1, ... , \sigma_n)$, $\sigma_1 \geq ... \geq \sigma_n > 0$. If the norm of the vector $(\sigma_{\nu+1}, ... , \sigma_n)^T$ if below a given tolerance, take the rank of A to be ν. Compute $x = U(c_1/\sigma_1, ... , c_\nu/\sigma_\nu, 0)^T$ with a residual of norm $\|r\| = \|(c_{\nu+1}, ... , c_m)\|$.

In some applications, the size of the problem can be large [7], and in others the solution is needed in the shortest time possible as in real-time signal processing [9]. As pointed out in [14], in either case, the use of multiprocessors offers definite advantages.

We concentrate here on the first stage, and we discuss the performance of parallel algorithms to compute the orthogonal factorization of an m by n matrix via Givens rotations. Such a factorization requires $mn - n(n+1)/2$ steps on a sequential machine, each step being the time necessary to achieve a Givens rotation. This O(m*n) number of steps to achieve the factorization motivates its parallelization. Indeed, parallel versions for SIMD or MIMD computers (Flynn [4]) of the Givens transformations algorithm have been introduced in the literature: see Lord et al. [12], Sameh and Kuck [17] and the survey papers of Heller [8] or Sameh [15] [16]. Assuming that enough processors are available to perform at each step up to $\lfloor m/2 \rfloor$ independent rotations simultaneously (where $\lfloor \cdot \rfloor$

denotes the floor function), Sameh and Kuck [17] propose a scheme whose number of steps is m+n-2 if m>n and 2n-3 if m=n. Recently, Modi and Clarke [13] have introduced and partially analyzed a greedy algorithm to compute the QR decomposition of an m by n matrix. Their numerical experiments show that this new algorithm takes appreciably fewer stages than Sameh and Kuck's scheme, which is confirmed by an asymptotic theoretical analysis: Modi and Clarke obtain the approximation

$$\log_2 m + (n-1)\log_2\log_2 m$$

which is valid if m goes to infinity, n fixed. In their paper, Modi and Clarke introduce another class of algorithms, namely the Fibonacci schemes, and discuss in detail their performances. However, they observe that these schemes seem to be less efficient than the greedy algorithm.

Finally, Sameh and Kuck [17] and Modi and Clarke [13] detail how to modify their algorithms to handle the case where only p Givens tranformations ($p<\lfloor m/2 \rfloor$) can be performed simultaneously.

In view of these results, many questions remain open:

1. Given the values of m, n, and p, find an optimal algorithm to compute the Givens factorization of an m by n matrix, assuming that p Givens transformations can be performed simultaneously.
2. Evaluate the number of steps needed to achieve the factorization using an otpimal algorithm
3. Among the preceding algorithms, determine those which are optimal, or asymptotically optimal; does the answer depend on the relationships between m, n and p ?

Moreover, all the algorithms presented by Sameh and Kuck [17] and Modi and Clarke [13] are Givens sequences, that is sequences of Givens rotations in which zeros once created are preserved. The question whether temporarily annihilating elements and introducing zeros that will be destroyed later on can lead to any additional parallelism has not been discussed. This is a nice example of specific questions that parallelism can rise.

Section 2 is devoted to some definitions and notations, together with a brief review of existing algorithms. In section 3, we prove that the greedy algorithm is optimal, not only in the class of the Givens sequences, but more generally in the class of all possible parallel algorithms. This result is true for any m by n rectangular matrix and for $p = \lfloor m/2 \rfloor$. Next (section 4), we derive some complexity results from this optimality analysis. In section 5 we discuss the case $p < \lfloor m/2 \rfloor$: there are several greedy algorithms in this case, but we prove that given m, n, p, it is always possible to select one optimal algorithm among them.

2. DEFINITIONS AND PREVIOUS RESULTS

Throughout the paper, A is an m by n rectangular matrix. We assume that p Givens rotations can be performed simultaneously. We denote lim f(x) the limit of a function f as x goes to infinity, and we recall that f(x) = o(x) means that f(x)/x tends to zero as x goes to infinity.

We let R(i,j,k), $i \neq j$, $1 \leq i,j \leq m$ and $1 \leq k \leq n$, denote the rotation in plane (i,j) which annihilates the element A(i,k).

To introduce existing algorithms, let us fix m = 13 and n = 6 for the purpose of illustration. The table of figure 1 describes Sameh and Kuck's annihilation scheme [17]. An integer r is entered when zeros are created at the r-th step.

*					
12	*				
11	13	*			
10	12	14	*		
9	11	13	15	*	
8	10	12	14	16	*
7	9	11	13	15	17
6	8	10	12	14	16
5	7	9	11	13	15
4	6	8	10	12	14
3	5	7	9	11	13
2	4	6	8	10	12
1	3	5	7	9	11

Figure 1 : Sameh and Kuck 's scheme

We do not specify completely each rotation. A zero can be created in a row using any other row with the same number of annihilated elements. For instance at step 3, we can perform simultaneously the rotations R(11,10,1) and R(13,12,2). R(13,12,2) is the only possible choice, but R(11,10,1) can be replaced by R(11,x,1) for any $x \leq 10$. Sameh and Kuck's scheme is easy to program and to analyze. Clearly, the total number of steps is m+n-2 if m>n (17 in the example) and 2n-3 otherwise.

A slight modification of the previous scheme [17] consists of performing $2^{\lfloor \log_2 m \rfloor - 1}$ rotations at the first step, and using the zeros created to annihilate one element per step (instead of every second step) in the last row, up to column $\lfloor \log_2 m \rfloor - 1$. The total number of steps is reduced to $n+m-\lfloor \log_2 m \rfloor - 1$ (15 in the example, see figure 2).

*					
9	*				
8	11	*			
7	10	12	*		
6	9	11	13	*	
5	8	10	12	14	*
4	7	9	11	13	15
3	6	8	10	12	14
2	5	7	9	11	13
1	4	6	8	10	12
1	3	5	7	9	11
1	2	4	6	8	10
1	2	3	5	7	9

Figure 2 : Sameh and Kuck modified

However, it is possible to reduce A to upper triangular form in only 14 steps, using a "greedy" scheme [1], [13] which performs at each step as many rotations as possible, annihilating the elements in each column from bottom to top and in each row from left to right. This scheme is depicted in the table of figure 3: it begins very fast since it performs 6 rotations from step 1 to step 2, 5 rotations from step 4 to step 8, 4 rotations at the steps 9 and 10, but it terminates slowly with only 2 rotations at the steps 11 and 12 and one at the last two steps.

*					
4	*				
3	6	*			
3	5	8	*		
2	5	7	10	*	
2	4	7	9	12	*
2	4	6	8	11	14
1	3	6	8	10	13
1	3	5	7	9	12
1	3	5	7	9	11
1	2	4	6	8	10
1	2	4	6	8	10
1	2	3	5	7	9

Figure 3 : the Greedy scheme

Finally, we introduce th Fibonacci schemes of Modi and Clarke [13]. The Fibonnacci annihilation scheme of order 1 is derived as indicated in figure 4. We fill up the first column from top to bottom . There is a single zero, below are u_2=2 successive copies of -1, then u_3=3 copies of -2, and so on (there are $u_k=u_{k-1}+1$ va as of -(k-1), with $u_1=1$). The second column is like the first except that all the entries are increased by 2, and the whole column moved down one place. Other columns are filled up using the

same rule. Now, to get an annihilation scheme, just add u+1 to all integers in the lower part of the matrix, where u is the integer in the left hand bottom corner. It is not straightforward to ensure that this scheme is actually a Givens annihilation scheme: we would need to check carefully whether it is possible to annihilate the elements of the matrix in such a way. However, this can be done owing to the very systematic construction of the scheme.

*					
0	*				
-1	2	*			
-1	1	4	*		
-2	1	3	6	*	
-2	0	3	5	8	*
-2	0	2	5	7	10
-3	0	2	4	7	9
-3	-1	2	4	6	9
-3	-1	1	4	6	8
-3	-1	1	3	6	8
-4	-1	1	3	5	8
-4	-2	1	3	5	7

Add 5 to all entries to get the annihilation scheme
Figure 4: Fibonnacci scheme of order 1

The Fibonacci scheme of order 2 is obtained in a similar way, replacing the relation $u_k=u_{k-1}+1$ by $u_k=u_{k-1}+u_{k-2}+1$ (with $u_1=1$, $u_2=2$), and adding 3 instead of 2 to the entries of column j to get the entries of column j+1.

The number of steps for the Fibonnacci scheme of order 1 is equal to $k+1+2(n-1)$, where k is the least integer such that $k(k+1)/2 \geq m-1$ (leading to 15 in our example). The number of steps for the scheme of order 2 is 19 in our example. Rather than providing the exact expression (there is no simple formula), we recall the asymptotic value $\log_\alpha m + 3n$ (see [13]), where $\alpha = (1+\sqrt{5})/2$.

For large values of m and/or n, it is not evident to select the best algorithm among the preceding ones. The following complexity analysis will give criterions for such a choice.

3. OPTIMALITY OF THE GREEDY ALGORITHM

In this section, we show that the greedy algorithm is optimal for decomposing any rectangular matrix A of size m x n, m ≥n, assuming that p = ⌊m/2⌋. An algorithm M requiring T steps to achieve the Givens factorization of A is represented by M = (M(1),...,M(T)) where for 1≤t≤T, M(t) is a group of r(t) disjoint rotations (r(t)≤p), which can be performed simultaneously. M is used in order to construct a sequence A(t) such that : A(0)=A and for 1≤t≤T, A(t) is obtained by applying in parallel the rotations in M(t) to A(t-1). For short we use (M,T,R) where R = r(1)+...+r(T). We say that (M,T,R) is a Givens sequence [13] if zeros once created are preserved, i.e. for any couple of distinct rotations R(i,j,k) and R(i',j',k') in M we have (i,k) ≠ (i',k'). We say that (M,T,R) is a Standard Parallel Givens Sequence (SPGS for short) if it is a Givens sequence which reduces A to upper triangular form and annihilates elements from left to right and from bottom to top, i.e. if R(i,j,k) ∈ M(t) and R(i',j',k') ∈ M(t') where t≤t', then k<k' or (k=k' and i≥i'). It is clear that all the algorithms introduced in the previous section are SPGS, in particular the Greedy algorithm. Finally, for any algorithm (M,T,R), we call r(i,t) the number of zeros in column i introduced at time t and s(i,t) the total number of zeros in column i after step t. Hence for Givens Sequences we have s(i,t) = r(i,1) + ... + r(i,t). A simple proof (see [1]) enables us to characterize the SPGS among the Givens Sequences:

Lemma 1 : Let M be a Givens Sequence. M is a SPGS if and only if :

$s(0,t) = m$ for $t \geq 0$

$s(i,0) = 0$ for $1 \leq i \leq m$

$s(i,t) \leq s(i,t-1) + \lfloor (s(i-1,t-1) - s(i,t-1))/2 \rfloor$ for $i,t \geq 1$.

It is clear that the length of a SPGS is equal to :

$$T = \min \{ t / s(n,t) \geq m - n \}$$

Informally, Lemma 1 means that no more zeros than half the difference between the number of zeros in column i and i-1 at step t-1 can be created in column i at step t.

Lemma 2 : In the set of opt..al algorithms there exists a SPGS.

Sketch of proof : Although the result could appear to be intuitively evident, the proof is long and tedious. Lemma 1 has been proved in Cosnard and Robert [3]. We consider an algorithm (M,T,R) which reduces A to upper triangular form and we show that there exists a SPGS of length T.

In the first step, we construct by induction an algorithm (M',T,R) which annihilates the elements in any row from left to right, in such a way that given any i,t, the number of zeros in column i at step t is greater in

the algorithm M' than in M, i.e. $s'(i,t) \geq s(i,t)$. Because the rotations in each step act on disjoint pairs of rows, we consider each pair separately. Let R(i,j,k) belong to M(t+1). We must construct an operation in M'(t+1) that preserves the induction properties. There are two cases :
- If a new zero is required in row i of A'(t), use a rotation to introduce it or use a permutation to bring it up from row j.
- If row i of A'(t) already has enough zeros, do nothing (identity).
What is actually done is determined by an analysis of cases; see [3] for details.

In the second step, we construct a Givens sequence M'' of length T, which triangularizes A. M'' contains the same rotations as M' but those which reduce to the identity or a permutation are suppressed: instead of permuting the rows of A, we permute the indices of the rows. Thus any rotation in M'' strictly increases the number of zeros: M'' is a Givens sequence which reduces A to upper triangular form using only $mn-n(n+1)/2$ rotations and annihilates the elements from left to right. Clearly, the length of M'' is T.

From this Givens sequence, the last part of the proof consists in constructing a Standard Parallel Givens Sequence of the same length : this can be done by constructing a new permutation of the indices of the rows in a very straightforward manner. Again, see [3] for details.

Theorem 1 : For any value of m and n (with $m \geq n$), and for $p = \lfloor m/2 \rfloor$, the greedy SPGS is an optimal algorithm.

Sketch of proof : From Lemma 2 we deduce that we have only to show the optimality of the greedy SPGS among the SPGS. This optimality follows from the inequality:

$$s(i,t) \leq sg(i,t) \text{ for } i,t \geq 1$$

We prove it by induction on i and t. For i or t equal to 0, we have :

$$s(0,0)=sg(0,0)=m,\ s(0,t)=sg(0,t)=m \text{ for } t \geq 1,\ s(i,0)=sg(i,0)=0 \text{ for } i \geq 1$$

Assume that for j and u such that $j<i$ or ($j=i$ and $u<t$) we have :

$$s(j,u) \leq sg(j,u)$$

Then from Lemma 1 :

$$\begin{aligned} s(i,t) &\leq s(i,t-1) + \lfloor (s(i-1,t-1) - s(i,t-1))/2 \rfloor \\ &\leq \lfloor (s(i-1,t-1) + s(i,t-1))/2 \rfloor \\ &\leq \lfloor (sg(i-1,t-1) + sg(i,t-1))/2 \rfloor \\ &\leq sg(i,t-1) + \lfloor (sg(i-1,t-1) - sg(i,t-1))/2 \rfloor \\ &\leq sg(i,t). \end{aligned}$$

This ends the proof of theorem 1.

4. COMPLEXITY RESULTS

The complexity of the QR decomposition using plane rotations is the number of steps needed to perform an optimal algorithm. From theorem 1, we deduce that the complexity equals the number of steps of the greedy SPGS. However we do not succeed in obtaining an exact formula for the complexity OPT(m,n) for all m and n. But we can prove the following results:

Theorem 2 : Let $p = \lfloor m/2 \rfloor$.
1. If $n = m$ (A is a square matrix), then $OPT(m,n) = 2n + o(n)$.
2. If n is fixed, then $OPT(m,n) = \log_2 m + o(\log_2 m)$.
3. If $m = o(n^2)$, then $OPT(m,n) = 2n + o(n)$.
4. If $m = o(n^k)$, with $k \geq 3$, then $2n + o(n) \leq OPT(m,n) \leq 3n + o(n)$.

Proof: The proofs of 1 and 3 can be found respectively in [1] and [3]. 2 follows from theorem 1 and a result of Modi and Clarke [13] which states that, for n fixed, the number of steps of the greedy algorithm is $\log_2 m + o(\log_2 m)$. In order to prove 4, we show first that in this case the number of steps of the greedy algorithm is greater than or equal to $2n + o(n)$ following the proof of 3. The second part of the proof can be deduced from the analysis of the Fibonacci algorithm of order 2, whose number of steps is $3n + o(n)$.

When the resolution with p processors of a problem of size mxn requires a time of computation $T_p(m,n)$, the speed up of the parallelization is defined by $S_p(m,n)=T_1(m,n)/T_p(m,n)$. Moreover a good measure of the degree of parallelism is given by $E_p(m,n)= S_p(m,n)/p$. Clearly, E is not greater than 1 and a parallel algorithm is of good efficiency if E is constant. From the preceding results, we can evaluate the asymptotic efficiency ASE(m,n) of the problem.

Corollary 1 : Let $p = \lfloor m/2 \rfloor$.
1. If $m = n$, then $ASE(m,n) = 1/2$.
2. If $m = \beta n$, then $ASE(m,n) = 1 - 1/2\beta$.
3. If $m = o(n^2)$, then $ASE(m,n) = 1$.
4. If $m = o(n^k)$, then $ASE(m,n) \geq 2/3$.
5. If n is fixed, then $ASE(m,n) = 2n/\log_2 m$.

From the corollary, we deduce that, for m proportionnal to n or m of the order of n^k, the parallelization of the QR decomposition is very efficient since the efficiency is asymptotically constant. However for n fixed, it decreases to zero as m goes to infinity.

5. ASYMPTOTIC OPTIMALITY

Section 4 could appear to be only of theoretical importance. However the table of figure 5 shows the asymptotic estimations that we have obtained are good even for realistic (small) sizes of the matrix.

M=N	7	8	9	10	14	15	16	17	18
OPT	10	11	13	15	23	24	26	28	30

M=N	32	40	64	128	256	512	1024	2048	4096
OPT	56	72	118	243	495	1000	2015	4051	8129

Number of steps for the greedy SPGS, square matrix;
to be compared with 2N.
Figure 5

Moreover, this asymptotic analysis allows us to select among the algorithms introduced in section 2 those which are aymptotically optimal. This is important since the computation of the indices of the rows involved in plane rotations at a given step of the greedy algorithm is somewhat difficult. Hence it must be avoided. This can be done by replacing the greedy SPGS by another algorithm which, although not optimal, will have good performances and could be implemented in a simple way.

Corollary 2 : Let $p = \lfloor m/2 \rfloor$.

1. If $m = n$, then Sameh and Kuck scheme is asymptotically optimal.
2. If $m = \beta n$, then the Fibonacci scheme of order 1 is asymptotically optimal.
3. If $m = o(n^2)$, then the Fibonacci scheme of order 1 is asymptotically optimal.
4. If $m = o(n^k)$ and $m \geq n^2$, then the Fibonacci scheme of order 2 is of asymptotic efficiency 2/3.

When n is fixed and m goes to infinity, none of the previous algorithms but the greedy is asymptotically optimal.

6. WITH P ROTATIONS SIMULTANEOUSLY

In this section, we shall limit the number of plane rotations which can be performed simultaneously to a given integer $p < \lfloor m/2 \rfloor$. The first following results are straightforward extensions of results presented in section 3.

Lemma 3 : Let $p < \lfloor m/2 \rfloor$ and M be a Givens Sequence. M is a SPGS if and only if :

$s(0,t) = m$ for $t \geq 0$

$s(i,0) = 0$ for $1 \leq i \leq m$

$$r(i,t) \leq \min \{ p - \sum_{j=1}^{i-1} r(j,t) \ , \ \lfloor (s(i-1,t-1) - s(i,t-1))/2 \rfloor \}$$ for $i,t \geq 1$.

Lemma 3 means that there is now one more constraint: the total number of zeros introduced at time t cannot exceed p. Moreover, just as before, no more zeros than half the difference between the number of zeros in column i and i-1 at step t-1 can be created in column i at step t. We shall see later on that these constraints make the analysis of the problem much more difficult.

Lemma 4 : In the set of optimal algorithms there exists a SPGS.

Proof: The proof follows in a similar way as that of Lemma 2.

A great difference with the case $p = \lfloor m/2 \rfloor$ is the fact that there is not a unique greedy SPGS. The greedy algorithms can be unformally described in the following manner: at each step do the maximum number of rotations.

Definition : Let M be a SPGS. M is a greedy algorithm if and only if:

$$\sum_{j=1}^{n} r(j,t) = \min \{ p \ , \ \sum_{j=1}^{n} \lfloor (s(j-1,t-1) - s(j,t-1))/2 \rfloor \}$$

Clearly this can be achieved with different strategies. To illustrate this fact we present two algorithms: the vertical greedy, which fills up the columns, and the horizontal greedy, which fills up the rows.

Vertical greedy:

- $s_{vg}(0,t) = m$ for $t \geq 0$
- $s_{vg}(i,0) = 0$ for $1 \leq i \leq m$
- $r_{vg}(i,t) = \min \{ p - \sum_{j=1}^{i-1} r_{vg}(j,t) \ , \ \lfloor (s_{vg}(i-1,t-1) - s_{vg}(i,t-1))/2 \rfloor \}$

for any $i,t \geq 1$.

Horizontal greedy:

- $s_{hg}(0,t) = m$ for $t \geq 0$
- $s_{hg}(i,0) = 0$ for $1 \leq i \leq m$
- $r_{hg}(i,t) \leq \min \{ p - \sum_{j=i+1}^{n} r_{hg}(j,t) + \lfloor (s_{hg}(i-1,t-1) - s_{hg}(i,t-1))/2 \rfloor \}$.

for any $i,t \geq 1$.

The problem of optimality is still open. In particular none of the two preceding greedy algorithms is optimal. However for particular choices of m, n or p both can be optimal.

Theorem 3 : Given m, n and p, there exists an optimal algorithm in the set of greedy SPGS.

Proof: Given a SPGS, we construct a greedy SPGS performing the decomposition in fewer steps. This is done by induction on t.

Assume that $s(i,t) \leq sg(i,t)$ and let us prove this relation for t+1. Recall that $s(i,t+1) = s(i,t) + r(i,t)$ and set $\delta(i,t) = sg(i,t) - s(i,t)$. We define $rg_1(i,t+1) = r(i,t+1) - \delta(i,t)$. From Lemma X we deduce that :

$$r(i,t+1) \leq \lfloor (s(i-1,t) - s(i,t))/2 \rfloor$$

which implies that:

$$\begin{aligned} rg_1(i,t+1) &\leq \lfloor (s(i-1,t) - s(i,t))/2 \rfloor - sg(i,t) + s(i,t) \\ &\leq \lfloor (s(i-1,t) + s(i,t))/2 \rfloor - sg(i,t) \\ &\leq \lfloor (sg(i-1,t) + sg(i,t))/2 \rfloor - sg(i,t) \\ &\leq \lfloor (sg(i-1,t) - sg(i,t))/2 \rfloor \end{aligned}$$

Moreover

$$\begin{aligned} rg_1(i,t+1) &\leq r(i,t+1) \\ &\leq p - \sum_{j=1}^{i-1} r(j,t+1) \end{aligned}$$

$$\leq p - \sum_{j=1}^{i-1} rg_1(j,t+1)$$

Hence we obtain:

$$rg_1(i,t+1) \leq \min \{ p - \sum_{j=1}^{i-1} rg_1(j,t+1) , \lfloor (sg(i-1,t) - sg(i,t))/2 \rfloor \}$$

Lemma 1 implies that this scheme is a SPGS. It is now easy to derive a greedy SPGS.

If $\sum_{j=1}^{n} rg_1(j,t+1) < \min \{ p , \sum_{j=1}^{n} \lfloor (s(j-1,t) - s(j,t))/2 \rfloor \}$

saturate the inequality by adding as many zeros as possible, for instance from left to right.

We then have:

$$\begin{aligned} s(i,t+1) &= s(i,t) + r(i,t) \\ &\leq sg(i,t) + rg_1(i,t) \\ &\leq sg(i,t) + rg(i,t) = sg(i,t+1) \end{aligned}$$

which concludes the proof.

From the theorem, it can be easily seen that, although there exists an optimal greedy algorithm for any m, n and p, we do not know how to construct it. Figure 6 shows that this optimal greedy SPGS can be different from the vertical or the horizontal greedy. We choose m=n=15 and p=4.

*														
5	*													
4	8	*												
4	7	12	*											
3	7	11	14	*										
3	6	10	13	17	*									
3	6	10	13	16	19	*								
2	6	9	12	15	18	21	*							
2	6	9	12	15	17	20	23	*						
2	5	9	11	14	17	19	22	25	*					
2	5	8	11	14	16	19	21	24	27	*				
1	5	8	11	14	16	18	21	23	26	28	*			
1	4	8	10	13	16	18	20	22	25	27	29	*		
1	4	7	10	13	15	18	20	22	24	26	28	30	*	
1	3	7	9	12	15	17	19	21	23	25	27	29	31	*

Figure 6a : Vertical greedy

*													
18	*												
16	19	*											
15	17	20	*										
10	16	18	21	*									
7	12	17	19	22	*								
4	8	13	18	20	23	*							
4	6	9	14	19	21	24	*						
3	5	8	11	15	20	22	25	*					
2	5	9	10	12	16	21	23	26	*				
2	4	6	9	11	13	17	22	24	27	*			
1	3	6	8	10	12	14	18	23	25	28	*		
1	3	5	7	9	11	13	15	19	24	26	29	*	
1	2	4	6	8	10	12	14	16	20	25	27	30	*
1	2	3	5	7	9	11	13	15	17	21	26	28	31	*

Figure 6b : Horizontal greedy

*													
14	*												
13	16	*											
11	14	18	*										
8	12	16	19	*									
7	11	14	17	20	*								
4	8	12	16	19	22	*							
4	6	10	14	17	20	23	*						
3	5	9	12	15	19	21	24	*					
2	5	7	10	13	17	20	22	25	*				
2	4	6	9	12	15	18	21	23	26	*			
1	3	6	8	10	13	17	20	22	24	27	*		
1	3	5	7	9	11	15	18	21	23	25	28	*	
1	2	4	6	8	10	13	16	19	22	24	26	29	*
1	2	3	5	7	9	11	15	18	21	23	25	27	30	*

Figure 6c : Better greedy !

5. CONCLUDING REMARKS

We have shown that the greedy algorithm introduced independently by Modi and Clarke [13] and Cosnard and Robert [1] is always optimal for computing the QR decomposition of a rectangular m by n matrix using Givens rotations, assuming that $\lfloor m/2 \rfloor$ rotations can be performed simultaneously. This result includes all parallel algorithms using Givens rotations, and not only Givens sequences. Furthermore, whenever the greedy algorithm would be too difficult to implement on a parallel computer, the asymptotic expressions we give permit to select among the well-known algorithms the ones which are asymptotically optimal. In particular if m and n are proportional, the Fibonacci scheme of order 1 of Modi and Clarke [13] is to be preferred to Sameh and Kuck's scheme [17] unless the matrix is square. In the case where m is greater than $O(n^2)$, the Fibonacci scheme of order 2 seems to be preferred.

If the number of Givens rotations that can be performed simultaneously is reduced to $p < \lfloor m/2 \rfloor$ (e.g. due to the amount of available hardware), the situation appears more complicated, since there are many greedy algorithms in this case. However, we have proven that there always exists an optimal greedy algorithm, although its systematic construction remains open.

REFERENCES

[1] COSNARD M., ROBERT Y., Complexité de la factorisation QR en parallèle, C. R. Acad. Sc. Paris, 297, A, 549-552 (1983)

[2] COSNARD M., ROBERT Y, Complexity of parallel algorithms: the example of the QR decomposition of a rectangular matrix, Proc. COGNITIVA 85, Paris, June 3-7 (1985)

[3] COSNARD M., MULLER J.M., ROBERT Y, Parallel QR decompostion of a rectangular matrix, IMAG Grenoble Technical Report 466, October 1984, submitted to Numerische Mathematik

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

[5] GENTLEMAN W.M., Least squares computations by Givens transformations without square roots, J. Inst. Maths. Applics. 12, 329-336 (1973)

[6] GENTLEMAN W.M., Error analysis of QR decomposition by Givens tranformations, Linear Algebra and Appl. 10, 189-197 (1975)

[7] GOLUB G.H., PLEMMONS R., Large-scale geodetic least-squares adjustment by dissection and orthogonal decomposition, Linear Algebra and Appl. 38, 3-28 (1980)

[8] HELLER D., A survey of parallel algorithms in numerical linear algebra, SIAM Review 20, 740-777 (1978)

[9] HORWARTH S. Jr., A new adaptative recursive LMS filter, Proc. Digital Signal Processing, Cappellini and Constantinides eds., 21-26, Academic Press (1980)

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

[11] LAWSON C., HANSON R., Solving least squares problems, Prentice-Hall (1974)

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

[13] MODI J.J., CLARKE M.R.B., An alternative Givens ordering, Numerische Mathematik 43, 83-90 (1984)

[14] SAMEH A., Solving the linear least squares problem on a linear array of processors, Proc. Purdue Workshop on algorithmically-specialized computer organizations, W. Lafayettte, Indiana, September 1982

[15] SAMEH A., Numerical parallel algorithms - a survey, Proc. High Speed computer and algorithm organization, D.Kuck, D. Lawrie and A. Sameh eds., Academic Press, 207-228 (1977)

[16] SAMEH A., An overview of parallel algorithms, Bulletin EDF C1, 129-134 (1983)

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

Comparaison des méthodes parallèles de diagonalisation pour la résolution de systèmes linéaires denses

Michel Cosnard[+], Yves Robert[+] et Denis Trystram[++]

A paraître aux C.R.A.S. Paris

[+] C.N.R.S.Laboratoire TIM3- I.N.P.G. BP 68
38402 ST MARTIN D'HERES CEDEX

[++] Ecole Centrale PARIS Grande voie des vignes
92295 CHATENAY MALABRY CEDEX

INFORMATIQUE THEORIQUE. - Comparaison des méthodes parallèles de diagonalisation pour la résolution de systèmes linéaires denses. Note de Michel Cosnard [+], Yves Robert [+] et Denis Trystram [*], présentée par Jacques-Louis Lions.

Nous comparons les différentes parallélisations des méthodes de Jordan et des paramètres pour résoudre par diagonalisation des systèmes linéaires denses. Nous présentons les versions les plus efficaces: Jordan par lignes et les paramètres par colonnes. Pour $p=\alpha n$ processeurs, nous montrons que la méthode des paramètres est plus rapide que celle de Jordan pour $\alpha \leq 0.44$.

COMPUTER SCIENCE. - Comparison of parallel diagonalization methods for solving dense linear systems.

We compare different parallel algorithms for the methods of Jordan and of the parameters for solving dense linear systems. We introduce the most efficient versions: Jordan by rows and the parameters by columns. For $p=\alpha n$ processors, we show that the method of parameters is better than the Jordan method for $\alpha \leq 0.44$.

1. INTRODUCTION. - Pour résoudre un système linéaire dense Ax=b sur un ordinateur séquentiel classique, la méthode la plus utilisée est celle de Gauss. Celle-ci effectue deux opérations de nature apparemment différente: triangularisation de la matrice et inversion d'un système triangulaire. Une autre approche consiste à rassembler ces deux étapes en une seule: diagonaliser la matrice. En général, on utilise la méthode de Jordan dans ce cas. Cependant cette méthode a un coût supérieur à celle de Gauss ($n^3+O(n^2)$ opérations arithmétiques contre $2n^3/3+O(n^2)$). Récemment, Huard [14] a proposé une méthode, dite des paramètres, qui peut être vue comme une variante de la méthode de Gauss qui intègre dans l'étape de triangularisation, une étape de résolution et donc permet de diagonaliser la matrice en effectuant le même nombre d'opérations.

La méthode de Gauss, dans la première phase de triangularisation, revient à calculer une suite de matrices triangulaires inférieures unicolonnes $L_1, L_2, \ldots, L_n$ telles que $L_n L_{n-1} \ldots L_1 A = R$. De même, la méthode de Jordan conduit à $J_n J_{n-1} \ldots J_1 A = D$ (où D est la diagonale de R). La méthode des paramètres s'écrit $P_n P_{n-1} \ldots P_1 A = D'$.

Pour la méthode de Jordan, à l'étape k, $J_k = R_k L_k$, avec R_k triangulaire supérieure unicolonne. Comme L_k et R_h commutent pour k>h, on regroupe les termes en R et en L pour obtenir: $R_n R_{n-1} \ldots R_1 \cdot L_n L_{n-1} \ldots L_1 = D$

Pour la méthode des paramètres, à l'étape k, $P_k = G_k H_k$, avec H_k matrice triangulaire inférieure uniligne et G_k matrice triangulaire supérieure unicolonne. On obtient ainsi la relation: $G_n H_n G_{n-1} H_{n-1} \ldots G_1 H_1 = D'$.

On vérifie par récurrence que, pour tout k, $G_k = R_k$, et que les pivots

construits au cours de la factorisation sont les mêmes pour toutes les méthodes: D = D'. D'autre part, on a $H_nH_{n-1}...H_1A = R$. Cette égalité correspond à l'algorithme de triangularisation de Gauss par lignes de la matrice A: à chaque étape k, on annule les k-1 premiers éléments de la ligne k. Les coefficients des matrices H sont reliés à ceux des matrices L par les relations h_{kl} (de la matrice H_k) = l_{kl} (de la matrice L_l).

Si, après l'algorithme de triangularisation par lignes, on continue les calculs jusqu'à diagonalisation comme par la méthode de Jordan, alors $R_nR_{n-1}...R_1 \cdot H_nH_{n-1}...H_1 = D$, c'est-à-dire $G_nG_{n-1}...G_1 \cdot H_nH_{n-1}...H_1 = D$.

En comparant ces égalités, on vérifie que la méthode des paramètres consiste à intercaler une phase de diagonalisation entre chaque étape de triangularisation par lignes (et non par colonnes, comme dans l'algorithme usuel).

La programmation sur des ordinateurs parallèles de la méthode de Gauss a été étudiée par de nombreux auteurs (voir les revues de Heller [5], de Sameh [11] et les travaux de Lord et coll.[7]). Par contre, il n'existe aucun travail sur la parallélisation des méthodes de diagonalisation. Le but de ce travail est de proposer des algorithmes parallèles pour cette classe de méthodes, de comparer leurs performances et d'étudier leur complexité. Les démonstrations des résultats se trouvent dans [3].

2. PRESENTATION DES ALGORITHMES SEQUENTIELS. - Nous supposons par la suite que l'on cherche à résoudre simultanément m systèmes linéaires denses à n inconnues (de même matrice) et que m est inférieur ou égal à n. A sera donc une matrice de taille n x (n+m).

Nous rappelons que les algorithmes de Jordan et des paramètres sont:

```
(* algorithme de Jordan *)
  Pour k <- 1 à n
(* préparation de la ligne k *)
      c <- 1/a_kk
      Pour j <- k+1 à n+m
          a_kj <- a_kj *c
(* mise à zéro colonne k *)
      Pour i <- 1 à n , i ≠ k
          Pour j <- k+1 à n+m
              a_ij <- a_ij - a_ik* a_kj
```

```
(* algorithme des paramètres*)
  Pour k <- 1 à n
(* élimination de la ligne k *)
      Pour i <- 1 à k-1
          Pour j <- k à n+m
              a_kj <- a_kj - a_ki* a_ij
(* préparation de la ligne k *)
      c <- 1/a_kk
      Pour j <- k+1 à n+m
          a_kj <- a_kj *c
(* mise à zéro colonne k, lignes 1 à k *)
      Pour i <- 1 à k-1
          Pour j <- k+1 à n+m
              a_ij <- a_ij - a_ik* a_kj
```

Nous supposons que l'on peut accéder aux éléments de la matrice A, soit par lignes, soit par colonnes et que le temps d'exécution est le même pour chaque opération arithmétique: ce temps sera notre unité.

3. UN RESULTAT D'OPTIMALITE. -Considérons la résolution en parallèle, avec p processeurs, d'un ensemble de tâches T_{kj}, $1 \leq k \leq n$, $k+1 \leq j \leq n+m$, avec les contraintes d'ordonnancement:

(A) la tâche $T_{k,k+1}$ doit être exécutée avant les tâches $T_{k+1,j}$ $(j \geq k+2)$

(B) pour j fixé > k, la tâche $T_{k,j}$ doit être exécutée avant la tâche $T_{k+1,j}$

Pour une valeur donnée de k, il y a donc n+m-k tâches à effectuer. Nous supposons de plus que toutes les tâches T_{kj} $(j>k)$ d'un même niveau k ont le même temps d'exécution t_k. Le graphe d'ordonnancement des tâches est:

T_{12}	T_{13}	T_{14} ...	$T_{1,n+1}$	... $T_{1,n+m}$	niveau 1
	T_{23}	T_{24} ...	$T_{2,n+1}$	... $T_{2,n+m}$	niveau 2
		...		...	
			$T_{n,n+1}$	... $T_{n,n+m}$	niveau n

PROPOSITION 1. - Parmi les algorithmes optimaux, il en existe qui satisfont de plus à la contrainte d'ordonnancement (C):

(C) la tâche $T_{k,j}$ doit être exécutée avant les tâches $T_{k,j+1}$ $(j \geq k+2)$.

Introduisons maintenant l'algorithme glouton G, défini informellement de la manière suivante: l'algorithme G exécute les tâches en balayant les lignes du graphe d'ordonnancement de gauche à droite, en commençant à tout instant le maximum de tâches (d'où le nom de glouton). Plus précisément, l'algorithme G exécute les tâches dans l'ordre suivant:

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

où la notation $T \ll T'$ signifie que l'exécution de la tache T ne peut débuter après l'exécution de la tache T' (mais la simultanéité est possible).

PROPOSITION 2. - L'algorithme glouton G est optimal. Son temps d'exécution est égal à $TG = (1/p) \sum_{1 \leq i \leq n+m-p} (n+m-i).t_i + \sum_{n+m-p+1 \leq i \leq n} t_i$.

4. METHODE DE JORDAN PARALLELE. - Supposons que le nombre p de processeurs soit compris entre 1 et O(n). Suivant l'accès aux données, on peut paralléliser la méthode de Jordan soit sous la forme présentée précédemment que nous appelerons Jordan par lignes, soit en inversant, dans celle-ci, les boucles i et j imbriquées. On obtient alors:

```
(* JORDAN par lignes *)
  Pour k <- 1 à n
      Pour j <- k+1 à n+m
          akj <- akj / akk
       Pour i <- 1 à n, i≠k
           Pour j <- k+1 à n+m
               aij <- aij - aik*akj
```

```
(* JORDAN par colonnes *)
  Pour k <- 1 à n
      Pour j <- k+1 à n+m
          akj <- akj / akk
          Pour i <- 1 à n , i ≠ k
              aij <- aij - aik* akj
```

Dans la version par lignes, les tâches sont définies ainsi:

JL_{kk} : Pour j <- k+1 à n+m
$a_{kj} \leftarrow a_{kj} / a_{kk}$ coût=n+m-k unités

JL_{ki} : Pour j <- k+1 à n+m $i = 1, \ldots, n$
$a_{ij} \leftarrow a_{ij} - a_{kj}$ $i \neq k$ coût=2(n+m-k) unités

La complexité de cette version sera notée JL_p: c'est le minimum des temps d'exécution de tous les algorithmes et aussi le temps d'exécution d'un algorithme optimal. Nous noterons EJL_p l'efficacité asymptotique. Les contraintes du graphe d'ordonnancement sont

- la tâche JL_{kk} doit êtr exécutée avant les tâches JL_{ki}
- la tâche JL_{ki} doit être exécutée avant la tâche $JL_{k+1,i}$

PROPOSITION 3. - i) Si $p=n-1$, $JL_p=3n(m+(n-1)/2)$ et $EJL_p=2/3$.

ii) Si $n/2 \leq p=\alpha n \leq n-1$, $JL_p \leq n+3m + (n-1)(m+n/2)(2+(n-2)/(p-1))$ et $EJL_p \geq 2/(1+2\alpha)$

iii) Si $1 \leq p \leq n/2$, $JL_p=n^2(2m+n)/p+O(n^2/p)$ et $EJL_p=1$.

Dans Jordan par colonnes, les tâches sont:

JC_{kj} : $a_{kj} \leftarrow a_{kj} / a_{kk}$ $1 \leq k \leq n$
Pour i <- 1 à n , i ≠ k $k+1 \leq j \leq m+n$ coût : 2n-1 unités
$a_{ij} \leftarrow a_{ij} - a_{ik}* a_{kj}$

Les contraintes d'ordonnancement satisfont aux hypothèses de la section 3, ce qui conduit au résultat suivant:

PROPOSITION 4. - i) Parmi les algorithmes optimaux, il en existe qui satisfont de plus à la contrainte d'ordonnancement (C):
- la tâche $JC_{k,j}$ doit être exécutée avant les tâches $JC_{k,j+1}$ $(j \geq k+2)$ (C)

ii) L'algorithme glouton défini précédemment est optimal.

iii) $JC_p = (2n-1)((n+m-p)(n+m+p-1)/2p + p-m)$

Pour m=1 et $p = \alpha.n$, $JC_p = (1+\alpha^2)n^2/\alpha + O(n)$ et $EJC_p = 1/(1+\alpha^2)$

5. METHODE DES PARAMETRES PARALLELE. - La méthode des paramètres comporte deux étapes par itération:

- annulation des k-1 premiers éléments de la ligne k et modifications correspondantes des n+m-k+1 éléments suivants: étape de descente.
- annulation des k-1 premiers éléments de la colonne k et modifications correspondantes des n+m-k derniers éléments des k-1 premières lignes: étape de remontée.

Suivant l'accès aux éléments de la matrice, on peut écrire chacune des deux étapes de deux manières différentes: par lignes ou par colonnes.

La descente en lignes n'est pas parallélisable, puisque, pour chaque valeur de i, les n+m-k+1 éléments a_{kj} sont modifiés. En effet, ceci contredit notre hypothèse d'interdire des modifications simultanées d'un même élément. Par conséquent, deux versions sont possibles: descente colonnes - remontée lignes et descente colonnes - remontée colonnes.

Contrairement au cas précédent, ces deux versions n'ont pas le même temps d'exécution. La version des paramètres colonnes-lignes, étudiée dans [3], est moins parallélisable que son homologue colonnes-colonnes. L'algorithme retenu est:

```
                (* descente colonnes - remontée colonnes *)
                Pour k <- 1 à n-1
                    Pour j <- k+1 à n+m
                        a_kj <- a_kj / a_kk
                    Pour i <- 1 à k-1
P_kj:                       a_ij <- a_ij - a_ik * a_kj
                    Pour i <- 1 à k
                        a_k+1,j <- a_k+1,j - a_k+1,i * a_ij
```

On retrouve la même structure globale que pour l'algorithme de Jordan par colonnes, mais le temps d'exécution de chaque tâche est variable. P_{kj} s'effectue en 4k-1 unités. Les contraintes temporelles restent inchangées, et le graphe d'ordonnancement vérifie encore les hypothèses précédentes:

PROPOSITION 5. - i) L'algorithme glouton est optimal.

ii) $PCC_p = ((n+m-p)/p)\,[(n+m-p+1)(4n+4m+8p-1)/6 - (n+m)\,]$
$+ (p-m+1)(4n+2m-2p-5)$

iii) Pour m=1 et $p = \alpha n$, $PCC_p = (-2\alpha^3+6\alpha^2+2)\,n^2/3\alpha + O(n/\alpha)$

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

6. COMPARAISON. - L'étude précédente des méthodes de Jordan et des paramètres nous montre quelles stratégies adopter pour la parallélisation des méthodes.

PROPOSITION 6. - i) $JL_p \leq JC_p$, quel que soit α.

ii) Pour $\alpha \leq 0.44$, $PCC_p < JL_p$ et pour $\alpha \geq 0.45$, $JL_p < PCC_p$

7. CONCLUSION. - La meilleure parallélisation de la méthode de Jordan est par lignes alors que la meilleure parallélisation des paramètres est par colonnes. La méthode de Jordan est plus apte à la parallélisation: l'efficacité de ses versions parallèles est excellente. Cependant, cela ne permet pas de combler dans tous les cas l'écart en nombre total d'opérations qui existe entre Jordan et les paramètres. Il n'est donc pas étonnant que, pour un nombre faible de processeurs, la méthode des paramètres ait un temps d'exécution inférieur à celui de la méthode de Jordan et que ceci soit inversé pour un grand nombre de processeurs.

[1] A. BOJANCZYK, R.P. BRENT et H.T. KUNG, Preprint CMU, 1981
[2] M. COSNARD et Y. ROBERT, C.R. Acad. Sc. Paris 297, A, 1983, 549-552
[3] M. COSNARD, Y. ROBERT et D. TRYSTRAM, Rapp. Rech TIM3 Grenoble, 1985
[4] J. ERHEL et Coll., 6ème colloque international sur les méthodes de calcul scientifique et technique, 1983, Eds. Glowinski et Lions
[5] D. HELLER, SIAM Review 20, 1978, 740-777
[6] P. HUARD, Bulletin EDF, 1979, 79-98
[7] R.E. LORD, J.S. KOWALIK et S.P. KUMAR, J. ACM 30 (1), 1983, 103-117
[8] J.J. MODI et M.R.B. CLARKE, Numer. Math. 43, 1984, p 83-90
[9] Y. ROBERT et M. TCHUENTE, Rairo MAN, 1985
[10] A. SAMEH et D.J. KUCK, J. ACM 25 (1), 1978, 81-91
[11] A. SAMEH, Bulletin EDF, 1983, 129-134
[12] M. SRINIVAS, IEEE T.C. 32, 1983, 1109-1117

+ C.N.R.S. Laboratoire TIM3- I.N.P.G. BP 68
38402 ST MARTIN D'HERES CEDEX

* Ecole Centrale PARIS Grande voie des vignes
92295 CHATENAY MALABRY CEDEX

Techniques de l'Informatique, des Mathématiques, de la Microélectronique et de la Microscopie quantitative.
Unité associée au C.N.R.S. n° 397

BP 68
38402 Saint Martin d'Hères cedex
France
tél. (76) 51.46.00
secrétariat : poste 5257

RESOLUTION PARALLELE DE SYSTEMES LINEAIRES DENSES PAR DIAGONALISATION

Résumé: Nous étudions la parallélisation des méthodes de diagonalisation pour résoudre des systèmes linéaires denses. Nous présentons une approche matricielle des méthodes de Jordan et des paramètres et montrons leurs liens avec la méthode de Gauss. Diverses versions parallèles sont étudiées et, pour chacune d'elles, nous évaluons le temps d'un algorithme optimal.

Les versions parallèles des méthodes de Jordan par lignes et des paramètres par colonnes sont les plus efficaces. Pour un nombre de processeurs $p=\alpha n$, nous montrons que la version précédente des paramètres est plus rapide que celle de Jordan pour $\alpha \leq 0.44$.

Abstract: We study the parallelization of diagonalization methods for solving denses linear systems. We introduce the methods of Jordan and of the parameters through a matricial approach and show the relationship with Gauss method. Various parallel versions are studied, and for each, we evaluate the execution time of an optimal algorithm.

The parallel versions of the methods of Jordan by rows and of the parameters by columns are the most efficient. For $p=\alpha n$ processors, we show that the preceding version of the parameters is better than that of Jordan for $\alpha \leq 0.44$.

RESOLUTION PARALLELE DE
SYSTEMES LINEAIRES DENSES
PAR DIAGONALISATION

RR n° 554 - Juillet 1985
Michel COSNARD, Yves ROBERT
& Denis TRYSTRAM

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

RESOLUTION PARALLELE DE SYSTEMES LINEAIRES DENSES PAR DIAGONALISATION

M. COSNARD[+], Y. ROBERT[+] et D. TRYSTRAM[*]

[+] C.N.R.S. - Laboratoire TIM3
I.N.P.G. BP 68
38402 ST MARTIN D'HERES CEDEX

[*] Ecole Centrale PARIS
Grande voie des vignes
92295 CHATENAY MALABRY CEDEX

0. INTRODUCTION

La méthode la plus connue pour résoudre des systèmes linéaires sur des machines séquentielles est la méthode de Gauss. Pour résoudre $Ax = b$, elle consiste à travailler sur le couple (A,b) pour le transformer par éliminations successives en le couple équivalent $(R, L^{-1}b)$ où $A = LR$ avec L triangulaire inférieure et R triangulaire supérieure. Enfin, x est calculé en résolvant le système triangulaire $Rx = L^{-1}b$, c'est à dire en transformant le couple $(R, L^{-1}b)$ en $(I, R^{-1}L^{-1}b)$: le deuxième élément étant la solution du problème.

Pour des problèmes de stabilité, on a presque toujours recours à des techniques de permutation de lignes ou de colonnes. Dans ce travail nous n'aborderons pas cette question: nous concentrons notre étude sur l'évaluation du nombre d'opérations arithmétiques, sans prendre en compte les problèmes liés à l'échange de données.

La méthode de Gauss effectue donc deux opérations de nature apparemment différente: triangularisation de la matrice, inversion d'un système triangulaire. Une autre approche consiste à rassembler ces deux étapes en une seule: diagonaliser la matrice. De manière classique on utilise la méthode de Jordan dans ce cas. Cependant cette méthode a un coût supérieur à celle de Gauss ($n^3 + O(n^2)$ opérations arithmétiques contre $2n^3/3 + O(n^2)$). Assez curieusement, cette différence n'a pas reçue de justification théorique pendant longtemps.

Récemment Huard [14] a proposé une méthode, dite des paramètres, qui peut être vue comme une variante de la méthode de Gauss qui intègre dans l'étape de triangularisation, une étape de résolution et donc permet de diagonaliser la matrice en effectuant le même nombre d'opérations.

La programmation sur des ordinateurs parallèles de la méthode de Gauss a été étudiée par de nombreux auteurs [13], [19], [22], [23], [25], [27], ... Par contre il n'existe aucun travail sur la parallèlisation des méthodes de diagonalisation. Le but de cet article est de proposer des algorithmes parallèles pour cette classe de méthodes, de comparer leurs performances et d'étudier leur complexité.

1. RAPPELS : GAUSS, JORDAN et LES PARAMETRES.

Nous supposerons par la suite que l'on cherche à résoudre simultanément m systèmes linéaires denses à n inconnues (de même matrice). Ceci permet de traiter le cas d'un seul système (m =1), de l'inversion d'une matrice (m = n) et le cas d'un nombre quelconque de systèmes. A sera donc une matrice de taille n x (n + m). Par la suite, nous supposerons que m est inférieur ou égal à n.

a) GAUSS : Le principe de la méthode de Gauss séquentielle consiste à éliminer de proche en proche les termes sous diagonaux de la matrice A. On construit ainsi une suite de matrices $(A_k)_{1 \le k \le n}$ par récurrence :

$$A_{k+1} = L_{k+1} A_k$$

En partant de $A_0 = A$, on obtient une matrice triangulaire supérieure en n étapes: $L_n L_{n-1} \ldots L_1 A = R$. Dans le cas de m systèmes linéaires, la méthode conduit à l'algorithme suivant:

```
(* Triangularisation*)
Pour k <- 1 à n
     c <- 1/a_kk
     Pour j <- k+1 à n+m
          a_kj <- a_kj *c
     Pour i <- k+1 à n
          Pour j <- k+1 à n+m
               a_ij <- a_ij - a_ik* a_kj
(*Résolution*)
Pour  k <- n à 1
     Pour j <- n+1 à n+m
          Pour i <- k + 1 à n
               a_kj <- a_kj - a_kj* a_ij
```

b) JORDAN : Le principe de la méthode de Jordan séquentielle est le même que pour l'élimination de Gauss, à cette différence prés que l'on cherche à obtenir une matrice diagonale. Matriciellement, la méthode s'écrit: $J_n J_{n-1} \ldots J_1 A = D$ (matrice diagonale). On peut écrire l'algorithme correspondant, toujours dans le cas de m systèmes à résoudre :

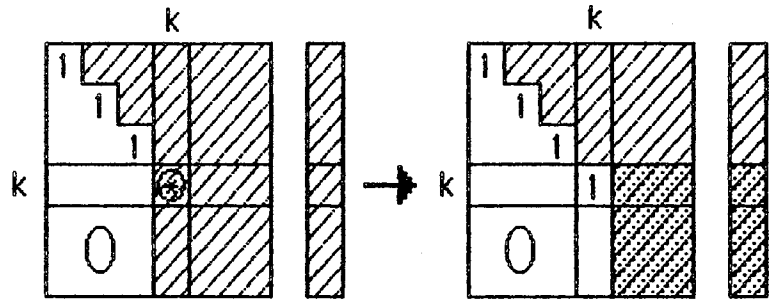
```
Pour k <- 1 à n
    (* préparation de la ligne k *)
    c <- 1/a_kk
    Pour j <- k+1 à n+m
        a_kj <- a_kj *c
    (* mise à zéro des éléments de la colonne k *)
    Pour i <- 1 à n , i ≠ k
        Pour j <- k+1 à n+m
            a_ij <- a_ij - a_ik* a_kj
```

c) LES PARAMETRES : Comme dans Jordan, la matrice est factorisée de manière à obtenir une matrice diagonale. L'organisation des calculs est différente : on élimine les éléments en ligne puis en colonne en gagnant à chaque étape un élément diagonal supplémentaire. De la même façon, on obtient une matrice diagonale D qui est la même que dans Jordan en n étapes : $P_n P_{n-1} \ldots P_1 A=D$. La méthode des paramètres n'est autre qu'une variante de la méthode de Gauss. L'algorithme séquentiel est le suivant :

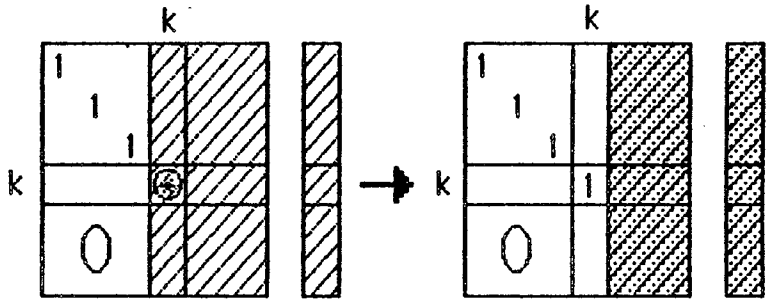
```
Pour k <- 1 à n
    (* élimination de la ligne k *)
    Pour i <- 1 à k-1
        Pour j <- k à n+m
            a_kj <- a_kj - a_ki* a_ij
    (* préparation de la ligne k *)
    c <- 1/a_kk
    Pour j <- k+1 à n+m
        a_kj <- a_kj *c
    (* mise à zéro de la colonne k jusqu'au terme diagonal *)
    Pour i <- 1 à k-1
        Pour j <- k+1 à n+m
            a_ij <- a_ij - a_ik* a_kj
```

d) RESUME : nous pouvons schématiser les passages des étapes k à k+1 pour les trois méthodes [14]:

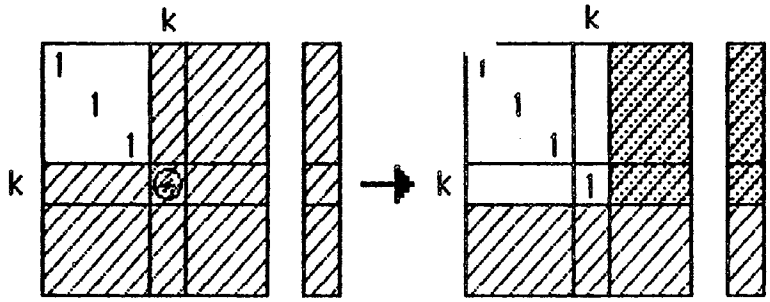
Gauss



Jordan



Les Paramètres



On en déduit le nombre d'opérations par méthode :

		div.	multiplications	additions
Gauss	n		$n^3/3 + mn^2 - n/3$	$n^3/3 + (m-1/2)n^2 - (m-1/6)n$
Jordan	n		$n^3/2 + (m-1/2)n^2$	$n^3/2 + (m-1)n^2 - (m-1/2)n$
Param.	n		$n^3/3 + mn^2 - n/3$	$n^3/3 + (m-1/2)n^2 - (m-1/6)n$

A la lecture du tableau précédent, on vérifie que les méthodes de Gauss et des Paramètres ont exactement le même coût en opérations élémentaires. Par contre, la méthode de Jordan est d'un coût supérieur en général. Le paragraphe suivant donne une interprétation matricielle de la méthode et permet d'expliquer sa similitude avec la méthode de Gauss.

e) INTERPRETATION MATRICIELLE : Nous avons vu que la méthode de Gauss, dans la première phase de triangularisation, revenait à calculer une suite de matrices triangulaires inférieures unicolonnes $L_1, L_2, \ldots, L_n$ telles que

$$L_n L_{n-1} \ldots L_1 \, A = R$$

De même, la méthode de Jordan conduit à

$$J_n J_{n-1} \ldots J_1 \, A = D \text{ (où D est la diagonale de R)}$$

et celle des paramètres s'écrit

$$P_n P_{n-1} \ldots P_1 \, A = D'.$$

Pour préciser les relations existant entre ces différentes méthodes, notons E_{ij} la matrice nxn dont tous les éléments sont nuls, sauf celui à l'intersection de la i-ème ligne et de la j-ème colonne qui vaut 1. Avec cette notation, on obtient

$$L_k = Id + \sum_{i=k+1}^{n} l_{ik} E_{ik}$$

Pour la méthode de Jordan, on a à l'étape k :

$$J_k = R_k L_k, \text{ avec } R_k = Id + \sum_{i=1}^{k-1} r_{ik} E_{ik}$$

Comme L_k et R_h commutent pour k>h, on regroupe les termes en R et en L :

$$R_n R_{n-1} \ldots R_1 \cdot L_n L_{n-1} \ldots L_1 = D$$

Pour la méthode des paramètres, on a à l'étape k :

$P_k = G_k H_k$, avec

- H_k matrice triangulaire inférieure uniligne :

$$H_k = Id + \sum_{i=1}^{k-1} h_{ki} E_{ik}$$

- G_k matrice triangulaire supérieure unicolonne :

$$G_k = Id + \sum_{i=1}^{k-1} g_{ik} E_{ik}$$

On obtient ainsi la relation

$$(*) \quad G_n H_n G_{n-1} H_{n-1} \ldots G_1 H_1 = D'$$

On vérifie facilement (par récurrence) que pour tout k, on a $G_k = R_k$, et que les pivots construits au cours de la factorisation sont les mêmes pour toutes les méthodes : donc, D = D'.

D'autre part, on a $H_n H_{n-1} \ldots H_1 \; A = R$: cette égalité correspond à l'algorithme de triangularisation de Gauss par lignes de la matrice A: à chaque étape k, on annule les k-1 premiers éléments de la ligne k. Les coefficients des matrices H sont reliés à ceux des matrices L par les relations h_{ki} (de la matrice H_k) = l_{ki} (de la matrice L_i).

Si après l'algorithme de triangularisation par lignes, on continue les calculs jusqu'à diagonalisation comme par la méthode de Jordan, on obtient la relation :

$$R_n R_{n-1} \ldots R_1 \cdot H_n H_{n-1} \ldots H_1 = D$$

c'est-à-dire

$$(**)\; G_n G_{n-1} \ldots G_1 \cdot H_n H_{n-1} \ldots H_1 = D$$

En comparant cette dernière égalité avec la relation (*), on vérifie bien que la méthode des paramètres consiste à intercaler une phase de diagonalisation entre chaque étape de triangularisation par lignes (et non par colonnes, comme dans l'algorithme usuel).

2. PARALLELISME

Lorsque l'on dispose de plus d'un processeur, une même méthode peut conduire à plusieurs implémentations différentes. Le nombre de processeurs n'est d'ailleurs pas le seul paramètre. Les contraintes imposées par la stucture de la machine (voir les diverses classifications proposées Flynn [9], Gajski [12], Treleaven [28], le modèle systolique, Bojanczyk et al. [2], Robert et Tchuente [23]) peuvent influencer notablement les performances d'une méthod De plus, le mode d'accès aux données prend une importance considérable et le rapport entre le temps de transmission et le temps de traitement devient une variable primordiale. Ces temps sont étroitement liés à la conception et à la fabrication de la machine parallèle sur laquelle est exécuté l'algorithme. Par conséquent nous n'aborderons pas ce problème.

Par contre, nous supposerons par la suite, que nous pouvons accéder aux éléments de la matrice A, soit par lignes, soit par colonnes. Les opérations de transfert possibles seront alors :

- la transmission d'une ligne ou d'une colonne de A de l'organe de stockage vers un organe de traitement.
- la transmission d'une ligne ou d'une colonne de A d'un organe de traitement vers l'organe de stockage.

La duplication d'une ligne ou d'une colonne aura un coût nul, c'est à dire que nous supposons qu'il est possible de transférer simultanément une même donnée vers plusieurs processeurs. Dans ce cas, aucun processeur ne pourra modifier cette donnée. Inversement, un processeur ne pourra modifier une donnée que s'il est le seul à en posséder un exemplaire.

Nous n'aborderons pas les mécanismes informatiques qui permettent de résoudre ce problème d'allocation des données sans conflit. Remarquons simplement que les contraintes précédentes et le mécanisme qui les gére permettent d'assurer un déroulement sans ambiguité de l'algorithme et une certaine synchronisation.

Ces contraintes étant définies, la première étape dans la parallélisation d'une méthode consiste en la définition des tâches élémentaires qu'exécuteront les processeurs et le graphe d'ordonnancement des tâches. Dans un tel graphe un arc relie la tâche T_1 à la tâche T_2 si T_1 modifie des données qu'utilise T_2. Ce graphe traduit donc la dépendance temporelle des opérations effectuées par l'algorithme: dans

l'exemple, T_2 ne pourra être exécutée qu'après l'exécution de T_1. Ceci garantit l'absence de conflit, l'intégrité temporelle des données et une parfaite adéquation entre la méthode et sa version parallèle.

Les tâches ainsi définies sont affectées aux processeurs disponibles en respectant le graphe d'ordonnancement des tâches et leur durée d'exécution. Ceci permet de déterminer le temps d'exécution de l'agorithme et d'obtenir une idée du gain apporté par l'utilisation de plusieurs processeurs. Une mesure plus fine du degré de parallélisme introduit par l'algorithme est le taux moyen d'utilisation des processeurs. l'efficacité est définie par :

$$e_p = T_1 / p\, T_p$$

où p est le nombre de processeurs, T_1 le temps d'exécution de l'algorithme sur un ordinateur séquentiel et T_p le temps d'exécution sur un ordinateur avec p processeurs.

Le problème de l'affectation des tâches aux processeurs est délicat et conduit à des études d'optimalité et de complexité. En ce qui concerne la décomposition QR, ce problème est étudié dans le cas de $\lfloor n/2 \rfloor$ processeurs ($\lfloor\ \rfloor$ est la fonction partie entière) par Cosnard et Robert [3] pour des matrices carrées et Cosnard, Muller et Robert [5] pour le cas rectangulaire ; voir aussi à ce sujet les travaux de Modi et Clarke [21] et de Sameh et Kuck [24]. Lord, Kowalik et Kumar [19] étudie la parallélisation de la méthode de Gauss en supposant un nombre suffisant de processeurs. Dans le cas d'un nombre fixé de processeurs, Srinivas [27] étudie la complexité de la parallélisation de la méthode de Gauss.

Dans la suite de ce travail, nous étudions la parallélisation des méthodes de diagonalisation de Jordan et des paramètres. Nous supposerons que le temps d'exécution est le même pour chaque opération élémentaire : ce temps sera notre unité.

3. UN RESULTAT D'OPTIMALITE

Nous démontrons dans ce paragraphe un résultat d'optimalité qui nous sera utile pour la parallélisation de plusieurs algorithmes dans la suite.

Nous considérons la résolution en parallèle, avec p processeurs, d'un ensemble de tâches T_{kj}, $1 \leq k \leq n$, $k+1 \leq j \leq n+m$. Nous supposons que les contraintes du graphe d'ordonnancement sont les suivantes:

(A) la tâche $T_{k,k+1}$ doit être exécutée avant les tâche $T_{k+1,j}$ $(j \geq k+2)$

(B) pour j fixé > k, la tâche $T_{k,j}$ doit être exécutée avant la tâche $T_{k+1,j}$

Pour une valeur donnée de k, il y a donc n+m-k tâches à effectuer. Nous supposons de plus que toutes les tâches T_{kj} $(j>k)$ d'un même niveau k ont le même temps d'exécution t_k.

Le graphe d'ordonnancement des tâches a l'allure suivante:

T_{12}	T_{13}	T_{14} ...	$T_{1,n+1}$	... $T_{1,n+m}$	niveau 1
	T_{23}	T_{24} ...	$T_{2,n+1}$	... $T_{2,n+m}$	niveau 2
		T_{34} ...	$T_{3,n+1}$	... $T_{3,n+m}$	niveau 3
		...	...		
			...		
			$T_{n,n+1}$	... $T_{n,n+m}$	niveau n

Proposition 1 :

Parmi les algorithmes optimaux, il en existe qui satisfont de plus à la contrainte d'ordonnancement (C):

(C) la tâche $T_{k,j}$ doit être exécutée avant les tâches $T_{k,j+1}$ $(j \geq k+2)$

Démonstration: Soit M un algorithme quelconque. Nous construisons à partir de M un algorithme M* de même temps d'exécution que M et qui respecte la contrainte supplémentaire (C)

- si à un instant t, l'algorithme M commence l'exécution de q tâches niveau k, M* fait de même, mais exécute les q premières tâches (selon

l'ordre imposé par la contrainte (C)) non encore commencées du niveau k
- si à un instant t, l'algorithme M ne commence l'exécution d'aucune tâche niveau k, M* non plus.

Ainsi, à tout instant au cours de l'exécution, M* aura exécuté le même nombre de tâches par niveau que l'algorithme M. Nous devons cependant démontrer que les contraintes d'ordonnancement (A) et (B) sont satisfaites pour M*. Pour la contrainte (A), c'est évident puisque $T_{k,k+1}$ est la première tâche exécutée par M* niveau k. Pour la contrainte (B), nous montrons que le nombre de tâches exécutées par l'algorithme M niveau k à l'instant t est supérieur ou égal au nombre de tâches exécutées par M niveau k+1 à l'instant $t + T_k$. Ce résultat assure que M* tel que nous l'avons construit respecte bien la contrainte (B).

Nous procédons par récurrence sur le temps t. Pour t=1, la seule tâche qui puisse être commencée par M est T_{21} en vertu de (A), d'où le résultat. Supposons le résultat vrai à l'instant t; pour toute tâche T_{kj} du niveau k, on a

$$fin(T_{kj}) > t+1 \Rightarrow fin(T_{k+1,j}) > t + 1 + t_k,$$

où fin(T) désigne l'instant où l'exécution de la tâche T est terminée. Par suite, card $\{j / fin(T_{kj}) \leq t+1\} >$ card $\{j / fin(T_{k+1,j}) \leq t+T_k+1\}$.

Nous introduisons maintenant l'algorithme glouton G, défini informellement de la manière suivante: l'algorithme G exécute les tâches en balayant les lignes du graphe d'ordonnancement de gauche à droite, en commençant à tout instant le maximum de tâches (d'où le nom de glouton). Plus précisément, l'algorithme G exécute les tâches dans l'ordre suivant:
$T_{12} \lessdot\lessdot T_{13} \lessdot\lessdot \dots \lessdot\lessdot T_{1,n+m} \lessdot\lessdot T_{23} \lessdot\lessdot T_{24} \lessdot\lessdot T_{2,n+m} \lessdot\lessdot T_{3,4} \lessdot\lessdot \dots \lessdot\lessdot T_{n,n+m}$
où la notation $T \lessdot\lessdot T'$ signifie que l'exécution de la tache T ne peut débuter après l'exécution de la tache T' (mais la simultanéité est possible).

A l'instant t=1, l'algorithme G commence l'exécution des tâches T_{12}, T_{13}, ..., T_{1p}. A tout instant $t \geq 1$, si $q \leq p$ processeurs sont disponibles, l'algorithme G les affecte à l'exécution des q tâches suivantes, du moins tant que la contrainte (B) le permet (sinon, on effectue le nombre maximum de tâches exécutables).

Plutôt que d'expliciter formellement l'algorithme glouton dans le cas général, nous allons détailler son exécution sur un exemple. Choisissant m=1, n=5 et p=3, nous avons le graphe d'ordonnancement suivant:

T_{12} T_{13} T_{14} T_{15} T_{16}
T_{23} T_{24} T_{25} T_{26}
T_{34} T_{35} T_{36}
T_{45} T_{46}
T_{56}

Considérons deux cas (modélisant chacun l'une des applications que nous rencontrons dans la suite):

- (I) $t_i = (n+m-i).\tau$, soit ici $t_i = (6-i)\tau$ pour $1 \leq i \leq 5$

- (II) toutes les tâches ont le même temps d'exécution T, c.a.d.
$$t_1 = t_2 = \ldots = t_n = T$$

Nous obtenons le séquencement suivant, où les temps indiqués sont relatifs au début de l'exécution des tâches:

<u>cas (I)</u>

temps	**processeur 1**	**processeur 2**	**processeur 3**
t=0	T_{12}	T_{13}	T_{14}
t=5τ	T_{15}	T_{16}	T_{23}
t=9τ			T_{24}
t=10τ	T_{25}	T_{26}	
t=13τ			T_{34}
t=14τ	T_{35}	T_{36}	
t=16τ			T_{45}
t=17τ	T_{46}		
t=18τ		T_{56}	

cas (II)

temps	processeur 1	processeur 2	processeur 3
t=0	T_{12}	T_{13}	T_{14}
t=T	T_{15}	T_{16}	T_{23}
t=2T	T_{24}	T_{25}	T_{26}
t=3T	T_{34}	T_{35}	T_{36}
t=4T	T_{45}	T_{46}	
t=5T	T_{56}		

Proposition 2 :

L'algorithme glouton G est optimal. Son temps d'exécution est égal à

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

Démonstration : En appliquant la proposition 1, nous montrons que l'algorithme glouton est optimal dans la classe des algorithmes qui vérifient les contraintes (A), (B) et (C). Sachant maintenant qu'une tâche T_{kj} doit être exécutée avant la tache $T_{k,j+1}$, on montre par récurrence que l'algorithme glouton débute le plus tôt possible l'exécution des tâches $T_{1,n+m}$, $T_{2,n+m}$, $T_{n,n+m}$.

Pour le temps d'exécution, nous notons que jusqu'à l'exécution de la tâche $T_{n+m-p,n+m}$ tous les p processeurs sont actifs sans interruption: en effet il y a plus de p tâches par niveau jusqu'à la ligne n+m-p. Le temps d'exécution est donc égal au temps d'exécution séquentielle de toutes les tâches précédentes, divisé par le nombre de processeurs p (en d'autres termes G est d'efficacité 1 jusqu'au niveau n+m-p). On rajoute à cette première valeur:

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

le temps d'exécution des tâches $T_{n+m-p+1,n+m}$, $T_{n+m-p+2,n+m}$, ..., $T_{n,n+m}$, ce qui conduit au résultat annoncé.

4. JORDAN PARALLELE

4.1. Avec $O(n^2)$ processeurs

Supposons pour commencer que chaque processeur peut accéder à chaque élément de la matrice et que nous disposons d'un grand nombre de processeurs. Une manière efficace de programmer la méthode de Jordan est la suivante :

```
Pour k <- 1 à n
    Pour j <- k + 1 à n + m
        Pour i <- 1 à n     i≠k
            a_ij <- a_ij - a_ik a_kj / a_kk
```

D'un point de vue séquentiel, cet algorithme est désastreux, puisqu'il répète plusieurs fois les mêmes opérations. Par contre si l'on dispose de $(n - 1)(n + m - 1)$ processeurs, il est possible, pour chaque valeur de k d'effectuer en parallèle la modification de tous les a_{ij}. Posons T_{ijk} la tâche suivante :

$$T_{ijk} : a_{ij} \leftarrow a_{ij} - a_{ik}\, a_{kj} / a_{kk}$$

Le graphe d'ordonnancement est le suivant

$T_{i,j,k-1}$ $T_{i,k,k-1}$ $T_{k,j,k-1}$ $T_{k,k,k-1}$

$T_{i,j,k}$

Une manière simple d'affecter les tâches aux processeurs est de faire exécuter simultanément toutes les tâches ayant même valeur de k. Chaque tâche s'exécutant en 3 unités, le temps d'exécution de l'algorithme est 3 n. Soit, pour $p = (n-1)(n+m-1)$

$$t_p = 3n$$

$$e_p = t_1/pt_p \sim (n^3 + 2mn^2)/(3n^2(n+m)) \sim (n+2m)/(3(n+m))$$

si m = 1, $e_p \sim 1/3$

si m = n, $e_p \sim 2/3$

En réalité, il est possible d'augmenter l'efficacité, en diminuant le nombre minimum de processeurs permettant de résoudre le problème en 3n unités de temps.

Ce problème a cependant un intérêt pratique assez faible puisqu'il n'existe pas de machines possédant un si grand nombre de processeurs. De plus le flot d'informations, circulant entre les processeurs et la mémoire, est énorme ce qui rend cette étude peu réaliste.

4.2. Jordan avec p processeurs

Supposons maintenant que le nombre p de processeurs soit compris entre 1 et O(n). Suivant l'accés aux données, on peut paralléliser la méthode de Jordan soit sous la forme présentée précédemment que nous appelerons Jordan par lignes, soit en inversant, dans celle-ci, les boucles i et j imbriquées. On obtient alors un Jordan par colonnes.

JORDAN par lignes:

```
Pour k <- 1 à n
    Pour j <- k+1 à n+m
        a_kj <- a_kj / a_kk
    Pour i <- 1 à n , i ≠ k
        Pour j <- k+1 à n+m
            a_ij <- a_ij - a_ik * a_kj
```

JORDAN par colonnes:

```
Pour k <- 1 à n
    Pour j <- k+1 à n+m
        a_kj <- a_kj / a_kk
    Pour i <- 1 à n , i ≠ k
        a_ij <- a_ij - a_ik * a_kj
```

4.2.1 Par lignes

Les tâches sont donc définies de la manière suivante :

JL_{kk} : Pour j <- k+1 à n+m

$a_{kj} \leftarrow a_{kj} / a_{kk}$

JL_{ki} : Pour j <- k+1 à n+m $i = 1, \ldots, n$

$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$ $i \neq k$

La tâche JL_{kk} s'exécute en n+m-k unités de temps alors que JL_{ki}, $i \neq k$, nécessite 2(n+m-k) unités.

Il est clair qu'il est sans intérêt de supposer p supérieur à n dans cette méthode puisqu'il n'y a que n-1 tâches JL_{ki}. Un algorithme est obtenu en affectant les tâches aux processeurs. Le temps d'exécution JL_p de la méthode est le minimum des temps d'exécution de tous les algorithmes. C'est aussi le temps d'exécution d'un algorithme optimal. Nous noterons EJL_p l'efficacité asymptotique de la méthode.

Les contraintes du graphe d'ordonnancement sont les suivantes:

- la tâche JL_{kk} doit être exécutée avant les tâches JL_{ki}
- la tâche JL_{ki} doit être exécutée avant la tâche $JL_{k+1,i}$
- pour i fixé > k, la tâche JL_{ki} doit être exécutée avant la tâche JL_{ii}.

ce qui conduit au graphe suivant:

JL_{11})

JL_{12} JL_{13} ... JL_{1n}) niveau 1

...

JL_{kk})

JL_{kk+1} JL_{kk+2} ... JL_{kn} JL_{k1} ... JL_{kk-1}) niveau k

JL_{k+1k+1}

...

Proposition 3:

i) Si $p=n-1$, $JL_p=3n(m+(n-1)/2)$ et $EJL_p=2/3$.

ii) Si $n/2 \leq p=\alpha n \leq n-1$, $JL_p \leq n+3m + (n-1)(m+n/2)(2+(n-2)/(p-1))$

et $EJL_p \geq 2/(1+2\alpha)$

iii) Si $1 \leq p \leq n/2$, $JL_p=n^2(2m+n)/p+O(n^2/p)$ et $EJL_p=1$.

Démonstration:

i) Le graphe d'ordonnancement des tâches implique qu'il ne sert à rien d'utiliser plus de n-1 processeurs. En effet la tâche JL_{kk} doit être exécutée avant les n-1 tâches JL_{ki}. Ceci implique en particulier que le plus long chemin du graphe est le suivant:

$$JL_{11} \mapsto JL_{12} \mapsto JL_{22} \mapsto JL_{23} \ldots JL_{k-1k-1} \mapsto JL_{k-1k} \mapsto \ldots JL_{nn} \mapsto JL_{n1}$$

Si $p=n-1$, on en déduit que

$$JL_p \geq 3 \sum_{k=1}^{n} (n+m-k) = 3(mn+n(n-1)/2)$$

En affectant à un processeur les tâches du plus long chemin et aux n-2 processeurs restants les n-2 tâches JL_{ki}, $i \neq k,k+1$, on obtient un algorithme dont le temps d'exécution est la borne inférieure précédente.

ii) Si $n/2 \leq p=\alpha n < n-1$, il n'est plus possible d'exécuter simultanément toutes les tâches JL_{ki}, $i \neq k$. Le nombre de tâches qui ne peuvent être exécutées en même temps que les autres est $n(n-1-p)$. Comme le temps d'exécution des tâches JL_{ki} est double de celui de JL_{kk}, un algorithme optimal est asynchrone (les processeurs débutent leurs tâches à des temps différents). Nous allons présenter un algorithme synchrone, non optimal (mais que nous pensons être asymptotiquement optimal), qui nous permettra d'obtenir les bornes de la proposition.

Le principe de cet algorithme repose sur les deux considérations suivantes:

- il faut exécuter les tâches JL_{kk} le plus tôt possible pour que les contraintes du graphe d'ordonnancement ne retardent pas l'exécution des tâches JL_{k+1i}

- il ne faut pas effectuer immédiatement les n-p-1 tâches non exécutées d'un même niveau, mais rassembler des tâches non exécutées jusqu'à ce que leur nombre dépasse p-1 et les affecter aux p-1 processeurs restants pendant l'exécution de $JL_{k'k'}$.

Pour décrire précisément cet algorithme, posons N=n(n-1) et r=n-p-1. Remarquons que $1 \leq r < n/2-1$. Définissons la suite (e_k) par:

$$e_k = \lfloor kr/(p-1) \rfloor - \sum_{j=1}^{k-1} e_j$$

e_k prend les valeurs 0 ou 1 et vaut 0 si le nombre de tâches non exécutées au niveau inférieur ou égal à k est inférieur à p-1 et 1 sinon. Renumérotons les tâches JL_{ki} de la manière suivante:

$$T_{(k-1)(n-1)+j} = JL_{k((k+j-1) \text{ MOD } n)+1} \quad \text{avec } 1 \leq j \leq n-1.$$

L'algorithme est le suivant:

```
(* initialisation *)
    compt <- 1 ;
    P1 effectue la tâche JL11 ;
    P2, P3, ..., Pn sont inactifs ;
(* on traite les k niveaux *)
    pour k <- 1 jusqu'à n-1 faire
        P1, ..., Pp effectuent les tâches Tcompt, ..., Tcompt+p-1 ;
        compt <- compt+p ;
(* le retard est inférieur à p-1 *)
        si ek vaut 0 alors
            P1 effectue la tâche JLk+1,k+1 ;
            P2, P3, ..., Pn sont inactifs ;
        fin si ;
(* le retard est supérieur à p-1 *)
        si ek vaut 1 alors
            P1 effectue la tâche JLk+1,k+1 ;
            P2, ..., Pp effectuent les tâches Tcompt, ..., Tcompt+p-2 ;
            compt <- compt+p-1 ;
        fin si ;
    fin pour ;
```

(* exécution des tâches restantes *)

tant que compt ≤ N **faire**

P1, ..., Pp effectuent les tâches T_{compt}, ..., $T_{compt+p-1}$;

compt <- compt+p ;

fin tant que ;

fin .

Pour illustrer cet algorithme, nous présentons dans le tableau suivant le cas de la diagonalisation d'une matrice de taille 9 avec 5 processeurs. ij dans la colonne Ps signifie que le processeur Ps effectue la tâche JL_{ij}.

n=9 p=5 r=3

k	1	2	3	4	5	6	7	8
e_k	0	1	1	1	0	1	1	1

	P1	P2	P3	P4	P5
	11				
k=1	12	13	14	15	16
e_1=0	22				
k=2	17	18	19	23	24
e_2=1	33	25	26	27	28
k=3	29	21	34	35	36
e_3=1	44	37	38	39	31
k=4	32	45	46	47	48
e_4=1	55	49	41	42	43
k=5	56	57	58	59	51
e_5=0	66				
k=6	52	53	54	67	68
e_6=1	77	69	61	62	63
k=7	64	65	78	79	71
e_7=1	88	72	73	74	75
k=8	76	89	81	82	83
e_8=1	99	84	85	86	87
k=9	91	92	93	94	95
	96	97	98		

Nous vérifions maintenant que cet algorithme respecte les contraintes du graphe d'ordonnancement. Pour cela, il suffit de montrer que:

- aucune tâche JL_{ki} n'est effectuée avant la tâche JL_{kk}, ce qui est équivalent à

$$\text{nb } JL_{ki} \text{ algo} = (k-1)p + (p-1)\sum_{j=1}^{k-2} e_j \leq (k-1)(n-1) = \text{nb max } JL_{ki} \text{ possibles}$$

- $JL_{k-1,k}$ est effectuée avant JL_{kk}, ce qui est équivalent à

$$\text{rang } JL_{k-1,k} = (k-2)(n-1)+1 \leq (k-1)p + (p-1)\sum_{j=1}^{k-2} e_j$$

Pour démontrer la première inégalité, on remarque que:

$$\sum_{j=1}^{k-2} e_j = \lfloor (k-1)r/(p-1) \rfloor - e_{k-1} \leq (k-1)r/(p-1)$$

et donc

$$(k-1)p + (p-1)\sum_{j=1}^{k-2} e_j \leq (k-1)p + (k-1)r \leq (k-1)(n-1).$$

La deuxième inégalité est équivalente à:

$$(p-1)\sum_{j=1}^{k-2} e_j \geq (k-1)r - (p-1) - r$$

c'est à dire:

$$\sum_{j=1}^{k-2} e_j \geq (k-2)r/(p-1) - 1$$

ce qui se déduit de:

$$\sum_{j=1}^{k-2} e_j = \lfloor (k-2)r/(p-1) \rfloor$$

Pour évaluer le temps d'exécution A de cet algorithme, nous devons déterminer, pour chaque étape, le temps maximal des tâches. La démonstration précédente montre qu'à l'étape k, l'algorithme termine le

niveau k-1 des tâches JL_{ki}, débute le niveau k des tâches JL_{ki} et exécute $JL_{k+1,k+1}$. On obtient alors:

$$A \leq (n+m-1) + \sum_{k=1,\, e_k=0}^{n-1} [(n+m-k-1) + 2(n+m-k)] + \sum_{k=1,\, e_k=1}^{n-1} [2(n+m-k) + 2(n+m-k)] + 2m$$

$$\leq (n+3m) + \sum_{k=1}^{n-1} 3(n+m-k) + \sum_{k=1}^{n-1} e_k(n+m-k)$$

Commençons par évaluer la partie de la somme contenant e_k:

$$\sum_{k=1}^{n-1} k e_k = \sum_{j=1}^{n-1} \sum_{k=j}^{n-1} e_k = \sum_{j=1}^{n-1} \left(\sum_{k=1}^{n-1} e_k - \sum_{k=1}^{j-1} e_k \right)$$

$$= (n-1) \sum_{k=1}^{n-1} e_k - \sum_{j=1}^{n-1} \sum_{k=1}^{j-1} e_k$$

et donc

$$\sum_{k=1}^{n-1} e_k (n+m-k) = (m+1) \sum_{k=1}^{n-1} e_k + \sum_{j=1}^{n-1} \sum_{k=1}^{j-1} e_k$$

$$= (m+1) \lfloor (n-1)r/(p-1) \rfloor + \sum_{j=1}^{n-1} \lfloor (j-1)r/(p-1) \rfloor$$

$$\leq (n-1)(m+n/2)\, r/(p-1)$$

Revenons maintenant à l'évaluation de A:

$$A \leq n+3m + 3(n-1)(m+n/2) + (n-1)(m+n/2)\, r/(p-1)$$
$$\leq n+3m + (n-1)(m+n/2)(2+(n-2)/(p-1))$$

EJL_p s'en déduit immédiatement.

iii) Si l'on dispose d'un nombre de processeurs p compris entre 1 et n/2, la difficulté consiste à répartir les processeurs de manière à ce qu'ils aient un taux d'occupation optimal. Nous proposons un algorithme pour effectuer Jordan par lignes avec p processeurs P1, P2, ... , Pp. On note k le nombre de niveaux traités (un niveau correspondant à l'élimination d'une colonne par Jordan) et l le nombre de tâches exécutées au commencement de l'étape k.

(* initialisation *)
Le processeur P1 effectue la tâche JL_{11} ;
pour k <- 1 **jusqu'à** n **faire**
    (* on traite le k ième niveau *)
    si l=0 **alors**
        les p processeurs effectuent $JL_{k+1,k} \cdots JL_{k+p,k}$ en parallèle
    fin si ;
    P1 effectue la tâche $JL_{k+1,k+1}$ en parallèle avec les p-1 autres qui effectuent $JL_{k+1+1,k} \cdots JL_{k+1+p-1 \text{ MOD } n,k}$;
    tant que (la ligne k n'est pas achevée) **faire**
    effectuer en parallèle p tâches dans l'ordre du graphe ;
    l <- nombre de tâches commencées au niveau k+1 ;
fin.

On vérifie facilement que l'algorithme est réalisable au sens des contraintes du graphe d'ordonnancement. En effet, la relation

$$p < n/2$$

nous assure que la tâche $JL_{k+1,k+1}$ est effectuée après $JL_{k+1,k}$ et avant $JL_{j,k+1}$ pour $j \neq k+1$. On peut facilement évaluer le temps d'exécution de l'algorithme précédent :

$$JL_p \leq n+m-1 + \lceil n/p \rceil \sum_{i=1}^{n} 2(n+m-i)$$

$$\leq n^2(2m+n)/p + O(n^2)$$

et donc, $EJL_p \geq 1$ ce qui montre que l'algorithme précédent est asymptotiquement optimal et que $JL_p = n^2(2m+n)/p + O(n^2)$ et $EJL_p = 1$.

4.2.2. Par colonnes

Les tâches sont maintenant définies de la manière suivante:

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

Pour i <- 1 à n , i ≠ k $\quad k+1 \leq j \leq m+n$

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

La tâche JC_{kj} s'exécute en 2n-1 unités de temps. Les contraintes du graphe d'ordonnancement sont les suivantes:

(A) la tâche $JC_{k,k+1}$ doit être exécutée avant les tâches $JC_{k+1,j}$ $(j \geq k+2)$
(B) pour j fixé > k, la tâche $JC_{k,j}$ doit être exécutée avant la tâche $JC_{k+1,j}$

Ici, à un niveau donné k, il y a n+m-k tâches à effectuer. Par contre, toutes les tâches JC_{kj} ont le même temps d'exécution. Le graphe d'ordonnancement est le suivant:

JC_{12} JC_{13} JC_{14} ... $JC_{1,n+1}$ ... $JC_{1,n+m}$

JC_{23} JC_{24} ... $JC_{2,n+1}$ ... $JC_{2,n+m}$

JC_{34} ... $JC_{3,n+1}$ ... $JC_{3,n+m}$

$JC_{n,n+1}$ ... $JC_{n,n+m}$

Ce graphe vérifie les hypothèses de la section 3 ce qui conduit au résultat suivant:

Proposition 4 :

i) Parmi les algorithmes optimaux, il en existe qui satisfont de plus à la contrainte d'ordonnancement (C):

- la tâche $JC_{k,j}$ doit être exécutée avant les tâches $JC_{k,j+1}$ $(j \geq k+2)$ (C)

ii) L'algorithme glouton est optimal.

iii) $JC_p = (2n-1)\,((n+m-p)(n+m+p-1)/2p + p-m)$

Pour m=1 et $p = \alpha.n$, $JC_p = (1+\alpha^2)n^2/\alpha + O(n)$ et $EJC_p = 1/(1+\alpha^2)$

Démonstration:

Toutes les tâches ont le même temps d'exécution: 2n-1 unités de temps. La formule donnée en 3 devient donc ici:

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

d'où le résultat annoncé:

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

Posons m=1 et $p = \alpha.n$, on obtient

$$JC_p = ((1+\alpha^2)/\alpha)\, n^2 + O(n).$$

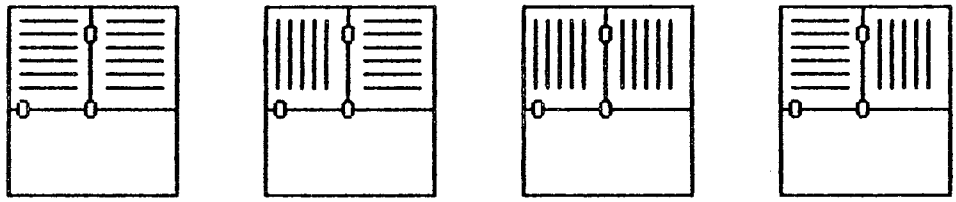
La valeur de EJC_p s'en déduit aisément, sachant que $JC_1 = n^3 + O(n^2)$.

5. LES PARAMETRES EN PARALLELE

La méthode des paramètres comporte deux étapes par itération:

- annulation des k-1 premiers éléments de la ligne k et modifications correspondantes des n+m-k+1 éléments suivants: nous la nommerons étape de descente.
- annulation des k-1 premiers éléments de la colonne k et modifications correspondantes des n+m-k derniers éléments des k-1 premières lignes: étape de remontée.

Suivant l'accés aux éléments de la matrice, on peut écrire chacune des deux étapes de deux manières différentes: par lignes ou par colonnes. On obtient donc quatre versions différentes schématisées dans la figure et présentées dans les programmes qui suivent.



ligne/ligne colonne/ligne colonne/colonne ligne/colonne

```
(* descente en lignes *)
Pour i <- 1 à k-1
    Pour j <- k à n+m
        a_kj <- a_kj - a_ki * a_ij
```

```
(* descente en colonnes *)
Pour j <- k à n+m
    Pour i <- 1 à k-1
        a_kj <- a_kj - a_ki * a_ij
```

```
(* remontée en lignes *)
Pour i <- 1 à k-1
    Pour j <- k+1 à n+m
        a_ij <- a_ij - a_ik * a_kj
```

```
(* remontée en colonnes *)
Pour j <- k+1 à n+m
    Pour i <- 1 à k-1
        a_ij <- a_ij - a_ik * a_kj
```

La descente en lignes n'est pas parallélisable, puisque, pour chaque valeur de i, les n+m-k+1 éléments a_{kj} sont modifiés. En effet, ceci contredit notre hypothèse d'interdire des modifications simultanées d'un même élément. Par conséquent, deux versions sont possibles: descente colonnes - remontée lignes et descente colonnes - remontée colonnes.

Comme dans la cas de Jordan, nous allons examiner la parallélisation en fonction du nombre de processeurs.

5.1 Avec $O(n^2)$ processeurs

Lorsqu'on dispose d'un nombre illimité de processeurs (n(n+m) suffisent dans ce cas), l'étape de remontée nécessite 2n unités de temps qu'elle soit effectuée par lignes ou par colonnes. Par contre, l'étape de descente conduit à l'exécution par chaque processeur de n-1 produits scalaires de vecteurs de longueur variant de 1 à n-1. La complexité d'un produit scalaire de taille k est log k. Le temps de descente est donc de:

$$D = \sum_{k=1}^{n-1} \log k = \log (n-1)! = O(n \log n)$$

Par conséquent, le temps total est de O(n log n).

On constatera que l'efficacité de cette parallèlisation (qui est en O(1/log n)) est mauvaise, comparée à celle obtenue dans le cas de la méthode de Jordan.

5.2 Avec O(n) processeurs

Contrairement au cas précédent, lorsqu'on ne dispose que d'un nombre limité de processeurs, les deux versions parallèles possibles: descente colonnes - remontée lignes et descente colonnes - remontée colonnes, n'ont pas le même temps d'exécution. Nous allons étudier successivement chacune d'elles. Il est clair que du strict point de vue du problème d'accessibilité aux données, la version colonnes - colonnes est préférable puisque l'accés aux données ne s'effectue que d'une seule façon: par colonnes.

5.2.1 Les paramètres colonnes - lignes

La version descente colonnes - remontée lignes de la méthode des paramètres est la suivante:

```
(* descente colonnes - remontée lignes *)
(* descente en colonnes *)
Pour j <- k à n+m
    Pour i <- 1 à k-1
        a_kj <- a_kj - a_ki * a_ij
(* préparation ligne k *)
Pour j <- k+1 à n+m
    a_kj <- a_kj / a_kk
```

(* remontée en lignes *)
Pour i <- 1 à k-1
    Pour j <- k+1 à n+m
        $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

Trois types de tâches sont donc à considérer :

- D_{kj}: Pour i <- 1 à k-1
        $a_{kj} \leftarrow a_{kj} - a_{ki} * a_{ij}$

- C_{kj}: $a_{kj} \leftarrow a_{kj}/a_{kk}$

- R_{ik}: Pour j <- k+1 à n+m
        $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

Il existe n+m-k+1 tâches D_{kj} dont le temps d'exécution est de 2(k-1) unités, n+m-k C_{kj} de temps 1 et k-1 R_{ik} de temps n+m-k. Les contraintes d'ordonnancement sont les suivantes:
- D_{kk} et D_{kj} précédent C_{kj}
- pour tout j et pour tout i, C_{kj} précédent R_{ik}
- pour tout i et pour tout j, R_{ik} précédent $D_{k+1,j}$.

On obtient alors le graphe d'ordonnancement:

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

Appelons PCL_p le temps d'exécution d'un algorithme optimal effectuant la méthode précédente avec p processeurs et $EPCL_p$ l'efficacité asymptotique.

Proposition 5:

Pour $p=\alpha n$ et $m=1$, $PCL_p \leq n^2(1 + 4\alpha^2 - 2\alpha^3 + 2/(3\alpha))$

$$EPCL_p \geq 1/(1 + 3\alpha/2 + 6\alpha^2 - 3\alpha^3)$$

Démonstration:

Des contraintes d'ordonnancement, nous déduisons que, pour débuter l'exécution des tâches R_{ik}, il est nécessaire d'avoir terminé l'exécution des tâches C_{kj} et D_{kj} qui ont respectivement la même durée. Par conséquent, il existe un algorithme synchrone optimal. Pour expliciter un tel algorithme, nous remarquons qu'il est possible de rassembler les tâches C_{kj} et D_{kj} en un seule tâche B_{kj} de durée 2k-1. Le graphe devient alors:

B_{22} B_{23} B_{24} ... $B_{2,n}$

R_{12}

... ...

B_{kk} B_{kk+1} ... $B_{k,n+m}$

R_{1k} R_{2k} ... $R_{k-1,k}$

... ...

Les tâches d'une même ligne doivent être entièrement effectuées avant de débuter l'exécution des tâches de la ligne suivante. Il est clair que l'algorithme glouton par lignes appliqué au graphe précédent est optimal. Son temps d'exécution est donné par la formule:

$$PLC_p = \sum_{k=2}^{n} \lceil (n+m-k)/p \rceil \, (2k-1) + \sum_{k=2}^{n} \lceil (k-1)/p \rceil \, 2(n+m-k)$$

qui se transforme en:

$$PLC_p = \sum_{k=m}^{n+m-2} \lceil k/p \rceil \, (2(n+m-k)-1) + \sum_{k=2}^{n} \lceil (k-1)/p \rceil \, 2(n+m-k)$$

Posons $m=pq-r$. Nous obtenons:

$$PLC_p = \sum_{k=m}^{qp} q(2(n+m-k)-1) + \sum_{j=q}^{\lfloor (n+m-2)/p \rfloor - 1} \; \sum_{k=jp+1}^{(j+1)p} (j+1)(2(n+m-k)-1$$

$$+ \sum_{k=\lfloor (n+m-2)/p \rfloor}^{n+m-2} \lceil (n+m-2)/p \rceil (2(n+m-k)-1)$$

Cette sommation est difficile. Nous ne continuerons pas plus avant ce calcul dans le cas général. Nous allons donc supposer que m=1. En reprenant ce qui précède, nous obtenons:

$$PLC_p = 4 \sum_{k=1}^{p} (n-k) + 4 \sum_{j=2}^{q-1} \sum_{k=(j-1)p+1}^{jp} j(n-k) + 4 \sum_{k=(q-1)p+1}^{qp-r} q(n-k)$$

$$\geq 4p\left(n - p/2 + (n+p/2)q^2/2 - pq^3/3\right)$$

Posons p=αn. Nous obtenons:

$$PLC_p \geq 4\alpha n\left(n - \alpha n/2 + (n+\alpha n/2)/(2\alpha^2) - 1/(3\alpha^3)\right)$$

$$\geq n^2\left(1 + 4\alpha - 2\alpha^2 + 2/(3\alpha)\right)$$

et donc:

$$EPCL_p \geq 1/(1 + 3\alpha/2 + 6\alpha^2 - 3\alpha^3)$$

La proposition 5 donne la complexité de la version dans le cas particulier où p=αn et m=1. Dans le cas général, l'évaluation du temps d'exécution de l'algorithme optimal est difficile. Nous l'avons omis, ce qui n'est pas trés important puisque nous verrons dans les pages suivantes que cette version est moins efficace que la vers') colonnes - colonnes.

5.2.2 Les paramètres colonnes - colonnes

La version descente colonnes - remontée colonnes de la méthode des paramètres est la suivante:

```
(* descente colonnes - remontée colonnes *)
Pour k <- 1 à n-1
    Pour j <- k+1 à n+m
        a_kj <- a_kj / a_kk
        Pour i <- 1 à k-1
            a_ij <- a_ij - a_ik * a_kj
        Pour i <- 1 à k
            a_k+1,j <- a_k+1,j - a_k+1,i * a_ij
```

Considérons les tâches P_{kj} suivantes :

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

Pour i <- à k-1 $\quad k+1 \leq j \leq n+m$

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

Pour i <- 1 à k

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

On retrouve la même structure globale que pour l'algorithme de Jordan par colonnes; ainsi on obtient le graphe d'ordonnancement suivant:

P_{12} P_{13} P_{14} ... $P_{1,n+1}$ ... $P_{1,n+m}$

P_{23} P_{24} ... $P_{2,n+1}$ ... $P_{2,n+m}$

P_{34} ... $P_{3,n+1}$ ... $P_{3,n+m}$

... ...

...

$P_{n,n+1}$ ... $P_{n,n+m}$

mais ici le temps d'exécution de chaque tâche est variable. La tâche P_{kj} s'effectue dans le temps (4k-1). Les contraintes temporelles restent inchangées:

(A) la tâche $P_{k,k+1}$ doit être exécutée avant les tâches $P_{k+1,j}$ ($j \geq k+2$)

(B) pour j fixé > k, la tâche $P_{k,j}$ doit être exécutée avant la tâche $P_{k+1,j}$

Ici encore, le graphe d'ordonnancement vérifie les hypothèses de la section 3. Nous évaluons donc le temps d'exécution de l'algorithme glouton, qui est toujours optimal.

Proposition 6 :

i) L'algorithme glouton est optimal pour la méthode des paramètres.

ii) $PCC_p = ((n+m-p)/p)\,[\,(4n+4m+1)(n+m-p+1)/2 - 2(n+m-p+1)(2n+2m-2p+1)/3 - (n+m)\,] + (p-m+1)(4n+2m-2p-5)$

iii) Pour $m=1$ et $p = \alpha n$,

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

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

Démonstration :

Les tâches P_{kj} ont le même temps d'exécution pour k fixé, à savoir 4k-1. La formule donnée au chapitre 3 devient ici:

$$PCC_p = (1/p) \sum_{i=1}^{n+m-p} (n+m-i).(4i-1) + \sum_{i=n+m-p+1}^{n} 4i-1$$

d'où le résultat annoncé:

$$PCC_p = ((n+m-p)/p)\,[\,(4n+4m+1)(n+m-p+1)/2 - 2(n+m-p+1)(2n+2m-2p+1)/3 - (n+m)\,] + (p-m+1)(4n+2m-2P-5)$$

Posons $m=1$ et $p = \alpha.n$, on obtient facilement

$$P_p = ((2 + 6\alpha^2 - 2\alpha^3)/ 3\alpha)\,n^2 + O(n/\alpha).$$

La valeur de $EPCC_p$ s'en déduit aisément, sachant que $PCC_1 = 2n^3/3 + O(n^2)$.

6. COMPARAISON DES METHODES DE JORDAN ET DES PARAMETRES EN PARALLELE

L'étude précédente des méthodes de Jordan et des paramètres nous montre quelles stratégies adopter pour la parallèlisation des méthodes. Quelle que soit la version considérée (par lignes ou par colonnes), nous avons pu mesurer le temps d'exécution des algorithmes. Il reste maintenant à comparer les algorithmes parallèles entre eux.

La parallélisation de la méthode de Jordan donne lieu à deux versions distinctes suivant qu'on élimine en lignes ou en colonnes et la méthode des paramètres à deux autres versions: descente et remontée en colonnes et descente en colonnes et remontée en lignes.

Dans le cas où l'on dispose de $p=\alpha n$ processeurs ($0<\alpha\leq 1$) et où $m=1$, les formules consignées dans le tableau suivant qui donnent le temps des algorithmes (à n^2 près) en fonction de α nous permettent de comparer la parallélisation des différentes versions entre elles.

	$\alpha\leq 1/2$	$\alpha\geq 1/2$
JL_p	$1/\alpha$	$(2\alpha+1)/2\alpha$
JC_p	$(\alpha^2+1)/\alpha$	
PCC_p	$(-2\alpha^3+6\alpha^2+2)/3\alpha$	
PCL_p	$(-6\alpha^3+12\alpha^2+3\alpha+2)/3\alpha$	

Proposition 7:

i) $JL_p \leq JC_p$, quel que soit α.

ii) $PCC_p \leq PCL_p$, quel que soit α.

iii) Pour $\alpha \leq 0.44$, $PCC_p < JL_p$

Pour $\alpha \geq 0.45$, $JL_p < PCC_p$

Démonstration :

i) Remarquons tout d'abord que, pour Jordan par lignes, nous n'avons qu'une majoration du temps d'exécution dans le cas $\alpha\geq 1/2$: $JL_p \leq (2\alpha+1)/2\alpha$

Dans ce cas, on vérifie que:

$$JL_p \leq (2\alpha+1)/2\alpha \leq JC_p$$

De même dans le cas $\alpha\geq 1/2$.

ii) Une remarque identique peut être formulée pour la méthode des paramètres. Pour la version colonnes - lignes, nous ne disposons que d'une minoration. Cependant, ceci est suffisant puisque:

$$PCC_p = (-2\alpha^3+6\alpha^2+2)/3\alpha \leq PCL_p(-6\alpha^3+12\alpha+3\alpha+2)/3\alpha$$

iii) Il nous reste donc à comparer la version de Jordan par lignes et la version colonnes - colonnes des paramètres. La comparaison des temps, pour $\alpha \leq 1/2$, conduit à trouver les racines de l'équation :

$$3 = -2\alpha^3 + 6\alpha^2 + 2$$

Il existe une racine comprise entre 0 et 1: $\alpha_0 = 0.442125301668$.

JL_p est supérieur à PCC_p si $\alpha \leq \alpha_0$ et inférieur sinon.

Pour $\alpha \geq 1/2$, la comparaison revient à annuler $-2\alpha^3+6\alpha^2-6\alpha+1$ qui n'a qu'une racine réelle égale à 2.42, donc $JL_p < PCC_p$ dans ce cas.

7. CONCLUSION

Le but de ce travail est l'étude de la parallélisation des méthodes de Jordan et des paramètres pour la résolution par diagonalisation de systèmes linéaires. Dans le cas séquentiel, la méthode des paramètres est plus efficace que celle de Jordan: le rapport des coûts est environ 2/3.

Nous avons concentré notre étude sur l'évaluation du nombre d'opérations arithmétiques, laissant de côté les temps de communication de mémoire à mémoire. Cette hypothèse n'est pas trop restrictive dans la mesure où nous avons limité l'accessibilité aux données: par lignes et par colonnes. Dans le cas important où le nombre de processeurs est inférieur à la taille du problème, le temps d'exécution de chaque tâche est de l'ordre de n unités: il n'est pas déraisonnable de supposer négligeable les temps d'accés.

Pour chacune des méthodes, deux versions parallèles ont été obtenues et des algorithmes efficaces proposés, en général optimaux. Dans la majorité des cas, les algorithmes optimaux sont asynchrones: les processeurs débutent leurs tâches dés que leurs données sont disponibles. Cependant, des versions synchrones peuvent être asymptotiquement optimales.

Nous avons donc comparé les temps d'exécution d'algorithmes optimaux, et les résultats s'interprètent directement en termes de complexité.

La meilleure parallélisation de la méthode de Jordan est par lignes alors que la meilleure parallélisation des paramètres est par colonnes.

La méthode de Jordan est plus apte à la parallélisation: l'efficacité de ses versions parallèles est excellente. Cependant, cela ne permet pas de combler dans tous les cas l'écart en nombre total d'opérations qui existe entre Jordan et les paramètres. Il n'est donc pas étonnant que, pour un nombre faible de processeurs, la méthode des paramètres ait un temps d'exécution inférieur à celui de la méthode de Jordan et que ceci soit inversé pour un grand nombre de processeurs. Nous avons prouvé que l'algorithme parallèle de Jordan par lignes est supérieur à l'algorithme parallèle des paramètres colonnes - colonnes dans le cas $\alpha \geq \alpha_0$ et inférieur dans l'autre cas, où α_0 est de l'ordre de 0.44 (le nombre de processeurs est pris égal à αn).

8. BIBLIOGRAPHIE

[1] H.M. Ahmed, J.M. Delosme, M. Morf, Highly concurrent computing structures for matrix arithmetic and signal processing, Computer Magazine, 1982, 65-82

[2] A. Bojanczyk, R.P. Brent, H.T. Kung, Numerically stable solution of dense systems of linear equations using mesh-connected processors, Preprint CMU, 1981

[3] M. Cosnard, Y. Robert, Complexité de la décomposition QR en parallèle, C.R. Acad. Sc. Paris 297, A, 1983, 549-552

[4] M. Cosnard, Y. Robert, Complexity of the Givens factorization for least squares problem, Actes Coll. "Vector and Parallel Processors for Scientific Computation", IBM Rome, 1985

[6] M. Cosnard, J.M Muller et Y. Robert, Parallel QR decomposition of a rectangular matrix, à paraître dans Numerische Math.

[7] J. Erhel, W. Jalby, A. Lichnewsky, F. Thomasset, Quelques progrès en calcul parallèle et vectoriel, 6ème colloque international sur les méthodes de calcul scientifique et technique, 1983, Glowinski et Lions eds

[8] M. Feilmeier, Parallel computers - Parallel mathematics, IMACS North Holland, 1977

[9] M.J. Flynn, Very high-speed computing systems, Proc. IEEE 54, 1966, 1901-1909

[10] J.M. Frailong, W. Jalby, J. Lenfant, XOR-schemes: a flexible data organization in parallel memories, 1985 International Conference on Parallel Processing

[11] J.M. Frailong, W. Jalby, Multipgrid processing on an SIMD architecture, Copper Mountain Multigid Conference, 31/3-3/4 1985)

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 5 de l'arrêté du 16 avril 1974

VU les rapports de présentation de Messieurs

. J. BERSTEL, Professeur
. P. QUINTON, Directeur de recherche
. F. ROBERT, Professeur

Monsieur Yves ROBERT

est autorisé à présenter une thèse en soutenance en vue de l'obtention du grade de DOCTEUR D'ETAT ES SCIENCES.

Fait à Grenoble, le 4 décembre 1985

Le Président de l'U.S.M.G

Le Président,

M. TANCHE

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble