Etude d'une architecture cellulaire programmable : définition fonctionnelle et méthodologie de programmation

Eric Payan

► To cite this version:

Eric Payan. Etude d'une architecture cellulaire programmable : définition fonctionnelle et méthodologie de programmation. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1991. Français. NNT : . tel-00339754

HAL Id: tel-00339754
https://theses.hal.science/tel-00339754

Submitted on 18 Nov 2008

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

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

THESE

présentée par

Eric PAYAN

pour obtenir le grade de **DOCTEUR**
de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**
(Arrêté ministériel du 23 novembre 1988)

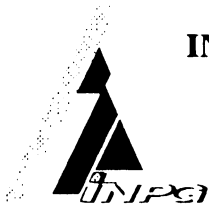
Etude d'une architecture cellulaire programmable :
Définition fonctionnelle et Méthodologie de programmation

Date de soutenance : 11 Juin 1991

Composition du jury :

Messieurs	Yves	CHIARAMELLA	Président
	Guy	MAZARE	
	Paul	CASPI	
	Jean Paul	SANSONNET	Rapporteurs
	Michel	MERIAUX	

Thèse préparée au sein du **Laboratoire** de **Génie Informatique**.



INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
46 avenue Felix Viallet
38031 GRENOBLE cedex

Tél. : 76.57.45.00

Année universitaire 1989

Président de l'Institut :
Monsieur Georges LESPINARD

Professeurs des Universités

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

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

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

Chercheurs du C.N.R.S

DIRECTEURS DE RECHERCHE CLASSE 0

LANDEAU	Ioan
NAYROLLES	Bernard

Directeurs de recherche 1ère Classe

ANSARA Ibrahim
CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
KRAKOWIAK Sacha
LEPROVOST Christian
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ARMAND Michel
AUDIER Marc
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
CHATILLON Chritiant
CLERMONT Jean-Robert
COURTOIS Bernard
DAVID René
DION Jean-Michel
DRIOLE Jean
DURAND Robert
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GARNIER Marcel
GUELIN Pierre
JOUD Jean-Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MEUNIER Jacques
PEUZIN Jean-Claude
PIAU Monique
RENOUARD Dominique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
VENNEREAU Pierre
WACK Bernard
YONNET Jean-Paul

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

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

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

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

E.N.S.H.M.G
ROWE Alain

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

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

Laboratoires extérieurs :

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

Situation particulière

PROFESSEURS D'UNIVERSITE

DETACHEMENT

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

SURNOMBRE

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

GRENOBLE 1

UNIVERSITE JOSEPH FOURIER

SCIENCES. TECHNOLOGIE. MEDECINE

Président de l'Université :
M. NEMOZ Alain

Année universitaire 1988-1990

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GÉOGRAPHIE

PROFESSEURS de 1ère Classe

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARNAUD Paul	Chimie Organique
ARVIEU Robert	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S.
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire I.S.N.
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerd	Informatique
GAGNAIRE Didier	Chimie Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean-Pierre	Mécanique
GIDON Maurice	Géologie
GUITTON Jacques	Chimie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures

JOSELEAU Jean-Paul	Biochimie
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PANNETIER Jean	Chimie
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRE Jean-Louis	Chimie Organique
RENARD Michel	Thermodynamique
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ARMAND Gilbert	Géographie
ATTANE Pierre	Mécanique
BARET Paul	Chimie
BERTIN José	Mathématiques
BLANCHI J. Pierre	STAPS
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BORRIONE Dominique	Automatique Informatique
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUANDET Jean-François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean-Pierre	Mécanique
COLOMBEAU Jean-François	Mathématiques (ENSL)
COURT Jean	Chimie
CUNIN Pierre-Yves	Informatique
DAVID Jean	Géographie

DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
HERINO Roland	Physique
JARDON Pierre	Chimie
KERCKHOVE Claude	Géologie
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MOREL Alain	Géographie
NEMOZ Alain	Thermodynamique CNRS - CRTBT
NGUYEN HUY Xuong	Informatique
OUDET Bruno	Mathématiques Appliquées
PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean-René	Physique
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L'IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
CHEHIKIAN Alain	E.E.A. IUT 1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	E.E.A. IUT 1

PROFESSEURS de 2ème Classe

BEE Marc	Physique IUT 1
BOUTHINON Michel	E.E.A. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHENAVAS Jean	Physique IUT 1

CHILO Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
FOSTER Panayotis	Chimie IUT 1
GOSSE Jean-Pierre	E.E.A. IUT 1
GROS Yves	Physique IUT 1
HAMAR Roger	Chimie IUT 1
KUHN Gérard, (détaché)	Physique IUT 1
LEVIEL Jean-Louis	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	E.E.A. IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
TURGEMAN Sylvain	Génie Civil
VINCENDON Marc	Chimie IUT 1
ZIGONE Michel	Physique IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Thérapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiène et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS Classe Exceptionnelle et 1ère Classe

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puériculture	C.H.R.G.
BEREZ Henri	Orthopédie-Traumatologie	Hôpital Sud
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hôpital Sud
COUDERC Pierre	Anatomie Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci
HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.

MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème Classe

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim-Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hôpital Sud
BERNARD Pierre	Gynécologie Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	Abidjan
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hôpital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et informatique médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.
MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie Obstétrique	Hôpital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

Remerciements

Je tiens à remercier

Monsieur Yves CHIARAMELLA, directeur du Laboratoire de Génie Informatique, de l'honneur qu'il m'a fait en acceptant de présider le jury de cette thèse.

Monsieur Guy MAZARE, professeur à l'ENSIMAG, de m'avoir accueilli dans son équipe et d'avoir assuré l'encadrement de cette thèse.

Messieurs Jean Paul SANSONNET, Directeur de Recherche au CNRS et Directeur du GDR "Architecture de machines nouvelles" et Michel MERIAUX, Directeur du LIFL, qui ont acceptés d'être rapporteurs de cette thèse.

Monsieur Paul CASPI qui a accepté de faire partie du jury de cette thèse et qui m'a aidé par ses conseils.

Je remercie également toutes les personnes que j'ai côtoyées pendant trois années de préparation de cette thèse. Grâce à leur aide et à leur amabilité, j'ai pu mener à bien ce travail dans de bonnes conditions.

Sommaire

1. Introduction 1

Vers une architecture
MIMD à grain fin

2. Les machines massivement parallèles : état de l'art .. 6
2.1 Introduction 6
2.2 le XILINX 7
2.2.1 Structure 8
2.2.2 Environnement logiciel 10
2.3 La Connection Machine 11
2.3.1 Structure 11
2.3.2 Le processeur élémentaire 11
2.3.3 Routeur 12
2.3.4 Environnement logiciel 13
2.4 Le Meganode 13
2.4.1 Structure 13
2.4.2 Environnement logiciel 14
2.5 Autres machines ou projets 14
2.6 Des problèmes sans architecture adaptée 16
2.6.1 Introduction 16
2.6.2 Etude d'un exemple : le produit de matrice creuse 17
2.6.3 Un modèle d'exécution mal exploité : les graphes dataflow 20

3. Etude d'une architecture cellulaire intégrée programmable: 23
3.1 Objectifs 23
3.2 Les topologies possibles 24
3.3 Réalisation des communications 25
3.3.1 Le mécanisme d'acheminement des messages 25
3.3.2 Les communications inter-cellules 29
3.3.3 Les contraintes sur le routage 31
3.4 Un traitement MIMD 33
3.4.1 Le Jeu d'instruction 34
Instruction de manipulation de l'accumulateur : 35
Opérations arithmétique 8 Bits : 35
Opérations logiques : 36
Décalages : 36
Arithmétique 16 bits : 37
Divers : 37
Communication : 38
3.4.2 Le chemin de donnée 40
3.5 Connexion à la machine hôte 44
3.5.1 Les entrées sorties 45
3.5.2 L'initialisation du réseau 45
L'accès aux cellules cachées : La transparence 46
La taille mémoire : le temps de programmation

....48

3.6 Performances..50

3.7 Conclusion 51

Une nouvelle méthode de programmation d'une architecture massivement parallèle, et sa compilation

4. Des langages de programmation pour machines parallèles 54

4.1 Des techniques de compilation spécifiques ?................54

4.2 Les langages inspirés de CSP..55

4.2.1 Sur les machines à base de Transputers.........55

4.2.2 Inadaptés à notre architecture.............................55

4.3 Les langages orientés objets...56

4.4 Les langages dataflow...57

4.4.1 Introduction 57

4.4.2 Le langage LUSTRE...58

Variables, équations, expressions.....................58

Opérateurs au niveau des valeurs.....................58

Opérateurs synchrones sur les suites..............58

Structuration des programmes...........................59

Changement d'horloge 59

Echantillonnage synchrone.................................60

Projection 61

Les tableaux...61

4.4.3 Pourquoi LUSTRE ? 63

5. La compilation de LUSTRE sur notre machine 65
5.1 Méthodologie de travail 65
5.2 Idée de départ 67
5.3 Construction du graphe dataflow 70
5.4 Le placement 72
5.4.1 Allocation dynamique ou allocation statique ?72
5.4.2 Limitation des performances: problèmes et solutions 73
5.4.3 Les impossibilités de placement et les cellules relais 79
5.4.4 Les méthodes de placement. 83
5.5 La synchronisation 84
5.5 Pourquoi synchroniser ? 84
5.6 De l'influence de l'instant d'émission des messages de synchronisations sur les performances ... 85
5.6 La génération des opérateurs LUSTRE 94
5.7 La méthode parallèle 95
5.7.1 Présentation 95
5.7.2 Intérêt et limitation 98
5.8 La méthode séquentielle 99
5.8.1 Présentation 99
5.8.2 Ordonnancement des opérateurs 100
5.8.3 Le problème du placement 104

5.8.4 Synchronisation.......... 108
5.8.5 Le problème des horloges et des relais.......... 109
5.9 Comparaison des deux méthodes.......... 110
5.9.1 Longueur et optimisation du code.......... 111
5.9.2 Performances temporelles du code.......... 116
5.9.3 Sensibilité au jeu d'instruction.......... 119
5.9.4 Sensibilité au placement.......... 120
5.9.5 Conclusion provisoire.......... 121
5.10 Confrontation compilateur LUSTRE, programmation en assembleur : étude d'un exemple, le produit de matrice.. 121
5.10.1 Introduction.......... 121
5.10.2 Ecriture des programmes.......... 122
5.10.3 Performances respectives.......... 124
5.11 Conclusion.......... 127

6. Conclusion.......... 129

Bibliographie.......... 133

ANNEXE 1 :
Les outils de compilation.......... 139
Introduction.......... 140
Le développement des tableaux.......... 142
L'analyse syntaxique et la génération du graphe dataflow 144
Limitation du degré de sortie des opérateurs.......... 146

Datation du graphe 149

Placement 150

Les performances 151

Les possibilités d'implantation 151

Simulations fonctionnelles 153

Les simulations OCCAM 153

Les simulations en HELIX 154

Le simulateur fonctionnel 154

Génération de code 155

Le générateur de code "pseudo parallèle" 155

Le générateur de code séquentiel 158

Simulation du réseau 160

ANNEXE 2 :
Exemples de programmes 163

Programmes LUSTRE 164

La transformée de Fourier rapide (FFT) 164

Le produit de matrice 166

Programme de tri 167

Produit de convolution 168

Exemple de programme assembleur : le produit de matrice 169

1. Introduction

Pour répondre à des besoins toujours croissants en puissance de calcul, on a vu se multiplier depuis quelques années les études concernant les architectures parallèles.

Sujet de recherche à l'origine, certaines de ces architectures parviennent maintenant à dépasser le cercle des laboratoires pour rencontrer leurs premiers succès commerciaux. Un des principaux facteurs de cette réussite est la disponibilité d'outils logiciels permettant une programmation plus aisée et facilitant la gestion du parallélisme.

Notre équipe travaille sur les architectures parallèles depuis le début des années 80. Ses premiers travaux concernaient la paralèlisation et l'implantation sur le silicium d'un certain nombre d'algorithmes destinés à la conception assisté par ordinateur de circuits intégrés (simulation logique, placement ...). Ces travaux nous ont amenés à définir une architecture nouvelle : le réseau cellulaire asynchrone à communication par passage de message. Au fil des recherches, le champ d'application de cette architecture c'est révélé très large, un certain nombre de points communs réunissant toutes ces applications :

- un grain de parallélisme très fin

- une circulation des données et un calcul des processus non synchrones

- des communications non régulières

Pourtant malgré ces similitudes la somme de travail exigée par l'implantation d'un seul de ces algorithmes restait considérable. Pour chaque algorithme nouveau à étudier, en plus de l'étude de la parallèlisation du problème (propre à chaqu'un), une bonne part des simulations du réseau est surtout le dessin du circuit devait être refait, ne bénéficiant que peut des études précédentes. Il était impossible pour une équipe de la taille de la notre d'assumer ces études et l'intérêt des algorithmes étudiés ne le justifié d'ailleurs pas toujours. L'idée c'est donc progressivement imposée à nous de d'implanter ces algorithmes non pas

de façon matériel mais de façon logiciel sur une architecture programmable que nous avons dut définir.

Il est maintenant impossible d'imaginer une architecture programmable sans ce soucier des outils nécessaire à ce travail. Etant donnée la finesse du grain de parallélisme et la nature non régulière et asynchrone des applications visées par notre architecture, il est impossible de laisser au programmeur la charge de parallèliser les problèmes. Les langages data flow de part l'expression naturel du parallélisme qu'il permettent et l'absence de contrôle centralisé qui les caractérisent, semblent les meilleurs candidat à une implantation sur notre architecture. C'est donc naturellement que nous nous avons étudié l'implantation de l'un d'eux : LUSTRE comme premier outil de programmation de notre réseau.

Dans la première partie de ce mémoire, après un rapide tour d'horizon des machines existantes, nous définirons notre architecture : le réseau cellulaire asynchrone. Venant à la suite de travaux portant sur la définition et l'intégration de plusieurs algorithmes, dans des architectures massivement parallèle dédiées; notre architecture actuelle généraliste et programmable a pu profiter de l'expérience acquise.

Notre réseau cellulaire asynchrone programmable, est le résultat du travail de l'ensemble de l'équipe. On peut en donner une répartition sommaire:

- P. Rubini a étudié le jeu d'instruction de la cellule ainsi que les différents modes de routage de messages possibles et leurs influences sur les performances. Il travaille actuellement sur l'écriture d'algorithmes adaptés à notre réseau.

- M. Karabernou réalise actuellement l'implantation sur le silicium d'un premier réseau prototype.

- Mon travail a porté sur l'étude d'un premier outil de programmation adapté à notre architecture : un compilateur LUSTRE.

- D'autre travaux sont en cours concernant l'interface du réseau avec l'hôte ainsi qu'une étude de faisabilité portant sur l'usage des langages objet pour programmer ce type d'architecture.

Il est difficile d'en expliciter certains aspects sans présenter l'ensemble du projet. Pour laisser à chacun le soin de développer son propre travail certains aspects de l'architecture seront introduits brièvement sans aucune justification des choix effectuées. Pour obtenir de plus amples renseignements il sera nécessaire de se reporter aux publications.

La deuxième partie de cette étude sera consacrée à la présentation de la méthode de programmation du réseau, que nous avons imaginée et réalisée. Dépassant le cadre d'une simple étude de faisabilité, nous nous sommes attachés, en nous appuyant sur de nombreuses expérimentations et sur la réalisation d'un prototype, à étudier les performances et par là même l'intérêt d'un tel outil.

Vers une architecture MIMD à grain fin

2. Les machines massivement parallèles : état de l'art

2.1 Introduction

Les progrès réalisés dans le domaine des circuits VLSI permettent l'intégration d'un nombre toujours croissant de composants élémentaires dans un seul circuit. On trouve déjà des circuits commercialisés comportant plus d'un million de transistors sur des surfaces dépassant le centimètre carré. Les progrès futurs devraient repousser encore plus loin ces limites (on parle déjà de 32 millions de transistors).

Profitant de ces progrès, les machines séquentielles (basées sur le modèle de Von NEUMAN) ont vu leurs performances décuplées en quelques années. Le niveau de performance atteint est tel que tout nouveau progrès ne peut se faire que grâce à des dispositifs extrêmement complexes et couteux en matériel.

Ces progrès technologiques rendent aussi possible la réalisation de circuits comportant un grand nombre de cellules identiques; chaque cellule n'étant qu'un automate très simple. Cette régularité associée à la simplicité du motif élémentaire qui les compose rend leur conception plus simple que celle des circuits séquentiels de taille équivalente.

On a vu se multiplier dans la littérature les appellations pour tenter de qualifier les architectures parallèles. Si la classification de [FLY72], basée sur le mode de parallélisme utilisé, est généralement admise, il n'existe pas encore de norme pour qualifier le degré de parallélisme d'une architecture. On a ainsi vu des machines "parallèle", "massivement parallèle", voire "hyper parallèle". Face à cette multiplicité d'appellations, il est nécessaire

de définir ce que signifie pour nous le parallélisme massif de notre architecture.

Si il est facile de faire la distinction entre machines séquentielles et parallèles, il est plus difficile de donner une limite chiffrée entre parallélisme massif et parallélisme "simple". Plutôt que de la fixer à un nombre arbitraire de processeurs 10,100, 1000... nous avons préféré la définir par son grain de parallélisme : une machine qui tend à utiliser le parallélisme maximal d'une application sera qualifiée de massivement parallèle (on parle alors de parallélisme à grain fin). Beaucoup de problèmes peuvent être décomposés en un très grand nombre de tâches exécutables en parallèle. Pour ce type d'algorithmes notre définition rejoint la notion du "grand nombre de processeurs".

Pour mieux comprendre les raisonnements qui nous ont guidés dans la définition de notre architecture, il est nécessaire de faire le point sur les machines existantes ou en cours d'études. Cette liste ne se veut pas un catalogue complet de toutes les architectures existantes, il s'agit plutôt d'un éventail représentatif des différentes idées circulants dans la communauté scientifique.

2.2 le XILINX

Le Xilinx n'est pas un "ordinateur" au sens classique du terme mais plutôt un circuit logique programmable [XIL87]. Sa présence dans ce mémoire se justifie par sa structure cellulaire qui peut être qualifiée de "massivement parallèle". Parmi les utilisations possibles de ce circuit, il en existe au moins deux qui rejoignent le domaine d'applications visées par notre machine.

- Un circuit cellulaire dédié à la simulation logique à été réalisé par notre équipe, sur le même principe, il est facilement envisageable d'implanter cette application sur le futur réseau programmable. On peut de même envisager d'utiliser les cellules du XILINX pour simuler des portes logiques ou des retards (compte tenu de la vitesse, on peut même parler d'émulation).

- Une étude est en cour au sein du Laboratoire de Génie Informatique (F. Rochetaux) sur l'utilisation de LUSTRE pour la programmation du XILINX [ROC89]; cette approche de la programmation des circuits massivement parallèles est voisine de celle présentée dans ce mémoire.

2.2.1 Structure

Globalement cette machine peut être vue comme :

- Un ensemble de cellules de très petite taille, comportant un point mémoire et un bloc combinatoire capable de calculer une fonction logique de quelques variables.

- Un réseau de communication programmable permettant de définir les liaisons entre cellules. Ce réseau utilise trois niveaux d'interconnexions :

Les liaisons directes permettent de relier entre eux deux cellules voisines.

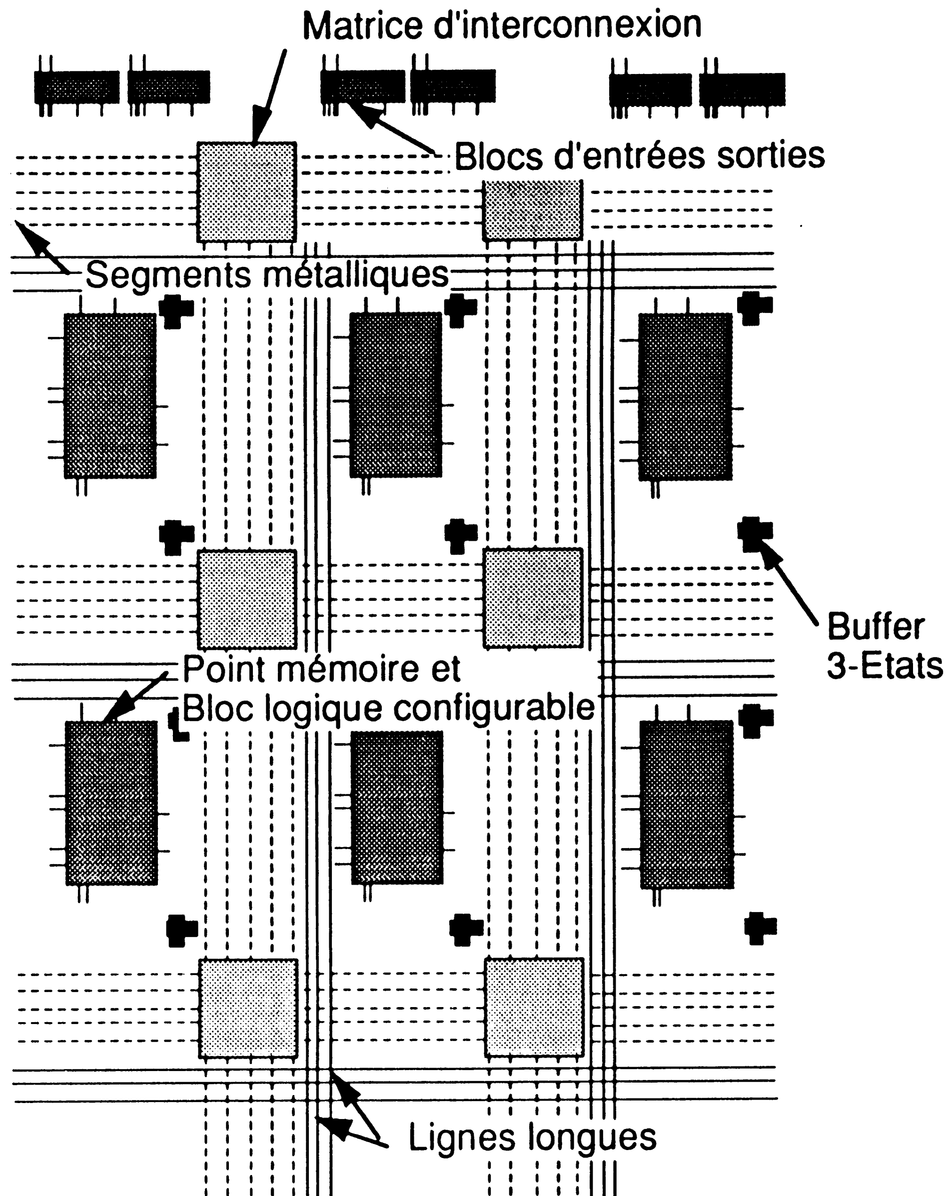
Les liaisons à usage générale permettent de relier entre eux deux blocs quelconques. A chaque intersection se trouve une matrice d'interconnexion qui permet de définir les connexions entre les segments métalliques.

Les lignes longues parcourent toute la longueur du circuit. Il y en a deux horizontales par rangée de cellules et trois verticales par colonne.

Le fonctionnement de cette machine est complètement synchrone puisque les points mémoires de toutes les cellules sont commandés par la même horloge.

Outre le réseau de routage et les cellules un circuit comporte aussi des blocs d'entrées-sorties permettant d'échanger des informations avec l'extérieur, ainsi que des buffers trois états qui peuvent être utilisés comme points de mémorisation.

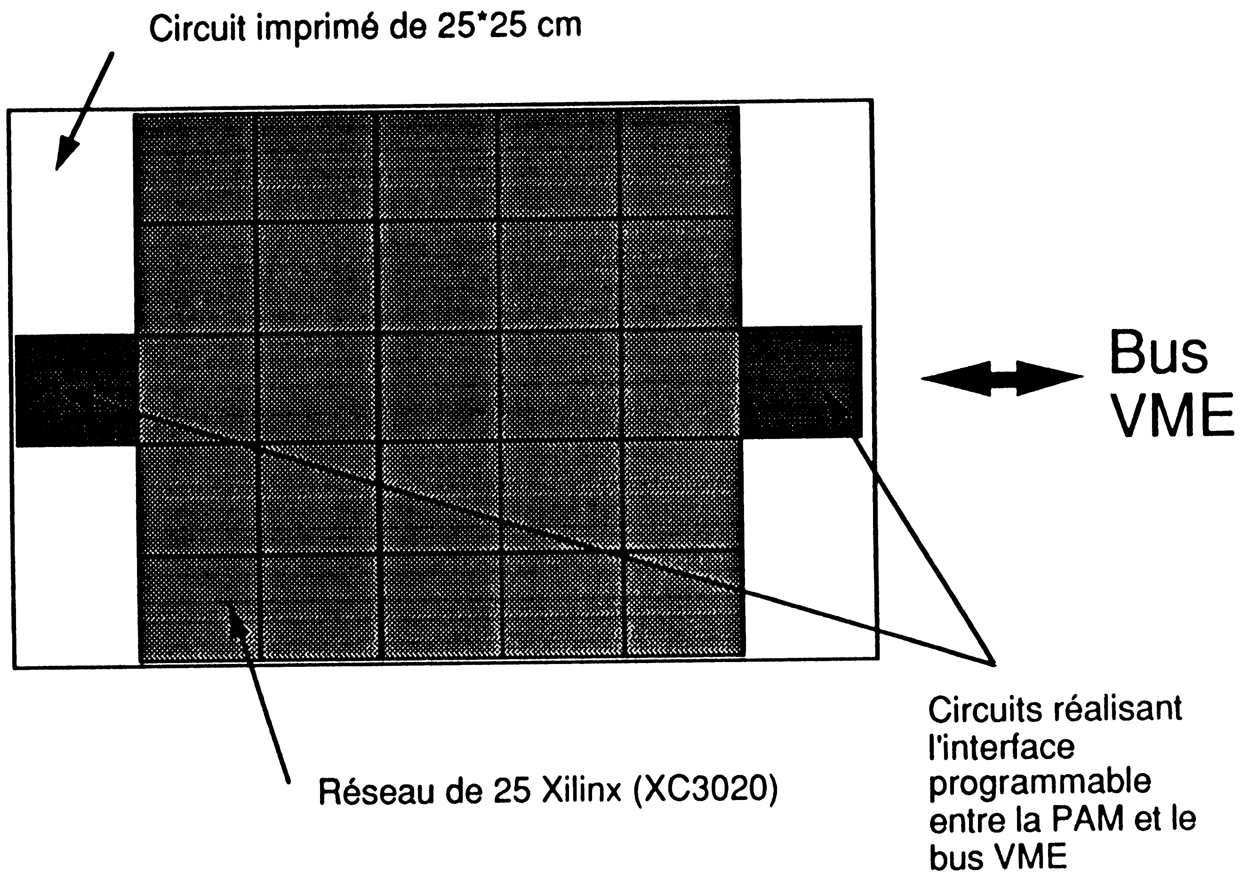
Partie de circuit XILINX



Un seul circuit XILINX de la série 3000 peut comporter jusqu'à 320 cellules programmables. Les circuits sont cascadables pour permettre la réalisation de réseaux de plus grandes tailles. Ce circuit est disponible commercialement et connaît un certain succès.

La possibilité de cascader plusieurs de ces circuits pour réaliser des réseaux de grandes tailles, est utilisée dans l'architecture de la PAM (Programmable Active Memories) [BER89]. Cette machine est constituée d'une matrice de 5*5 circuits soit approximativement 3K cellules.

L'ensemble est relié à un bus VME par l'intermédiaire de 2 XILINX supplémentaires, qui réalisent une interface programmable.



2.2.2 Environnement logiciel

L'environnement logiciel du XILINX est plus proche des outils de CAO de circuits intégrés que des langages de programmation classiques. On y trouve un éditeur de schémas et un placeur.

Des travaux sont en cours dans notre laboratoire pour permettre l'utilisation du langage de haut niveau LUSTRE pour programmer ce circuit. En effet il peut être vu comme un réseau d'opérateurs logiques programmables, il est donc naturel de tenter de placer un graphe dataflow sur cette machine. Néanmoins à l'usage il apparaît que ce circuit n'est adapté qu'à l'exécution de graphe donc les valeurs circulant sur les arcs

sont des booléens. Le travail sur des entiers nécessite une décomposition au niveau du bit et donc un grand nombre de cellules.

Grâce à son architecture originale et à son caractère cellulaire ce circuit peut présenter pour des programmes manipulant des booléens une alternative performante aux machines traditionnelles.

2.3 La Connection Machine

La Connection Machine utilise un parallélisme de type SIMD c'est à dire qu'elle travaille de manière synchrone en effectuant la même instruction sur des données multiples [DEL90].

Le principe d'utilisation de la Connection Machine est d'associer un processeur à chaque élément de l'ensemble sur lequel on veut travailler.

Ce principe de fonctionnement ainsi que le grand nombre de processeurs de la Connection Machine (de 16k à 64k processeurs) permet de la classifier indubitablement parmi les machines massivement parallèles.

2.3.1 Structure

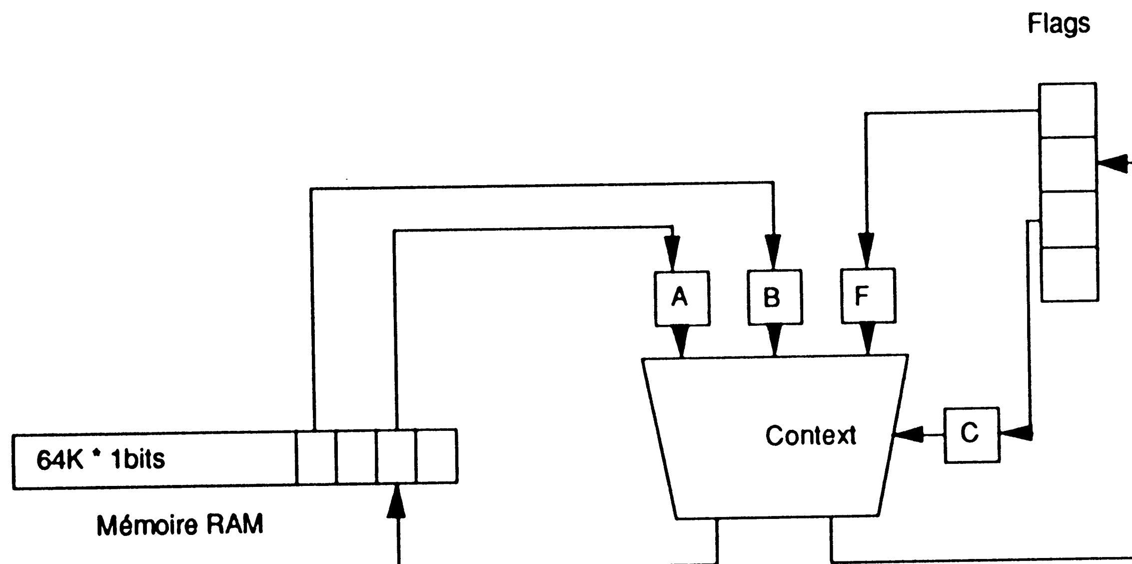
La Connection Machine peut comporter jusqu'à 65536 processeurs, repartis en 8 cubes. Les processeurs sont rassemblés par groupes de seize, les groupes étant eux même reliés entre eux pour former un hyper-cube de degré 12. Les seize processeurs d'un même groupe sont entièrement connectés entre eux par l'intermédiaire d'un réseau cross-bar.

2.3.2 Le processeur élémentaire

Le processeur élémentaire de la Connection Machine est un processeur 1 bit, il est capable d'effectuer n'importe quelle opération de $\{0,1\}^3$ dans $\{0,1\}^2$ soit 2^{16} opérations possibles.

Chaque processeur reçoit du frontal, à travers un séquenceur, l'instruction à exécuter. Pour permettre de différentier le calcul de certains processeurs en fonction de leurs données, ils disposent d'un context-flag : si il est à zéro les opérations sont inhibées sur ce processeur.

De plus chaque processeur dispose d'une mémoire de donnée de 64K bit.



Le processeur élémentaire de la Connection Machine

Pour accélérer les calculs numériques, chaque groupe de 32 processeurs dispose d'une unité de calcul flottant.

2.3.3 Routeur

Pour permettre des communications non locales (permettre à deux processeurs non directement connectés de communiquer), à chaque groupe de seize processeurs est associé un routeur.

Des instructions assembleur permettent alors d'envoyer des messages avec plusieurs niveaux de communications : communications générales ou en grille.

2.3.4 Environnement logiciel

L'environnement logiciel de la Connection Machine est remarquablement riche pour une machine parallèle. L'utilisateur dispose de plusieurs langages dérivés des langages usuels : CM-Fortran, C*, *Lisp. Il est aussi possible de programmer la Connection Machine à un niveau inférieur en utilisant ParIS (Parallel Instruction Set). Ce langage peut être comparé à un assembleur très évolué.

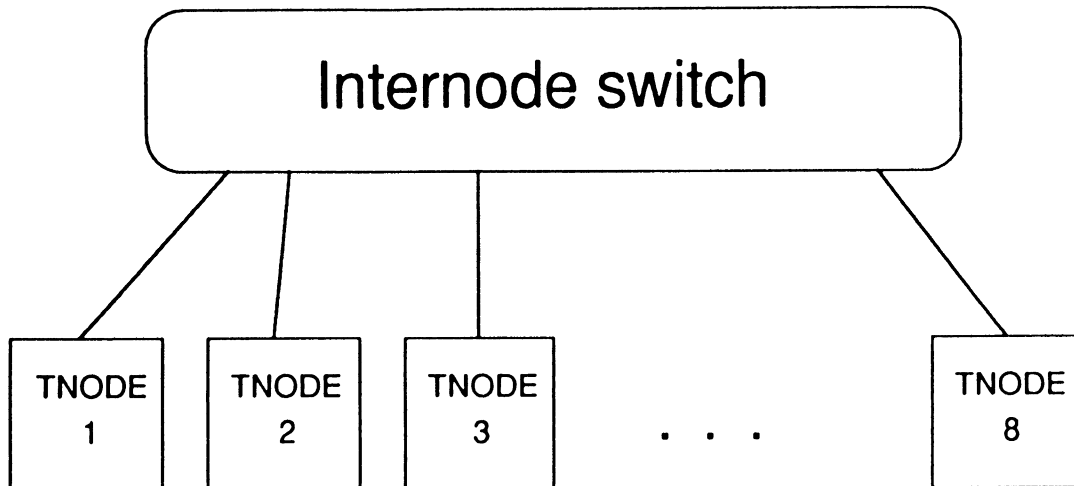
2.4 Le Meganode

Le Transputer est un puissant micro processeur 32 bits, réalisé par la société INMOS, qui dispose de 4 liens de communication, ce qui le rend particulièrement adapté à la réalisation d'architectures parallèles. De nombreuses machines utilisant cette brique de base sont apparues récemment. Le Meganode constitue l'un des projets les plus ambitieux articulé autour du Transputer .

2.4.1 Structure

Sur le Meganode [TOU90], les Transputers sont regroupés en modules appelés tandem-nodes. Chaque tamdem-node peut contenir jusqu'à 32 Transputers de travail, 2 Transputers de contrôle et quelques uns utilisés comme serveurs (mémoire, disque). Les Transputers d'un même tandem-node sont reliés entre eux par un switch permettant de réaliser n'importe quelle topologie de connexion.

Le Mega node est constitué de 8 tandem-nodes reliés entre eux par des "internode-switch" permettant de connecter n'importe quel Transputer d'un tamdem-node à un autre Transputer d'un autre tandem-node.



Tous ces niveaux de connexions à travers des switchs programmables font du mega node une machine reconfigurable. Cette machine n'à donc pas de topologie fixe tel l'hypercube ou la grille mais une topologie variable (éventuellement dynamiquement) suivant les applications.

2.4.2 Environnement logiciel

Bien que le langage "naturel" de programmation des machines basées sur le Transputer soit OCCAM, le langage de programmation le plus utilisé est le LOGICAL C (une version de C adaptée au parallélisme). L'environnement logiciel de cette machine présente encore de nombreuses faiblesses : problème de fiabilité des compilateurs, absence d'outils de mise au point. Tous ces problèmes freinent considérablement son développement. Cette expérience nous montre que le nombre et la qualité des outils logiciels disponibles contribuent pour une bonne part au succès d'une architecture.

2.5 Autres machines ou projets

Notre équipe travaille depuis plusieurs années à la réalisation d'architectures massivement parallèles dédiées à des applications particulières. Toutes ces implantations utilisent les mêmes concepts de base (qui sont aussi ceux de la machine programmable) : une structure en

grille à deux dimensions, des communications par passage de messages. Les applications étudiées sont :

- La simulation logique [COR88] [OBJ88] : Il est possible sur un tel réseau de placer un schéma logiques à simuler. Un circuit comportant 4 cellules, intégrant l'algorithme de simulation logique à délai unitaire d'une porte, à été réalisé. De nombreux problèmes liés à la technologie utilisée, n'ont pas permis un test satisfaisant ainsi que la réalisation d'une carte prototype.

- La reconstruction d'image [LAT89]: pour résoudre rapidement ce problème d'imagerie médicale très gourmant en temps de calcul, un circuit comportant une cellule (20 000 transistors) a été réalisé et testé avec succès. Une carte prototype intégrant un réseau de 25 cellules est en cours de réalisation.

- Réseau de neurones [FAU90]: Une étude visant à simuler le fonctionnement d'un réseau neuronal à l'aide d'une architecture cellulaire dédié est en cours. Parmi les problèmes traités, le choix de la précision du codage et la limitation du degré d'entrée des neurones (et donc des cellules) ont été résolu. Les simulations ont montré la qualité des performances obtenues par une telle machine.

De nombreux autres projets sont en cours de développement dans d'autres laboratoires, parmi lesquels:

- Le Warp [POM88] est une machine systolique programmable. C'est un anneau de puissants processeurs (10 MFLOPS par processeur), chacun disposant d'une mémoire données de 32KMots à accès rapide. L'environnement de programmation comporte un langage inspiré de Pascal : W2. Ce projet à été initialement développé à l'université de Carnegie Mellon, une version industrielle à depuis été réalisée par INTEL : le IWARP.

- La machine Mega [GER90] est un projet français de l'université d'Orsay qui vise à la réalisation d'une machine MIMD comportant jusqu'à 1 million de processeurs 16 bits, chacun disposant d'une mémoire d'environ 16K mots. La topologie retenue est le cube, un packaging spécial

permettant l'emboîtement en trois dimensions des circuits (chaque circuit comportant un processeur et sa mémoire). Un routeur permet par envoi de messages la communication dans le cube.

Nous voyons donc qu'il existe un grand nombre de projets dont quelques uns sont déjà commercialisés. Nous allons pourtant voir que malgré cette diversité d'architectures proposées, il existe encore une classe de problèmes non couverte par les machines actuelles.

2.6 Des problèmes sans architecture adaptée

2.6.1 Introduction

Notre expérience en matière d'algorithmique parallèle nous conduit à penser qu'il existe une classe de problèmes dont le schéma d'exécution est mal adapté aux machines actuelles.

Ces applications ont en commun un certain nombre de particularités:

- Un grain de parallélisme possible très fin

- Une exécution des opérateurs ou une circulation des données asynchrone.

Le grain de parallélisme fin de ce type d'algorithmes rend difficile son implantation sur des architectures à grain moyen (du type des réseaux de Transputers). Sur ces machines, le nombre de processeurs étant limité et de toute façon inférieur au nombre de grains de l'algorithme, on est obligé de procéder à un regroupement qu'il est difficile de rendre optimal.

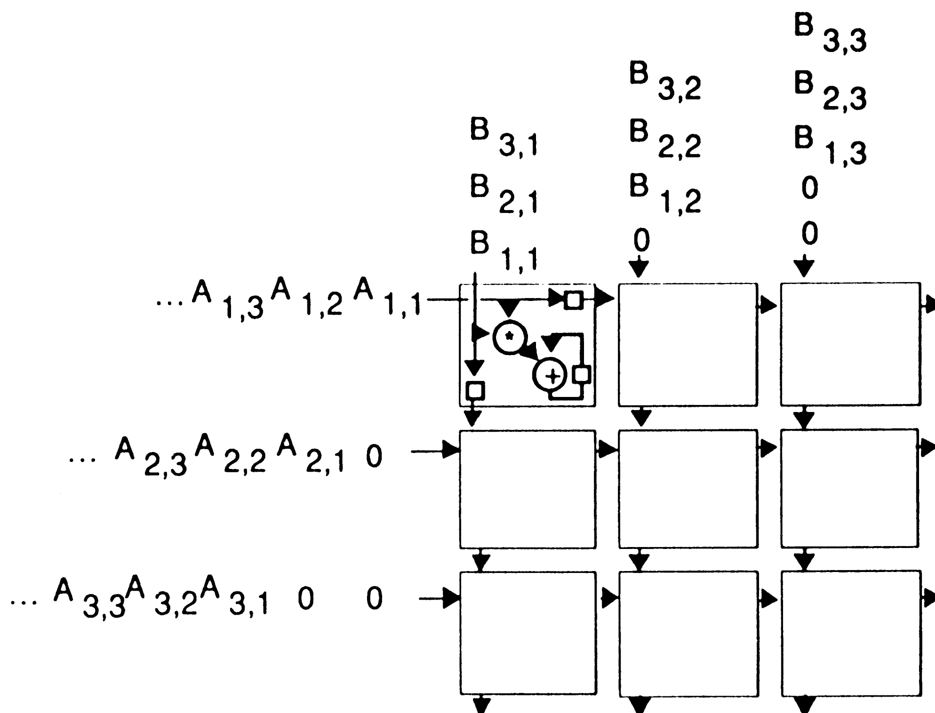
L'exécution asynchrone de ce type d'algorithme rend difficile et non optimal son implantation sur des machines SIMD de type Connection Machine. Pour illustrer notre propos nous allons examiner en détail quelques exemples.

2.6.2 Etude d'un exemple : le produit de matrice creuse

Le Produit de matrices est l'une des opérations les plus courantes en calcul numérique intensif. Son temps de calcul en $O(n^3)$ à rendu nécessaire le développement d'architectures spécialisées à cette application.

Sur les "supercalculateur" comme le CRAY-1 la technique du pipeline est utilisée pour accélérer l'opération. Les architectures systoliques apportent aussi une solution massivement parallèle à ce problème.

Produit de matrice systolique :



Dans bon nombre de problèmes numériques, les matrices sur lesquelles on travaille sont très grosses mais très creuses : seules quelques éléments par ligne ou par colonne sont non nuls. Les techniques dites de "Matrices creuse" permettent de réduire la quantité de calcul en conservant des matrices aussi creuses que possibles, et de ranger de telles matrices sous forme compacte en ne conservant que les éléments non nul. Malheureusement les techniques d'optimisation matérielles et logicielles sont difficiles à combiner avec cette mise en oeuvre parallèle :

les multiplications par zéro que l'on aimerait éviter sont en général contenues dans le flot de données alimentant le pipeline.

Notre idée consiste à ne faire circuler dans notre réseau que les éléments non nuls. Il nous faut alors associer à chaque élément sa position dans la matrice opérande. Chaque élément de la matrice A (pour le calcul de A*B) se voit ainsi associer son numéro de colonne alors que l'on associe les numéros de ligne avec les éléments de B.

L'association de deux éléments de A et B ne dépend plus alors d'une horloge mais uniquement de leurs positions dans les matrices opérandes.

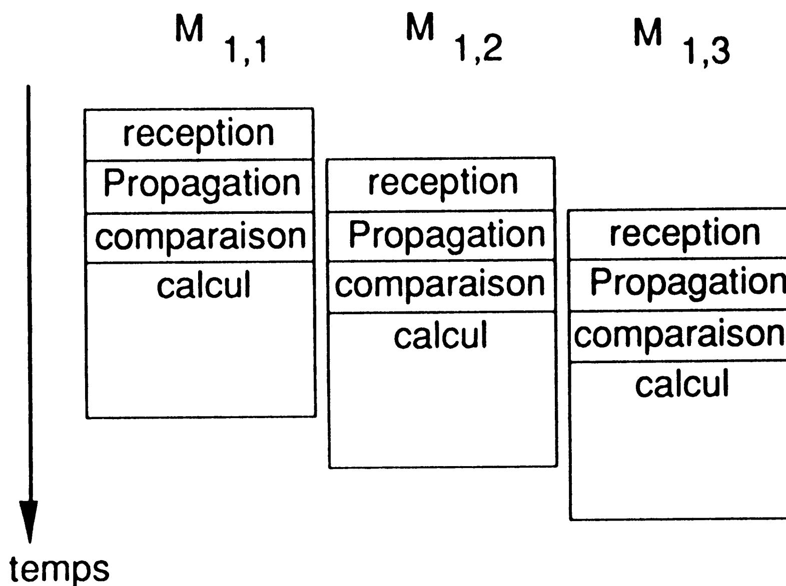
L'algorithme qu'effectue alors chaque cellule est proche d'une fusion de listes : une cellule qui dispose en entrée de deux valeurs ayant des positions différentes dans leurs matrices absorbe la valeur ayant la position la plus petite et demande à la cellule amont correspondante la valeur suivante. Un calcul n'est effectué que si les deux positions sont identiques.

```
forward, match and MAC (Ain,Bin:stream; returns Aout,Bout:stream)
a:=get(Ain)
b:=get(Bin)
put(Aout,a)
put(Bout,b)
s:=0
cycle
    if a.rank = b.rank
        s:=s+a.value*b.value
        a:=get(Ain) b:=get(Bin)
    else
        if  a.rank < b.rank
            a:=get(Ain)
        else
            if a.rank > b.rank
                b:=get(Bin)
fin cycle
```

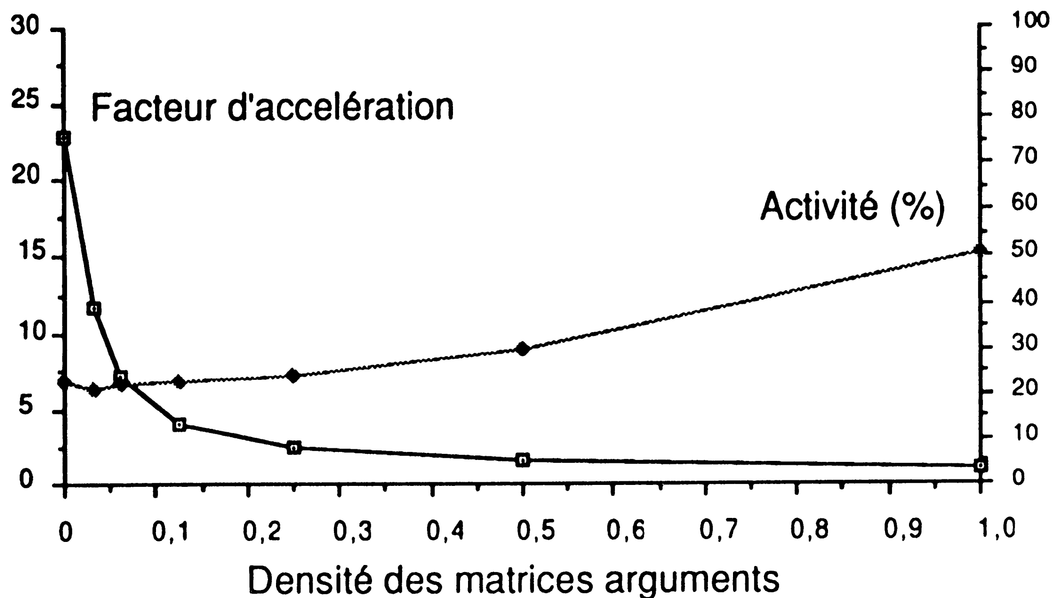
Ou get(Ain) effectue une lecture de l'entrée A bloquante si A est vide; put(Aout,x) envoie la valeur x sur sa sortie A. Chaque opérande comporte

deux champs : a.value contient la valeur de a proprement dite, a.rank indique la position de a dans la matrice opérande.

Il faut ici faire remarquer que les communications asynchrones permettent d'accroître le parallélisme dans cet exemple : comme la multiplication est beaucoup plus longue que la propagation des valeurs, plusieurs noeuds peuvent travailler simultanément sur les mêmes valeurs. Ainsi pour trois éléments de la matrice M, d'abscice 1 et d'ordonnées 1, 2 et 3 on à le chronogramme suivant :



P. Rubini a simulé l'exécution de cet algorithme avec différentes densités de matrices (générées aléatoirement). Les éléments des matrices sont codés avec le format en virgule flottante IEEE simple précision.



On voit dans cet exemple comment l'asynchronisme entre les différents processus peut être utilisé pour augmenter les performances. Il faut aussi remarquer que si l'on veut tirer pleinement partie du parallélisme de cet exemple il faut disposer d'un grand nombre de processeurs, mais que la tâche demandée à chacun est simple. On comprend alors que cet exemple soit mal adapté au parallélisme SIMD de la Connection Machine mais aussi que les tâches élémentaires sont trop nombreuses et de trop petite taille pour le Meganode.

2.6.3 Un modèle d'exécution mal exploité : les graphes dataflow

Dans la plupart des systèmes de calcul, l'exécution d'un programme est réalisée sequentiellement en suivant l'ordre d'écriture des instructions du programme.

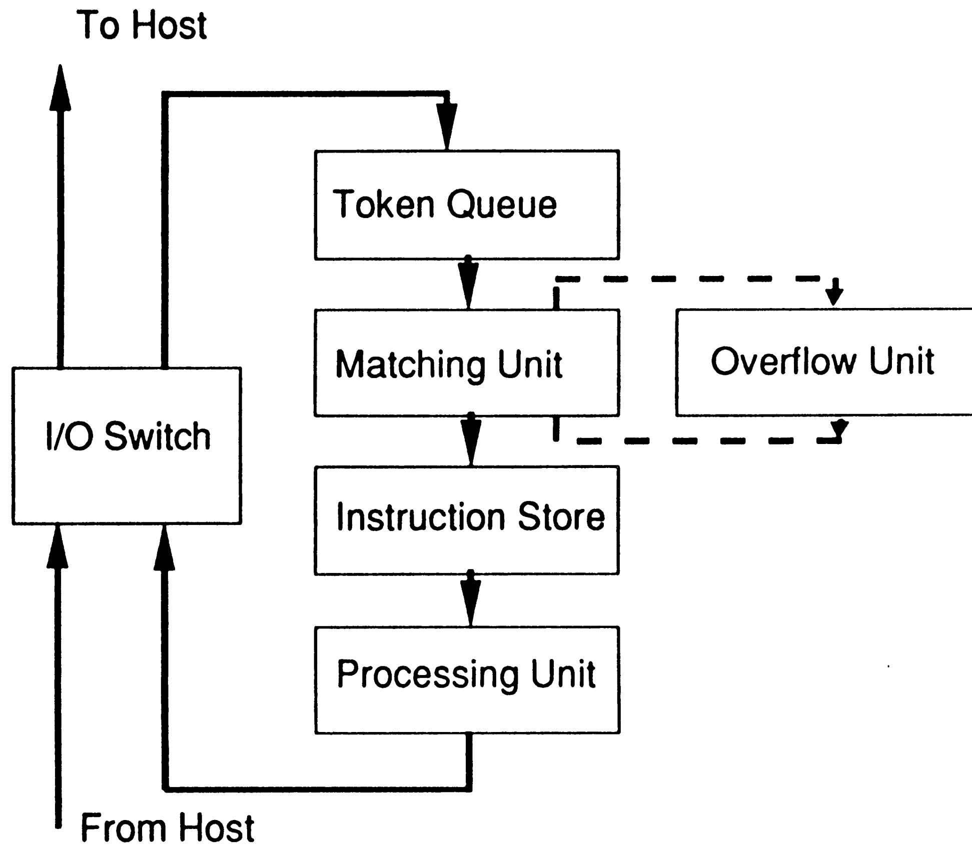
L'approche utilisée par les machines dataflow est différente, elle consiste à exploiter le parallélisme inhérent à chaque algorithme, sans demander au programmeur de l'indiquer explicitement. Un graphe dataflow représente explicitement les dépendances entre les données d'un programme. Dans un tel graphe il est donc facile d'identifier les opérations qui peuvent être faites en parallèle.

Les graphes dataflow sont donc des graphes non réguliers, constitués de petites tâches indépendantes les unes des autres.

Placer un graphe dataflow sur la Connection Machine est impossible (sauf pour des graphes très réguliers) car dans cette machine de type SIMD tous les processeurs effectuent la même action au même instant alors que les noeuds du graphe sont de nature différente.

Il est plus vraisemblable de distribuer le graphe sur une machine MIMD telle le Meganode. Le nombre de noeuds à placer est alors nettement supérieur au nombre de processeurs et l'on est obligé de procéder à de nombreux regroupements. L'étude d'exemples concret (3.4.4) nous à montré que le niveau de parallélisme que l'on peut extraire d'un tel graphe est largement plus massif que les quelques dizaines de Transputers dont on peut disposer; il correspond à un regroupement de seulement quelques noeuds par processeurs. On peut de plus remarquer que même si l'on dispose d'un nombre de Transputers suffisant pour ne procéder qu'à peu de regroupement, la tâche qui leur sera alors allouée (quelques opérations simples du type addition ou sélection) est ridiculement petite en regard de la puissance et du coût d'un Transputer et de sa mémoire.

Ce modèle d'exécution original a donné lieu au développement d'un certain nombre d'architectures dédiées telles la machine de Manchester [GUR85] ou celle du MIT[DEN80]. Ces machines utilisent ce schéma d'exécution des programmes bien adapté à un parallélisme massif. Et pourtant elles font appel à un nombre relativement restreint de processeurs. Leur organisation autour d'un bus en anneau qui constitue un goulot d'étranglement rendrait inefficaces un plus grand nombre de processeurs. On ne peut donc pas les qualifier de massivement parallèle.



Un exemple de machine dataflow traditionnelle : la machine de Manchester

Ce faible nombre de processeurs ne permet pas de tirer complètement partie du parallélisme intrinsèque de tels graphes. Pire encore, pour limiter les calculs inutiles (par exemple ne calculer que la branche utile d'un test) qui engorgeraient leurs processeurs, ces machines sont obligées d'utiliser des schémas d'exécution différents tel le demand driven.

3. Etude d'une architecture cellulaire intégrée programmable:

3.1 Objectifs

Nous venons de voir que les machines parallèles actuelles ne couvrent pas l'ensemble des algorithmes que l'on souhaiterait pouvoir exécuter de manière efficace en parallèle. Il est donc logique de tenter de définir une architecture venant combler ces lacunes. Le cahier des charges que nous nous sommes défini est donc :

- Un réseau constitué d'un grand nombre de cellules : pour la majorité des applications visées, plusieurs milliers semblent être une valeur minimale. Pour réaliser une telle machine tout en restant dans les limites de l'acceptable (pour l'encombrement comme pour la consommation électrique) l'intégration devra être poussée : plusieurs cellules sur un seul circuit. Pour permettre cette intégration, la taille de chaque cellule doit être la plus réduite possible, la topologie du réseau ne doit pas compliquer leur intégration. De plus, comme la technologie actuelle ne permet pas l'implantation d'un réseau complet sur un seul circuit, ceux-ci doivent être cascadables.

- Disposer de communications non locales entre les cellules du réseau, ce qui permet l'exécution d'algorithmes non réguliers ou de topologies différentes de celle de notre machine. En conséquence les communications ne doivent pas être limitées aux voisins immédiats de chaque cellule, mais au contraire permettre, à partir d'une cellule donnée, d'accéder au plus grand nombre possible d'éléments du réseau.

- Le traitement effectué par une cellule doit être totalement programmable et indépendant de celui des autres processeurs (l'ensemble du réseau est donc une architecture de type MIMD).

3.2 Les topologies possibles

Un rapide calcul montre qu'il est impossible d'envisager une machine massivement parallèle dans laquelle chaque processeur serait connecté à tous les autres processeurs de la machine (pour n processeur il faut factorielle(n-1) connections, soit par exemple pour 10 éléments 3628800 connections). Le choix de la topologie fixe le nombre de voisins d'une cellule, mais quel que soit ce choix, ce nombre est limité. Pour obtenir une architecture généraliste, il est nécessaire de disposer d'un mode de communication permettant de s'affranchir de cette limite. Lui seule permet de placer sur le réseau des graphes non réguliers ou de topologie différente de celle de la machine.

L'intégration d'un grand nombre de cellules sur un seul circuit est l'un des objectifs fixés. Un grand nombre de topologies sont possibles pour une telle machine parallèle:

- L'hypercube est la topologie utilisée par la Connection Machine. Il présente l'avantage d'une faible distance entre les deux éléments les plus éloignés (log du nombre total de processeurs). Il est facile de placer sur une telle topologie des anneaux, des grilles ... Les problèmes liés à son implémentations sur le silicium sont nombreux :

- degré d'entrées sorties dépendant de la taille du réseau (et atteignant 16 pour 65536 processeurs).

- difficile à placer sur le silicium qui ne comprend que quelque niveaux d'interconnexions dessinés sur le plan (surtout si l'on veut pouvoir cascader les circuits).

- Les cubes, plus proches de l'univers réel (à 3 dimensions) peuvent séduirent. La plus longue distance entre 2 éléments est de $\sqrt[3]{\text{nombre de processeurs}}$. Superposer des circuits pour former un cube nécessite des boîtiers non standards [GER90] et pose de gros problèmes

de refroidissement et d'accessibilité aux circuits situés au centre du cube.

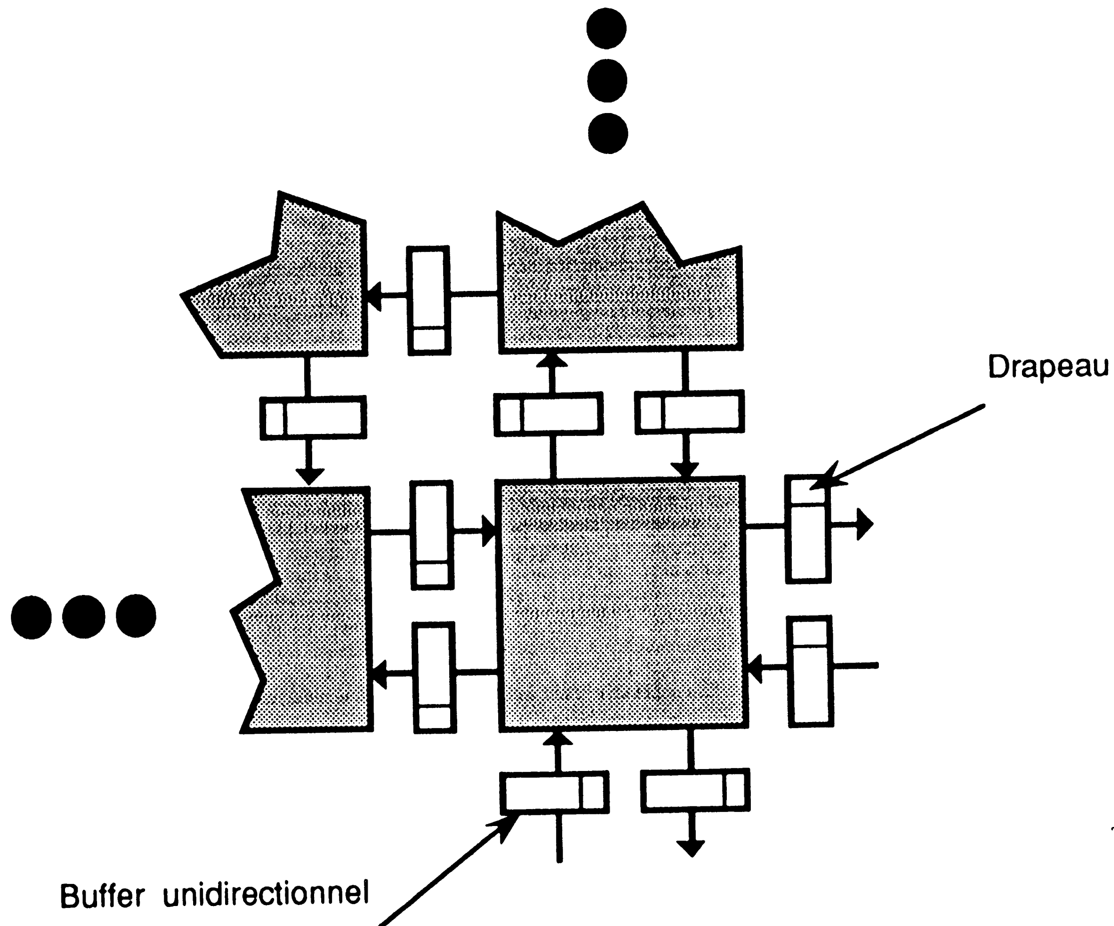
- La grille à été choisie car elle permet une implantation aisé et compacte du réseau sur le silicium. La plus longue distance entre deux processeurs est de $\sqrt[2]{\text{nombre de processeurs}}$, ce qui est légèrement moins bon que le cube mais tout de même bien meilleur que celle offerte par un anneau. Le périmètre d'une cellule carrée est suffisant pour permettre une communication des données en parallèle entre deux cellules voisines. La cascabilité des circuits est facile à assurer avec une telle technologie.

D'un strict point de vue théorique l'hypercube apparaît donc comme la topologie la plus performante. Mais des études [DAL86] prenant en compte les problèmes technologiques liés à l'implantation de ces structures sur les VLSI, contredisent l'approche mathématique et établissent que les grilles (à deux ou trois dimensions) donnent de meilleurs résultats après intégration dans un circuit.

3.3 Réalisation des communications

3.3.1 Le mécanisme d'acheminement des messages

La topologie retenue est une grille à deux dimensions, dans laquelle chaque cellule peut communiquer de manière asynchrone avec ses 4 voisines par l'intermédiaire de buffers de communication. Chaque buffer est unidirectionnel, pour permettre des communications bidirectionnelles entre cellules, il est nécessaire d'en utiliser deux. Un drapeau associé à chaque buffer permet aux deux cellules de connaître leur état : vide ou plein.



Les communications entre deux cellules, directement connectées ou non, se font par l'intermédiaire de messages qui transitent de cellules en cellules jusqu'à destination.

De nombreuses stratégies d'acheminement de messages sont envisageables parmi lesquelles :

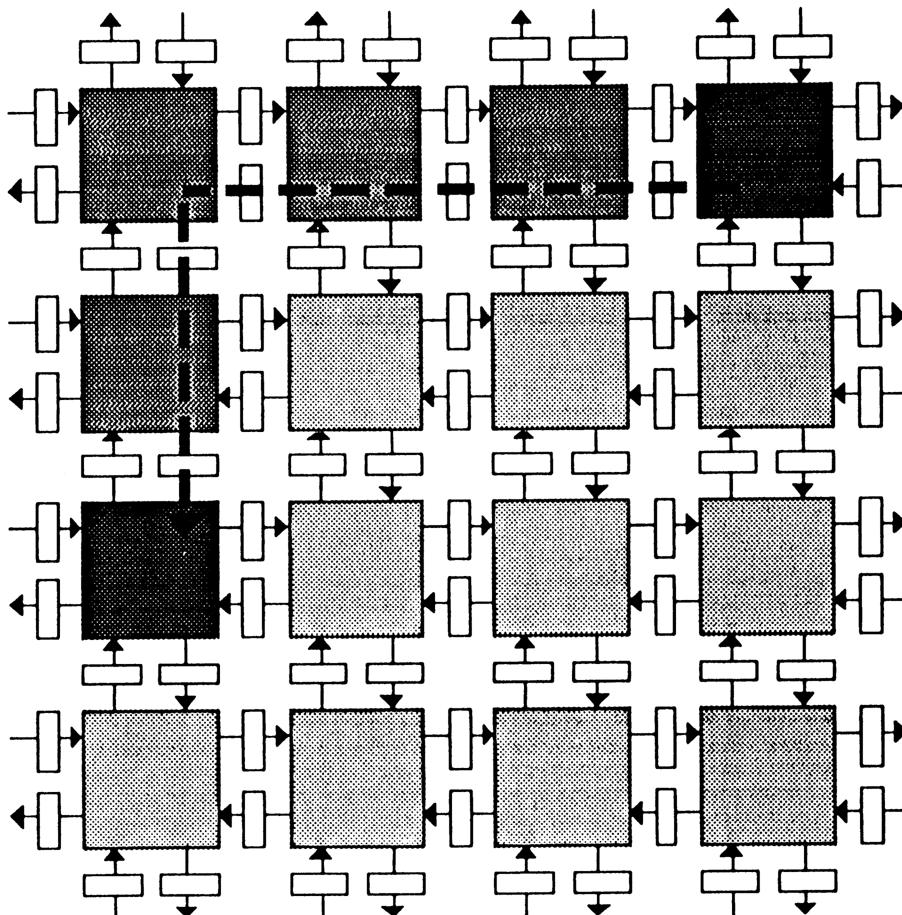
- Acheminement en abscisse, puis en ordonnée, avec une attente en cas de contention de messages. Cette stratégie est utilisée par notre équipe depuis le début de nos études sur les réseaux cellulaires [BER85]. Elle présente une grande simplicité de mise en oeuvre et de nombreuses simulations ont montré que ces performances étaient largement suffisantes [KAR90c].

- On peut vouloir faire contourner aux messages les zones encombrées, par un acheminement en abscisse puis en ordonnée avec un changement de direction en cas de contention de messages [GER90].

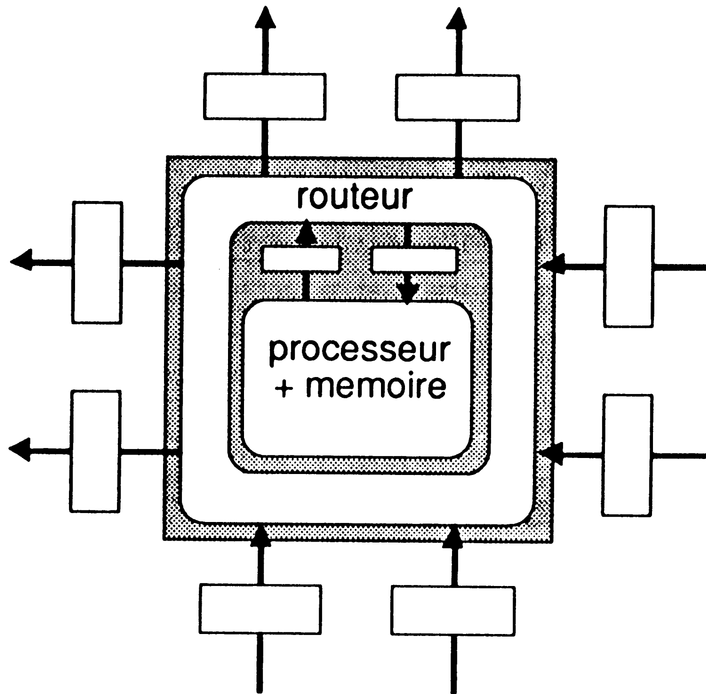
Cette méthode est légèrement plus complexe à implanter que la précédente; de plus le caractère non déterministe de ce mode de routage pose des problèmes supplémentaire de programmation (les messages peuvent arriver dans un ordre différent de celui d'émission).

- Un acheminement avec contournement de cellules défectueuses est intéressant dans le cadre de l'utilisation d'une technologie WSI (Wafer Scale Integration); elle a fait l'objet d'une étude précédente de l'équipe [FAU86].

Pour la machine notre réseau nous avons choisi la stratégie la plus simple : l'acheminement en X puis en Y.

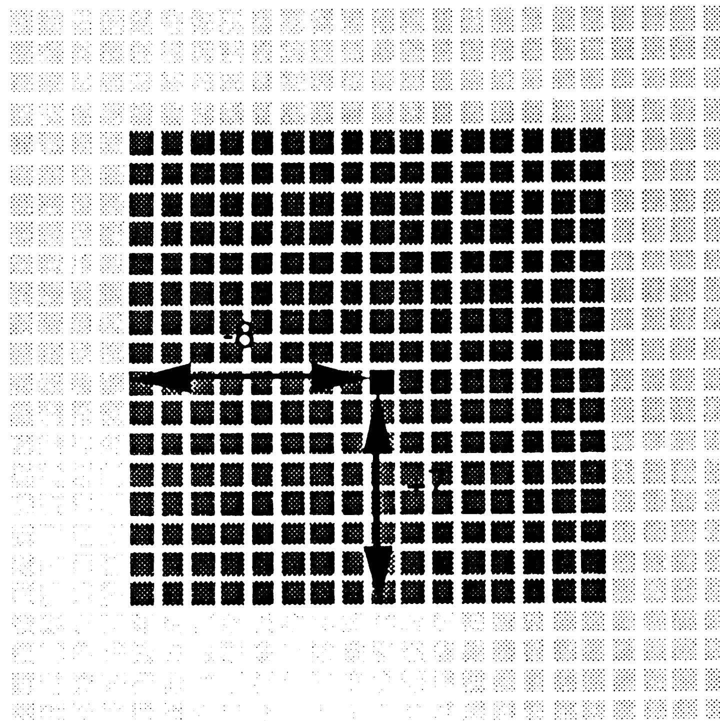


Afin que le routage de messages ne ralentisse pas trop le calcul en cours dans les cellules, un mécanisme câblé indépendant prend en charge l'acheminement des messages au niveau de chaque cellule : l'aiguilleur. La communication entre l'aiguilleur et la partie traitement de la cellule se fait par l'intermédiaire de buffers de même type que les buffers inter-cellules.



Chaque message doit donc comporter en plus du champ donnée un champ adresse spécifiant le déplacement relatif vers la cellule destination. Cette adresse (dx,dy) est décrémentée à chaque transfert du message au travers d'une cellule. Les champs dx et dy sont codés sur 4 bits ce qui donne une capacité d'adressage de -8 à +7 cellules.

Ensemble des cellules adressables :



3.3.2 Les communications inter-cellules

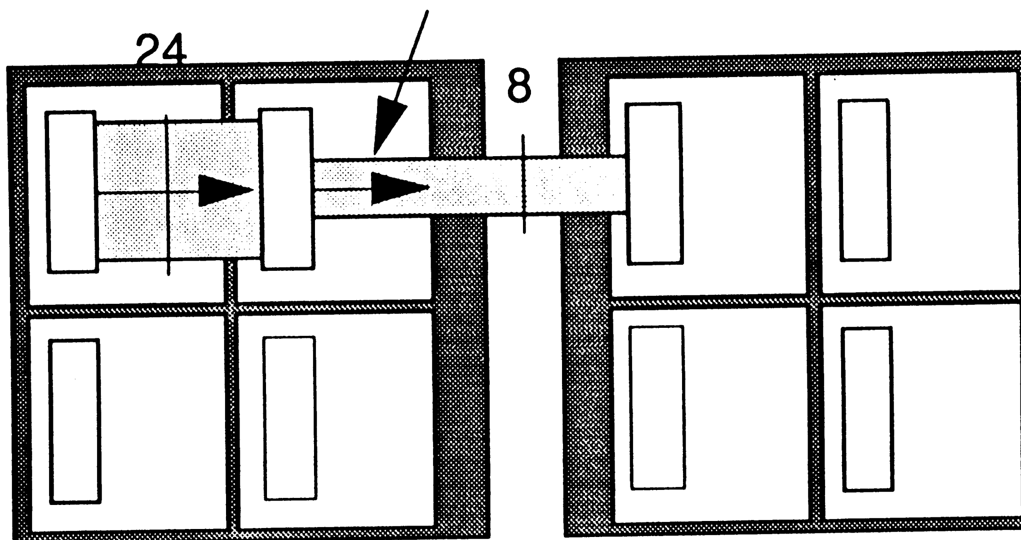
Parallèlement au choix du chemin que les messages doivent emprunter, il existe aussi de nombreuses possibilités quant à la méthode de transmission des messages entre deux cellules. Pour des raisons technologiques, le nombre de plots reliant deux cellules situées dans deux boîtiers différents est beaucoup plus faible que le nombre de connexions possibles entre deux cellules situées dans le même circuit. Des méthodes de transmissions mixtes combinant deux principes différents à l'intérieur d'un circuit et entre deux circuits sont donc à envisager.

- Parallèle : Cette méthode permet une grande vitesse de transmission. Le nombre de signaux à transmettre par circuit est égal à 8 fois le nombre de bits des messages : 24 (à multiplier par n dans le cas de circuit N*N cellules), on comprend qu'elle ne soit pas utilisable entre deux circuits différents. De plus, elle implique de stocker dans chaque cellule le message complet. Cette méthode combinée à une liaison série entre les circuits à été utilisée dans les précédents travaux de l'équipe [OBJ88].

- Série : Permet de réduire le nombre de connexions entre circuits mais au prix d'une baisse de performances. Comme pour la précédente, il est nécessaire de stocker dans chaque cellule le message complet.

- Mixte : On peut combiner ces deux méthodes de façons multiples pour obtenir toutes sortes de combinaisons : passage en parallèle et en 3 fois de 3 mots de 8 bits (pour une longueur de message de 24), passage en 12 mots de 2 bits .... De plus il est clair que la méthode peut ne pas être la même entre cellules du même circuit et entre circuits.

Un exemple de transmission en parallèle avec sérialisation partielle aux frontières du circuit :
les octets du message sont transmis en 3 cycles

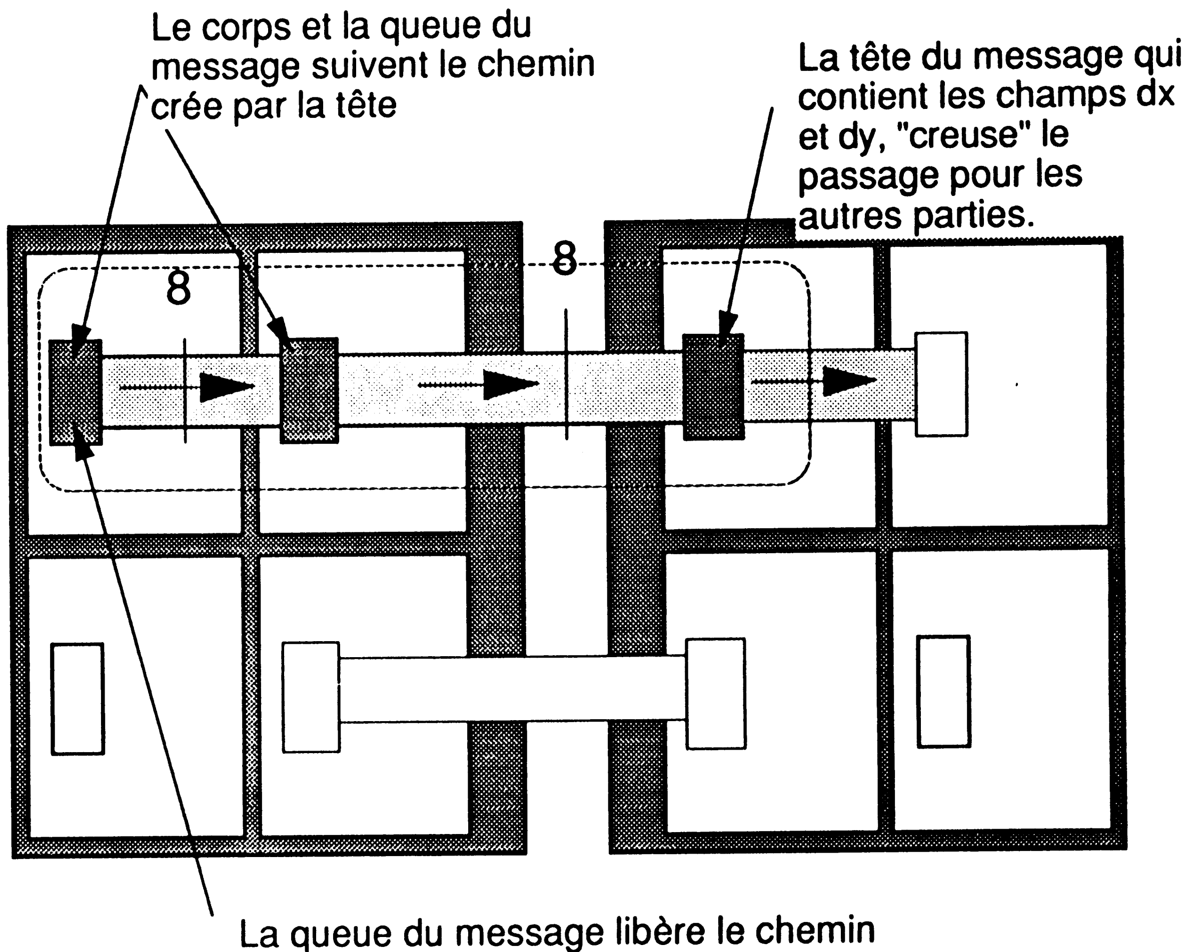


Ces méthodes peuvent être mises en oeuvre selon la technique "Store and Forward", c'est à dire en rangeant à chaque pas la totalité du message dans chaque cellule traversée; mais il est plus intéressant d'utiliser la technique du "whormehole routing" : le message est stocké sur n cellules consécutives. La première partie (la tête) contient le déplacement vers la cellule destination, elle "creuse" un passage qu'empruntent successivement les autres parties du message.

Le travail de simulation de P. Rubini [KAR90c][FAU91] nous a permis de fixer de façon précise les performances respectives de toutes ces méthodes. Notre choix s'est alors porté sur l'utilisation de la technique du

whormehole routing avec des messages coupés en trois tronçons de huit bits.

les messages sont dans 3 tampons successifs



3.3.3 Les contraintes sur le routage

Choisir la méthode de routage n'est pas tout, il faut aussi s'assurer que le routage ne provoquera ni famine ni interblocage. Si les problèmes de famine sont faciles à éviter par un mécanisme de priorité tournante, il n'est pas aussi simple de bien comprendre les mécanismes nécessaires pour éviter les interblocages. Une longue pratique des architectures massivement parallèles nous permet d'établir la règle suivante : si l'on veut éviter le blocage du réseau, il faut s'assurer que le retrait d'un message du réseau par la cellule destination aura lieu au bout d'un temps

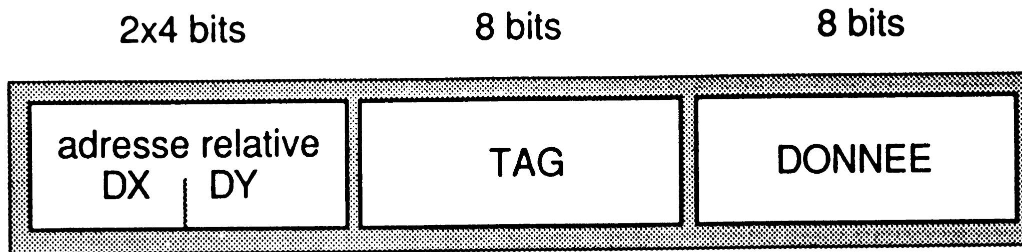
fini, quelque soit l'algorithme déroulé par la partie traitement. Cette condition a une conséquence évidente : le retrait des messages du réseau doit être assuré par un processus différent de celui effectuant le calcul. Pour assumer cette tâche plusieurs solutions sont envisageables :

- Disposer d'une partie traitement de la cellule multi-processus consiste à implanter des mécanismes matériels (sauvegarde de contexte, réveil de processus inactif lors de l'arrivée d'un message ...) facilitant la gestion de processus. On pourrait alors programmer en fonction des algorithmes le traitement à effectuer sur les messages entrants. De plus, de nombreux algorithmes tireraient partie de la possibilité de décrire plusieurs processus sur une même cellule. Malheureusement le coût hardware d'un mécanisme permettant une gestion complète des processus ne nous a pas permis d'opter pour cette solution.

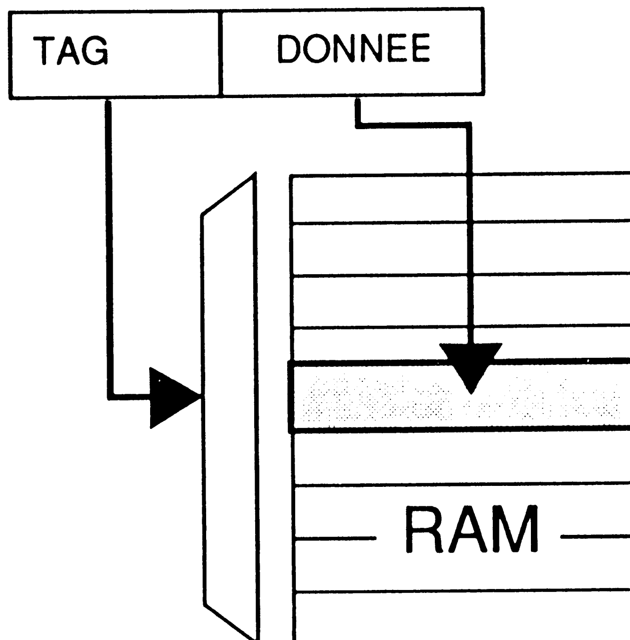
- Implanter comme un mécanisme d'interruption le lancement de programmes traitant les messages entrants. Cette méthode permet de garder une certaine souplesse sur le traitement à effectuer sur les messages, car le programme exécuté par l'interruption consécutive à leur l'arrivée est en mémoire, et peut donc dépendre de l'algorithme. Elle nécessité tout de même de disposer d'un mécanisme d'interruption avec changement de contexte.

- Utiliser la partie routage (qui constitue déjà un deuxième processus, indépendant dans la cellule) pour ranger les messages entrants. Ce mécanisme fige le traitement à effectuer, mais il a l'avantage d'être extrêmement simple à implémenter.

Nous avons choisi pour notre implantation de confier à la partie routage le rangement des messages entrants en mémoire. Pour permettre de choisir l'adresse à laquelle ranger le message, un troisième champ du message est nécessaire : Le champ TAG. On a alors la structure de messages suivante :



Le champ TAG est donc utilisé par la partie routage comme adresse pour ranger le message dans la mémoire de la partie traitement :



3.4 Un traitement MIMD

Nous avons vue que seul le traitement MIMD (Multiple Instruction, Multiple Data) permet d'apporter une réponse à la classe de problèmes que nous nous proposons de résoudre. Le traitement MIMD fait donc partie du cahier des charges initiales de notre machine.

Le choix du jeu d'instructions est une tâche délicate; les contraintes à respecter sont nombreuses, on peut citer :

- Le jeu d'instruction doit permettre la programmation de l'ensemble des applications visées. Cela peut paraître évident mais le

simple oubli d'un mode d'adressage ou d'une instruction peut compliquer voire rendre impossible l'écriture d'un programme.

- Il doit respecter un certain équilibre entre la taille du séquenceur nécessaire à son exécution et le volume de mémoire occupé par le programme. Un jeu d'instruction simple, de type RISC, donnera un séquenceur de petite taille, mais les programmes écrits avec lui nécessiteront plus d'instructions et donc de mémoire que s'ils étaient écrits avec un jeu d'instructions type CISC.

Ces problèmes ont fait l'objet d'une étude complète menée par M. Karabernou et P. Rubini qui sera présentée dans leur thèse. La méthodologie de travail à consisté à s'appuyer sur un outil de simulation et une large palette d'applications. La simulation des différents jeux d'instructions a permis de parvenir au compromis actuel :

3.4.1 Le Jeu d'instruction

Dans la version actuelle du réseau, chaque cellule comporte une mémoire locale de 256 octets (il n'y a pas de mémoire globale) utilisée pour le stockage du programme à exécuter sur la cellule et des données. Cet espace adressable volontairement limité permet de coder les adresses sur 8 bits et donc d'utiliser un codage très compact pour les programmes. Nous sommes partis du jeu d'instruction que l'on rencontre sur les microprocesseurs 8 bits à accumulateur classiques tels le 6502 ou le 6800.

L'espace mémoire limité, permet de simplifier les modes d'adressages, nous avons retenu les modes suivants :

Absolu :

la donnée se trouve à l'adresse rangée dans le mot suivant le code opération.

Absolu court :

la donnée se trouve à l'adresse rangée dans le demi-octet de poids faible de l'instruction. Cette adresse a un format court (4 bits) et fait donc implicitement référence à une certaine page de la mémoire. Il s'agit de la page $Ø pour LDA et STA (page des variables de travail) et de la page $F pour PUT, GET et TRY (page des variables de communication).

Immédiat :

la donnée se trouve après le code opération.

Indirect :

la donnée se trouve à l'adresse contenue dans le registre d'indirection I.

Indirect postincrémenté :

la donnée se trouve à l'adresse contenue dans le registre d'indirection I. Ce registre est incrémenté du nombre de mots utilisés (1 sauf pour SEND et les instruction 16 bits : respectivement 3 et 2) après l'indirection.

Le jeu d'instruction comporte les instructions suivantes :

Instruction de manipulation de l'accumulateur :

LDA : Chargement de l'accumulateur.

STA : Stockage de l'accumulateur.

Opérations arithmétique 8 Bits :

ADD : Addition.

SUB : Soustraction.

ADC : Addition avec retenue.

SBC : Soustraction avec emprunt.

CMP : Comparaison.

INC : Incrémentation.

DEC : Décrémentation.

TST : Comparaison à 0.

NEG : Complément à 2.

NGC : Complément à 2 en multiprécision.

CLR : Remise à 0.

MUL : Multiplication 8 bits fois 8 bits résultat sur 16 bits

Opérations logiques :

XOR : Ou exclusif bit à bit.

OR : Ou logique bit à bit.

AND : Et logique bit à bit.

NOT : Complément à 1.

Décalages :

ROL : Rotation à gauche avec la retenue.

ROR : Rotation à droite avec la retenue.

ASR : Décalage arithmétique à droite.

ASL : Décalage arithmétique à gauche.

Arithmétique 16 bits :

LDAW : Chargement de l'accumulateur étendu.

STAW : Stockage de l'accumulateur étendu.

ADCW : Addition 16 bits avec retenue.

SBCW : soustraction 16 bits avec emprunt.

Divers :

Les instructions de contrôle ont été réduites au minimum, ne conservant que les branchements conditionnels. Cela ne pose pas de problème particulier car les programmes ont souvent une structure très simple.

Bcc : Branchements conditionnels.
(Même condition que sur le 68000)

LDI : Chargement du registre d'index I.

CLC : Remise à 0 de la retenue.

SEC : Mise à 1 de la retenue.

TAPC : Transfert de A dans le PC.

TPCA : Transfert du PC dans A.

TIA : Transfert de I dans A.

TAI : Transfert de A dans I.

Communication :

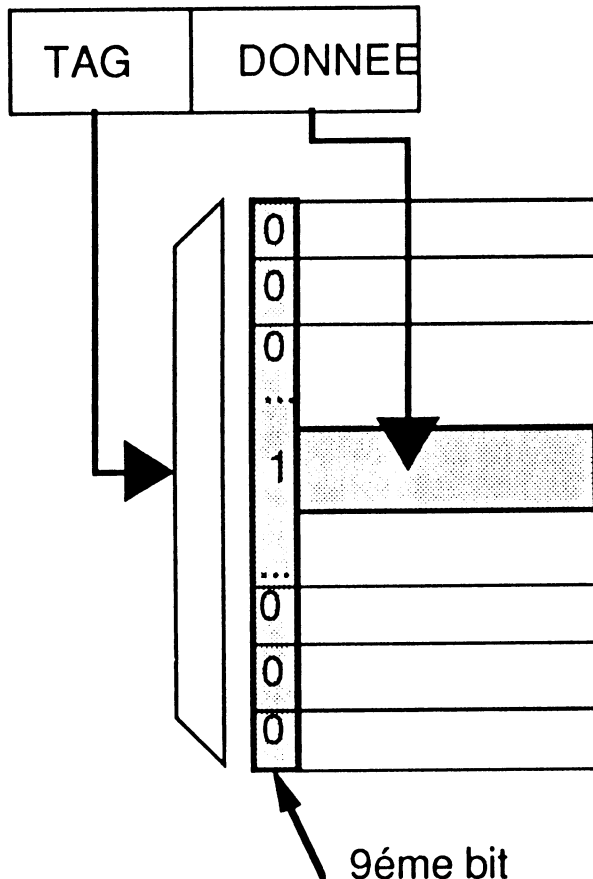
Les instructions de communication constituent une part importante et originale de notre jeu d'instruction.

Comme nous l'avons vu, si l'on veut éviter tout interblocage du réseau, il faut s'assurer que le retrait d'un message du réseau ne dépend pas du traitement en cours dans la cellule destination: les messages doivent être pris dans l'ordre où ils arrivent, et le processeur ne doit pas avoir à attendre de pouvoir émettre un message avant de retirer le message incident. Nous avons choisi une méthode qui peut se voir d'un point de vue fonctionnel de la façon suivante :

- Tout message entrant dans sa cellule destination est déposé par le mécanisme de routage dans la mémoire de la partie traitement de la cellule.

- Le champ TAG du message sert d'adresse au routage pour effectuer ce rangement en mémoire.

- Chaque adresse de la mémoire dispose d'un $9^{ème}$ bit qui est mis à 1 quand le routage dépose un message dans la mémoire, et remis à 0 quand ce message est lu.



Ce bit supplémentaire est utilisé comme bit de synchronisation par les instructions de communication

Les instructions de communication retenu sont :

SEND : Envoi du message présent en mémoire à partir de l'adresse donnée en argument : <data ; tag ; adresse relative(dx,dy)>. Le processeur est bloqué tant que le buffer de sortie n'est pas libre et que le message ne peut y être recopié.

GET : Réception d'un message : attend que le flag de l'opérande soit à 1, puis lit le contenu, le range dans l'accumulateur, et remet le flag à zéro.

PUT : Envoi d'un message local au processeur : écrit le contenu de l'accumulateur à la position mémoire donnée en argument, et met le flag correspondant à 1. Comme pour les messages incident, aucune vérification n'est effectuée, d'autant moins que si le processeur devait être bloqué sur une condition de ce type, rien ne permettrait de le débloquer (il

ne pourrait pas faire de GET, puisque bloqué...)

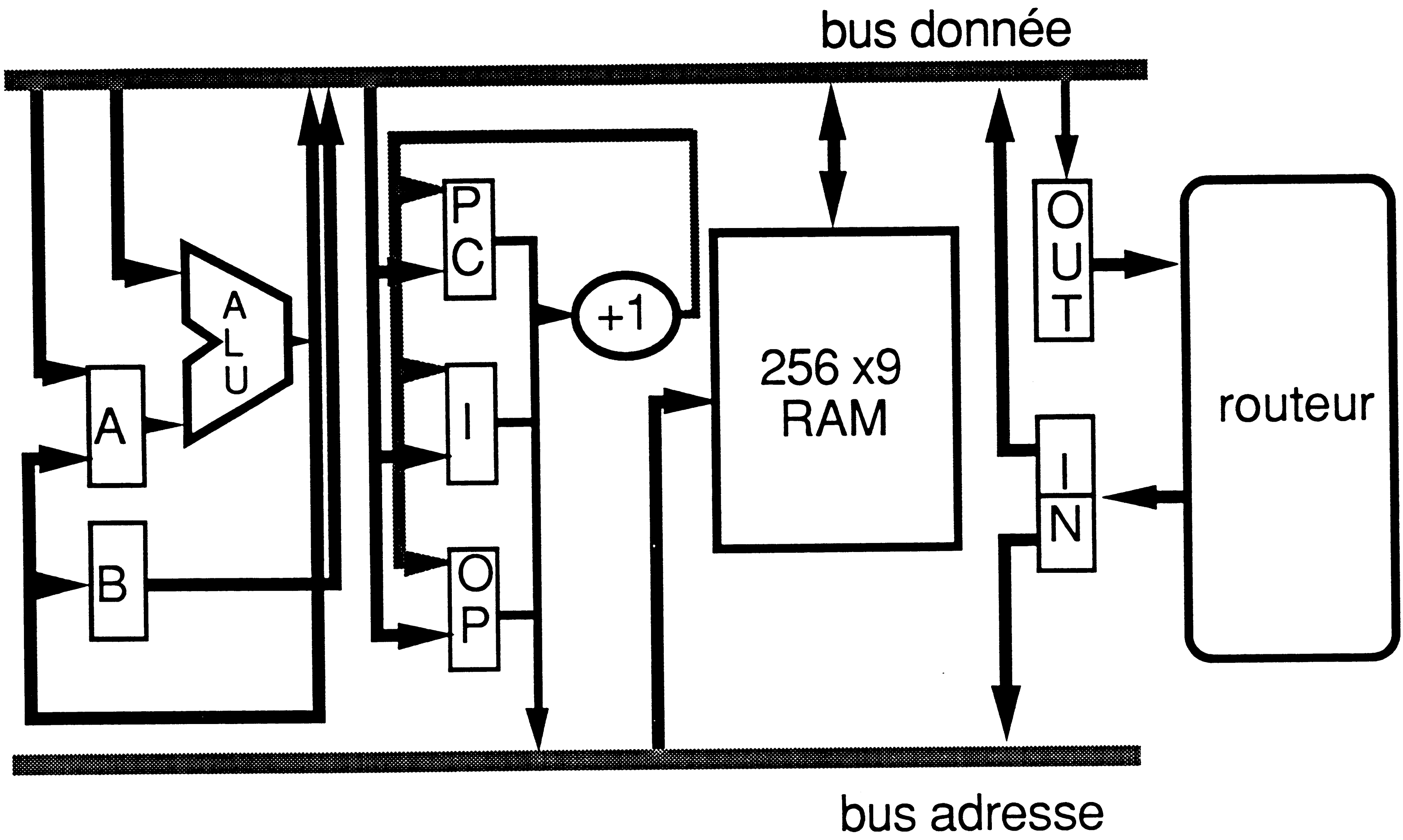
TRY : Test de présence d'un message : teste le flag de la position mémoire donnée en argument ; si le flag est nul, se déroute à l'adresse contenue dans le vecteur d'échec (Mem[0]), sinon continue en séquence.

Il faut noter qu'il n'y a pas de gestion d'interruption dans la cellule, la communication se faisant grâce à des instructions de plus haut niveau.

3.4.2 Le chemin de donnée

Le chemin de donnée doit permettre l'exécution du jeu d'instruction d'une manière efficace; mais il doit aussi répondre à des impératifs de taille très sévères.

L'étude menée par M. Karabernou a montré que le chemin de donnée suivant était adapté :



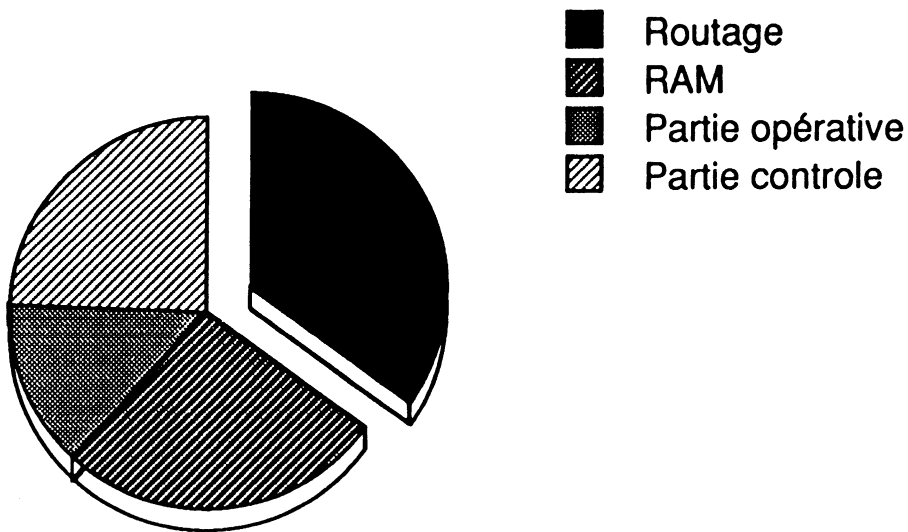
bus donnée
ALU
A
B
PC
I
OP
+1
256 x9
RAM
OUT
IN
routeur
bus adresse

Conformément aux objectifs initiaux, la cellule de base de la machine du réseau est de petite taille : approximativement 20000 transistors. Leur répartition entre les différents blocs fonctionnels est la suivante :

	Nombre de transistors	Surface estimé (mm^2)
ROUTAGE	5000	5
TRAITEMENT :		
RAM	13000	3,8
Partie opérative	2000	2
Partie contrôle	Impossible à obtenir avec les outils utilisés (VTI)	3,5

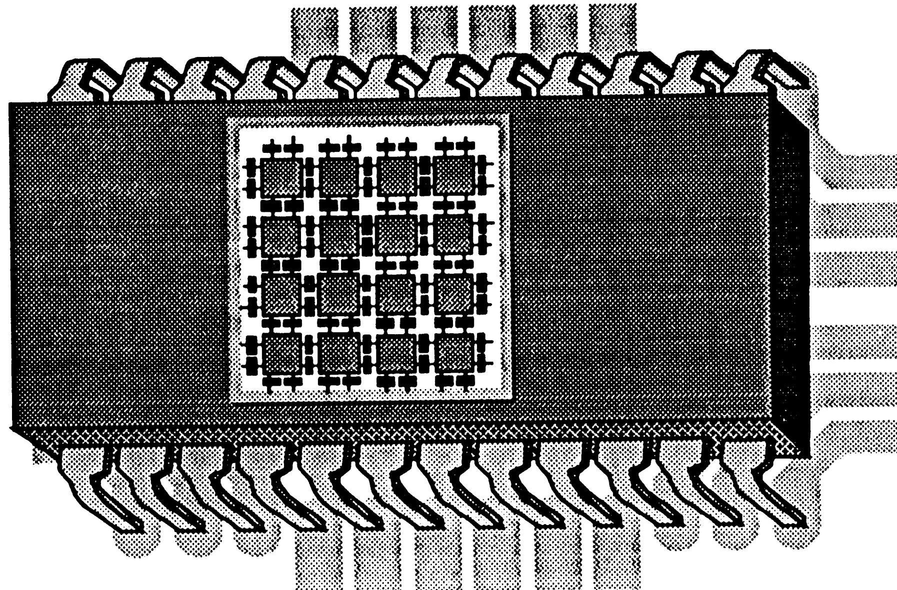
Ces chiffres ont été obtenus en utilisant des évaluations qui se basent sur une bibliothèque de cellules déssinées avec la technologie CMP 1,5 micron.

La proportion de surface utilisée par chacun des blocs fonctionnels est la suivante :



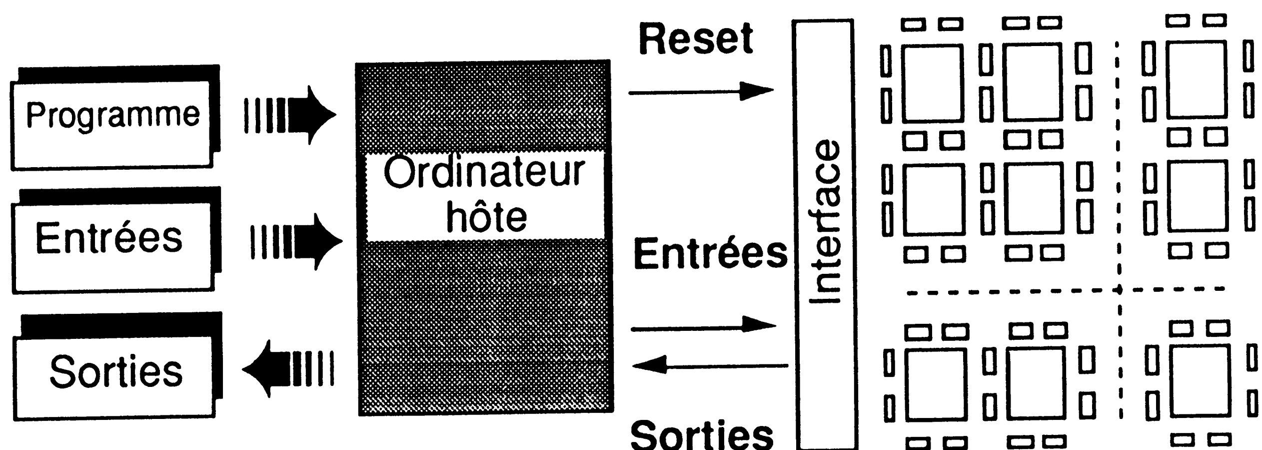
On voit ainsi que l'objectif initial consistant à équilibrer la surface du circuit utilisée par les différentes fonctions est respecté. Il est remarquable de noter que la partie contrôle et la RAM occupent approximativement la même surface.

Par ailleurs, on voit qu'en utilisant pour la réalisation du circuit des technologies industrielles plus fines et autorisant des chips de grande taille, il est d'ores et déjà possible d'intégrer un grand nombre de cellules dans un seul circuit ; dans peu de temps, les 1,3 Millions de transistors que necessitent 8*8 cellules pourront être intégrés dans un seul chip.



3.5 Connexion à la machine hôte

Pour fonctionner notre architecture doit être reliée à un ordinateur hôte qui lui fournit le programme et les données nécessaires au calcul, le réseau lui rendant les résultats.



3.5.1 Les entrées sorties

Les entrées sorties du réseau se font par l'intermédiaire des cellules périphériques du réseau. Le nombre de cellules connectées à l'extérieur est ainsi au maximum de 4 fois la largeur du réseau alors que le nombre total de cellules croît en fonction du carré de la largeur du réseau. On pourrait alors craindre un engorgement des communications de la périphérie du réseau pour la gestion des entrées sorties. Comme nous allons le voir il n'en est rien, le goulot d'étranglement est constitué par le bus de l'ordinateur hôte dont les performances ne varient pas en fonction de la taille du réseau.

La gestion des entrées sorties se fait grâce à l'ordinateur hôte. La fréquence de communication entre le réseau et son hôte est égale à la fréquence du plus lent parmi :

- Le bus de l'ordinateur hôte. Sur une machine de type compatible PC/AT cette fréquence est de : 6 Mhz. Un cycle d'accès au bus nécessite 3 cycles d'horloge soit 500ns

- Le temps de routage d'un message divisé par le nombre de cellules connectées à l'extérieur.

Le temps de routage d'un message étant de 300 ns, il suffit donc de connecter 1 cellule vers l'extérieur pour dépasser en performance la vitesse de communication du bus de l'hôte.

Le réseau étant asynchrone, il est possible pour des exemples où les entrées sorties sont un problème important, de le connecter à plusieurs machines hôte en affectant à chacune une partie des communications.

3.5.2 L'initialisation du réseau

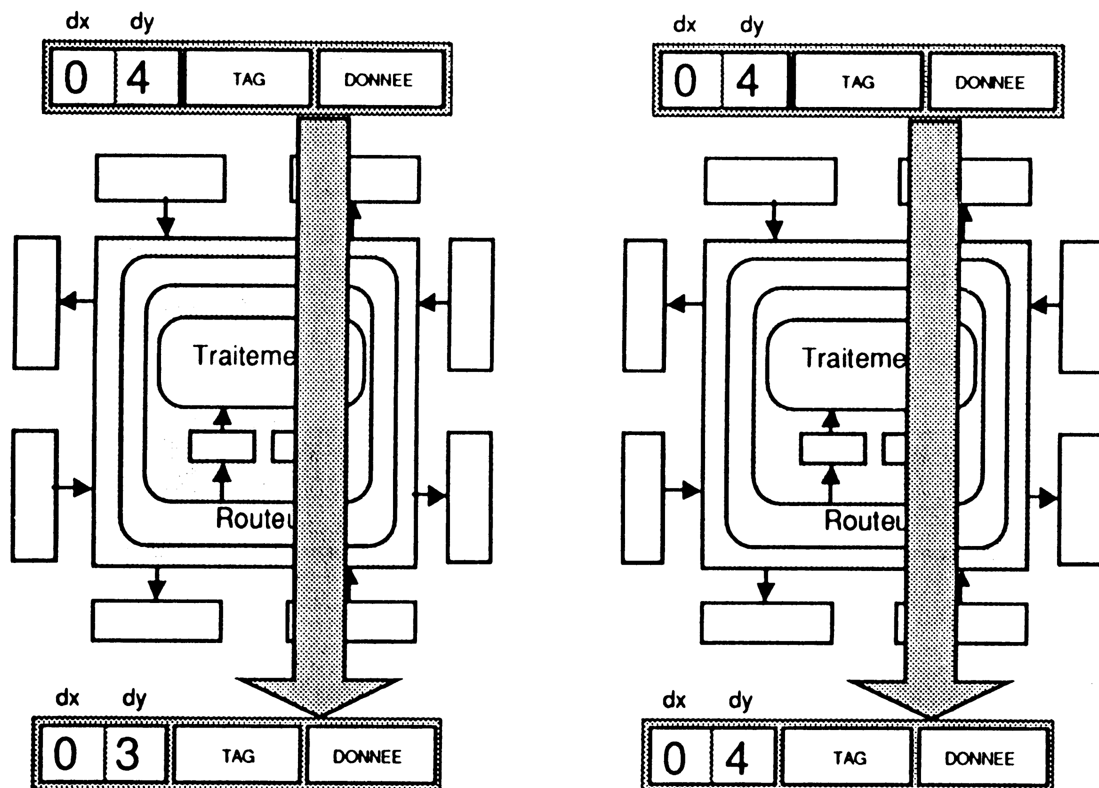
L'ordinateur hôte se voit attribuer la tâche de gérer la programmation du réseau. C'est dans l'ordinateur hôte que sont assemblés ou compilés les programmes (assembleur ou LUSTRE). Notre équipe a déjà étudié pour des projets précédents une interface reliant un réseau cellulaire

asynchrone à une machine traditionnelle (compatible PC).

L'accès aux cellules cachées : La transparence

Dans l'architecture telle que nous l'avons définie, les champ dx et dy des messages sont codés chacun sur 4 bits. Si l'on veut pouvoir programmer des réseaux de grande taille, il faut donc disposer d'un mécanisme permettant d'envoyer leur programme aux cellules non directement accessibles de l'extérieur.

Après avoir examiné plusieurs solutions, nous avons choisi d'utiliser une méthode que nous avons appelée : "transparence". Elle consiste à disposer sur chaque cellule d'un signal transparence généré par l'ordinateur hôte. Quand ce signal est à 1 dans une cellule, le routeur correspondant transmet toujours les messages mais il cesse de décrémenter le champ dx ou dy correspondant :

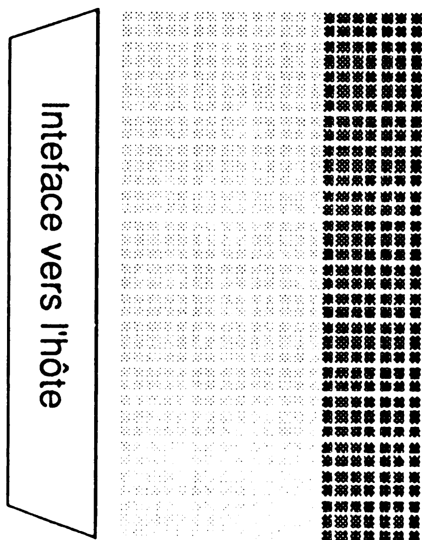


Communication d'un message sans transparence   Communication d'un message avec transparence

Pour pouvoir programmer l'ensemble du réseau, l'ordinateur hôte doit mettre en transparence les cellules situées entre lui et la zone qu'il désire atteindre. Il peut ainsi programmer par vagues successives l'ensemble des zones.

Cela est illustré par le schéma qui suit :

1ère phase : programmation des cellules les plus éloignées

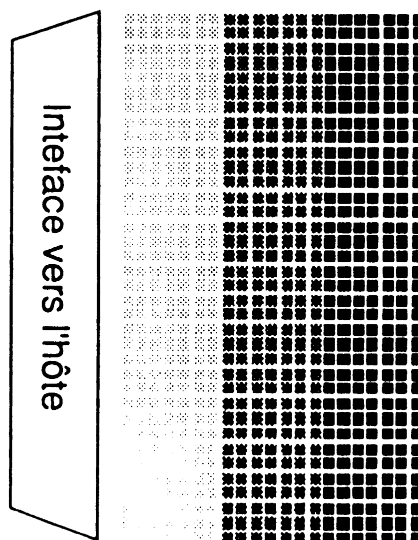


☐ Cellules en mode transparence

▩ Cellules en cours de programmation

■ Cellules déja programmées

2ème phase : programmation des cellules intérmédiaires

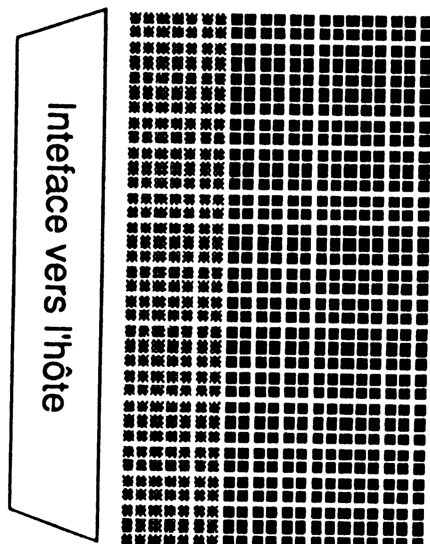


☐ Cellules en mode transparence

▩ Cellules en cours de programmation

■ Cellules déja programmées

3ème phase : fin de la
programmation du réseau



Cellules en mode transparence

Cellules en cours de programmation

Cellules déja programmées

Cette méthode très simple permet donc d'accéder à l'ensemble des cellules d'un réseau de grande taille.

Des études ont montré que le signal transparence peut de plus être utilisé pour permettre un démarrage synchrone de l'ensemble des cellules du réseau.

La taille mémoire : le temps de programmation

Comme on l'a vu, les messages contiennent un champ nommé TAG; ce champ est utilisé par le routage de la cellule destination du message pour savoir à quelle adresse ranger le champ valeur dans la mémoire de la cellule. L'espace mémoire de la cellule (programme plus données) est de 256 octets. Un champ TAG sur 8 bits permet donc d'adresser l'ensemble de la mémoire. Pour envoyer son programme à une cellule, il suffit donc de lui envoyer 256 messages en incrémentant le TAG du message à chaque fois.

La taille mémoire d'une cellule est faible : 256 octets; mais quelle est la taille mémoire de l'ensemble du réseau ?

Taille du réseau	nombre de cellules	taille mémoire de l'ensemble du réseau
10*10	100	25 Koctets
50*50	2500	625 Koctets
100*100	10K cellules	2,5 Moctets
200*200	40K cellules	10Moctets
500*500	250 K cellules	62,5Moctets
1000*1000	1M cellules	250Moctets

On voit donc que même pour des réseaux de très grande taille l'espace mémoire total est du même ordre que sur des "Mainframe" classiques.

Nous savons que la vitesse de communication entre l'ordinateur hôte et le réseau est fixée par la vitesse du bus de ce dernier, les temps de programmation du réseau peuvent donc être estimés :

Taille du réseau	Temps de programmation
10*10	256 millième de seconde
50*50	640 millième de seconde
100*100	2,5 secondes
200*200	10 secondes
500*500	64 secondes
1000*1000	256 secondes

Les temps de programmation reste donc dans les limites de l'acceptable même pour un réseau de grande taille et un ordinateur hôte modeste. Néanmoins, ce délai limite l'utilisation de notre machine à des exemples dans les quels un calcul identique est répétés un grand nombre de fois.

3.6 Performances

La définition complète de notre architecture est trop récente pour que nous puissions déjà disposer de benchmarks fiables pour comparer les performances attendues avec des machines déjà existantes.

De nombreux algorithmes ont été programmés et simulés par P. Rubini, et montrent des résultats intéressants; mais la seule étude comparative dont nous disposons est l'implantation d'un réseau neuronal: le NetTalk, sur notre réseau. L'utilisation d'un réseau cellulaire asynchrone disposant de communications non locales pour l'émulation de réseaux neuronaux, a été étudiée par B. Faure [FAU90] dans le cadre de la réalisation d'une architecture dédiée. Nous sommes partis de son modèle de neurones et nous avons pu obtenir les résultats suivants [KAR90c].

	Apprentissage MPCUS (millions of connection updates per second)	Reconnaissance MPCS (millions of connections per second)
parallélisme synaptique	21	135
parallélisme neuronal	2,6	17
VAX 780	0,05	
Convex C1	1,8	
16K Connection Machine	2,6	
64K Connection Machine	13	

Le nombre de cellules nécessaires à la simulation de cet exemple avec un parallélisme synaptique est d'environ 1 millier. Soit un plaque de 8*8 boîtiers de 4*4 cellules.

Les performances atteintes par notre architecture sont donc prometteuses et justifient pleinement la réalisation d'un circuit prototype.

3.7 Conclusion

Le réseau cellulaire programmable constitue la synthèse et l'évolution logique de plusieurs années de travail de l'équipe Circuit du Laboratoire de Génie Informatique sur les architectures massivement parallèles.

Nos choix de conception se sont toujours portés vers les solutions les moins coûteuses à implémenter. Le résultat est une cellule simple à réaliser, simple à programmer, équilibrée du point de vue de l'encombrement sur le silicium des différents blocs qui la composent. Mais comme nous le verrons cette solution minimaliste est trop simple pour pouvoir résoudre des problèmes plus complexes comme l'utilisation des langages objets.

Un premier circuit comportant 4 cellules est en cours de développement. Il devrait dans un premier temps nous permettre de réaliser un réseau prototype de quelques centaines de cellules.

Une nouvelle méthode de programmation d'une architecture massivement parallèle, et sa compilation

4. Des langages de programmation pour machines parallèles

4.1 Des techniques de compilation spécifiques ?

Avec le développement récent des architectures hautement parallèles programmables (du grand réseau de TRANSPUTERS au XILINX) le besoin de nouvelles méthodes de programmation est apparu. On retrouve sur ces architectures les problèmes classiques de la programmation de bas niveau : jeu d'instructions pauvre, complexité de la mise au point ... A ces problèmes s'ajoutent ceux spécifiques des architectures parallèles : synchronisations entre processus, interblocages...

L'ensemble de ces problèmes constitue des obstacles tels qu'il nous est apparu indispensable de disposer de langages de haut niveau pour permettre une programmation plus aisée du réseau. L'expérience des architectures traditionnelles montre que leur utilisation permet :

- Une programmation plus rapide,
- Une méconnaissance de l'architecture cible,
- La portabilité des programmes entre différentes machines.

Ceci se fait au prix de :

- Une baisse des performances des programmes,
- L'impossibilité d'utiliser pleinement les ressources de l'architecture.

Notre objectif est de mettre au point une méthode de “compilation” de langages de haut niveau sur architectures hautement parallèles offrant les mêmes types d'avantages et pas plus d'inconvénients que la compilation sur machine traditionnelle.

4.2 Les langages inspirés de CSP

Le langage CSP (pour Communicating Sequential Processes) a été développé par C. A. R. Hoare [HOA78] pour permettre l'écriture de programmes parallèles. Son dérivé OCCAM [INM84] est plus connu, il a servi de base aux développements effectués à base de Transputer.

Ce langage algorithmique permet la description de processus soit à exécuter séquentiellement : primitive SEQ, soit à exécuter en parallèle : primitive PAR. La communication entre processus se fait par des canaux grâce à un mécanisme de rendez-vous.

4.2.1 Sur les machines à base de Transputers

Occam a servi de cible lors de la définition du Transputer, il n'est donc pas étonnant de constater que ce dernier lui est particulièrement bien adapté.

A l'usage, on constate que les implémentations actuelles d'Occam comportent de nombreux défauts : absence de pointeur, de structure de données construite ... Ce qui explique le certain déclin qu'il subit actuellement au profit de langages plus proche des habitudes de programmation comme Logical C.

4.2.2 Inadaptés à notre architecture

De plus, ce type de langage laisse à l'utilisateur le soin de paralléliser les algorithmes. Trouver une décomposition d'un problème en processus parallèles est déjà une tâche difficile avec quelques processus, faire de

même pour une machine massivement parallèle tient du casse-tête. De plus la complexité des problèmes de synchronisation, de communication, et de mise au point des programmes croît avec le nombre de processus. On peut imaginer qu'à l'exception des problèmes très réguliers, laisser la gestion de ces tâches au programmeur est illusoire.

De part sa sémantique, ce langage permet la description de parties de programmes à exécuter de façon séquentielle. La taille de ces parties n'est pas limitée. Si l'on voulait compiler ce langage sur notre architecture, il faudrait soit limiter la taille des processus à ce que peut contenir la cellule, soit parvenir à répartir ces portions de code sur plusieurs cellules tout en garantissant la séquentialité de l'exécution. La très faible taille de la mémoire rend peut vraisemblable la première solution, et l'efficacité de la seconde est pour le moins douteuse.

On voit donc que les langages de la famille d'OCCAM sont difficiles à compiler sur une architecture comme la nôtre. Ils ne remplissent pas l'objectif initial qui était de décharger le programmateur de la gestion du parallélisme. Toutes ces raisons nous ont conduit à nous tourner vers d'autres familles de langages, dans lesquelles le parallélisme est exprimé plus naturellement.

4.3 Les langages orientés objets

Il existe plusieurs raisons qui font des langages orientés objets des outils de programmation adapté aux architectures parallèles [DAL86]:

- Ces langages encouragent la localité en associant à chaque objet (ou donnée) la méthode qui le manipule.

- La sémantique des communications par message des langages comme Smalltalk, est bien adaptée au mécanisme de communication d'une architecture à passage de messages.

- L'utilisation de nom pour identifier les objets permet d'obtenir un espace d'adressage uniforme, indépendant de leur placement physique.

Pour permettre une exécution efficace de tels langages, il est nécessaire de disposer d'un minimum de fonctionnalités de base sur la cellule (pour permettre par exemple une gestion du parallélisme dynamique). La taille mémoire actuelle de la cellule est probablement trop réduite pour permettre d'y implanter la plupart des méthodes.

Une étude est actuellement en cours au sein de notre équipe, visant à évaluer les mécanismes nécessaires à l'exécution des langages orientés objet, ainsi qu'à déterminer une architecture adaptée. Ce travail devrait nous permettre d'entrevoir les évolutions futures de notre architecture cellulaire programmable.

4.4 Les langages dataflow

4.4.1 Introduction

Nous avons vu (2.2.3) que les langages dataflow permettent une expression naturelle du parallélisme. L'absence de contrôle centralisé qui les caractérise les rend particulièrement adaptés à une implantation sur une machine massivement parallèle.

De nombreux langages dataflow ont été développés, parmi les plus connus il faut citer :

VAL (Value Oriented Langage) développé au MIT pour permettre la programmation de leur machine dataflow [ACK79].

LAU (Langage à Assignation Unique) a lui aussi été développé autour d'une machine à Toulouse [PLA76].

Lucid dont Lustre est un dérivé [Wad85].

4.4.2 Le langage LUSTRE

Le langage LUSTRE à été développé au sein du Laboratoire de Génie Informatique de Grenoble par l'équipe SPECTRE, c'est une évolution temps réel du langage LUCID.

Ce chapitre est une introduction informelle (et incomplète) au langage LUSTRE. Pour plus de détails se référer à [PLA87]

Variables, équations, expressions

Une variable X représente une suite infinie $x_0,x_1,...,x_n,...$ de valeurs appartenant à un domaine D_X caractérisé par son type.

Tout domaine contient une valeur indéfinie, notée nil.

On définit une variable au moyen d'une équation X=E, où E est une expression de même type que X, définissant une suite $e_0,...,e_n,...$ Si on a X=E alors pour tout indice n, $x_n=e_n$, autrement dit, d'un point de vue temporel, à chaque instant n les valeurs de X et de E sont égales.

Opérateurs au niveau des valeurs

Tous les opérateurs classiques sur les valeurs sont étendus pour opérer point par point sur les suites :

[op(A,B,...)] = [op(a_n,b_n,...)] quelque soit n, avec la convention que tous les opérateurs sont stricts vis à vis de nil (sauf les opérateurs conditionnels).

Opérateurs synchrones sur les suites

Si E est une expression, définissant la suite $e_0,...,e_n,....$ alors pre(E) définit la suite nil, e_0, ..., e_{n-1},...

Si F est une expression du même type que E, définissant la suite $f_0,...,f_n,...$ alors E->F définit la suite $e_0, f_1, f_2, ..., f_n,...$

Exemple: L'équation X = 0 -> pre(X) + 1

définit la suite croissante des entiers naturels.

Structuration des programmes

Un noeud est un sous programme LUSTRE; il reçoit des variables d'entrée, calcule des variables de sortie et éventuellement des variables locales, au moyen d'un système d'équations.

Exemple:

```
node FRONT(C:bool) returns (H:bool)
let
    H = C -> (C and not pre (C))
tel
```

Ce noeud est vrai à chaque front montant de C. D'après cette déclaration, si E est une expression booléenne, FRONT(E) est synonyme de E -> E and not pre(E)).

Changement d'horloge

Jusqu'à présent nous n'avons présenté que l'aspect purement synchrone de LUSTRE : le comportement d'un programme est cyclique et l'on décrit la valeur de chaque variable à chaque cycle. Dans cette partie nous allons donner la possibilité de définir des variables évoluant à différentes "vitesses", c'est à dire dont l'horloge de renouvellement est une sous suite du cycle de base du programme. Cette possibilité présente divers avantages :

- Elle permet d'abréger la programmation, en n'indiquant que les changements effectifs de valeur d'une variable.

- Elle facilite la programmation de systèmes asynchrones.

Echantillonnage synchrone

Nous appellerons horloge, une variable booléenne. Si C est une horloge et X une expression définissant la suite des valeurs de E correspondant à des instants où C vaut TRUE. Le n-ième terme de cette suite est donc la valeur de E au n-ième instant où C vaut TRUE, comme le montre la table suivante:

C	true	true	false	true	true	false	false
E	e_0	e_1	e_2	e_3	e_4	e_5	e_6
X = E when C	e_0	e1		e_3	e4		

Cette définition appelle plusieurs remarques:

- Dans l'exemple ci-dessus, X est calculé selon l'horloge C, ce qui signifie que la seule notion de temps "connue" par X est la suite des cycles où C vaut TRUE. Par suite, la question "que vaut X quand C vaut FALSE ?" n'a pas de sens.

- Deux variables peuvent maintenant décrire la même suite de valeurs sans pour autant être égales: une variable est caractérisée non plus seulement par sa suite de valeur, mais aussi par son horloge. A toute variable est associée syntaxiquement une horloge, qui peut être la condition toujours vraie si la variable évolue à la cadence du cycle de base du programme.

- Les opérateurs définis jusqu'ici ne peuvent opérer sur des variables d'horloges différentes

<u>Projection</u>

Si E est une expression sur l'horloge C, alors current(E) est une expression dont l'horloge est la même que l'horloge de C, et dont la valeur à chaque cycle est la valeur prise par E au dernier cycle où C valait TRUE.

La table suivante illustre la combinaison des opérateurs when et current :

C	true	false	true	true	false	false	true
Z	z_0	z_1	z_2	z_3	z_4	z_5	z_6
X=Z when C	z_0		z_2	z_3			z_6
Y=current(X)	z_0	z_0	z_2	z_3	z_3	z_3	z_6

L'opérateur current permet d'opérer sur des variables d'horloges différentes, puisque si X et X' sont des variables d'horloges respectives C et C' et si C et C' sont sur la même horloge, alors current(X) op current(X') est une expression correcte, quel que soit l'opérateur op.

<u>Les tableaux</u>

En LUSTRE il existe des tableaux de toutes dimensions, mais pour rendre possible une vérification statique de leur sémantique, leur utilisation est restreinte par les régles suivantes :

- Les bornes des tableaux doivent être connues lors de la compilation.

- Les tableaux ne doivent pas être indexés par des variables: un index doit être une expression faite d'opérateurs simples appliqués aux constantes et index du constructeur "for...let...tel".

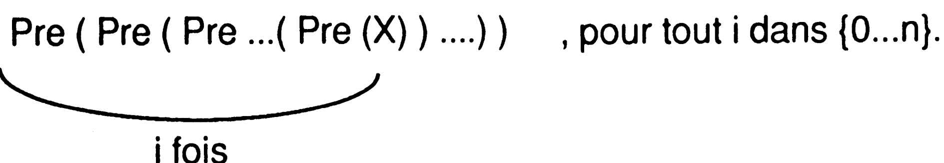
Exemple :

On peut écrire

```
PX[0] = X;
for i in 1..n
    let
    PX[i] = pre(PX[i-1]);
    tel;
```

qui définie PX[i] comme étant

Pre (Pre (Pre ...(Pre (X)) ....))   , pour tout i dans {0...n}.



Cette définition des tableaux permet la description en LUSTRE de tableaux systoliques [Halb87] ce qui constitue une intéressante classe d'applications pour notre compilateur.

Cette syntaxe pose de nombreux problèmes pour la preuve de programmes, ses créateurs travaillent actuellement sur une nouvelle méthode de description des parties de programmes répétitives.

4.4.3 Pourquoi LUSTRE ?

Les raisons qui nous ont poussées à choisir le langage LUSTRE pour l'implanter sur notre architecture sont multiples :

Des raisons géographiques :

Le langage LUSTRE a été développé au sein du Laboratoire de Génie Informatique à Grenoble. Cette proximité et l'aide de ces auteurs nous a permis d'utiliser une partie du compilateur mise au point pour les machines séquentielles. La partie concernant l'analyse syntaxique et la génération du graphe dataflow a ainsi pu être réutilisée pratiquement sans modification, nous permettant ainsi d'économiser un précieux temps.

Des raisons liées au langage :

Sa sémantique bien définie a permis la mise au point d'outils de preuve de programme. Des techniques de compilation de LUSTRE sur architecture séquentielle existent, on peut ainsi imaginer développer et mettre au point le programme sur machine traditionnelle puis n'avoir qu'à le recompiler pour passer à notre architecture massivement parallèle. L'objectif initial de portabilité des programmes est ainsi atteint.

Le choix des applications ciblées :

Le langage LUSTRE a été conçu pour permettre l'écriture de programmes de traitement du signal. Ce domaine d'application comporte plusieurs spécificités :

- Il nécessite une très grande fiabilité logicielle : il est difficile de tolérer des erreurs de programmation dans le programme de contrôle d'une centrale nucléaire ou d'un avion. De part les outils de preuve de programme qui ont été développés autour de lui, LUSTRE offre une solution élégante à ces problèmes.

- Le temps de calcul des algorithmes (et donc de réponse de la machine) est une donnée critique. Or de nombreux traitement de ce type sont très gourmands en temps de calcul; il est donc intéressant de pouvoir

les programmer sur notre réseau pour les faire bénéficier de la puissance d'une machine massivement parallèle.

La possibilité de pouvoir décrire des tableaux systoliques [KUN82] en LUSTRE constitue un atout supplémentaire pour ce langage car elle permet de programmer facilement sur notre réseau les nombreux algorithmes de ce type déjà étudiés et publiés.

On peut donc parler de concordance d'applications visées entre notre machine et le langage LUSTRE; néanmoins une grande partie des raisonnements et des méthodes qui vont être présentés dans cette étude sont applicables à d'autres langages dataflow.

5. La compilation de LUSTRE sur notre machine

5.1 Méthodologie de travail

Dès le début de l'étude sur une machine massivement parallèle programmable, nous avions à l'esprit les problèmes de programmation spécifiques à ce type de machines. Pour permettre de nombreuses interactions entre la définition d'outils logiciels et celle de l'architecture, les deux études ont été menées en parallèle.

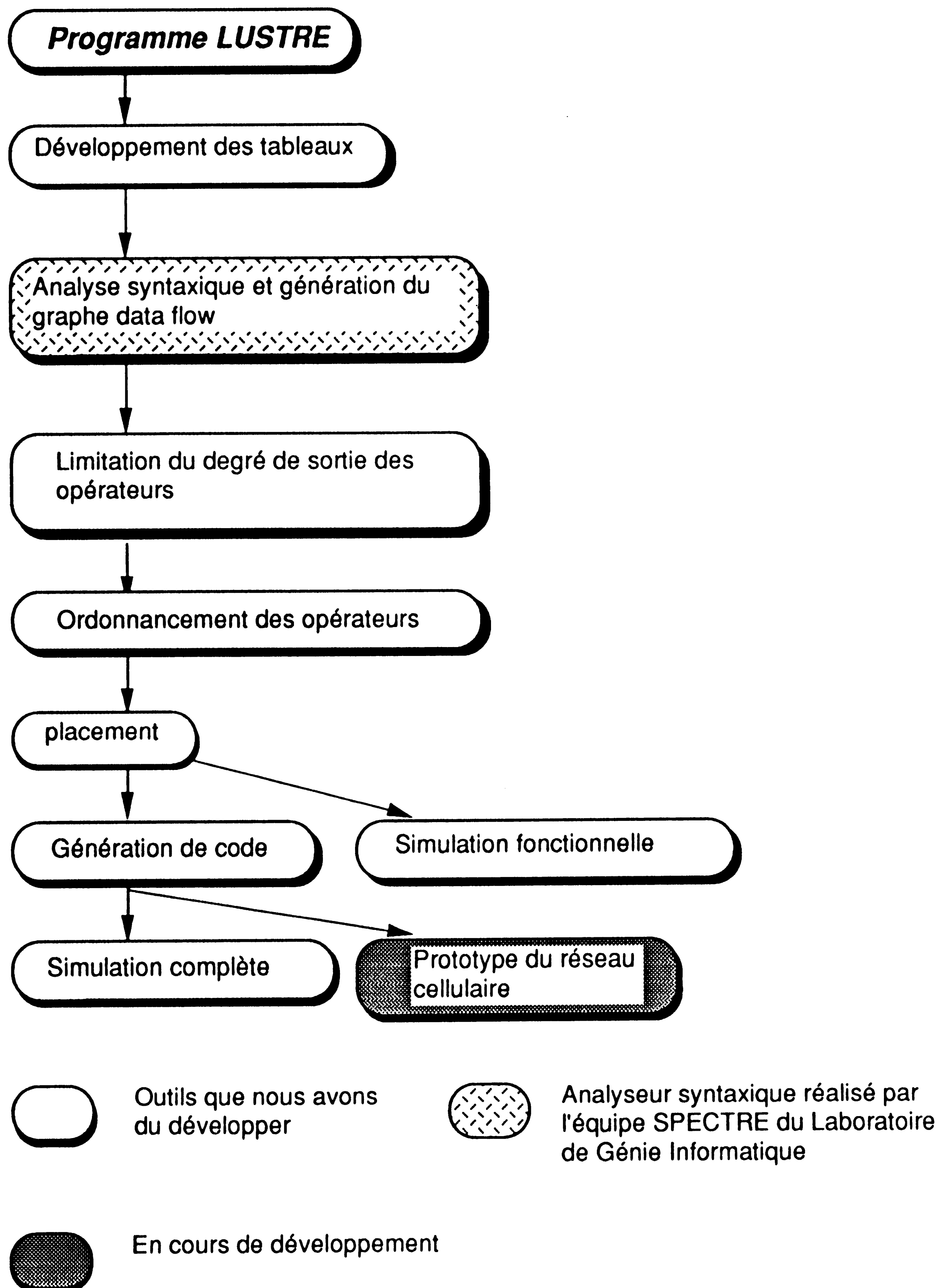
Cette méthodologie de travail a permis d'évaluer les performances de nos outils de compilation avec les variantes successives du jeu d'instructions qu'a connues la cellule.

Sans vouloir faire de notre réseau une machine dédiée à l'exécution du langage LUSTRE, nous espérions faire apparaître quelques instructions permettant une compilation plus efficace de ce langage, à l'image des instructions LINK et UNLINK du 68000 qui facilitent l'exécution des langages procéduraux sur ce microprocesseur.

Le résultat de nos réflexions est plus diffus, l'apport essentiel des outils de compilation à la définition de notre architecture est une meilleure compréhension de l'algorithmique parallèle. On peut tout de même voir la trace de cette méthode de travail dans l'instruction TRY, ainsi que dans le choix d'un mécanisme de routage, capable de ranger directement les messages dans la mémoire de la cellule destination.

Pour permettre une évaluation précise des performances du réseau en fonction des différents choix étudiés, nous avons du programmer un

certain nombre d'outils. L'ensemble de ces outils constitue une chaîne complète de programmation et de simulation du réseau :



Les deux simulateurs de cette chaîne permettent de choisir le niveau de précision et de rapidité souhaitées dans la simulation.

Le simulateur fonctionnel permet d'évaluer rapidement les performances d'un programme. L'ensemble du mécanisme de communication est reproduit par cet outil. L'exécution des programmes est simulée au niveau des opérateurs et non des instructions assembleurs. Pour affiner la simulation, chacun d'entre eux se voit attribuer une constante représentant sa durée de calcul.

Le simulateur complet permet de reproduire l'exécution des programmes dans les moindres détails de ses instructions. On peut, grâce à lui, obtenir une image fidèle du fonctionnement du réseau.

Ces outils sont détaillés avec précision dans l'annexe 1. Il faut noter que quand un réseau prototype sera disponible, il viendra naturellement trouver sa place dans cette chaîne d'étude.

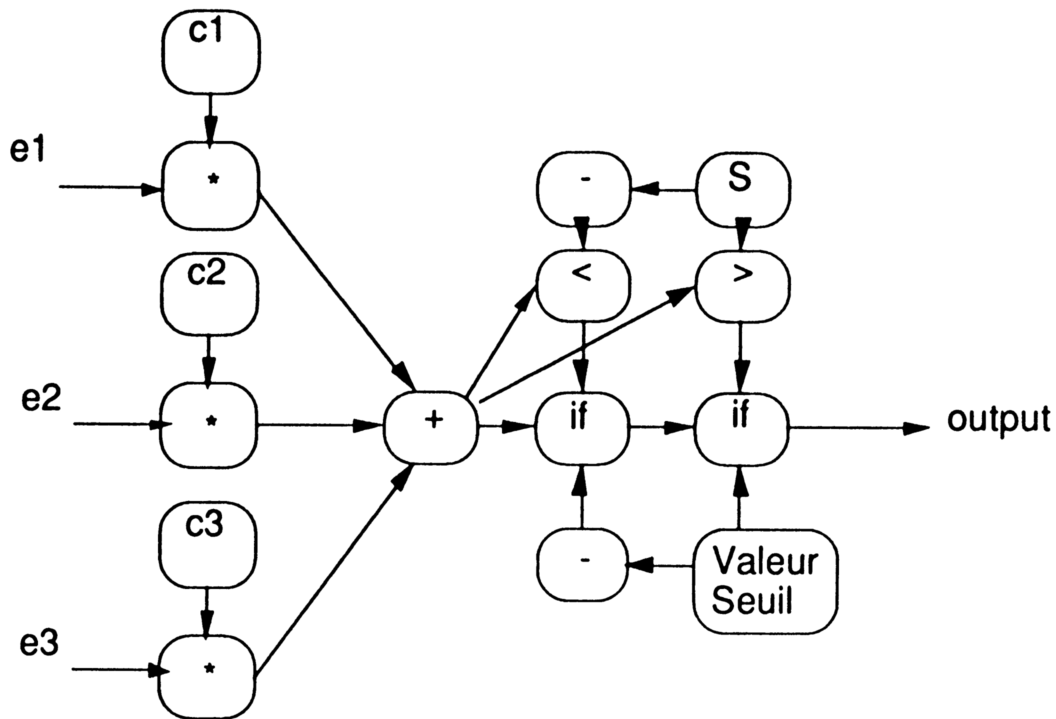
5.2 Idée de départ

Le langage LUSTRE étant un langage dataflow, tout programme peut se représenter sous forme d'un graphe d'opérateurs :

Ainsi le programme LUSTRE d'un neurone à 3 entrées :

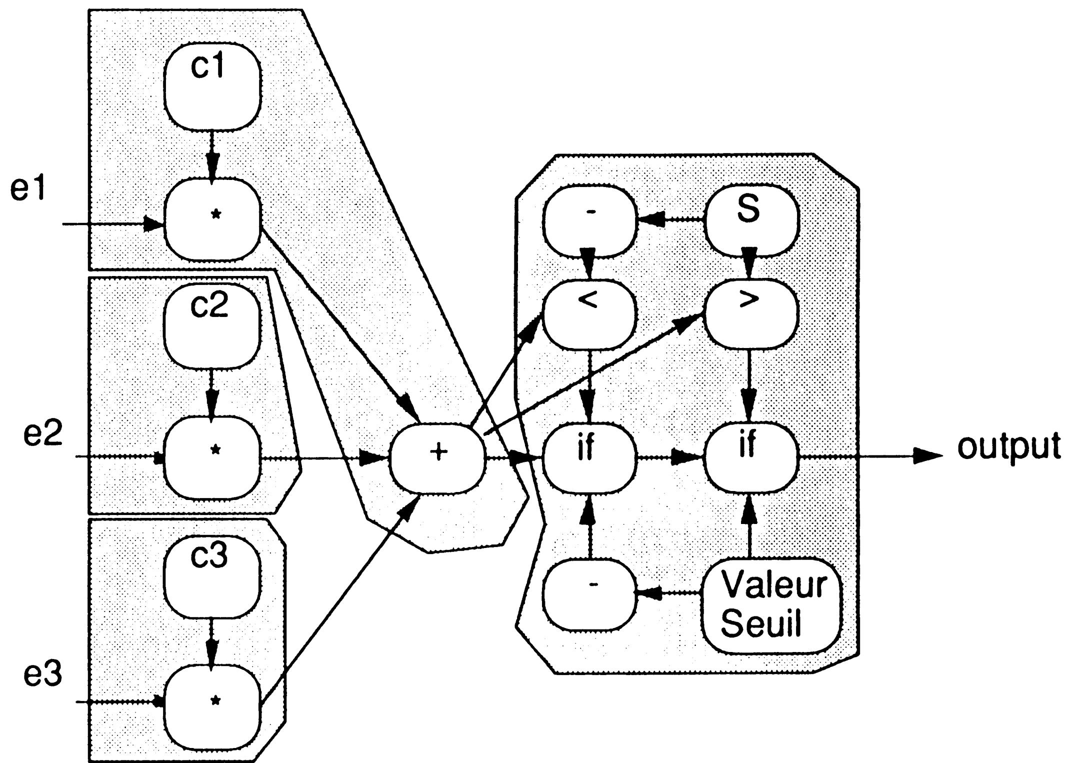
```
I = Entrée1 * C1 + Entrée2 * C2 + Entrée3 * C3
Sortie =  if I < -Seuil
      then -Valeur_seuil
      else
            if I > Seuil
            then Valeur_seuil
            else I
```

donne le graphe suivant :

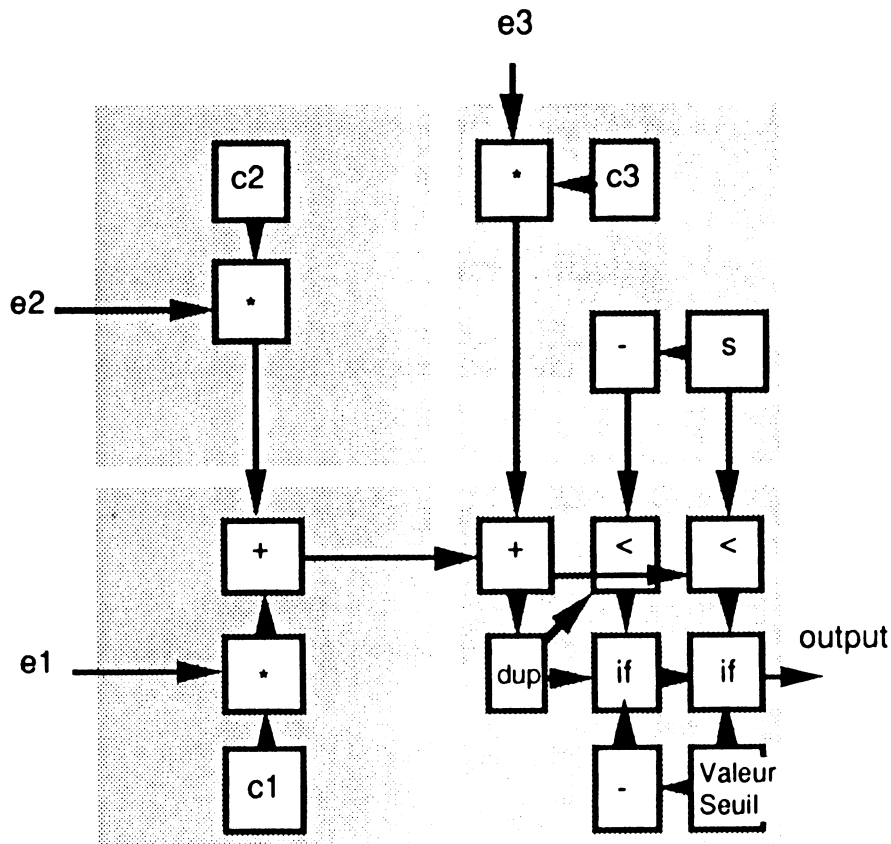


Ce graphe est alors découpé en plusieurs sous graphes, afin de répartir la charge de calcul sur les différentes cellules. Le placement doit aussi faire attention à minimiser les communications :

Exemple de découpage en 4 sous graphes :



Lors d'une phase de placement, chaque sous graphe se voit attribuer une cellule:



Il ne reste plus ensuite qu'à générer le programme assembleur correspondant au sous graphe associé à chaque cellule .

Notre compilateur utilise ce principe pour permettre l'exécution de programme LUSTRE; nous allons maintenant en détailler les étapes.

5.3 Construction du graphe dataflow

Cette étape consiste à extraire de la forme textuelle du programme le graphe correspondant. Un certain nombre d'opérations sont efféctuées lors de cette génération :

- Analyse syntaxique et lexicale : comme tout langage de haut niveau, Lustre répond à une syntaxe précise. C'est à ce niveau que l'on

vérifie que toutes les variables et les noeuds (procédures) appelés sont bien déclarés.

- Extension des tableaux et des noeuds : à ce niveau chaque appel d'un noeud est remplacé par sa déclaration. Ainsi, contrairement à un langage classique où le code d'une procédure n'est généré qu'une fois, le sous graphe d'un noeud est généré à chaque appel. De même les tableaux sont mis à plat : une opération portant sur les éléments 1 à 10 d'un tableau sera dupliquée en 10 opérations portant chacune sur l'un de ses éléments.

Exemple :

Le programme :

```
PX[0] = X;
for i in 1..5
    let
    PX[i] = pre(PX[i-1]);
    tel;
```

devient après développement des tableaux :

```
PX[0] = X;
PX[1] = pre(PX[0]);
PX[2] = pre(PX[1]);
PX[3] = pre(PX[2]);
PX[4] = pre(PX[3]);
PX[5] = pre(PX[4]);
```

- Génération du graphe : On met à jour à ce niveau une structure de données décrivant chaque arc du graphe

Dans le prototype logiciel que nous avons développé nous avons pu réutiliser l'analyseur syntaxique et lexical ainsi que le générateur de graphe du compilateur LUSTRE développé pour des machines séquentielles par

l'équipe Spectre. Les tableaux n'étant pas implémentés dans la version que nous avons utilisée, il nous à fallu écrire un préprocesseur réalisant leur développement.

5.4 Le placement

5.4.1 Allocation dynamique ou allocation statique ?

Le problème de placement de tâches sur une architecture parallèle a fait l'objet de nombreuses études. Parmi les solutions que l'on trouve dans la littérature, on peut discerner deux grandes tendances :

L'allocation statique de tâches : chaque tâche se voit associer un processeur pour toute la durée de vie du programme, et seules les données circulent sur les liens de communications. Cette méthode présente l'avantage de la simplicité ; son principal défaut est de ne pas adapter le placement et donc la répartition du travail en fonction des données et de la charge des processeurs à un instant donné.

L'allocation dynamique de tâches consiste à distribuer en cours d'exécution les tâches en fonction de la charge des processeurs. Cette méthode pose de nombreux problèmes qui ne sont pas tous résolus au jour d'aujourd'hui :

- Elle nécessite des moyens pour allouer dynamiquement un processeur inoccupé à un nouveau travail; il faut donc savoir repérer les processeurs inutilisés. Sur une machine massivement parallèle, pour être efficace, ces stratégies doivent être distribuées sur l'ensemble des processeurs.

- Pour permettre le transfert dynamique de programme sur un processeur, chacun doit comporter un logiciel de base type moniteur, permettant le chargement puis le lancement du code.

- Parmi tout les processeurs inoccupés, lequel choisir lorsque

l'on veut attribuer une tâche ? Il n'existe pas encore de réponse à cette question dans les travaux déjà publiés.

Deux raisons nous ont fait choisir un placement statique des processus :

- Dans un programme LUSTRE qui ne comporte qu'une horloge, tous les opérateurs sont exécutés avec la même fréquence : à chaque cycle de l'horloge de base tous les arcs sont parcourus par des valeurs et tous les noeuds sont évalués. Ainsi l'écoulement des valeurs dans le graphe dataflow, et l'ensemble des tâches à effectuer, pour un programme ne comportant qu'une horloge, est statique.

- La taille de la cellule, et notamment de la mémoire autorise difficilement l'implantation des mécanismes de ramasse miette et de moniteur nécessaires à une allocation dynamique.

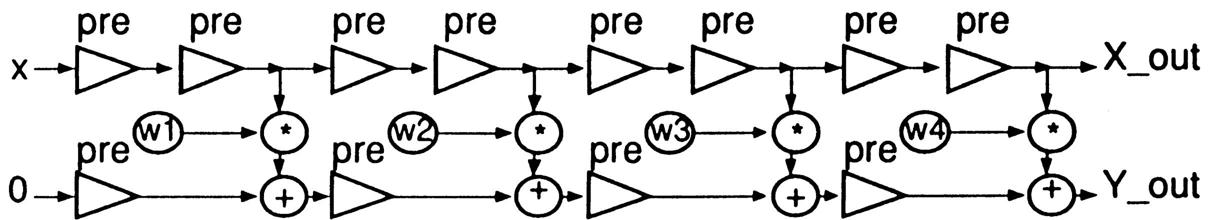
5.4.2 Limitation des performances; problèmes et solutions

L'idée la plus simple consiste à assigner à chaque cellule un opérateur du graphe dataflow.

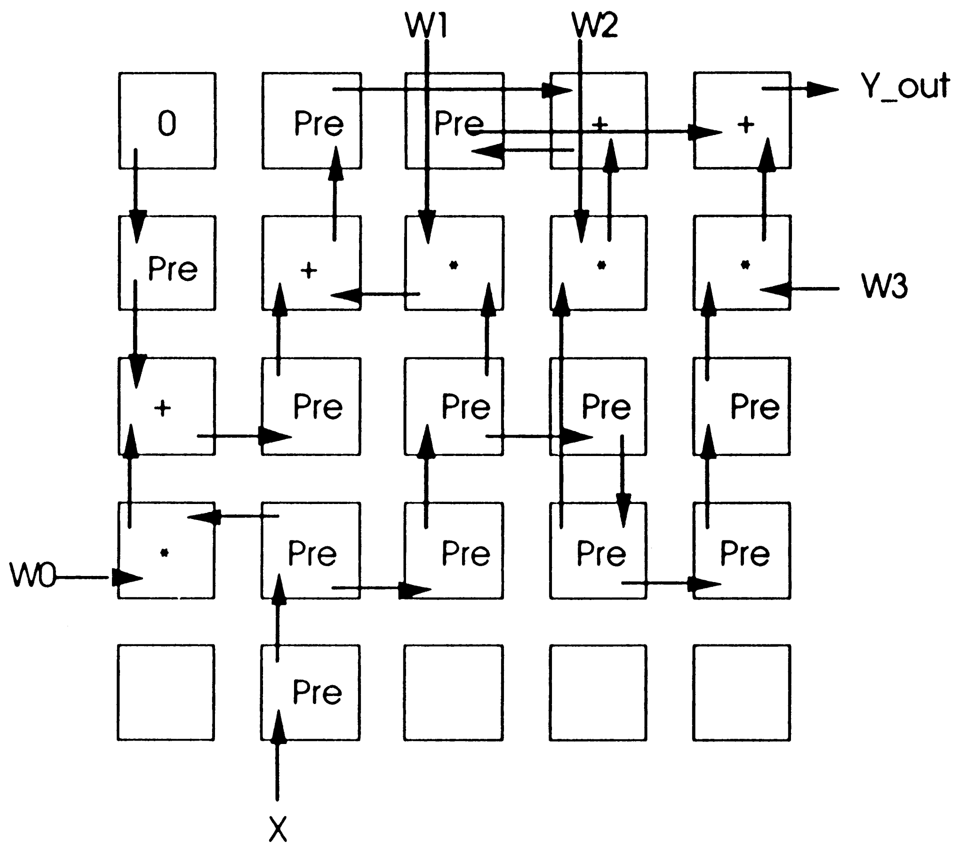
Le programme LUSTRE du produit de convolution

```
node convolution (
(x:int) when true;
for k in 0..4
      let
      (c[k]:int) when true;
      tel
)
returns (
(y:int) when true;
);

var
for k in -1..4
      let
      (x[k]:int) when true;
      (y[k]:int) when true;
      tel

let
for k in 0..4
      let
      y[k] = pre (y[k-1]) + c[k]*x[k];
      x[k] = pre (pre (x[k-1]));
      tel;
x[-1] = x;
y[-1] = y;
y = y[4];
tel.
```

Nous donne le graphe dataflow suivant:



que l'on place alors sur le réseau cellulaire :



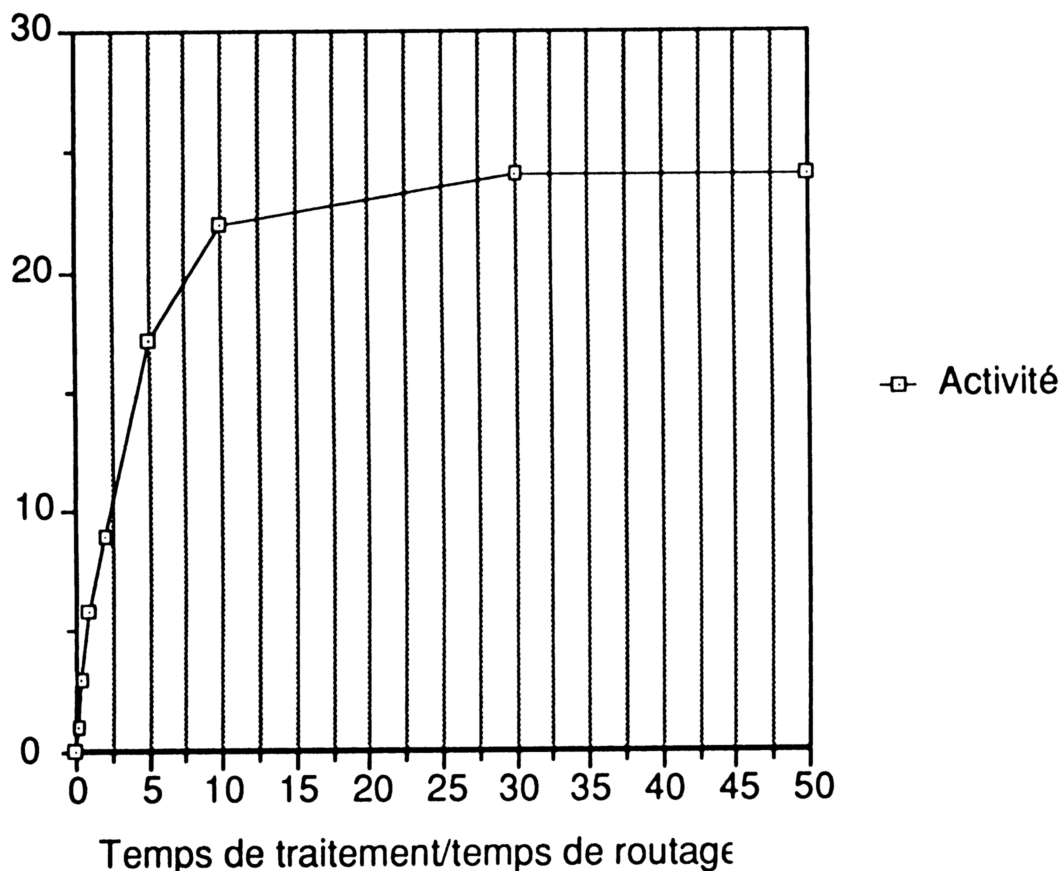
On constate immédiatement que:

- Le programme n'est pas dépendant de l'architecture (il s'écrit de la même manière pour une exécution sur machine séquentielle).

- Seule sa taille maximale dépend de l'architecture (du nombre de cellules), comme elle dépend de la taille de la mémoire sur une machine traditionnelle.

- Le placement automatique (annexe 1) conduit à une implantation désordonnée des opérateurs très éloignée de la régularité des tableaux systoliques.

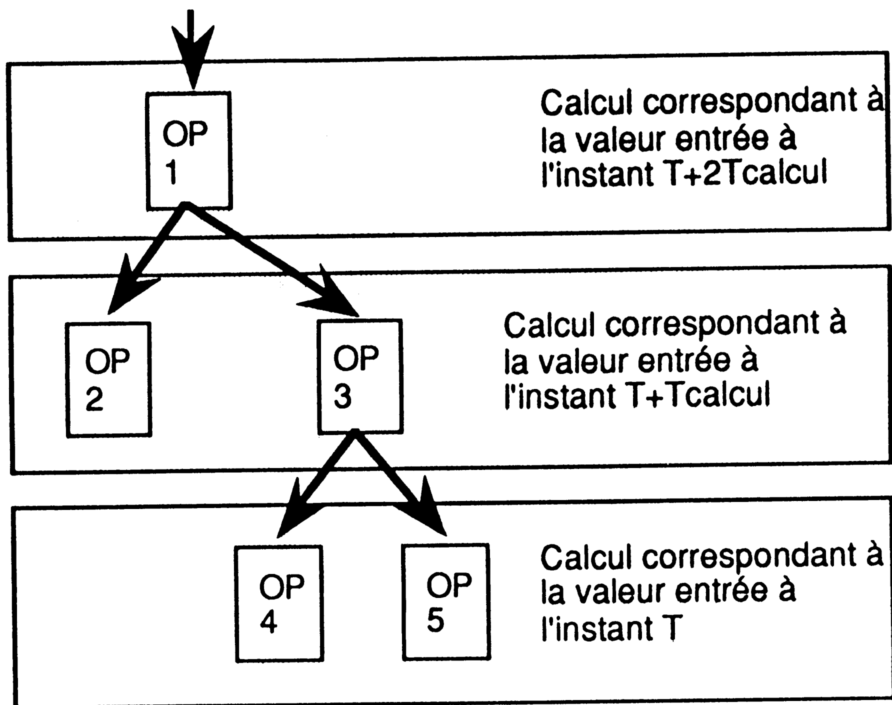
Une simulation de cet exemple à l'aide de notre simulateur fonctionnel (Annexe 1) fait apparaître un problème important : quelques soient les performances du mécanisme de communication, l'activité du réseau ne peut dépasser une certaine valeur (ici 25%).



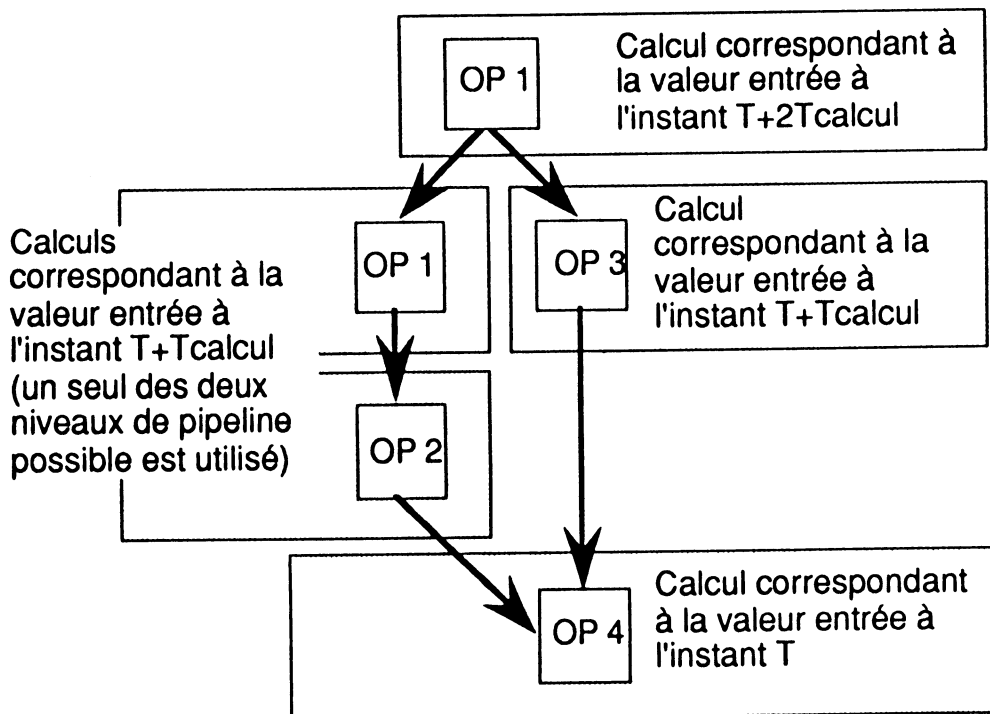
Il existe deux raisons à cette limitation :

- Les opérateurs ont un temps de calcul différent suivant leur fonction : entre une multiplication et une addition il y a un rapport 10. Tous les opérateurs se synchronisent donc sur le plus lent.

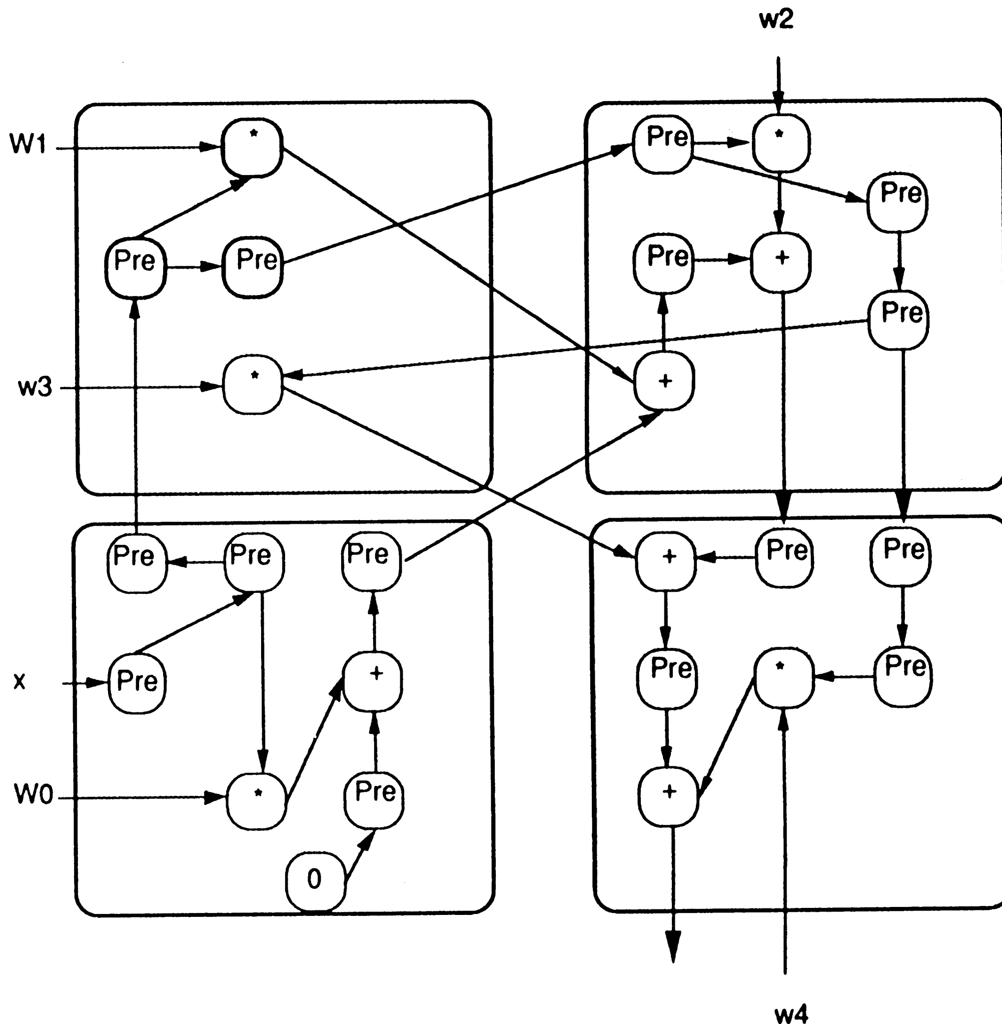
- La différence de longueur entre les chemins: dans le cas idéal et avec un temps de communication nul, chaque opérateur devrait permettre d'introduire un niveau de pipe-line:



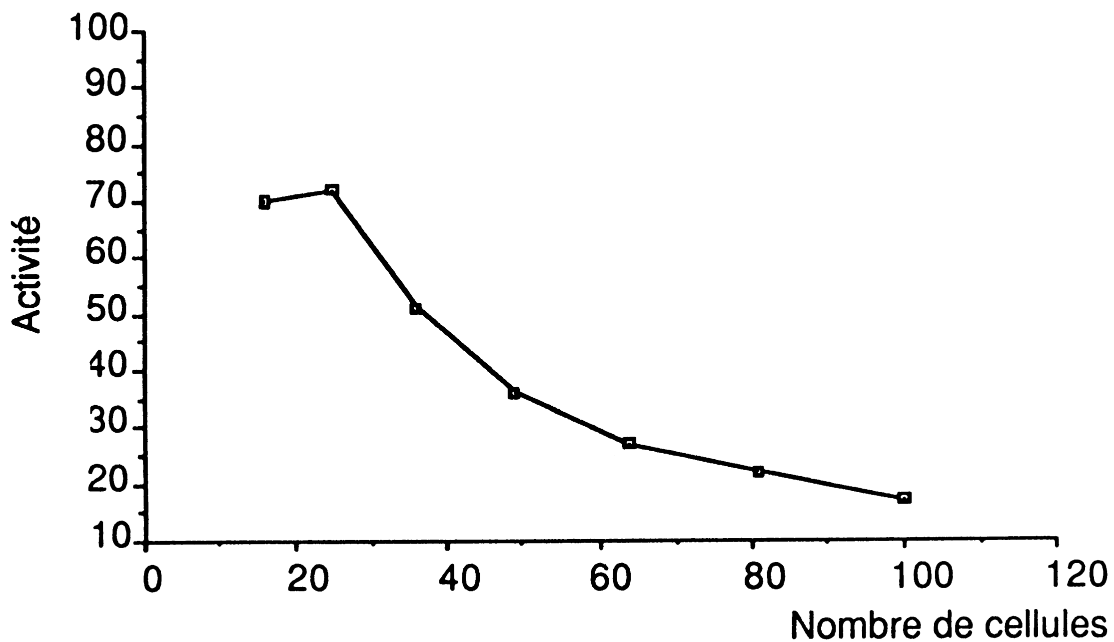
Mais parfois la structure du graphe introduit des anomalies limitant les niveaux de pipeline.



Le premier problème peut se résoudre en regroupant plusieurs opérateurs "rapides" sur une même cellule afin d'égaliser les charges de calcul entre celles ci. Ainsi pour un produit de convolution 4 points on obtient le placement suivant :



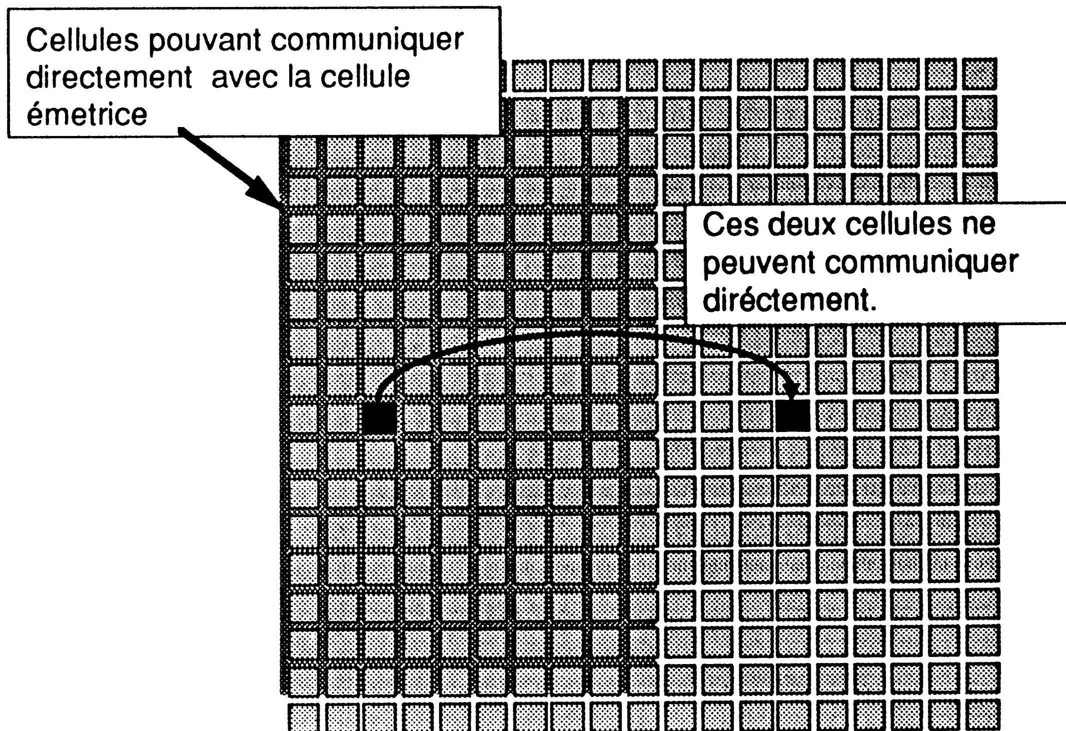
Une simulation du même exemple que ci-dessus mais avec un nombre de points accru (produit de convolution 10 points : graphe de 69 noeuds) nous donne l'activité du réseau en fonction du nombre de cellules utilisées:



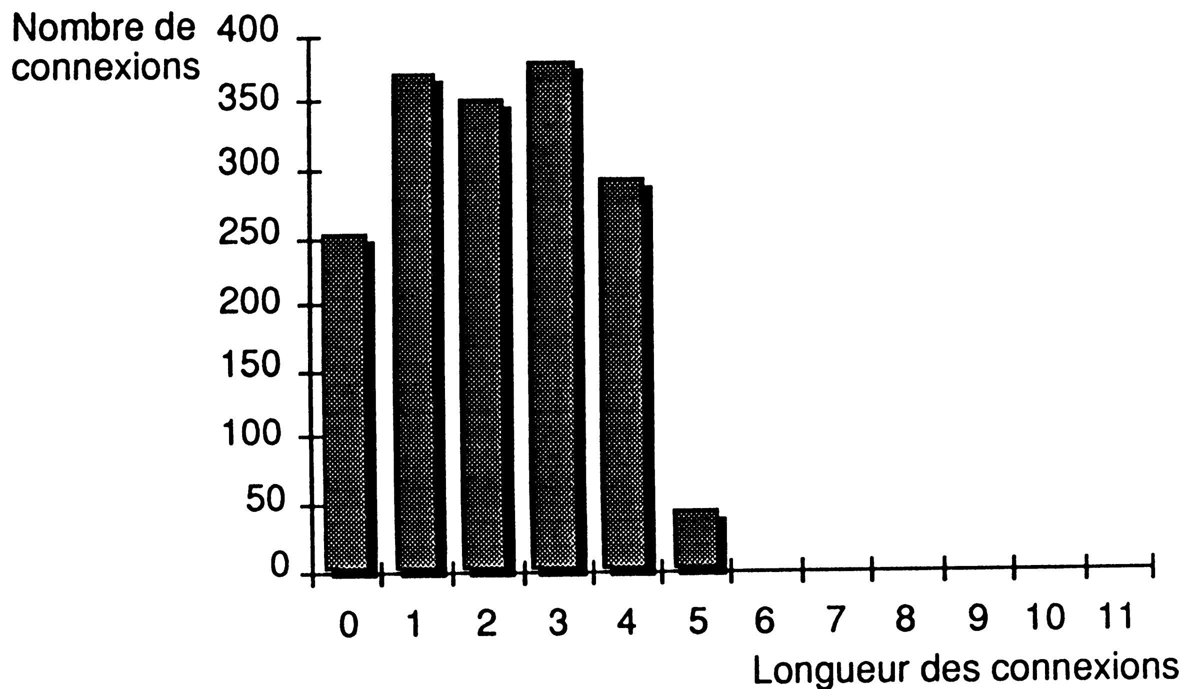
Pour améliorer la structure du graphe et limiter les anomalies concernant les niveaux de pipeline, il faut mettre en place des files à l'entrée des chemins de données trop courts. La complexité de leur gestion fait craindre une perte de performance supérieure au gain ce qui nous a conduits à abandonner l'étude d'une telle solution.

5.4.3 Les impossibilités de placement et les cellules relais

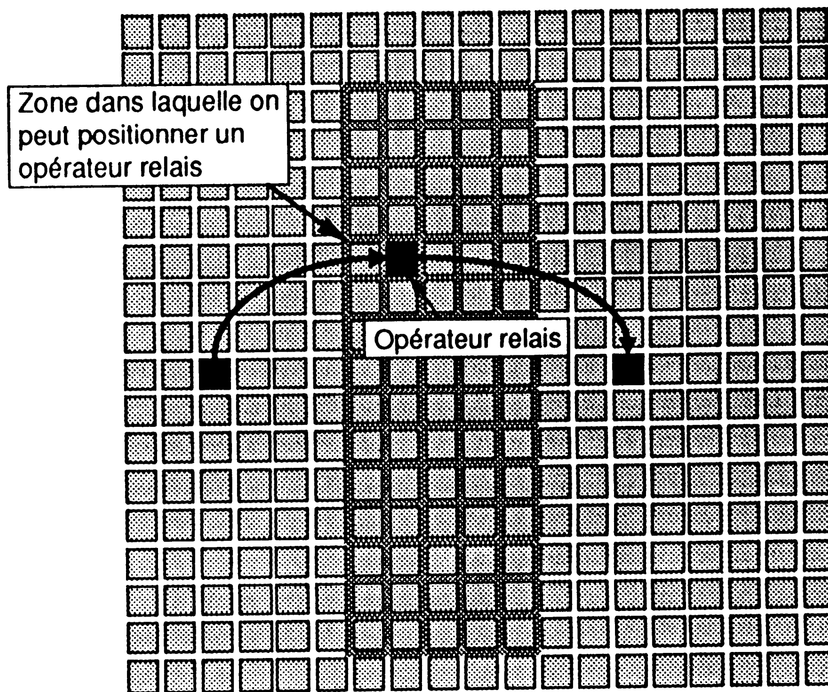
Comme nous l'avons vu (2.3.3) notre architecture utilise pour le codage du déplacement des messages, dx et dy, 4 bits soit des valeurs comprises entre -8 et +7. Il est impossible de garantir que quelque soit le graphe on puisse assurer un placement ayant une localité telle qu'aucun couple d'opérateurs connectés ne se trouve à une distance dépassant les capacités d'adressage des messages.



Pour évaluer l'étendue du problème il est nécessaire d'examiner le résultat du placement sur des exemples pratiques. Nous avons utilisé le placeur de notre chaîne de compilation (Annexe 1) pour mesurer (ici pour une FFT placée sur un réseau de 11*11 cellules) la répartition du nombre de connexions en fonction de leurs longueurs. Le graphe obtenu est le suivant :



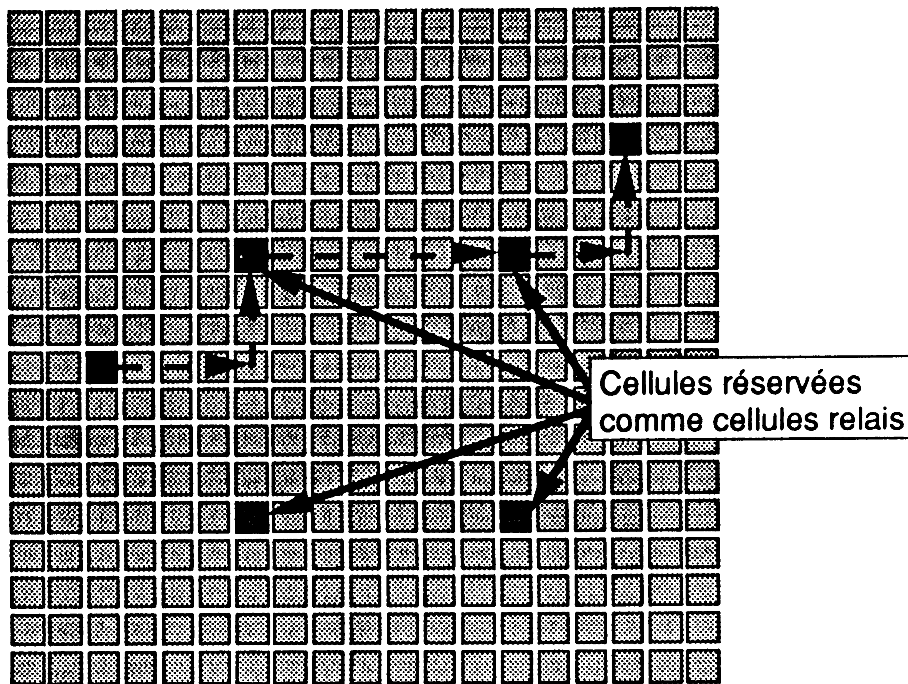
On voit donc que dans la pratique le nombre de connexions dépassant les capacités d'adressage des messages reste nul pour des graphes et des réseaux de taille moyenne. C'est pour cette raison, que la version actuelle de notre compilateur, n'intègre pas de mécanisme permettant de remédier à ce problème. Pour les versions futures, la solution la plus simple sera de prévoir des opérateurs relais permettant une ré-émission des messages dont les destinataires sont trop lointains pour être adressés directement.



Plusieurs stratégies sont possibles à ce niveau :

- On peut, après le placement, espérer trouver dans l'ensemble des cellules utilisables comme relais (représenté en gris sur le schéma ci-dessus) au moins une cellule, disposant d'assez de place en mémoire pour jouer ce rôle. Cette solution ne garantit pas que l'on pourra toujours trouver la place nécessaire dans la mémoire d'une cellule pour jouer le rôle de relais. L'expérience nous montrant que le nombre de communications nécessitant un relais était très faible, cette méthode doit être suffisante dans la pratique.

- On peut réserver une cellule par carré de 7*7 pour jouer le rôle de relais. Cette méthode présente sur la précédente la garantie de pouvoir établir une communication entre deux cellules distantes. Son principal défaut est d'immobiliser une cellule sur 49 soit approximativement 2% de l'ensemble du réseau.



: Chemins empruntés par le message

5.4.4 Les méthodes de placement.

Le problème du placement étant de type Np-complet, il est illusoire d'espérer trouver une solution optimale pour un exemple de taille significative.

De nombreux algorithmes ont été présentés dans la littérature, trois principaux se détachent [RAY90]:

-L'algorithme glouton

- L'algorithme de Bokhari

- Le recuit simulé

La facilité de programmation et d'adaptabilité à de nouveaux critères de placements, nous ont fait retenir le recuit simulé pour être implanté dans notre prototype. Une étude précédente [COR88] menée par R. Cornu Eymieux dans notre équipe nous à déjà montré qu'il était possible

d'implanter ce type d'algorithme sur notre machine. Il est ainsi possible d'imaginer que dans une évolution future de notre compilateur cette étape importante et coûteuse en temps de calcul, qu'est le placement soit exécutée par le réseau.

5.5 La synchronisation

La génération de code consiste à associer à chaque sous graphe placé sur une cellule, un programme assembleur équivalent. C'est à ce niveau que l'on doit se soucier du type des opérandes et des problèmes de synchronisation.

5.5 Pourquoi synchroniser ?

La communication entre deux cellules peut être vue comme un problème producteur consommateur. Dans le cas de notre réseau, ce problème est encore compliqué par le fait que le producteur et le consommateur ne sont pas forcément voisins, et donc que l'un ne peut pas connaître l'état de l'autre autrement que par des échanges de messages.

Pour assouplir la synchronisation, il est possible d'insérer entre eux une file permettant une légère désynchronisation. Malheureusement la taille de cette file est forcément bornée (ne serait-ce que par la taille de la mémoire disponible dans la cellule) et pour éviter tout débordement il faut tout de même ajouter un mécanisme de synchronisation. La complexité de gestion d'une file et le peu de gain attendu d'une telle méthode nous a fait abandonner cette solution.

Nous avons vu dans la partie précédente que lorsqu'un message parvient dans sa cellule destination, il est automatiquement rangé par le routage dans la mémoire de la cellule. Aucun test n'est effectué par le routage pour s'assurer que l'adresse destination est bien vide de toute valeur. Pour éviter la perte d'opérandes il faut s'assurer avant l'envoi de chaque message que son adresse d'arrivée dans la mémoire de la partie traitement de la cellule est vide.

Il aurait été possible d'ajouter au routage un mécanisme de test, afin de n'autoriser l'arrivée de messages qu'à des adresses libres. Même avec ce mécanisme il était tout de même nécessaire de synchroniser les cellules car une absence complète de synchronisation aurait conduit des messages à attendre la libération de leur destination à la périphérie de la cellule interdisant toute autre communication par ce chemin. Une telle situation conduit rapidement à un blocage du réseau.

Il nous faut donc mettre en place un mécanisme de synchronisation. Le réseau cellulaire étant asynchrone, il n'existe aucun signal global capable de remplir cette fonction (comme on pouvait en trouver dans l'architecture dédiée à la simulation logique réalisée dans notre équipe par P. Objois [OBJ88] et R. C. Emieux [COR88]). Il nous est apparu beaucoup plus performant d'avoir recours à une synchronisation par des messages "demande de valeur".

5.6 De l'influence de l'instant d'émission des messages de synchronisations sur les performances

Il existe de nombreuses façon d'utiliser les messages "demande de valeur". Nous allons constater que la position des ordres "envoyer un message demande" et "attendre un message demande" dans l'algorithme de l'opérateur, influence grandement les performances du réseau.

Le schéma le plus naïf :

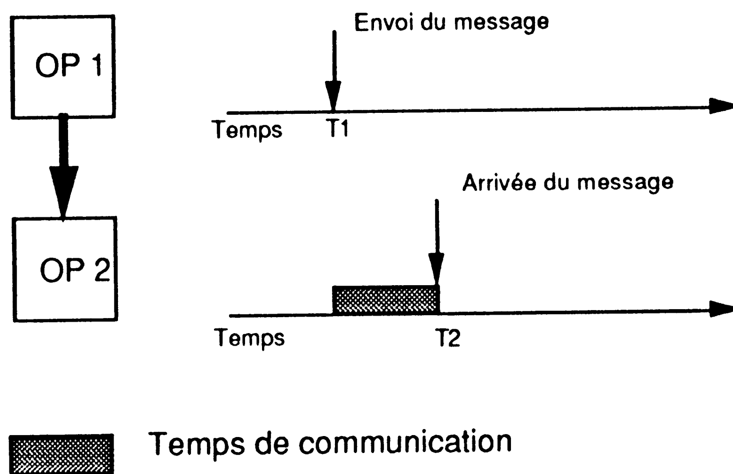
```
répéter
      attendre les demandes
      envoyer les demandes
      attendre les opérandes
      faire le calcul
      envoyer les résultats
tant que vrai
```

Cet algorithme fonctionne suivant le principe "demand driven" : L'ordinateur hôte émet une demande, celle-ci se propage dans le graphe

jusqu'à parvenir à des opérateurs n'attendant pas de valeur pour fournir un résultat (une entrée, un PRE ou une constante). Ces résultats se propagent ensuite dans le graphe en sens inverse des demandes jusqu'à la sortie.

Représentation des chronogrammes

Le graphe sera représenté sur la partie gauche du schéma, la partie droite étant utilisée comme représentation sous forme de chronogramme du travail du noeud correspondant. Les temps de communication sont reportés sur le chronogramme du noeud destination. Ainsi le schéma :



signifie OP1 envoie un message à OP2 à l'instant T1 et OP2 reçoit le message à l'instant T2

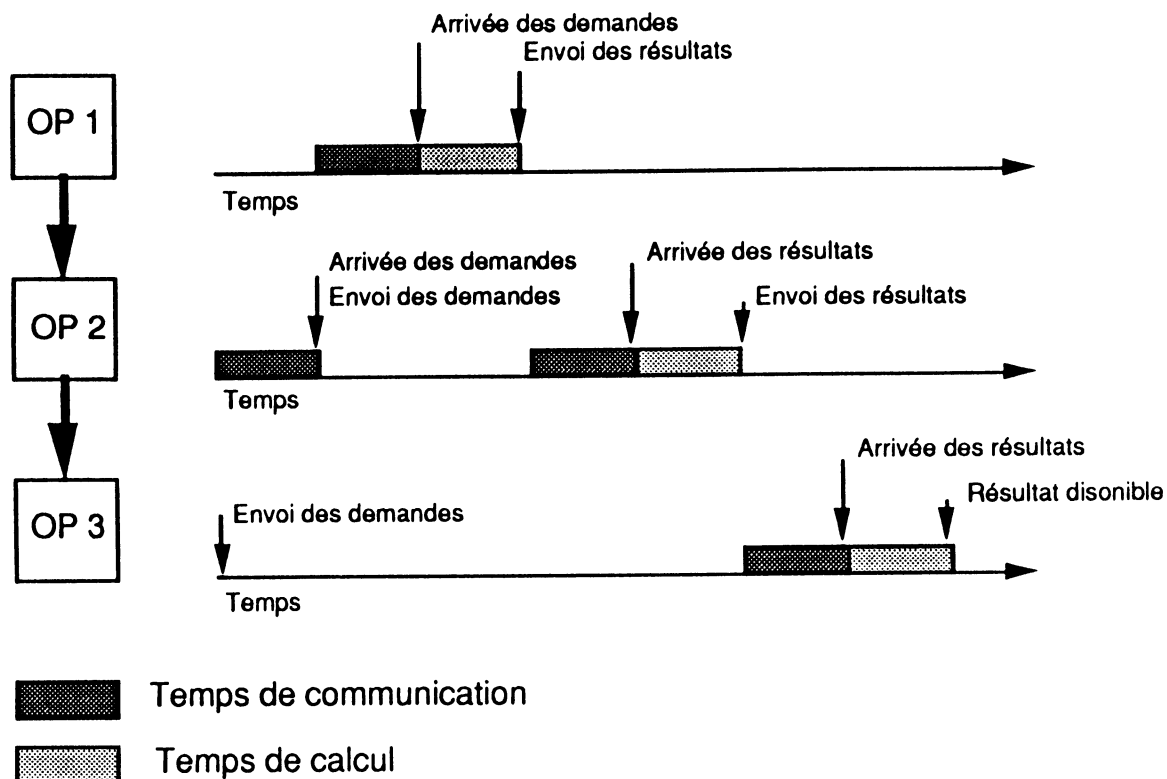
Examinons le fonctionnement de cet algorithme pour un cas dégénéré comportant trois opérateurs :

- OP1 opérateur sans antécédent

- OP3 opérateur sans suivant

- OP2 opérateur classique

Ce graphe simple n'est pas réaliste car on ne tient pas compte du dialogue entre le réseau et l'hôte, mais il est suffisant pour comprendre les mécanismes mis en jeu.



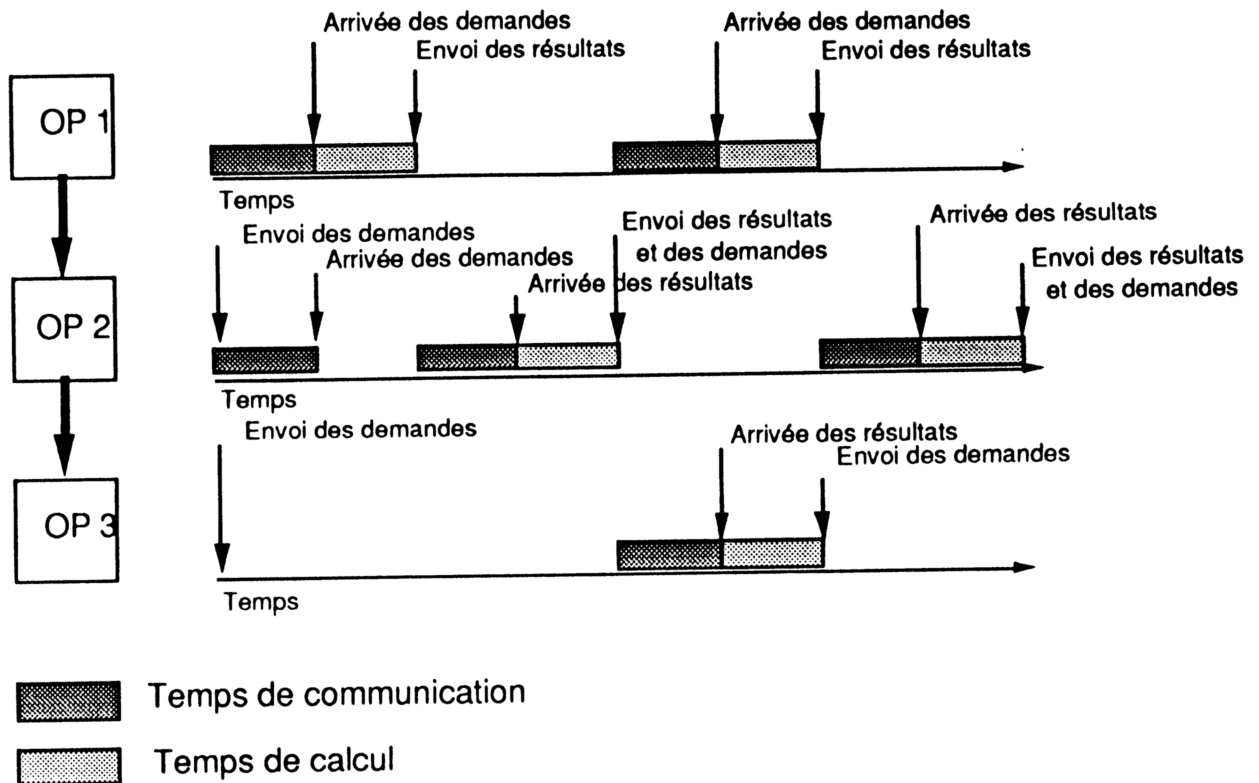
On voit sur cet exemple que le parallélisme est nul (en fait il dépend de la forme du graphe). Cette méthode empêche un fonctionnement en pipeline du réseau limitant ainsi fortement les performances.

Pour augmenter le recouvrement d'exécution, il est tentant pour un opérateur de ne pas attendre de recevoir de "demande" pour envoyer les siennes.

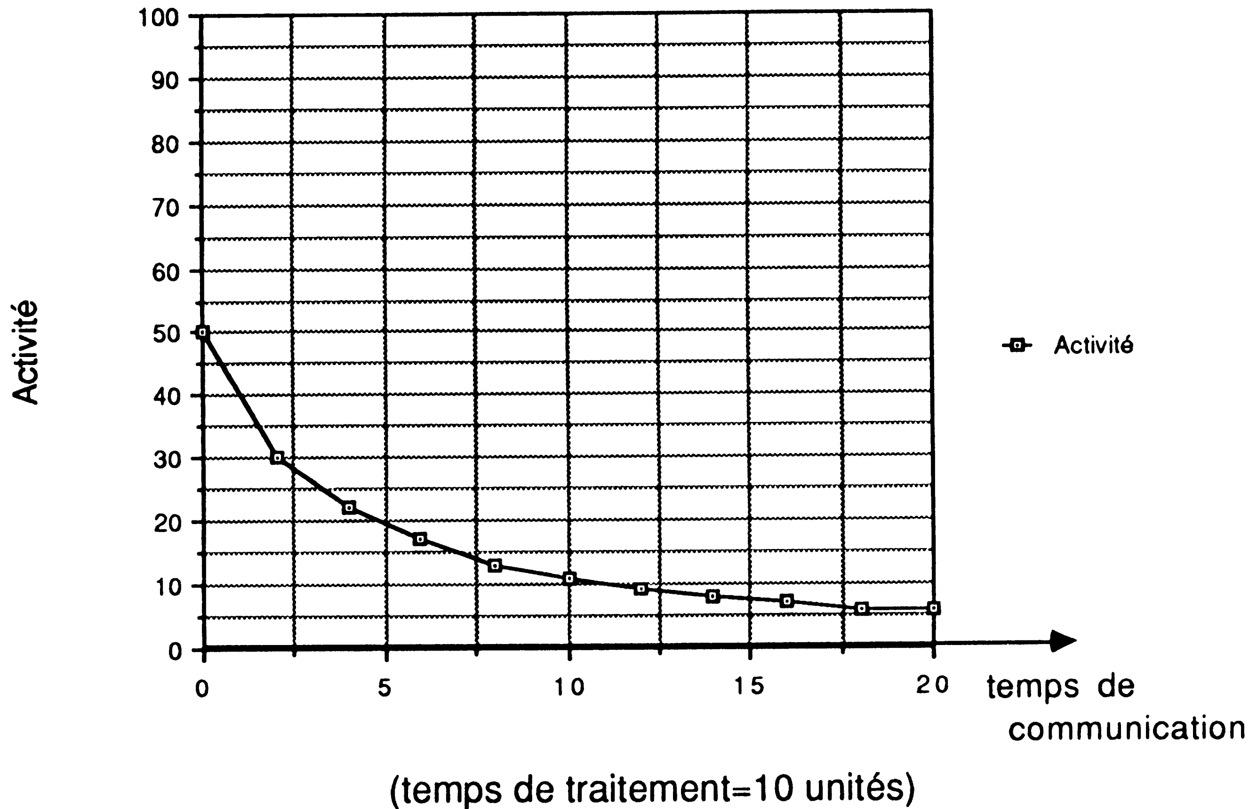
L'algorithme utilisé par les cellules serait alors:

```
répéter
    envoyer les demandes
    attendre les opérandes et les demandes
    faire le calcul
    envoyer les résultats
tant que vrai
```

Pour les mêmes opérateurs que ci-dessus on obtient le chronogramme suivant :



Ce schéma montre bien que l'activité du réseau dépend directement du temps de communication ce que confirme le graphe suivant:



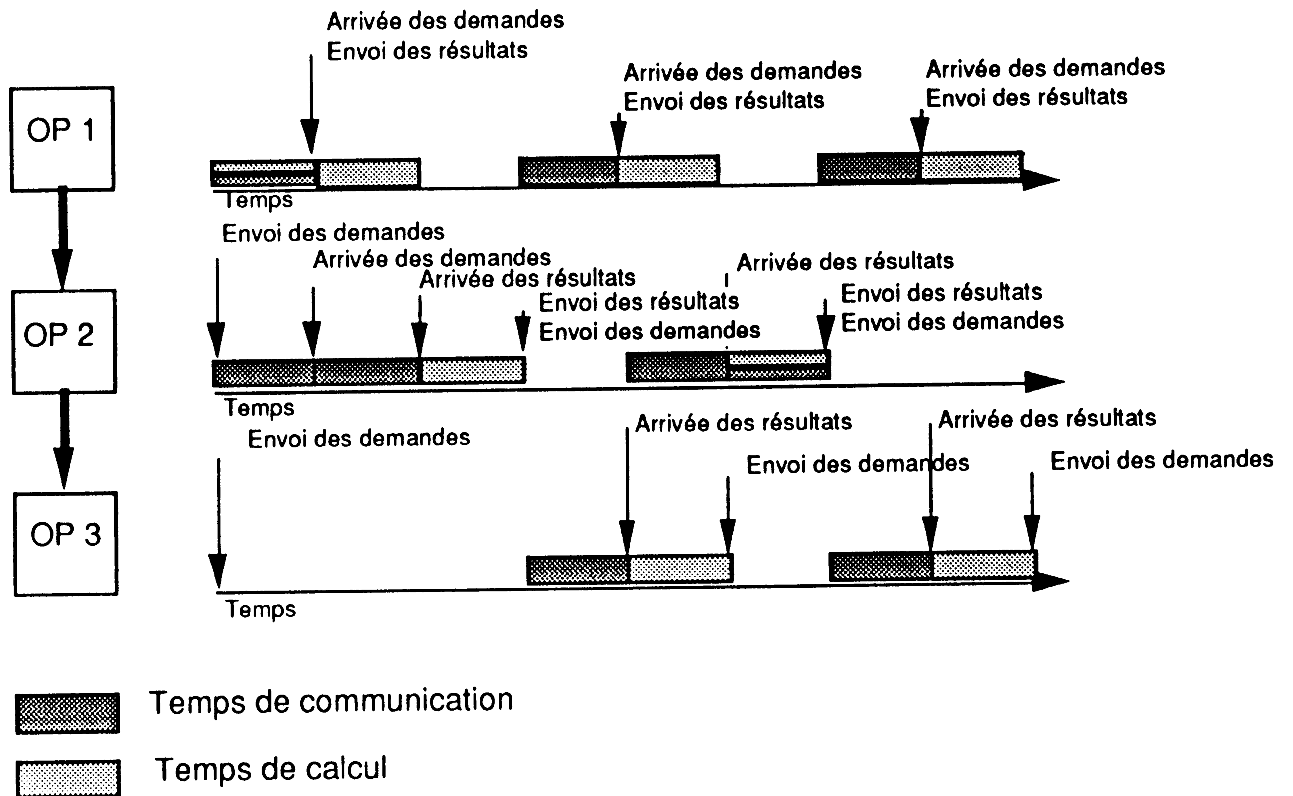
(temps de traitement=10 unités)

Pour améliorer encore les performances, on peut faire la remarque suivante:

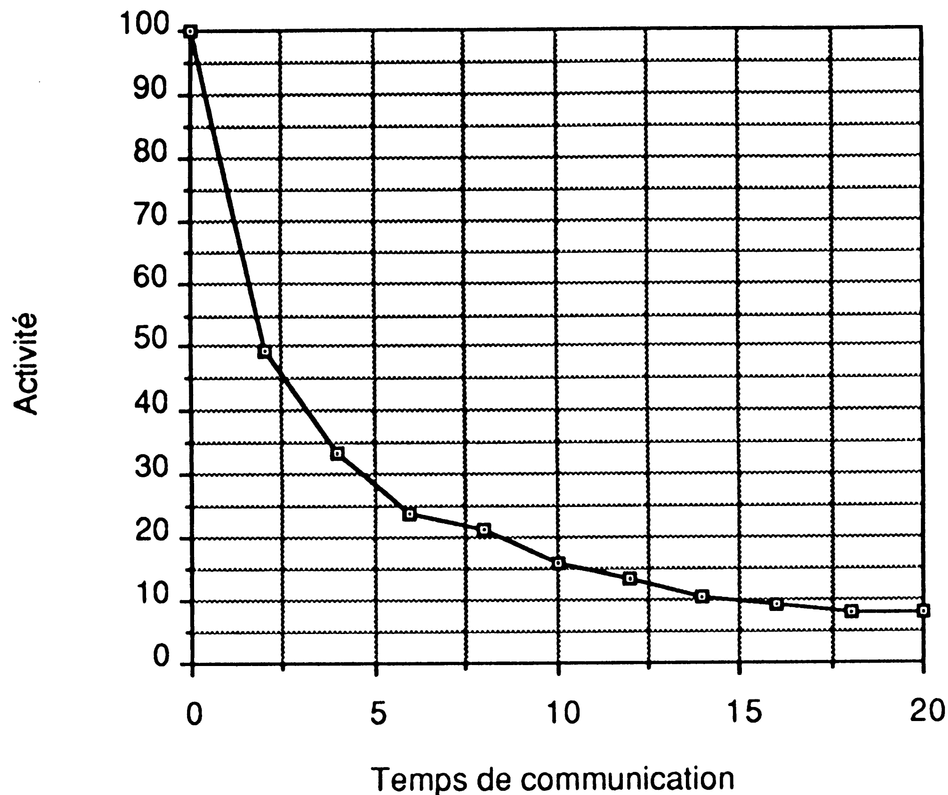
- On peut commencer le calcul sans attendre les demandes, pour s'assurer d'une synchronisation correcte il suffit de les attendre avant d'envoyer le résultat :

```
répéter
    envoyer les demandes
    attendre les opérandes
    faire le calcul
    attendre les demandes
    envoyer les résultats
tant que vrai
```

Ce qui pour les trois mêmes opérateurs que ci-dessus nous donne le schéma suivant:



Et nous obtenons alors le graphe:



(temps de traitement = 10 unités)

Il apparaît donc qu'une simple anticipation des demandes permet d'accroître les performances de façon significative.

Pour accroître les performances, il est encore possible d'aller plus loin dans l'anticipation des demandes:

L'idée consiste à ne pas attendre d'avoir besoin des opérandes pour les demander aux cellules amonts, mais à envoyer les requêtes dés que tous les opérandes sont arrivés. Pour que cette méthode fonctionne correctement il faut prendre une des deux précautions suivantes :

- Une opérande ne doit pas pouvoir entrer dans la cellule tant que l'opérande précédente correspondante n'a pas été utilisé dans le calcul.

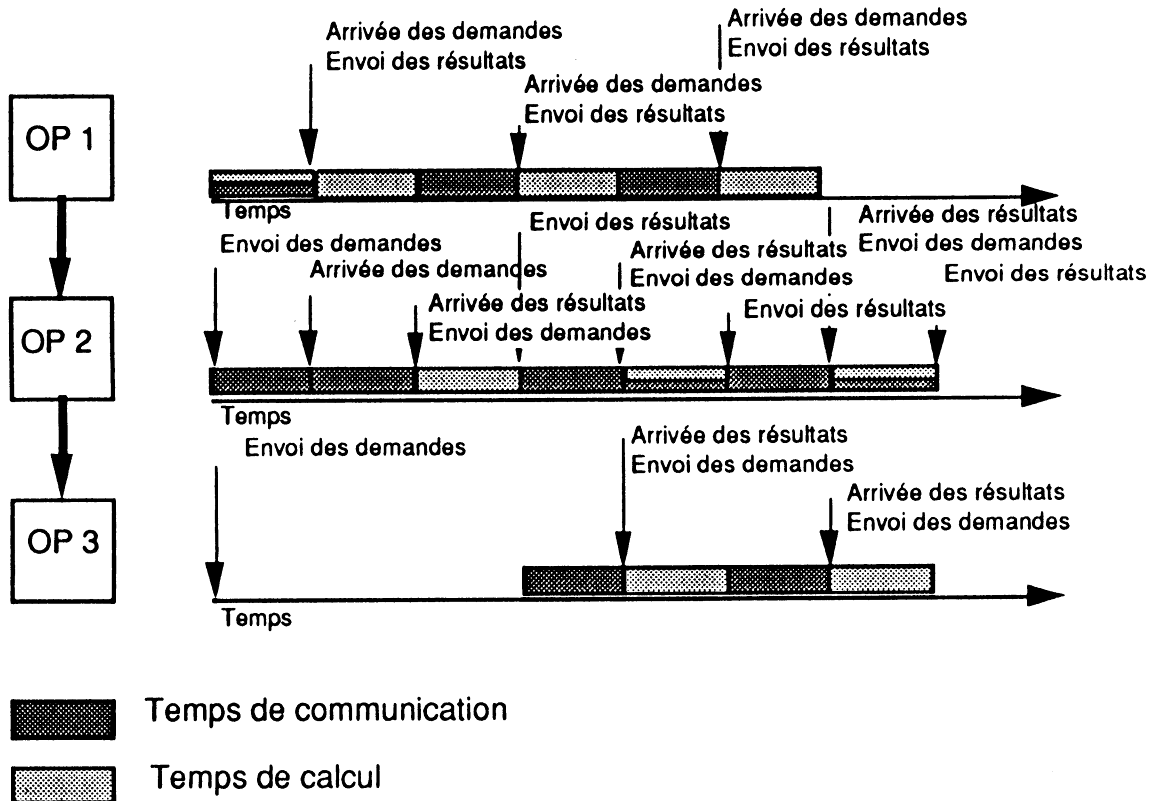
- On doit sauvegarder toutes les opérandes avant d'envoyer les demandes.

Si aucune de ces deux conditions n'est respectée, on court le risque de voir un opérande venir écraser la précédente avant qu'elle n'ait été lue (et utilisée) par le calcul. Pour éviter tout risque d'interbloquage du réseau de communication, il n'a pas été possible d'implanter de façon matérielle un mécanisme réalisant la première condition, c'est donc la deuxième solution que nous avons dû adopter.

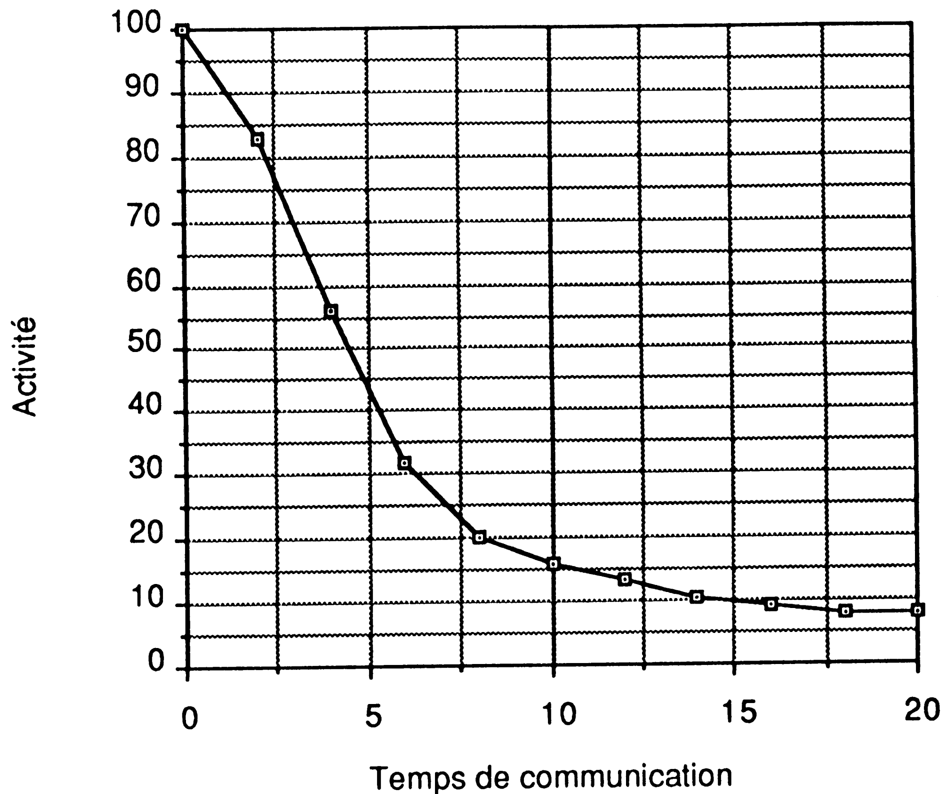
L'algorithme devient alors:

```
envoyer les demandes
répéter
    attendre les opérandes
    envoyer les demandes
    faire le calcul
    attendre les demandes
    envoyer les résultats
tant que vrai
```

Ce qui donne le chronogramme suivant :



Les performances obtenues par cette méthode montrent que non seulement on atteint une activité de 100% pour un temps de communication nul, mais que quelles que soitent les performances des communications l'efficacité est meilleure :



5.6 La génération des opérateurs LUSTRE

Cette étape essentielle de la compilation d'un programme LUSTRE sur notre architecture consiste à générer, pour chaque sous graphe placé sur une cellule, un programme équivalent. Cette étape est la plus dépendante de la machine cible car très liée au jeu d'instruction.

Comme nous l'avons vu (3.4.2), à l'issue du placement chaque cellule se voit attribuer un sous graphe. Celui-ci peut être de toute forme, complètement connexe ou non.

Deux méthodes apparaissent à ce niveau :

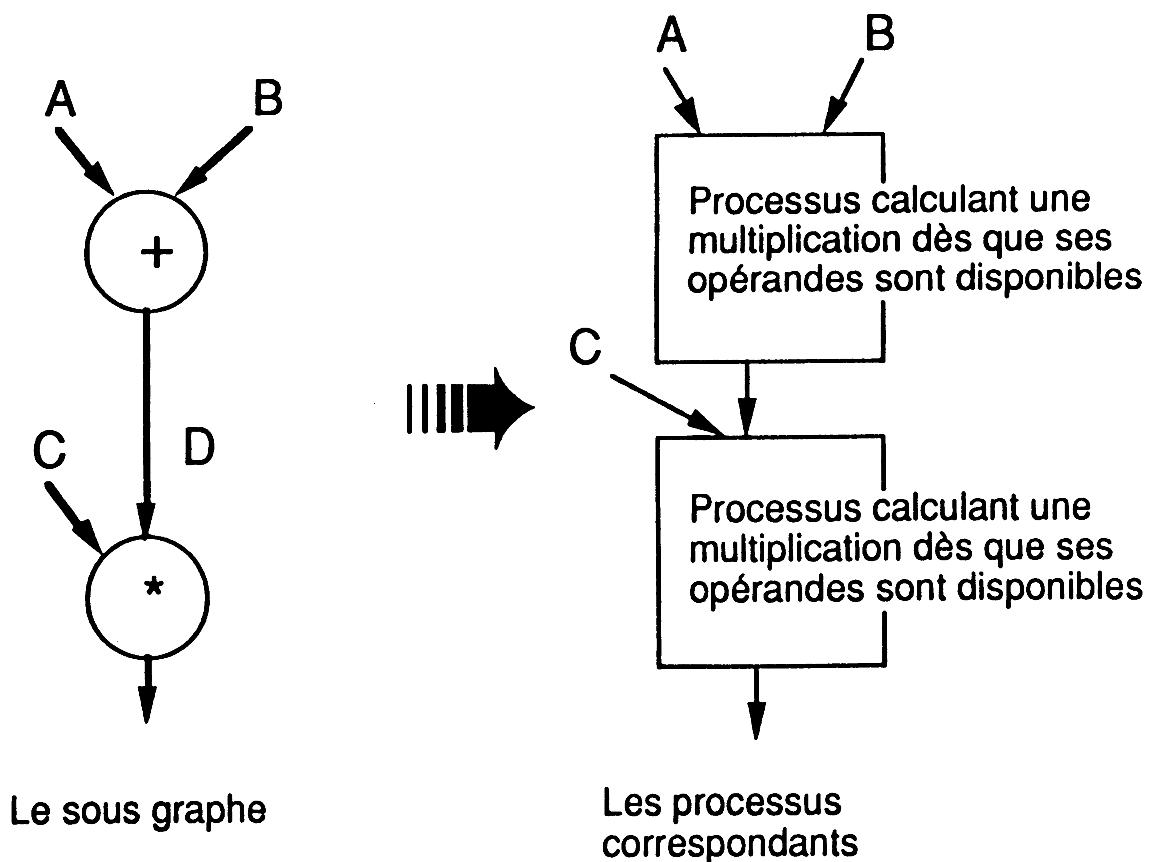
- On peut considérer que chaque opérateur est indépendant de ses colocataires placés sur le même processeur, on générera alors un programme dans lequel chacun se voit associer un processus indépendant des autres noeuds présents dans la cellule. Cette stratégie sera intitulée "méthode parallèle".

- On peut considérer qu'il existe dans le graphe des dépendances. On va alors tenter de fixer à la compilation un ordre d'exécution des opérateurs placés sur la cellule, et l'on générera un programme séquentiel correspondant. Cette stratégie sera intitulée “méthode séquentielle”.

5.7 La méthode parallèle

5.7.1 Présentation

Cette méthode de génération de code est la plus naturelle. Comme chaque opérateur d'un graphe dataflow agit indépendamment des autres, attendant de disposer de toutes ses opérandes pour pouvoir commencer son calcul, il est naturel pour reproduire ce comportement de vouloir associer à chacun un processus.



Le sous graphe

Les processus correspondants

La partie traitement de chaque cellule est un processeur séquentiel. Comme nous l'avons vu (5.6) nous avons renoncé, pour des questions de complexité, à implémenter de façon matérielle une partie traitement multi-processus. On ne dispose donc d'aucun mécanisme d'interruption ou de changement de contexte comme on peut en trouver dans les micro-processeurs classiques. L'instruction Try vient heureusement combler cette lacune.

Rappelons que l'instruction Try fonctionne de la façon suivante :

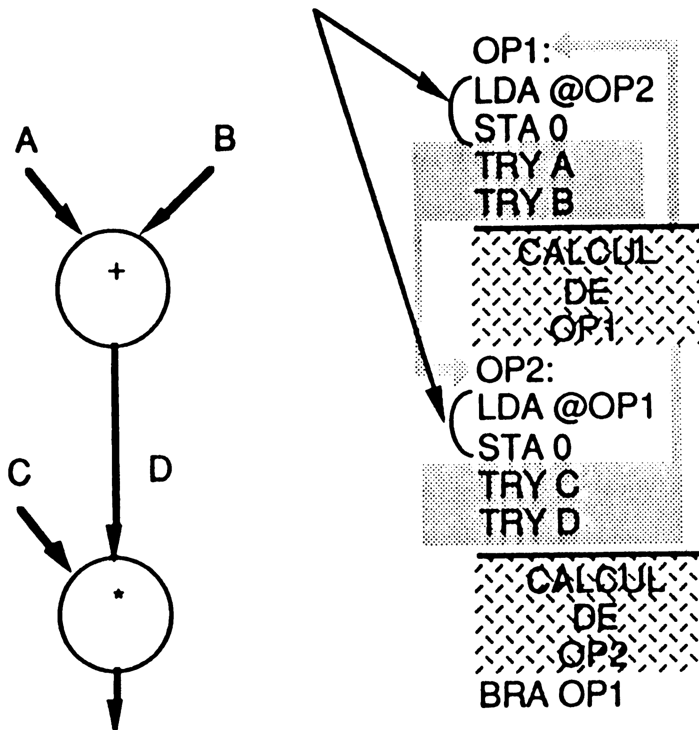
Lors de l'exécution de Try addr la partie traitement de la cellule teste l'état de l'adresse spécifiée :

- Si cette adresse est pleine, on continue l'exécution en séquence.

- Si cette adresse est vide, on effectue alors un saut à une adresse contenue dans un vecteur de branchement (constitué en fait de l'adresse 0 de la mémoire).

Pour les deux opérateurs donnés dans l'exemple ci-dessus, on a alors le schéma de fonctionnement suivant :

Stockage de l'adresse
de l'opérateur suivant
dans le vecteur de
branchement



Le sous graphe placé sur une cellule

Utilisation des try pour une attente active des opérandes de chaque noeud du graphe : Avant le calcul de l'opérateur on teste la présence de ces opérandes grâce à l'instruction TRY, si l'une d'elles est absente on effectue un branchement à l'opérateur suivant

Cette méthode permet une implémentation aisée et compacte d'un pseudo-parallélisme en laissant le soin à la cellule qui est en attente active de tester alternativement si les opérandes des différents "processus" sont arrivés; néanmoins elle oblige à faire le test de présence de tous les opérandes (y-compris les messages de synchronisation) avant de commencer l'exécution de l'opérateur. Des deux stratégies performantes d'émission des messages de synchronisation vues ci-dessus (3.4.5) seule la première est utilisable, car la deuxième ne teste la présence des "demandes" qu'après le calcul de l'opérateur.

Algorithme N°1	Algorithme N°2

répéter
    envoyer les demandes
    attendre les opérandes et
    les demandes
    faire le calcul
    envoyer les résultats
tant que vrai
	envoyer les demandes
	répéter
	    attendre les opérandes
	    envoyer les demandes
	    faire le calcul
	    attendre les demandes
	    envoyer les résultats
	tant que vrai

5.7.2 Intérêt et limitation

Avec cette méthode, l'ordre d'exécution des différents opérateurs placés sur une cellule n'est pas fixé à la compilation; il dépend uniquement de l'ordre d'arrivée des différents messages. On peut donc dire que l'on est dans ce cas fidèle à la stratégie dataflow, ou de séquencement des opérations par les valeurs. Les avantages de cette implantation fidèle de l'approche dataflow découlent en grande partie de leur modèle :

- Chaque opérateur placé sur une cellule s'exécute indépendamment des autres affectés à la même cellule; il dépend uniquement de l'arrivée des opérandes. Aucun ordre d'exécution décidé arbitrairement à la compilation ne vient gêner une exploitation maximale du parallélisme.

- On peut placer sans difficulté des opérateurs d'horloges différentes (et donc de fréquences d'exécution différentes) dans la même cellule. Les arrivées de leurs opérandes respectives viendront naturellement réguler leur fonctionnement.

Malheureusement le prix à payer pour la gestion d'un pseudo parallélisme sur notre machine est élevé :

- L'absence de mécanisme de gestion des processus et de changement de contexte câblé (comme on en rencontre dans le Transputer), conduit à une certaine lourdeur de programmation : à chaque opérande correspond une instruction Try pour tester sa présence. La longueur du programme associé à un opérateur s'en trouve ainsi augmentée.

- Les opérateurs placés sur une même cellule constituant des processus indépendants, on est obligé de les synchroniser par des envois de messages de la même façon que ceux disposés dans des cellules différentes. Ces messages ne sont évidemment pas des messages "physiques" transitant par le réseau, mais de simples positionnements de flags en mémoire grâce à l'instruction PUT.

5.8 La méthode séquentielle

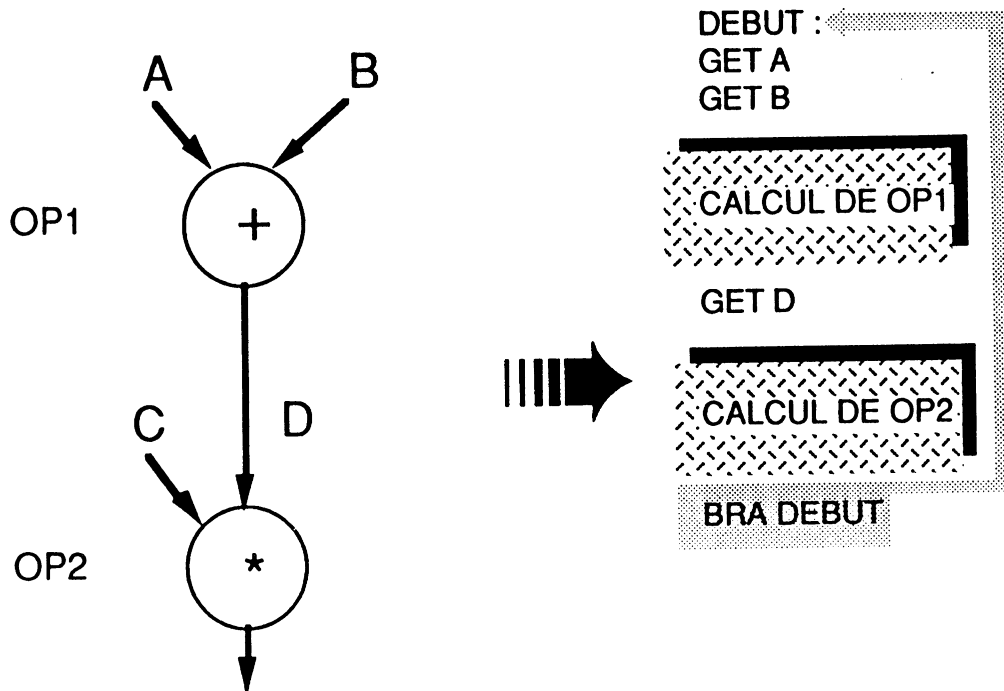
5.8.1 Présentation

Dans la méthode précédente chaque opérateur se voyait attribuer un processus dans la cellule. Cette méthode a l'avantage de la simplicité mais elle ne permet pas de tirer parti des dépendances présentes dans le graphe. Ainsi dans l'exemple ci-dessus, la multiplication ayant besoin du résultat de l'addition comme opérande, il est inutile d'essayer d'effectuer la multiplication avant d'avoir calculer l'addition. Il est donc tentant de vouloir utiliser ces dépendances entre opérateurs, pour simplifier et raccourcir le programme généré.

La méthode séquentielle vise à fixer à la compilation un ordre d'exécution entre les opérateurs afin d'utiliser les dépendances du graphe pour simplifier le code généré.

5.8.2 Ordonnancement des opérateurs

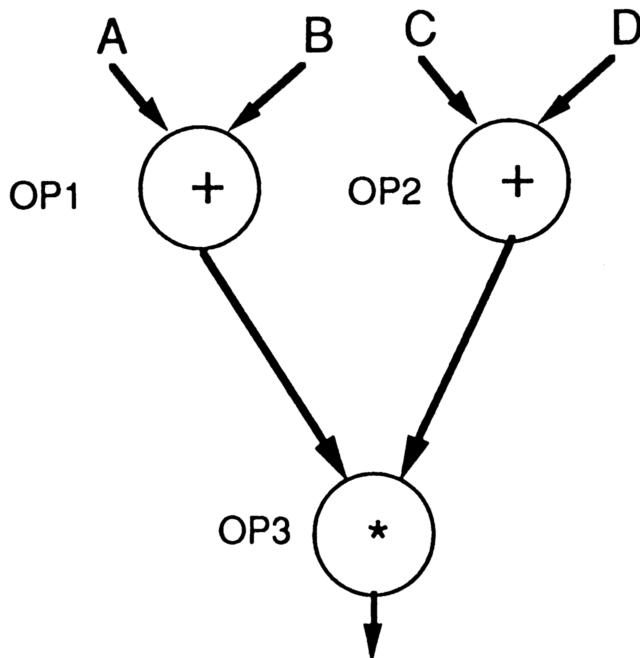
Il est facile de fixer un ordre d'exécution à deux opérateurs directement connectés et placés dans la même cellule :



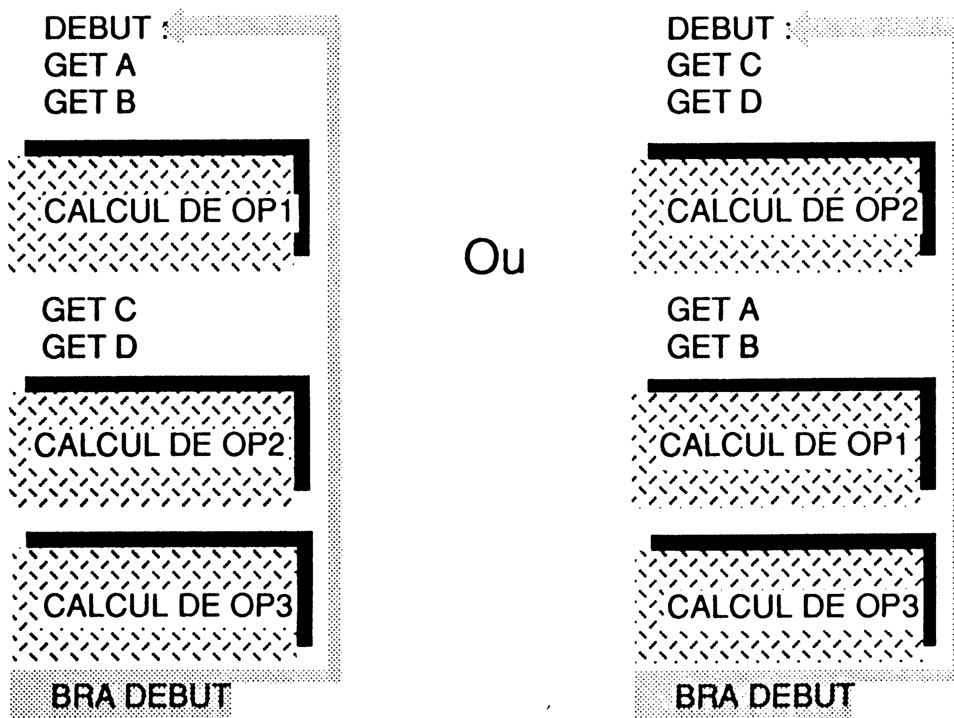
Le sous graphe placé dans la cellule

L'ordre d'exécution des opérateurs correspondants

Mais il est beaucoup plus délicat de fixer un ordre arbitraire d'exécution entre deux noeuds n'ayant pas de dépendance directe :

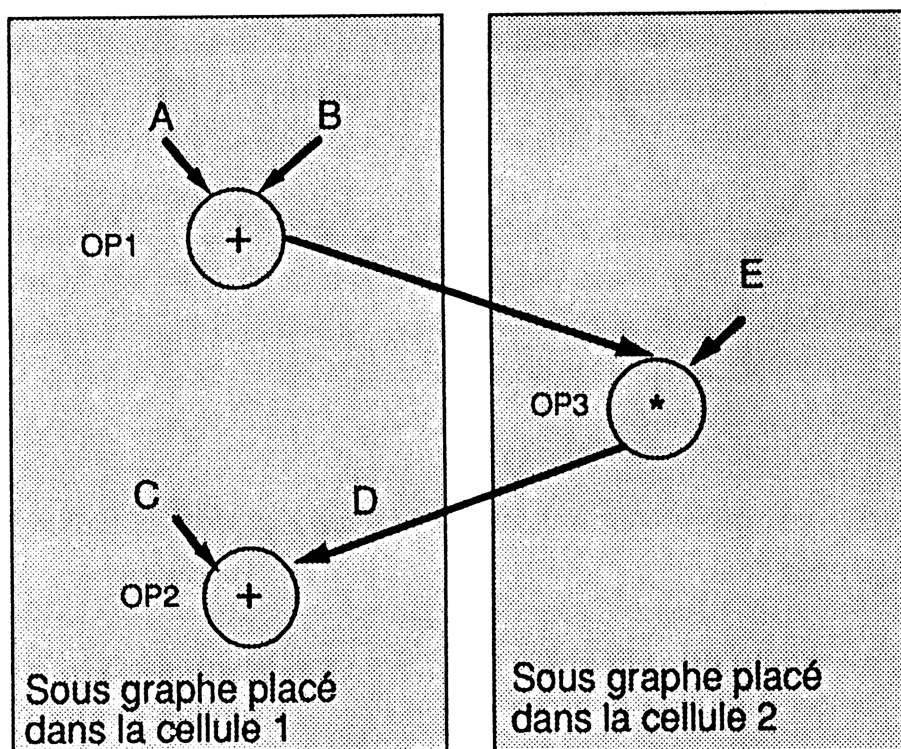


Ce sous graphe placé dans une cellule peut donner deux programmes séquentiels équivalents

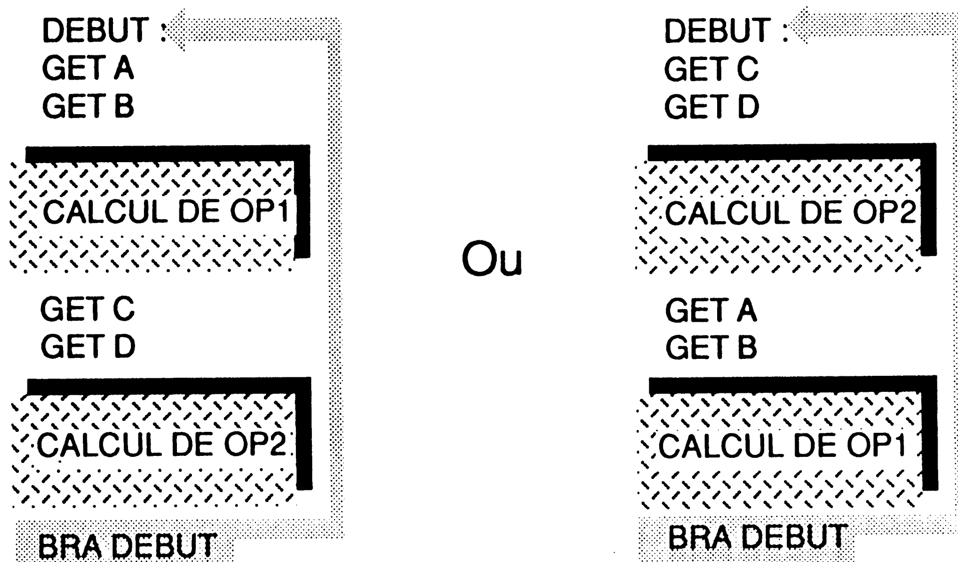


Il est impossible à la compilation de connaître l'ordre d'arrivée des opérandes A,B,C,D et donc de savoir lequel des deux opérateurs sera exécutable en premier. Cet ordre peut même dépendre des entrées du programme. On voit donc que fixer l'ordre d'exécution va introduire une perte de performance.

Pour connaître les dépendances il est nécessaire d'examiner le graphe en totalité, et pas seulement le sous graphe placé dans une cellule. Ainsi dans le cas suivant :

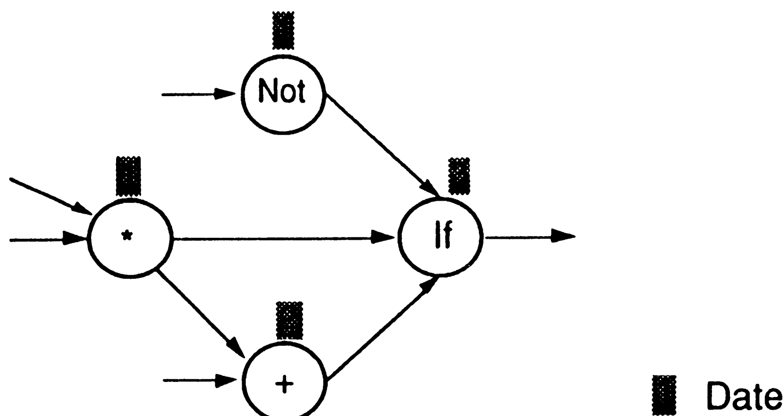


On a pour la cellule 1 deux ordres de génération possibles :



Il est évident que seul le premier programme fonctionne. Dans le deuxième, on attend D avant d'avoir calculer OP1, dont le résultat est nécessaire à la cellule 2 pour calculer D, et l'ensemble se bloque

Il faut donc, pour pouvoir générer un code fonctionnant correctement, fixer un ordonnancement des tâches prenant en compte la totalité du graphe. La solution retenue consiste à mettre en place une relation d'ordre entre les noeuds. L'utilisation d'un algorithme classique d'ordonnancement (de type PERTH), permet d'affecter à chacun une date qui définit sa position dans l'ensemble des tâches à effectuer.



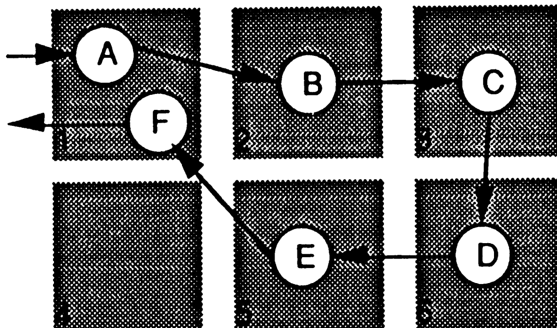
Une fois ces dates mises en place, même en observant une seule cellule à la fois, on a l'assurance que si une dépendance relie deux noeuds,

alors la date du noeud amont est inférieur à celle du noeud aval. Il suffit donc de générer le code correspondant aux opérateurs, par ordre de dates croissantes, pour s'assurer du fonctionnement correct du programme. Dans la pratique les dates ne seront pas seulement incrémentées d'une unité quand on passe d'un opérateur au suivant, mais elles prendront en compte le temps de calcul de chaque opérateur (voir annexe 1).

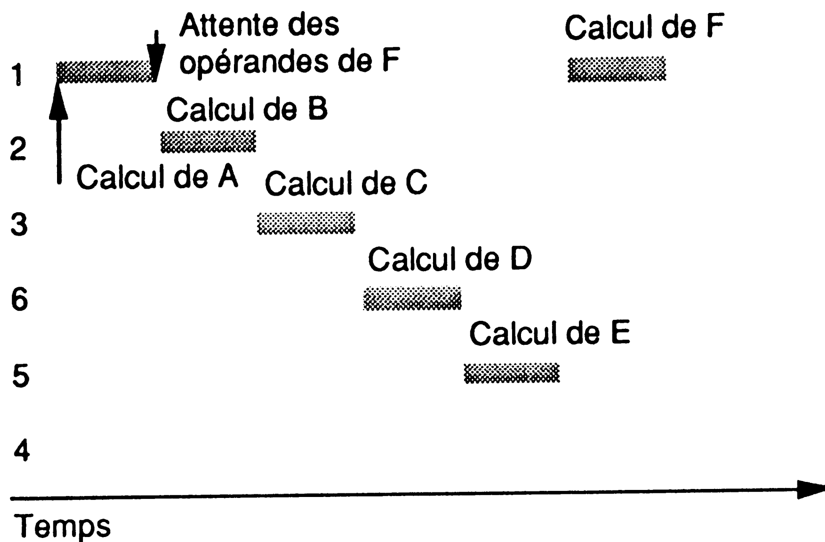
5.8.3 Le problème du placement

Si dans la méthode parallèle le placement vise à minimiser les longueurs de connexions et à répartir la charge de calcul entre les cellules, dans la méthode séquentielle il doit aussi préserver le "pipe line" naturel d'un graphe dataflow.

Imaginons que les hasards du placement nous donnent le résultat suivant :

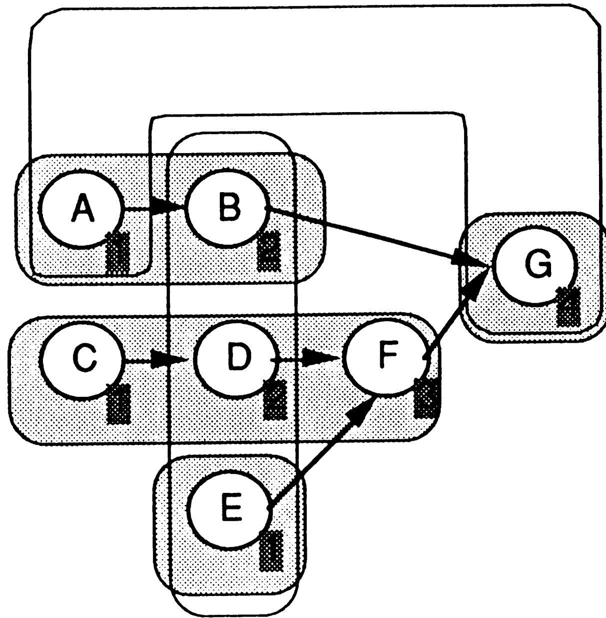


On à alors le chronogramme suivant :



Le résultat d'un tel placement est de réduire à néant tout recouvrement d'exécution entre les opérateurs. L'expérience nous a montré sur des problèmes réels, que si l'on ne tient pas compte de ce problème lors du placement les performances dépassent rarement quelques pour-cent d'activité.

Ce problème a été résolu en introduisant lors du placement la notion de position dans le pipe line. Les dates (voir ci-dessus) utilisées pour assurer un ordonnancement correct des opérateurs lors de la génération de code, constituent une caractérisation de la position des opérateurs dans le "pipe line". Lors du placement on essaie de ne regrouper sur une même cellule que des opérateurs dont les dates se suivent sans se recouper.



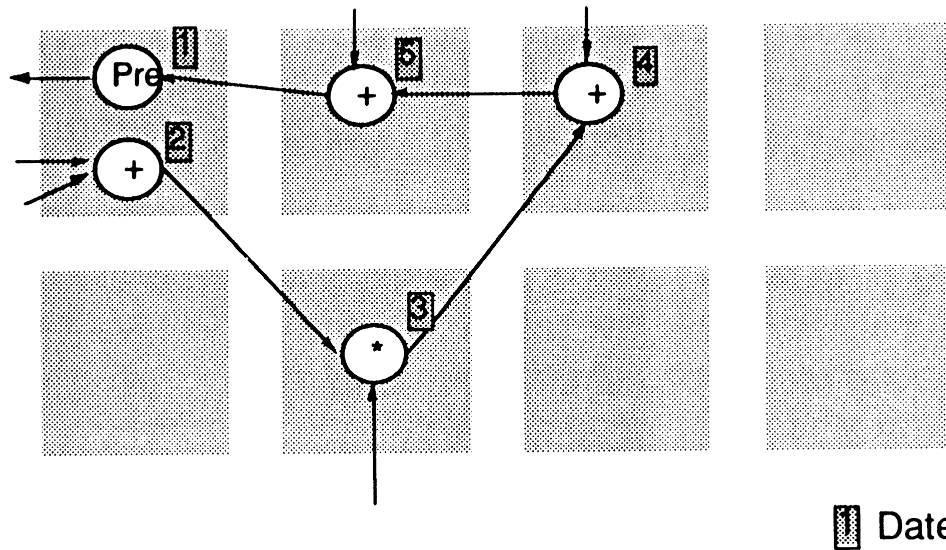
Regroupements correct

Regroupements degradant le parrallèlisme

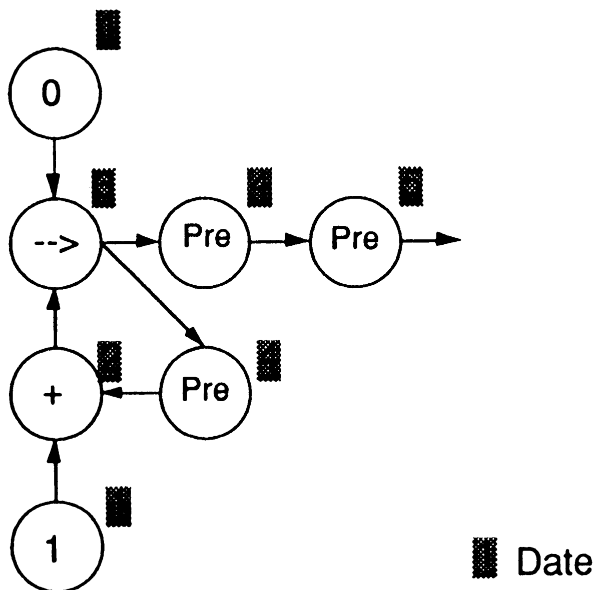
Date

Le problème est compliqué par la présence des noeuds PRE, ceux ci n'ont pas besoin d'une opérande pour leur premier calcul, il est donc tentant de leur attribuer une date égale à 1 (comme pour les entrées). Une étude attentive des cas suivants montre que cette idée est fausse.

En effet les noeuds Pre n'ont pas besoin d'opérande pour leur premier calcul, mais ils ne pourront commencer le deuxième avant d'avoir reçu une valeur. Dater ces noeuds comme des entrées conduit donc à les voir regroupés à l'issue du placement sur une même cellule qu'un voisin immédiat d'une entrée.



Comme on le voit dans ce schéma, on limite alors le degré de pipeline de la partie de graphe comprise entre l'entrée et le Pre à à un seul niveau. Pour éviter cette limitation, nous avons choisi de numéroter les "Pre" ne figurant pas dans une portion de graphe cyclique en suivant la même méthode que les autres noeuds.



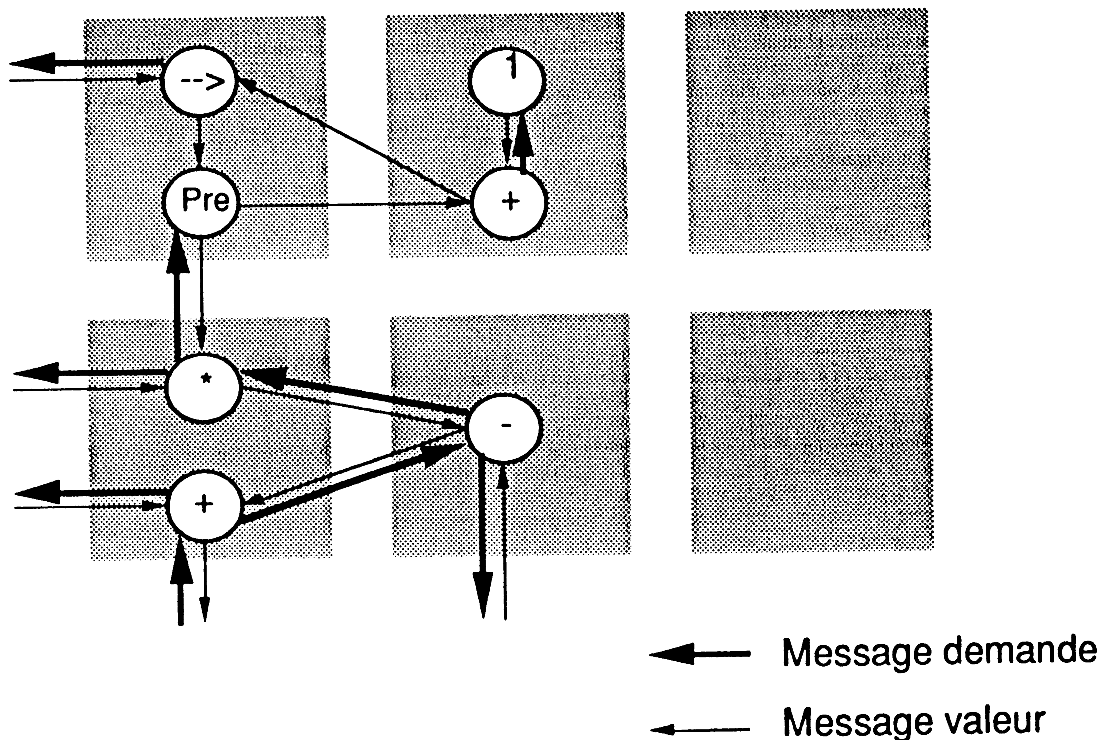
Le placement et le regroupement d'opérateurs sur le réseau est donc une tâche beaucoup plus délicate à assurer pour la méthode séquentielle que pour la méthode parallèle. Lors du placement, le respect du recouvrement entre les opérateurs détériore quelque peu l'optimisation des longueurs de communication entre les cellules.

5.8.4 Synchronisation

Comme nous l'avons vu, pour obtenir un fonctionnement correct du réseau, il est nécessaire de synchroniser le fonctionnement des cellules.

L'ajout d'un message de synchronisation permet de créer une boucle de contrôle afin d'empêcher l'arrivée d'opérandes supplémentaires tant que le traitement en cours n'est pas terminé. Dans les parties cycliques du graphe un tel bouclage existe naturellement, et il est alors possible de supprimer toute synchronisation supplémentaire.

Dans le cas de la méthode parallèle, le grain de parallélisme n'est pas la cellule mais l'opérateur; c'est donc chaque opérateur qu'il convient de synchroniser.



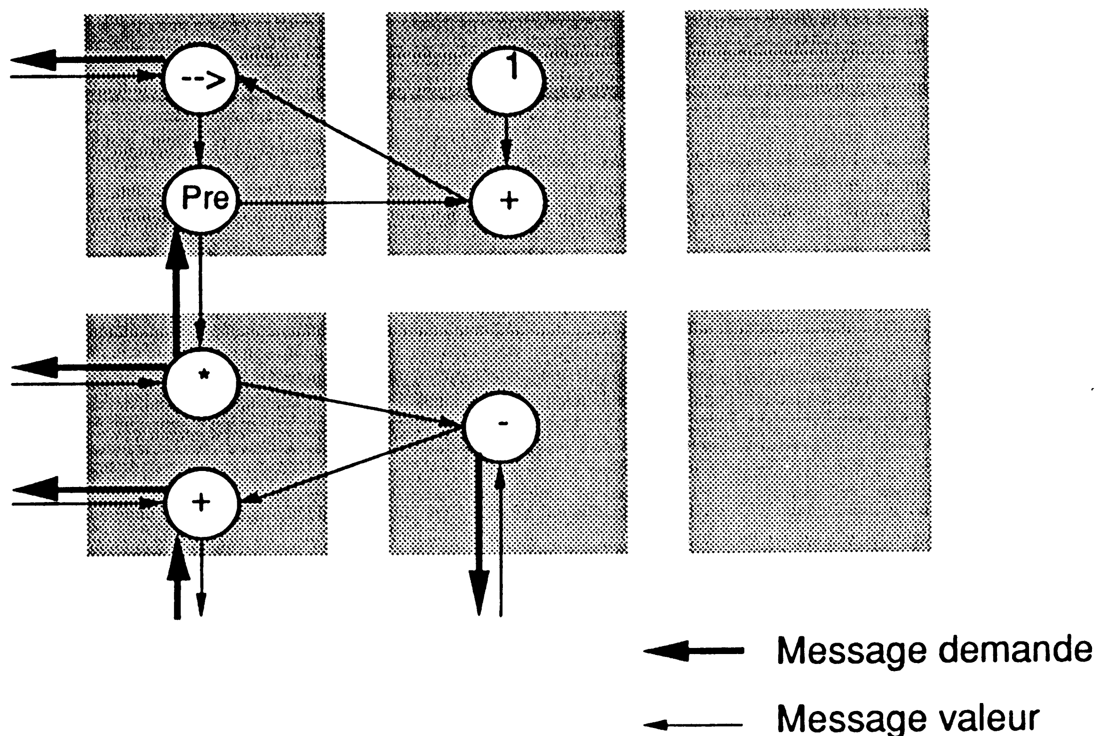
Dans la méthode séquentielle, l'ordre d'exécution des opérateurs est fixé à la compilation, il n'est donc plus nécessaire de synchroniser ceux placés sur une même cellule.

La nature séquentielle du programme généré pour chaque cellule permet de limiter la synchronisation à un échange bidirectionnel de

messages. Deux cas peuvent se présenter :

- Il existe un échange de valeurs bidirectionnel entre les cellules; dans ce cas les deux processeurs se trouvent naturellement synchronisés.

- L'échange de valeurs entre les cellules est unidirectionnel, il convient alors d'ajouter un message de synchronisation pour établir une communication dans l'autre direction.



On voit donc que la méthode séquentielle permet de simplifier la synchronisation et par là même de réduire le nombre de messages et la taille du code qui lui est nécessaire.

5.8.5 Le problème des horloges et des relais

Le langage LUSTRE offre la possibilité de définir des variables évoluant à différentes "vitesses", c'est à dire dont l'horloge de renouvellement est une sous-suite du cycle de base. Dans le cadre d'une génération de code de type séquentiel cette possibilité pose problème : un ordonnancement statique des opérateurs sur les cellules force tous les

opérateurs ainsi regroupés à fonctionner sur une même horloge. Parmi les solutions possibles, interdire tout regroupement entre noeuds d'horloges différentes apparait comme la seule réaliste. Cette contrainte supplémentaire lors du placement complique encore cette tâche délicate. Heureusement les exemples étudiés ne font qu'un faible usage de cette possibilité (en général pour extraire les résultats) et cette contrainte sur le placement reste donc très relative.

Nous avons vu que si l'on veut garder à cet outil toute sa généralité, il faut prévoir un mécanisme de cellules relais. Sur les deux méthodes présentées (3.4.4) celle consistant à trouver une place dans une cellule intermédiaire pour héberger un relais, semble irréaliste. Il est déjà impossible de garantir pouvoir trouver cette place sans autre limitation que la taille mémoire. Espérer parvenir à satisfaire les contraintes supplémentaires sur le placement qu'impose la méthode séquentielle, comme la position dans le pipe line ou les problèmes d'horloges, est peut-être illusoire.

La seule méthode utilisable est donc celle qui consiste à réserver lors du placement une cellule sur 49 pour servir de relais.

5.9 Comparaison des deux méthodes

Face à toutes ces contraintes imposées par la méthode séquentielle, la question se pose de savoir s'il y à plus a perdre dans les complications qu'à gagner dans la simplification du code. La réponse réside dans une étude comparative des deux méthodes.

Il est très difficile d'évaluer leurs performances respectives de manière théorique, pour les départager nous avons programmé deux générateurs de code (annexe 1) : un pour chaque méthode. Grâce à ces outils ils nous est facile d'obtenir leurs performances exactes pour des exemples donnés. Nous allons donc pouvoir examiner leur comportement sur plusieurs points pour les départager.

5.9.1 Longueur et optimisation du code

La longueur du programme généré constitue une caractéristique importante du programme : un code plus compact permet de placer plus d'opérateurs sur une cellule, et donc se contentera d'un réseau de taille réduite pour fonctionner. Le tableau suivant nous donne la taille du code (en nombre d'instructions) de différents exemples (voir annexe 2) générés avec les deux méthodes

	Méthode séquentielle	Méthode pseudo parallèle
F.F.T.	4806	6469
Tri	4182	7590
Produit de matrice	3058	4148

La différence est donc sensible entre ces deux méthodes, mais elle varie suivant les exemples. Il est donc intéressant pour obtenir une meilleure compréhension du phénomène, d'étudier sur quelques exemples la répartition du code par fonctions.

Description des légendes

Nous avons choisi de découper le code généré en 6 parties (5 pour la méthode séquentielle) : Env Sync, Rec val, Calcul, Rec Sync, Env Val, Attente.

- Env Sync correspond au code nécessaire à l'envoi des messages de demandes aux cellules amonts.

- Rec Val est la partie du programme concernant la réception et le stockage des valeurs (opérande)

- Calcul représente les instructions permettant l'évaluation des opérateurs à l'exclusion de toutes instructions de manipulation de données en mémoire.

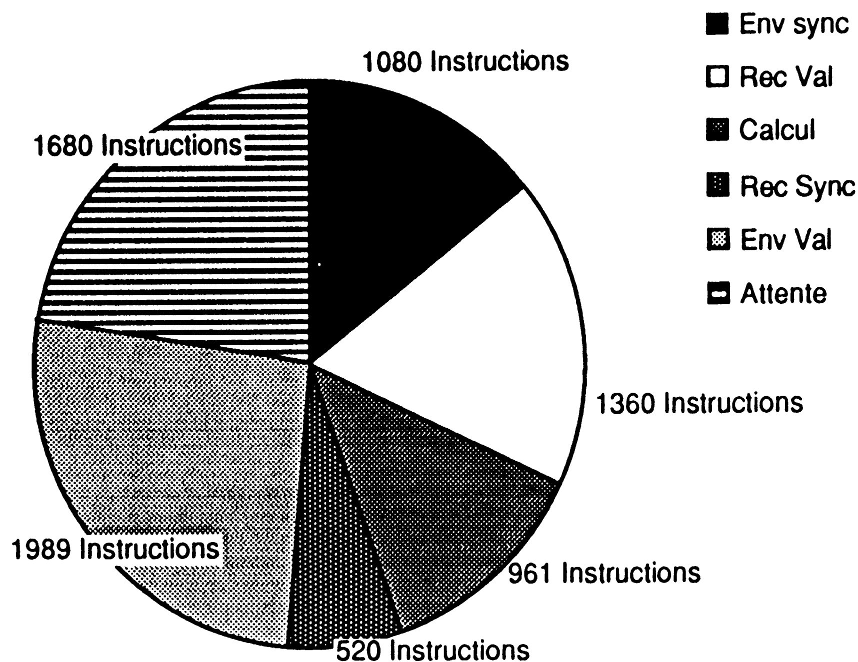
- Rec Sync est la partie du programme concernant la réception des messages de demandes.

- Env Val correspond au code nécessaire à l'envoie des messages résultat aux cellules aval.

- Attente est le code utilisé par la méthode parallèle pour gérer le pseudo parallélisme nécessaire à son fonctionnement.

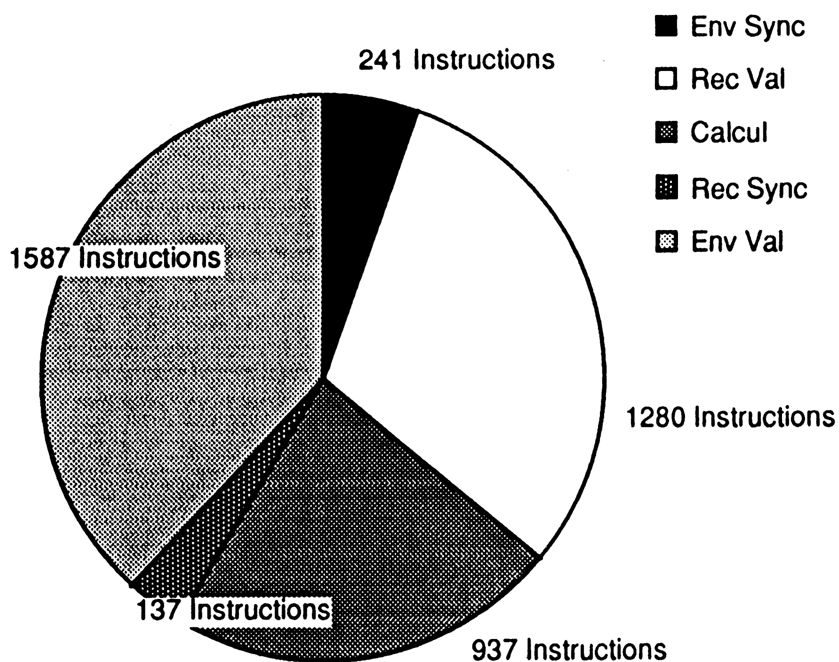
Un exemple de découpage de programme avec ces 6 zones est donné dans l'annexe 2.

Ainsi le programme de tri généré avec la méthode parallèle, nous donne la répartition suivante :



Longueur totale du programme : 7590 instructions.

Avec le même exemple, placé sur un réseau de même taille, mais en utilisant la méthode séquentielle nous obtenons :

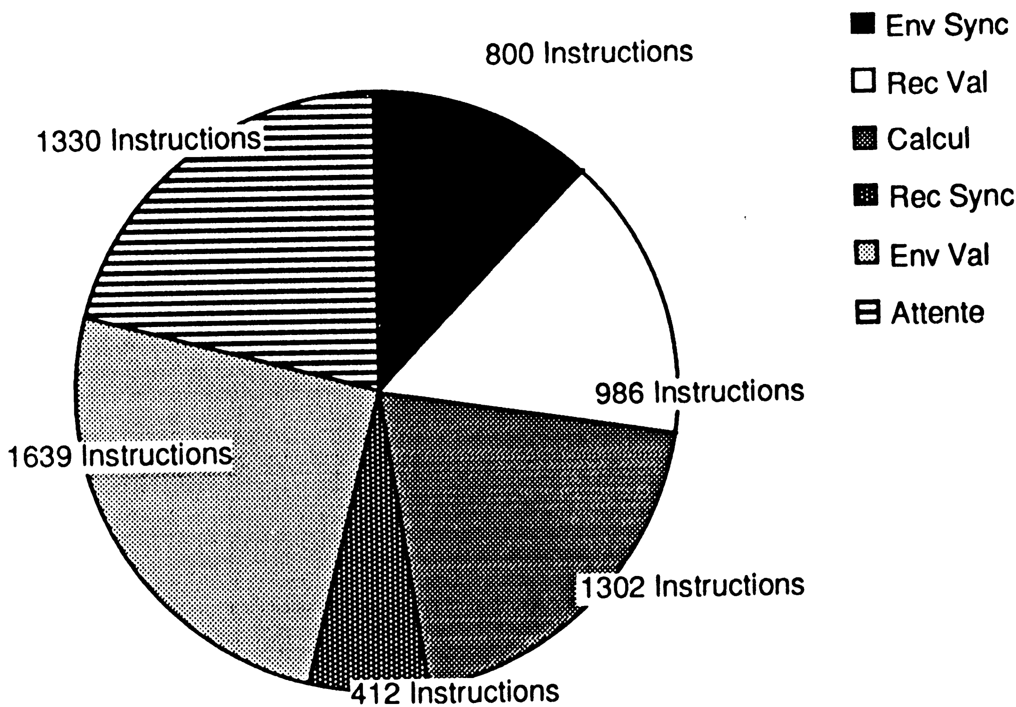


Longueur totale du programme : 4182

La différence de taille du code entre les deux méthodes est particulièrement sensible pour les instructions de synchronisation et, plus légèrement pour les instructions de communication de valeurs. L'observation de ces graphes nous montre dans les deux cas l'importance de la part relative du code dédié à la communication. Ceci est facilement compréhensible si l'on sait (Annexe 2) que l'essentiel de l'algorithme de tri choisi réside dans la communication de valeurs, et que lès calculs se limitent à des comparaisons.

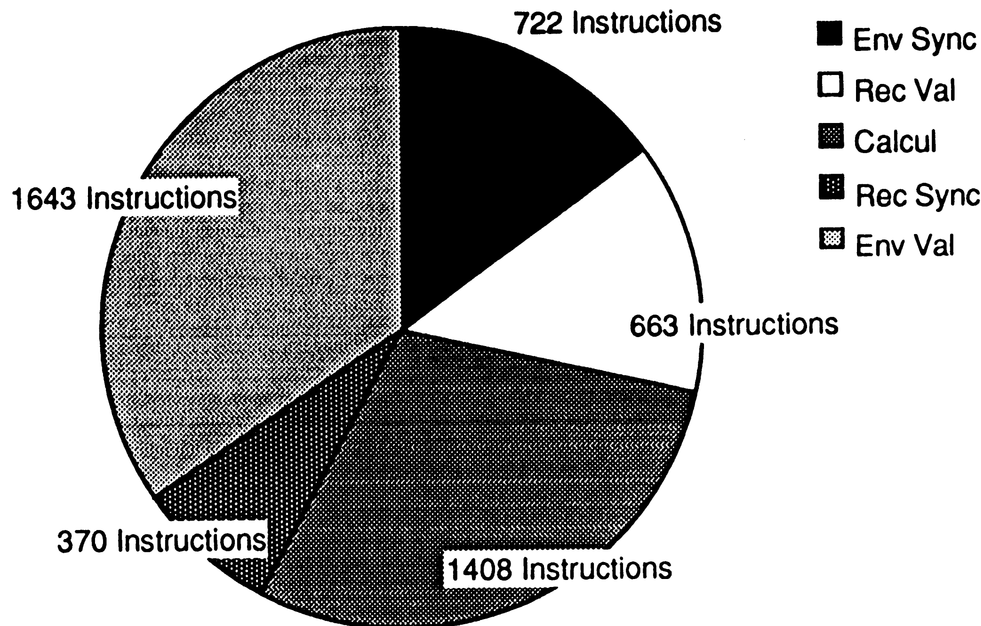
Le même principe de découpage appliqué à la transformée de Fourier nous donne:

Pour la méthode pseudo parallèle



Longueur totale du programme : 6469

Pour la méthode séquentielle :



Longueur totale du programme : 4806

La répartition par fonctionnalité dépend donc de la nature du programme source. Il est vraisemblable que si nous avions pu effectuer la même opération avec des calculs en virgules flottantes (pas encore implantés dans la version actuelle de nos outils : Annexe 1) la part du code dédiée au calcul en aurait été fortement accrue.

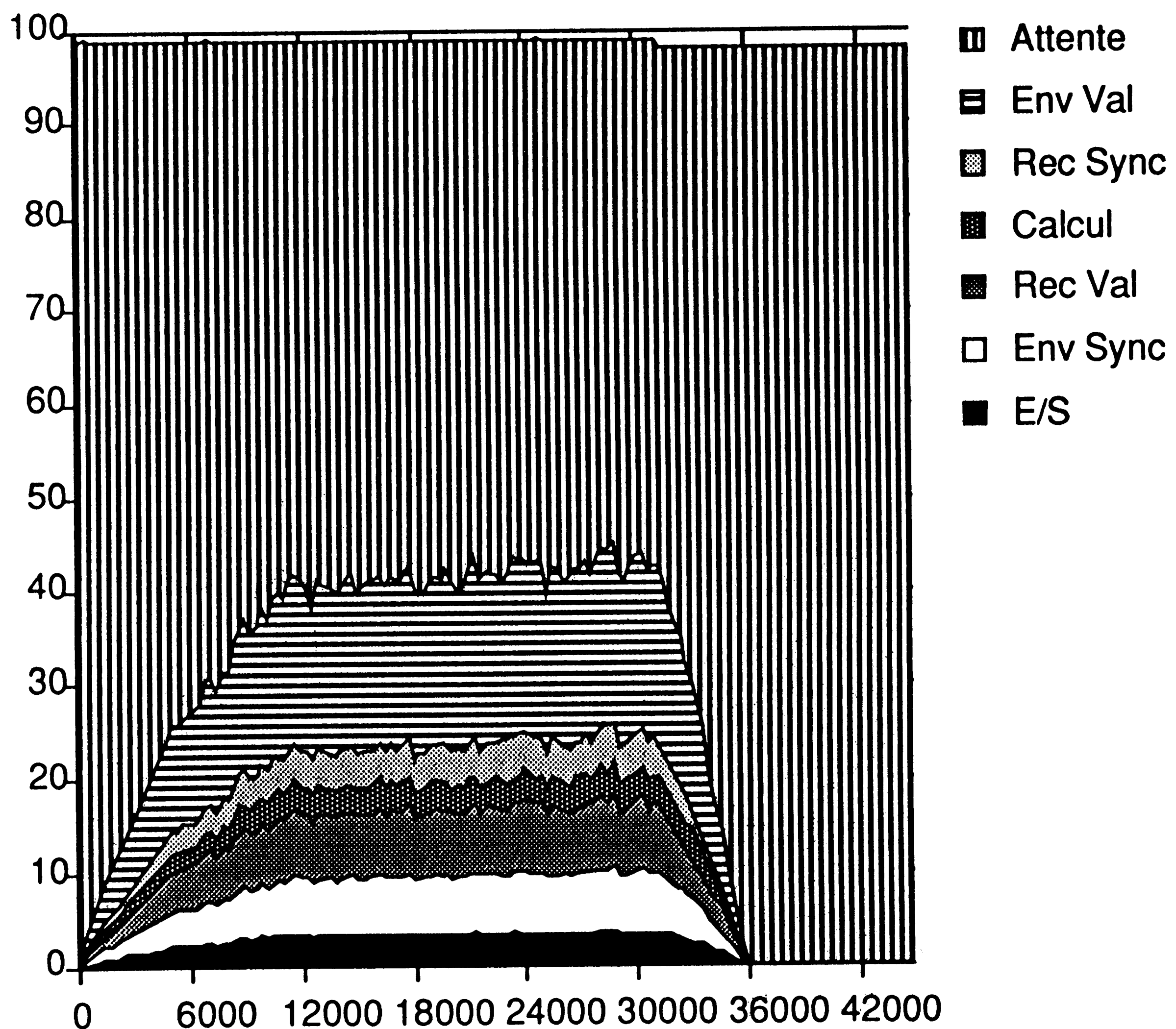
Du strict point de vue de la longueur du code, la méthode séquentielle apparaît donc comme largement supérieure à la méthode "pseudo-parallèle".

Pour les évolutions futures de ces deux outils, il faut noter que si le générateur de code "pseudo-parallèle" ne paraît pas pouvoir beaucoup évoluer, il en est tout autrement du deuxième outil. La simplification du code généré grace à l'absence de méchanisme de géstion du pseudo-parallèlisme, permet d'appliquer dans ce cas des techniques d'optimisation classiques : utilisation de modes d'adressages post-incrémentés, passages de paramètres par l'accumulateur ...

5.9.2 Performances temporelles du code

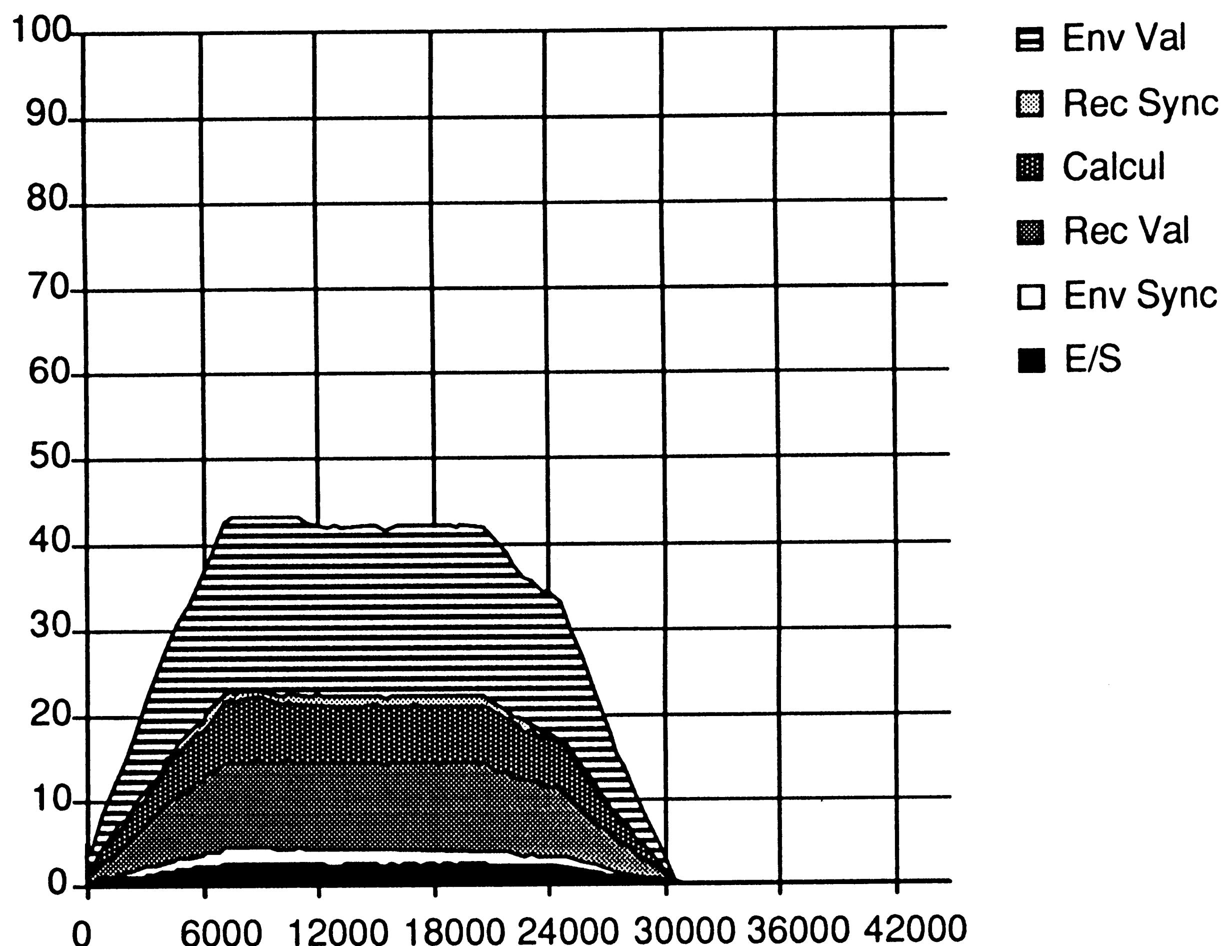
Si la composition et la taille du programme généré constituent un premier critère d'appréciation des deux méthodes, le point le plus important pour toute évaluation reste la vitesse de calcul. Pour permettre une comparaison aisée nous avons compilé des programmes identiques avec les deux outils de génération de code.

Ainsi pour le programme de tri on obtient avec la méthode parallèle le graphe d'activité suivant :



Le léger décrochement de l'attente vers la fin du calcul est dû au blocage des cellules communicant avec l'ordinateur hôte, celui-ci n'ayant plus de valeur à leur fournir en entrée.

La génération d'un code séquentiel pour le même exemple nous donne un résultat meilleur :

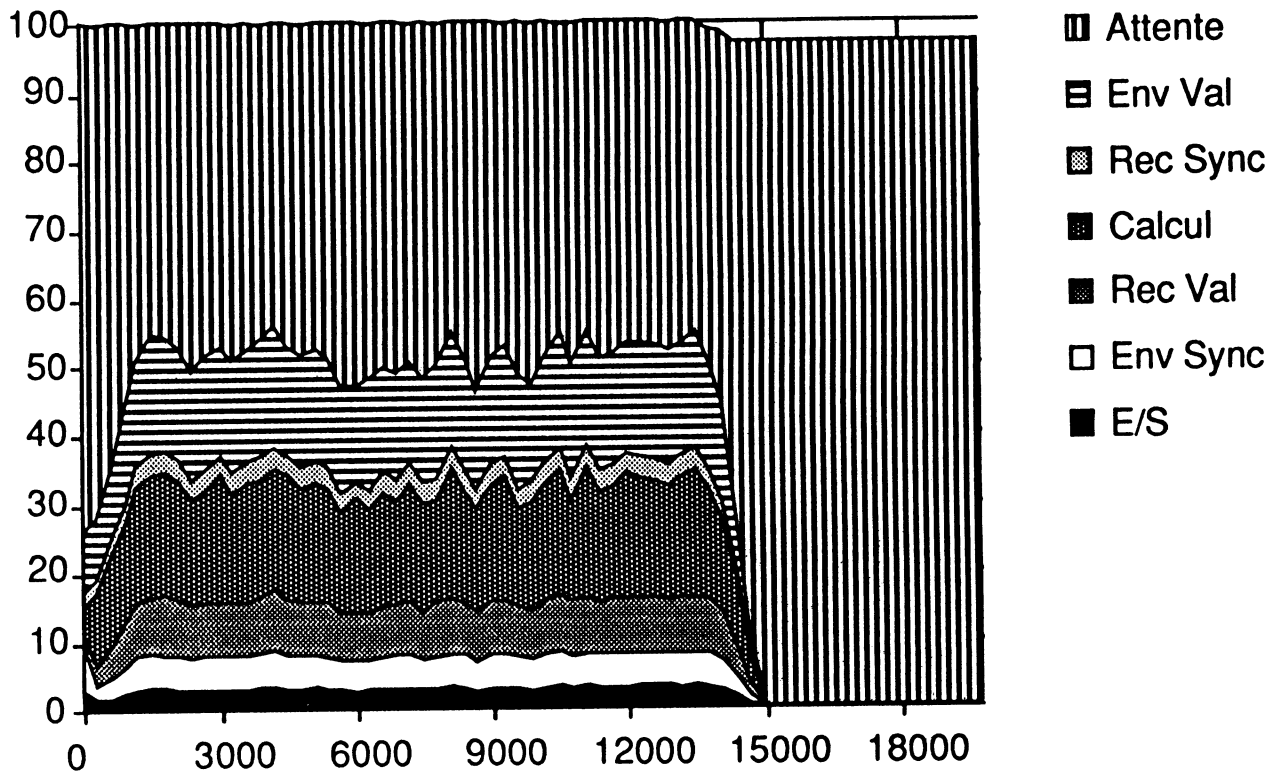


Ainsi pour le programme de tri, qui comporte de nombreux opérateurs de petite taille, la réduction du code nécessaire à la synchronisation et la suppression de celui utilisé pour simuler le parallélisme dans la cellule, compensent largement la baisse d'activité due à l'ordonnancement statique des opérateurs.

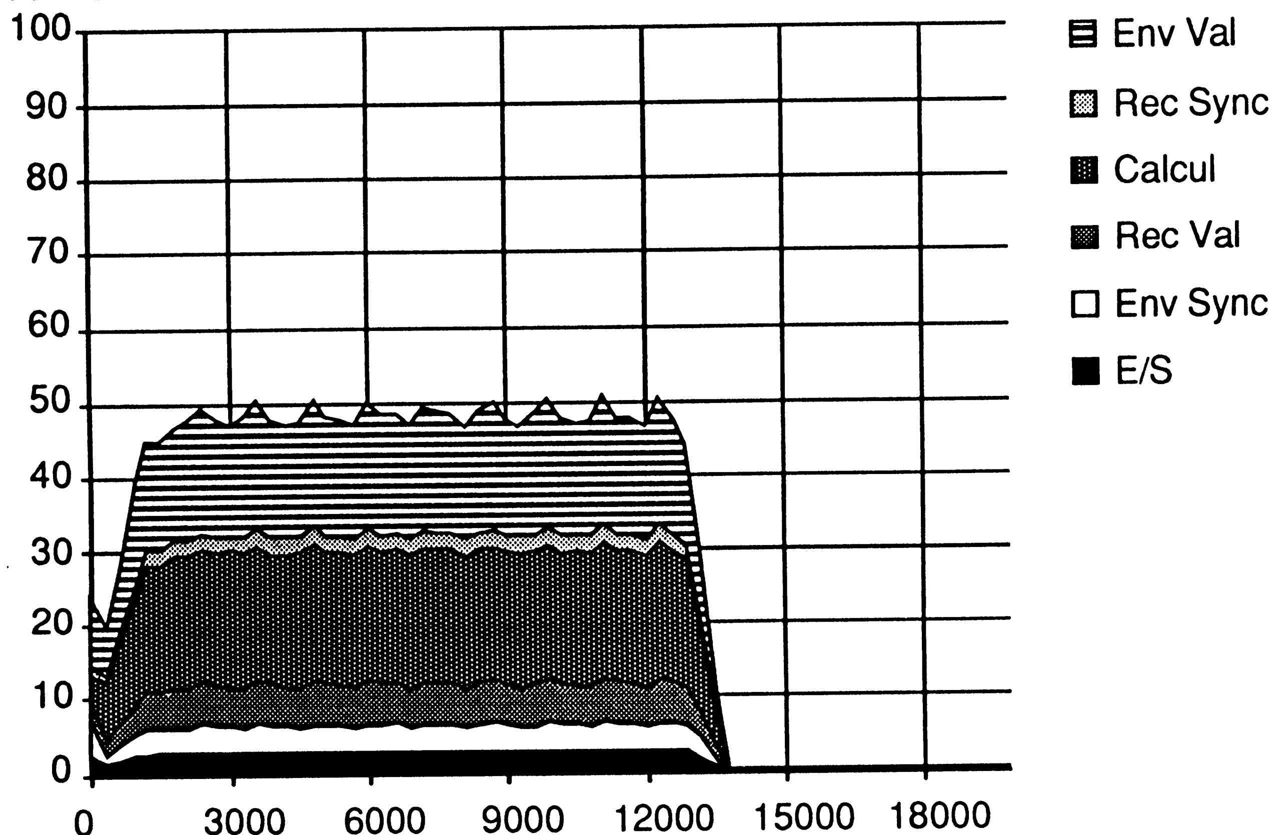
Mais pour un programme dont les opérateurs sont plus "gros" et donc plus longs à calculer, la gestion du parallélisme ne deviendra t'elle pas alors négligeable? Pour obtenir un premier élément de réponse, il est intéressant de faire la même opération sur un exemple utilisant des

opérateurs arithmétiques telle la multiplication. Nous avons choisi de simuler un programme calculant une FFT.

Avec la méthode parallèle on obtient :



Une exécution séquentielle des opérateurs placés sur la cellule, nous donne:



On constate donc que comme on pouvait s'y attendre les différences de performances s'estompent quand les tailles des opérateurs croissent. Pour pousser ce raisonnement à l'extrême, il faut noter que si la taille des opérateurs est telle que l'on ne peut en placer qu'un par cellule (ce qui est le cas de l'arithmétique flottante), les programmes générés par les deux méthodes sont identiques; il en est donc de même pour leurs performances.

5.9.3 Sensibilité au jeu d'instruction

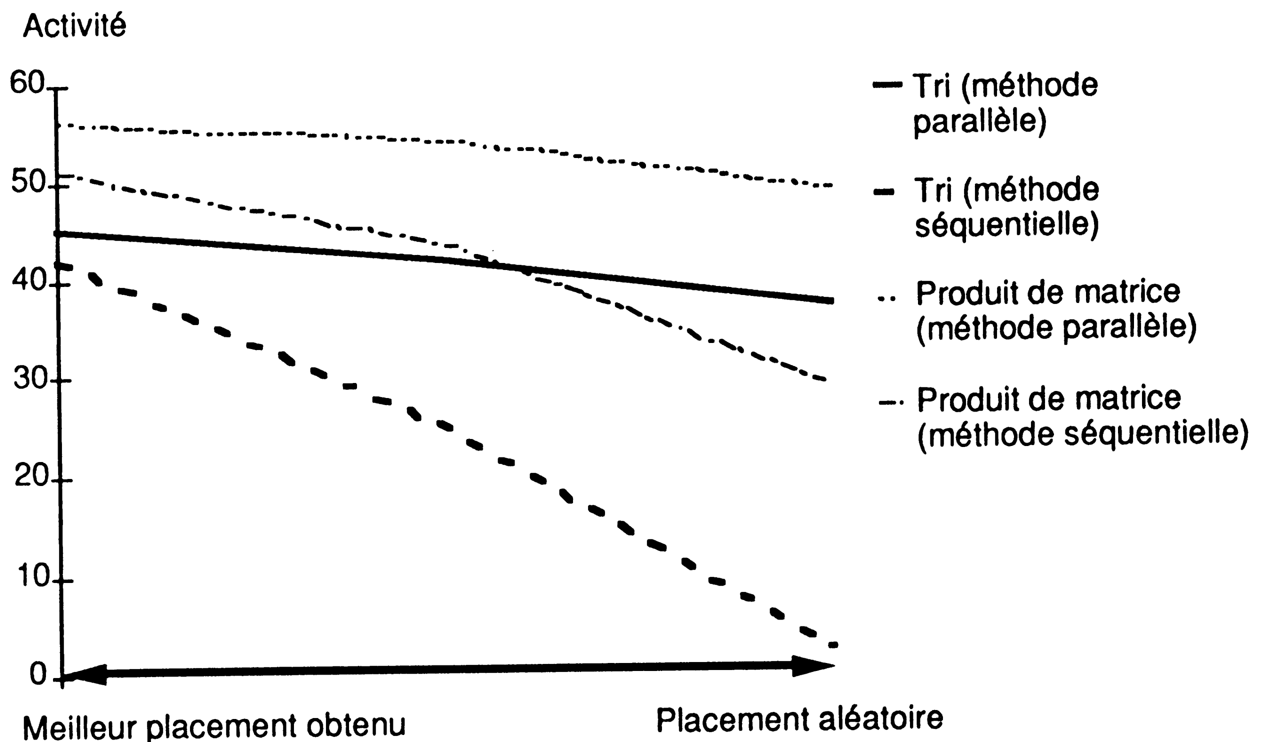
Le résultat que nous venons d'obtenir concernant les performances, ne doit pas être généralisé trop rapidement à toutes les architectures. Le mécanisme de gestion du "pseudo parallélisme" sur notre architecture est extrêmement lourd à manier. La gestion des processus par programmes grâce à l'instruction Try nécessite un nombre d'instructions, et donc de

cycles pour leur exécution, élevé. On dénombre un Try par opérande et un autre Try par message de synchronisation. On a donc un nombre d'opérations nécessaires à la gestion du parallélisme égal au degré de la cellule.

Il est facile d'imaginer que sur une architecture moins minimaliste, comportant par exemple un mécanisme câblé de gestion des processus, la taille du programme généré par nos deux outils se rapprocherait. Il est probable que sur une telle architecture l'intérêt de la méthode séquentielle ne soit plus aussi évident.

5.9.4 Sensibilité au placement

Pour évaluer la sensibilité des performances à la variation de la qualité du placement, nous avons simulé l'exécution de deux programmes avec différents placements :



Notre outil part d'une configuration aléatoire (placement aléatoire) et procède par échange de paires. En faisant varier le nombre d'itérations

(d'échange de paire) de l'algorithme il est possible de faire varier la qualité du placement, d'un placement aléatoire pour un nombre d'itération nul, jusqu'à un placement de qualité obtenu par un grand nombre d'itérations.

La méthode parallèle résiste donc mieux à la dégradation du placement. Il faut noter que ses performances varient peu, même pour un placement totalement aléatoire. La méthode séquentielle est beaucoup plus sensible, et les performances peuvent même s'effondrer totalement pour un exemple comportant un grand pipeline comme le tri.

5.9.5 Conclusion provisoire

Comme nous l'avons vu il est difficile de faire un choix définitif entre les deux approches quelque soit l'architecture. Sur notre machine où l'espace mémoire est très limité et la gestion de processus embryonnaire, l'exécution séquentielle des opérateurs placés sur une cellule nous apparaît pour l'instant, comme la méthode la plus intéressante.

5.10 Confrontation compilateur LUSTRE, programmation en assembleur : étude d'un exemple, le produit de matrice.

5.10.1 Introduction

La comparaison entre les deux méthodes précédentes n'est pas suffisante pour permettre une évaluation précise des performances de la machine. La seule référence pour situer notre outil est la programmation en assembleur du réseau. Il a donc été nécessaire d'écrire un même programme à l'aide de ces deux langages.

Le choix d'un programme de test est toujours difficile; nos critères de choix ont été les suivants :

- Un algorithmique simple et déjà parfaitement défini.

- Une topologie de l'algorithme en grille afin de ne pas trop compliquer la programmation en assembleur.

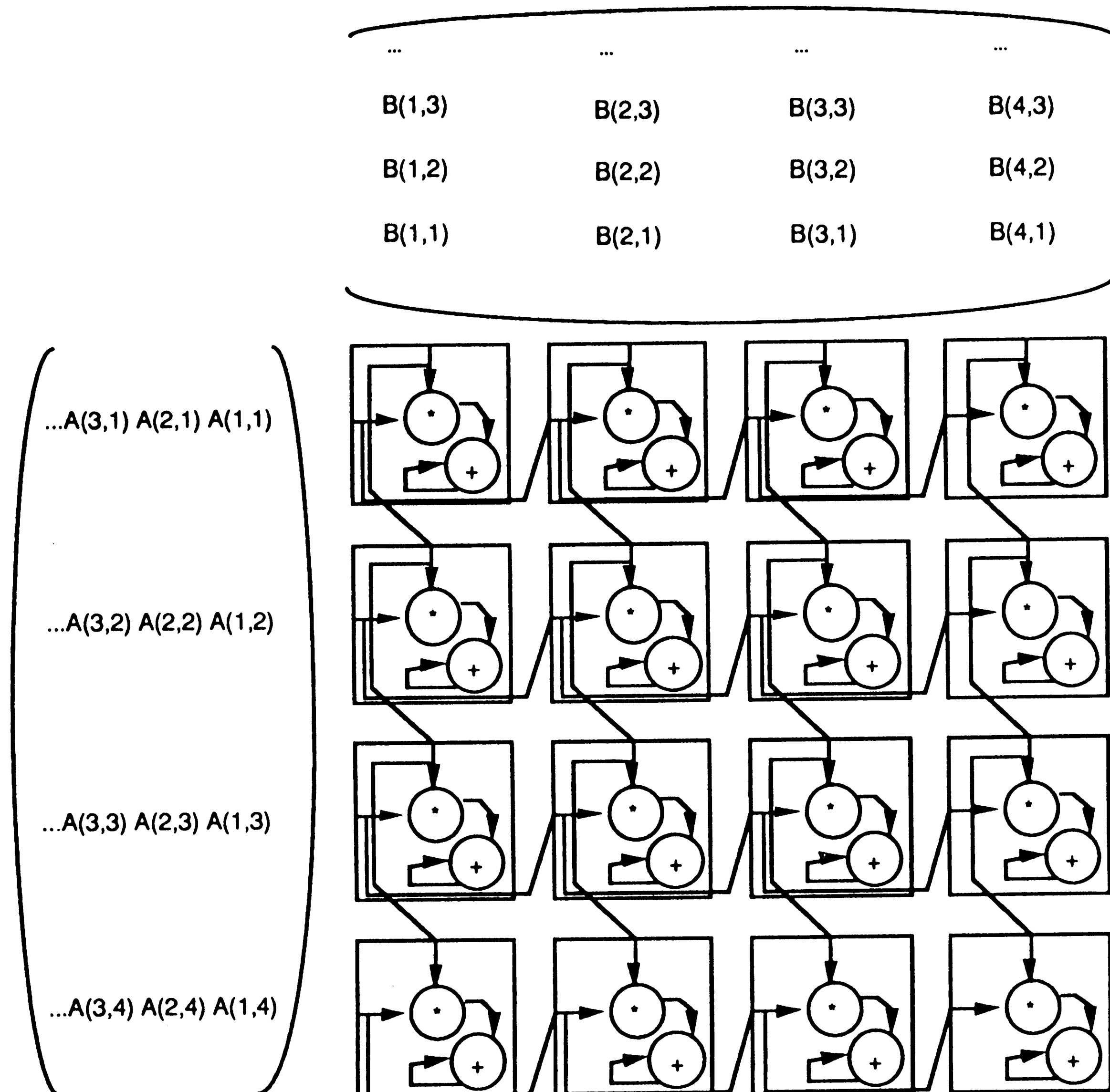
- La version actuelle de notre compilateur LUSTRE n'intégrant pas les opérations en virgule flottante, les variables devront être de type entiér.

Notre choix s'est porté vers l'algorithme classique du produit de matrice, pour simplifier encore plus, nous avons choisi de ne nous intéresser qu'au calcul du résultat et pas à sa sortie du réseau.

Nous sommes conscient du fait que ce choix est discutable, sa structure régulière en simplifie en effet grandement la programmation en assembleur.

5.10.2 Ecriture des programmes

L'implantation du produit de matrice sur machine parallèle est un algorithme classique de la littérature. Les deux matrices sont présentées sur deux côtés du réseau :



Il faut noter que, contrairement à la version systolique de cet algorithme, le caractère asynchrone de notre architecture rend un décalage des matrices opérandes inutile : il sera de fait réalisé par les opérateurs eux même.

Le listing des deux programmes sources de cet exemple (produit de deux matrices 5*5) est donné dans l'annexe 2.

La comparaison entre les deux écritures est donnée par le tableau suivant:

	Programmation en assembleur	Programmation en LUSTRE
Temps de développement :	≈4 heures	≈15 minutes
Temps de compilation (ou d'assemblage)	≈30 secondes	≈30 minutes principalement du placement par recuit simulé (sur Macintosh 2fx)
Taille du réseau nécessaire à l'exécution	5*5 cellules	6*6 cellules
Taille du programme source	180 lignes	36 lignes
taille du programme assembleur exécutable	180 lignes	596 lignes

La différence de taille entre les deux programmes assembleur exécutables est due au fait que dans la version assembleur le programme est le même pour chaque cellule (au effet de bord près voir Annexe 2), il n'est donc décrit qu'une fois en précisant qu'il s'adresse à l'ensemble des cellules. Le programme généré par notre compilateur LUSTRE est différent pour chaque cellule, il ne peut donc pas être compacté de la même façon.

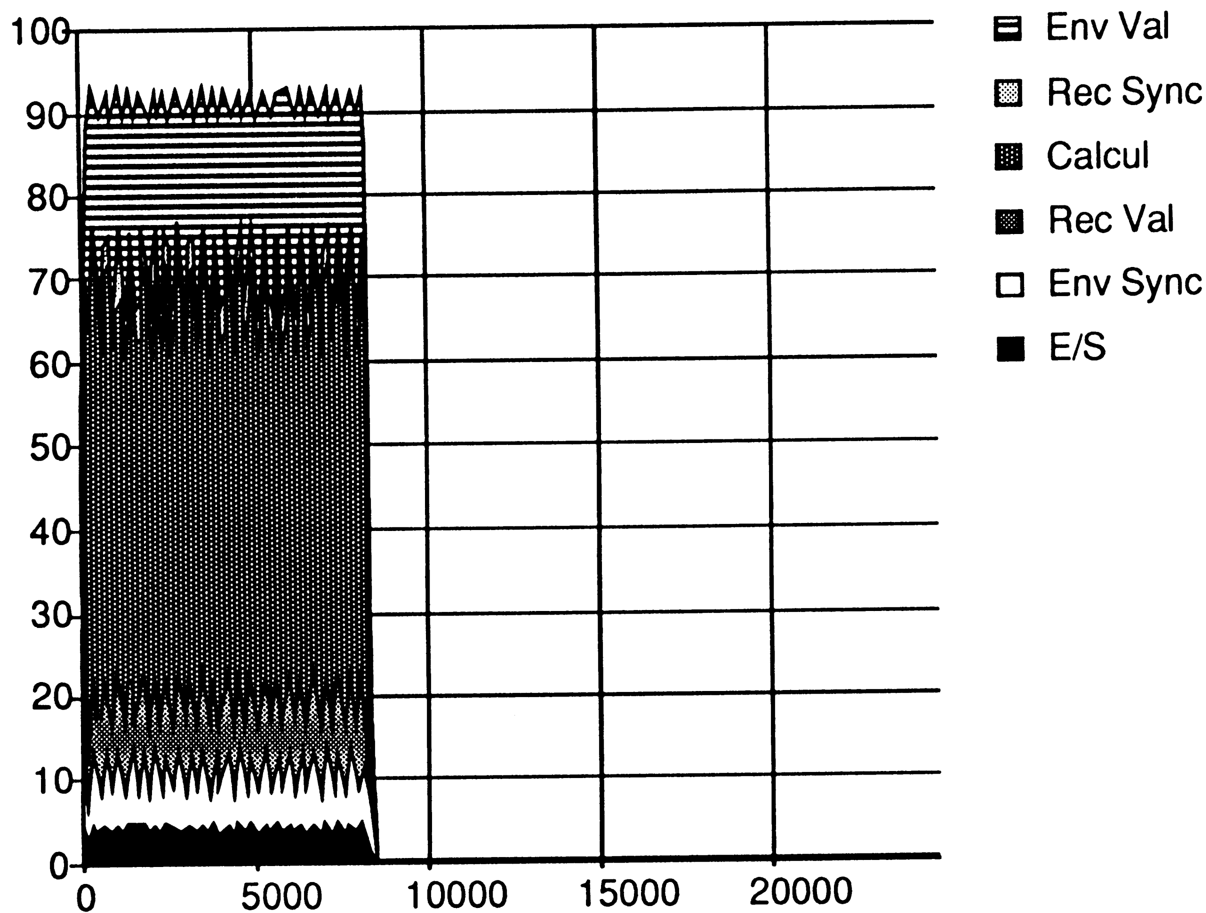
5.10.3 Performances respectives

Pour un jeu d'entrées comportant 120 valeurs (notre programme calcul alors la sous matrice 5*5 située en haut à gauche de la matrice résultat 120*120) on obtient les performances suivantes :

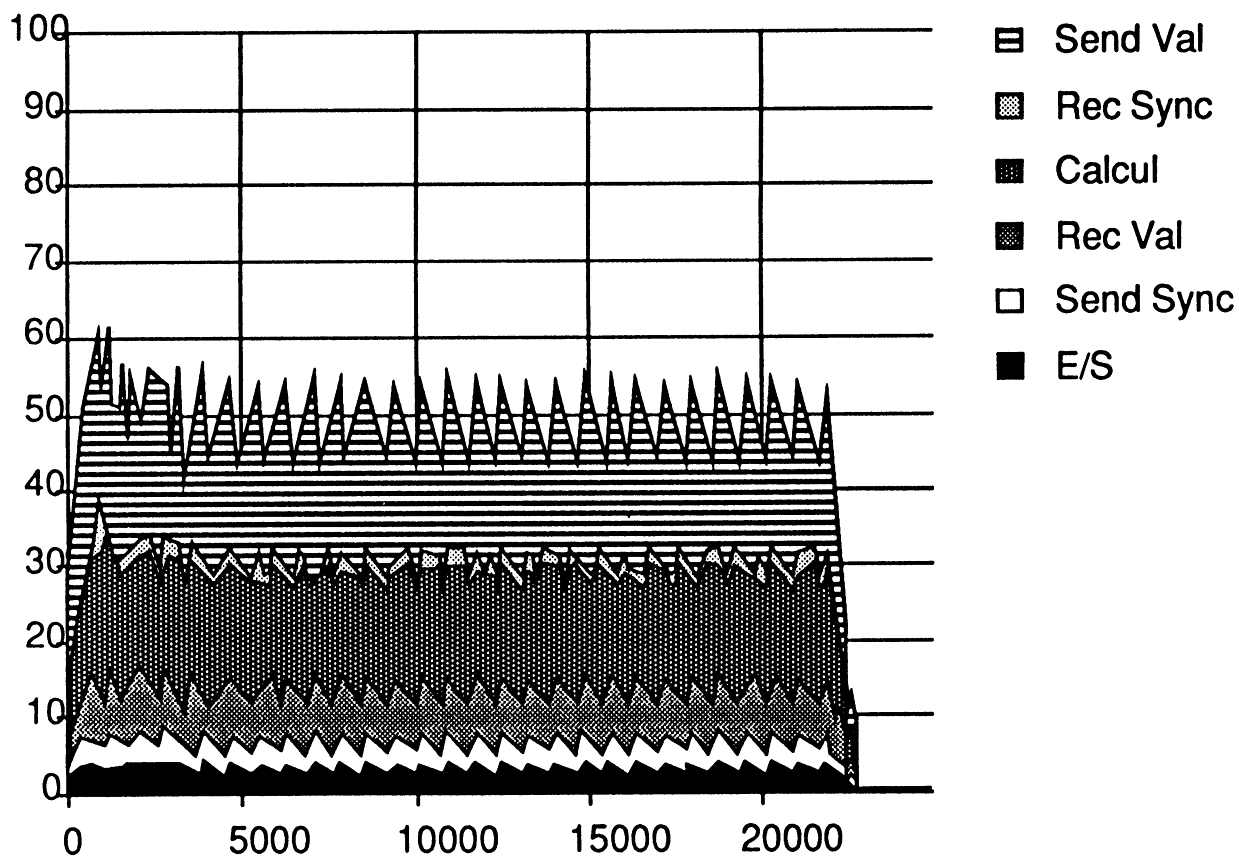
Les 10*120 éléments des matrices opérandes sont absorbés par le réseau en 8400 cycles d'horloge pour le programme assembleur contre 21273 cycles pour le programme LUSTRE. Le rapport de performances entre les deux méthodes est donc de ≈ 2,5. Le réseau utilisé par le programme LUSTRE est de 6*6 cellules (contre 5*5 pour le programme

assembleur), le rapport de performance ramené à une seule cellule est donc de 3,6.

L'exécution de la version assembleur du programme sur notre machine nous donne le chronogramme de fonctionnement suivant :



La version Lustre nous donne le résultat suivant :



On voit dont que les communications représentent la même portion du travail des cellules pour les deux programmes. La nette baisse d'activité du réseau (45% contre 90%) se ressent surtout dans le pourcentage de calcul (18% contre 45%).

5.11 Conclusion

Notre outil de compilation à permis une écriture plus compacte et plus rapide d'un programme que le simple assembleur. Le gain en temps de développement et en lisibilité des programmes est considérable. Le rapport de performance de 2,5 (3,6 si l'on ce ramène à un même nombre de cellules) apparait comme un prix à payer raisonnable en regard de ces facilitées.

Les performances de ce premier prototype logiciel de compilateur LUSTRE sur notre architecture répondent donc au cahier des charges que nous nous étions fixé (3.1)

De nombreux développements futurs de ce travail sont possibles, parmi les plus intéressants il faut citer :

- Le portage de l'ensemble du compilateur sur la futur machine hôte du réseau (Actuellement l'ensemble des outils nécessité 2 machines distincts voir annexe 1)

- L'ajout à notre compilateur de la possibilité d'utiliser dans des programmes LUSTRE des noeuds externes programmés en assembleur. Cette possibilité doit permettre au programmeur d'ajouter au langage de manière efficace, des opérateurs absents ou difficiles à programmer.

6. Conclusion

Le but de cette thèse était de définir une architecture massivement parallèle programmable, permettant par une programmation appropriée, de résoudre une grande variété d'applications. Nôtre second objectif (sur le quel j'ai personnellement travaillé) consistait à imaginer puis développer une méthode programmation adaptée à la fois à notre nouvelle architecture et à la classe d'algorithmes visée.

Ceci nous a amené à découper nôtre étude en deux grandes parties : la définition de l'architecture et la réalisation de notre outil de programmation. malgré ce découpage nécessaire à la bonne compréhension de ce mémoire, il est important de comprendre que ces deux développements ont été réalisés en parallèle, de manière à permettre une interaction entre ces deux réalisations.

Une architecture originale

Dans ce mémoire nous avons vu qu'il existe une classe d'applications très couteuse en temps de calcul et pourtant difficiles voir impossibles à parallèliser de manière efficace sur les architectures existantes. Notre équipe à depuis le début des années 80 exploré plusieurs d'entre eux à travers la réalisation de circuits intégrés spécifiques. Le résultat de ces travaux est la définition d'une architecture cellulaire intégrée originale apportant des solution aux problèmes posés par cette classe d'algorithme. Ces expériences nous ont convaincu de la nécessité de dépasser la réalisation de circuits spécifique en définissant une machine plus générale, programmable en fonction de l'application souhaité. Notre architecture est donc une grille à deux dimensions de cellules élémentaires. Chaque cellule intègre deux parties distinctes : un mécanisme de communication permettant l'envoi de messages entre deux cellules non directement connectées et une partie traitement qui n'est pas autre chose qu'un petit microprocesseur 8 bits. Une mémoire de 256 octets propre à chaque cellule sert à la fois de mémoire programme et de mémoire données. L'architecture ainsi définie est donc de type MIMD (chaque cellule pouvant recevoir un programme différent des autres) avec un grain de parallélisme très fin.

Une méthode de programmation adaptée

Pour permettre le développement et l'utilisation dans des conditions acceptables de cette nouvelle classe d'architecture, Nous avons du imaginer et réaliser des outils de compilation adaptés. La nature massive du parallélisme visé ainsi que le caractère irrégulier et asynchrone de la circulation des données dans les algorithmes étudiés rend très complexe (voir impossible) la gestion du parallélisme par le programmeur. Il nous à donc fallu abandonner les langages de programmation habituel des architectures MIMD du type OCCAM dans lesquels la description du parallélisme est explicite. Les langages data flow se sont rapidement imposés à nous en regard de leurs nombreuses qualités:

la gestion du parallélisme est implicite pour le programmeur

le degré de parallélisme extrait est élevé

le graphe de tâche constitue un modèle bien connue sur lequel il est possible d'effectuer de nombreuses optimisations.

Il était important pour nous de dépasser la simple étude de faisabilité et d'approfondir ce travail par une évaluation aussi précise que possible des performances que l'on peut attendre d'un tel outil. En l'absence de modèle assez précis pour réaliser une étude théorique, nous avons choisi de réaliser une chaîne de développement complète allant de l'écriture d'un programme LUSTRE jusqu'à la simulation du réseau dans ses moindres détails. Malgré de nombreuses imperfections qui lui confère un caractère de prototype logiciel, cette chaîne complète existe maintenant. Les limites de ces outils viennent plus du domaine d'applications visées par le langage, que de problèmes d'implémentation.

En conclusion, il commence à être admis dans la communauté scientifique qu'il existe une voie différente du schéma de calcul Von Neuman : le parallélisme. Nos travaux nous ont convaincu de l'intérêt d'architectures utilisant un grand nombre de cellules, chacune travaillant indépendamment des autres et sachant communiquer. Cette étude nous à de même montrée que l'utilisation des langages data flow pour la programmation de tels architectures constitue une voie prometteuse. Il est vraisemblable que malgré les changements d'habitudes de programmation qu'implique de tels langages, leurs disponibilités favoriserait la diffusion de cette nouvelle génération d'architectures.

Bibliographie

[ACK79] "VAL - A Value-Oriented Algorithmic Language: Preliminary Reference Manual"
W. B. Ackerman, J. B. Dennis
Technical Report, Laboratory for Computeur Science, MIT, June 1979

[ANS88] "OCCAM for functional simulation of highly parallel architectures"
Y. Ansade, R. Cornu-Emieux, B. Faure
Esprit Project meeting, Saint pierre de Chartreuse, fevrier 1986

[BER85] "Etude d'une machine cellulaire pour la simulation logique de circuits intégrés"
J. P. Bernard
Thèse de Docteur Ingénieur, INPG, Grenoble, Juillet 1985

[BER89] "Introduction to Programmable Active Memories"
P. Bertin, D. Roncin, J. Vuillemin
Rapport téchnique, Digital, Paris, June 1989

[COR88] "Réseau de cellule intégré : Etude d'architecture pour des applications de CAO de VLSI"
R. Cornu-Emieux
Thèse de l'INPG, Grenoble, Septembre 1988

[DAL86] "A VLSI architecture for concurent data structures"
W. J. Dally
Thesis, Califonia Institue of Technology, 1986

[DEL90] "Tout ce que vous voulez savoir sur la connection machine (sans oser le demander)"
D. Delesalle, D. Trystram, D. Wenzek
Rapport technique, LMC-IMAG, Avril 1985.

[DEN80] "Data Flow Super Computeurs"
J. B. Dennis
Computer, November 1980, pp 48-56

[FAU86] "WSI asynchronous cell network"
B. Faure, Y. Ansade, R. Cornu-Emieux, G. Mazaré
Proceedings of IFIP Workshop on WSI, Grenoble, mars 86

[FAU90] "A cellular architecture dedicated to neural net emulation"
B. Faure, G. Mazaré,
Microprocessing and Microprogramming (The Euromicro Journal), volume 30, North-Holland, 1990, pp. 249-256.

[FAU91] "Une architecture massivement parallèle intégrée"
B. Faure, S. M. Karabernou, G. Mazaré, E. Payan, P. Rubini
Journal des télécoms, à venir

[FLY72] "Some computer organizations and their effectiveness"
M.J. Flynn,
IEEE Transaction on Computers, 21,9 Sept 1972.

[GER90] "Une stratégie de routage pour la machine à parallèlisme massif MEGA"
C. Germain
Actes du deuxième symposium Architectures Nouvelle de Machines, Toulouse, Septembre 1990, pp 1-16

[GUR85] "The Manchester Prototype Dataflow Computeur"
J. R. Gurd, C. C. Kirkham, I. Watson
Communication of the ACM, January 1985, pp. 34-52

[Hal86] "Use of a real-time declarative language for systolic array design and simulation"
N. Halbwachs, D. Pilaud
Proceedings of the International Workshop on Systolic Arrays, Oxford, 1986

[HIL85] "The connection Machine"
D. Hillis
MIT Press, Cambridge, Mass, 1985

[HOA78] "Communicating sequential processes"
C. A. R. Hoare,
Communication ACM, Vol. 21, N°8, 1978, pp. 666-677

[INM84] "Occam programming manual"
INMOS limited, Prentice-Hall, 1984

[KAR90a] "A network with small general processing units for fine grain parallelism"
S. M. Karabernou, G. Mazaré, E. Payan, P. Rubini
Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures, Pont à Mousson, Juin 1990, pp 197-200.

[KAR90b] "Une architecture massivement parallèle programmable : la machine RAP"
S. M. Karabernou, E. Payan, P. Rubini
3éme rencontres sur les algorithmes & architectures massivement parallèles, Luminy, Octobre 1990.

[KAR90c] "Une machine générale massivement parallèle pour le parallélisme à grain fin"
M. Karabernou, E. Payan, P. Rubini
Actes du deuxième symposium Architectures Nouvelle de Machines, Toulouse, Septembre 1990, pp 17-38

[Kor88] "A Data Driven VLSI Array for Arbitrary Algorithms"
I. Koren, B. Mendelson, I. Pelet, G. M. Silberman
Computer, Octobre 1988, pp 30-43

[KUN82] "Why systolic architecture ?"
H. T. Kung
IEEE Computer, Vol. 15, N°1, 1982 pp. 37-46.

[LAT89] "Architecture massivement parallèle : un réseau de cellules intégré pour la reconstruction d'images"
D. Lattard
Thèse de l'INPG, Grenoble, novembre 1989

[MAZ89] "A programmable highly parallel architecture for Digital Signal Processing"
G. Mazaré, E. Payan
Proceedings of the IEEE International Symposium on Circuits and Systems 89, Portland, mai 1989, pp 1332-1335.

[MAZ90] "A programmable Highly Parallel Architecture : Functional Definition and Performance Evaluation"
G. Mazaré, E. Payan
Proceedings of the International Conference on Parallèle Processing in Neural Systems and Computeur (ICNC), Düsseldorf, Mars 1990, pp 27-30.

[MUL88] "Architecture pour le calcul de la transformée de Fourier Discrète"
J. M. Muller, D. Trystram
Rapport téchnique, TIM3 IMAG, Grenoble, Avril 1988

[OBJ88] "Réseau de cellules intégré : méchanisme de communication inter-cellulaire et application à la simulation logique"
P. Objois
Thèse de l'INPG, Grenoble, Septembre 1988

[PAY90a] "A new programming method for Highly parallel Architecture"
E. Payan G. Mazaré
Proceedings of the 1990 International Symposium on Lucid and Intensional Programming, Kingston, Mai 1990, pp 90-95.

[PAY90b] "A New Programming Method for Highly Parallel Architecture"
E. Payan, G. Mazaré
proceedings of the 7th international conference On Systems Enginering, Las Vegas, July 1990, pp 788-794

[PLA76] "LAU System Architecture : A Parallel Data-Driven Processor Based on Single Assignement"
A. Plas, D. Comte, O. Gelly, J. C. Syre

proceedings of the 1976 International Conferrencee on Parallel Processing, August 1976

[POM88] "Neural network simulation at Warp speed : how we got 17 million connections per second"
D. A. Pommerleau, G. L. Gusciora, D. S. Touretzsky,H. T. Kung
Proceedings of the IEEE International Conference on Neural Networks, San Diego, California, juillet 1988.

[RAY90] "Placement de tâches sur une architecture multiprocesseur"
P. Raynal
Rapport de DEA d'informatique, LMC-IMAG, Juin 1990

[ROC89] "Programmation d'un circuit massivement parallèle à l'aide d'un langage déclaratif synchrone"
F. Rocheteau
Rapport de DEA d'informatique, LGI-IMAG, Juin 1989

[ROC90] "Pollux, une extension de Lustre pour la description de circuits matériels"
F. Rocheteau
Rapport interne, LGI-IMAG, 1990

[SIL86] "SL2000/Helix : User documentation",
Silvar Lisco, 1986.

[TOU90] "Evaluation des performances du méganode à 128 processeurs"
A. Touzene, B. Plateau
Rapport interne, LGI-IMAG, Mars 1990

[Wad85] "LUCID, the data flow programming language"
W. W. Wadge, E. A. Ashcroft
Academic Press, 1985.

[Xil87] Xilinx. Technical Data
XC3020 XC3030 XC3042 XC3064 XC3090 Logic Cell array. 1987

ANNEXE 1 :

Les outils de compilation

Introduction

Pour évaluer les performances d'une architecture, deux méthodes sont possibles. On peut tenter de définir un modèle théorique de son fonctionnement, ou au contraire se faire une opinion grâce à l'exécution de nombreux exemples (sur un simulateur ou un prototype).

Obtenir un modèle théorique permettant d'évaluer les performances de notre réseau est une tâche délicate. Comment en effet modéliser finement le routage du réseau et les contentions de messages ?

Pour parvenir à évaluer le plus finement possible notre architecture et ces outils logiciels, nous avons donc choisi d'avoir massivement recours à la simulation. La réalisation d'un prototype du compilateur LUSTRE et d'un simulateur du réseau fût donc la part la plus importante ce travail.

Le prototype de compilateur LUSTRE pour machine massivement parallèle, que nous avons réalisé, se compose d'un certain nombre de programmes, chacun correspondant à une étape de la compilation.

L'ensemble de ces outils constituent l'organigramme suivant :

Programme LUSTRE

↓

Développement des tableaux

↓

Analyse syntaxique et génération du graphe data flow

↓

Limitation du degré de sortie des opérateurs

↓

Ordonnancement des opérateurs

↓

placement → Simulation fonctionnelle

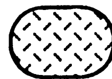
↓

Génération de code → Prototype du réseau cellulaire

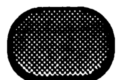
↓

Simulation complète

Outils que nous avons du développer



Analyseur syntaxique réalisé par l'équipe SPECTRE du Laboratoire de Génie Informatique



En cours de développement

Nous allons présenter dans les pages qui viennent chacun des modules réalisés.

Le développement des tableaux

Pour simplifier l'écriture du compilateur nous avons choisi de réutiliser une partie du travail réalisé par l'équipe SPECTRE du Laboratoire de Génie Informatique, pour l'implantation de LUSTRE sur machine séquentielle, et notamment l'analyse syntaxique. La possibilité qu'offre LUSTRE de décrire des programmes répétitifs sous la forme de tableaux est l'une des caractéristiques intéressantes de ce langage en vue d'une implantation sur architecture cellulaire. La disponibilité de cette fonctionnalité était indispensable à notre étude. Malheureusement, la description de telles structures n'étant pas un point d'intérêt central pour les personnes travaillant sur la base de machines séquentielles, elles ne sont pas implémentées dans l'analyseur syntaxique actuel.

Pour résoudre ce problème, nous avons choisi de réaliser un préprocesseur réalisant un développement des parties répétitives du programme avant la phase d'analyse syntaxique.

La syntaxe choisie à été définie dans [Hal86] pour la description des tableaux systoliques. Cette syntaxe posant des problèmes dans la preuve de programme, elle a depuis était remplacée [ROC90]. Il faut noter que cette nouvelle version autorise toujours un développement statique des tableaux lors de la génération du graphe.

A partir du programme source suivant :

```
node produit_de_matrice (
    for x in 1..5
        let
        (a[x][1]:int) when true;
        (b[1][x]:int) when true;
        tel
)
```

```
returns ( );

var
    for x in 1..5
        let
        for y in 1..5
            let
            (r[x][y]:int) when true;
            tel
        tel

let
for x in 1..5
    let
    for y in 1..5
        let
        r[x][y] = 0 -> (a[x][1]*b[1][y] + pre(r[x][y]));
        tel
    tel
tel.
```

On obtient le programme développé :

```
node matrice (
    (ai1i1:int) when true;
    (ai2i1:int) when true;
    (ai3i1:int) when true;
    (ai4i1:int) when true;
    (ai5i1:int) when true;
    (bi1i1:int) when true;
    (bi1i2:int) when true;
    (bi1i3:int) when true;
    (bi1i4:int) when true;
    (bi1i5:int) when true;
)
returns ( );
```

```
var
    (ri1i1:int) when true;
    (ri1i2:int) when true;
    (ri1i3:int) when true;
    (ri1i4:int) when true;
    (ri1i5:int) when true;
    ...
    (ri4i5:int) when true;
    (ri5i1:int) when true;
    (ri5i2:int) when true;
    (ri5i3:int) when true;
    (ri5i4:int) when true;
    (ri5i5:int) when true;

let
ri1i1 = 0 -> (ai1i1*bi1i1 + pre(ri1i1));
ri1i2 = 0 -> (ai1i1*bi1i2 + pre(ri1i2));
ri1i3 = 0 -> (ai1i1*bi1i3 + pre(ri1i3));
...
ri5i5 = 0 -> (ai5i1*bi1i5 + pre(ri5i5));
tel.
```

Ce programme de 366 lignes est écrit en Pascal sur Macintosh, il ne comporte aucune vérification syntaxique ou sémantique (le programme fourni en entrée est supposé correct).

L'analyse syntaxique et la génération du graphe dataflow

L'analyse syntaxique et lexicale constitue traditionnellement la première phase de la compilation d'un programme. De nombreux outils tel LEX et YACC existent pour simplifier et automatiser la réalisation de cette partie de compilateur. Nous avons préféré à leur utilisation, réutiliser une portion du compilateur LUSTRE développée pour les architectures séquentielles par l'équipe SPECTRE du Laboratoire de Génie

Informatique. Notre travail n'a consisté qu'à greffer sur cet outil un module permettant l'extraction sous forme d'un fichier texte du graphe dataflow. Le coût de cette réutilisation est un manque de souplesse dans l'analyseur, qui nous a par exemple empêché d'y ajouter les tableaux.

Le résultat de l'exécution de ce programme est un fichier texte qui décrit chaque arc du graphe :

Type de l'arc, Nœud destination,Code opération du noeud destination, Nœud source

Pour les entrées et les constantes (qui ne sont des destinations d'aucun arc) on ajoute à la fin de la ligne :

code-op du nœud source,nom de l'entrée ou valeur de la constante.

Exemple de fichier décrivant un graphe :

```
inte, 3,arr,4,cst,0
 inte, 3,arr,5
 inte, 5,plu,6
 inte, 6,tim,7,inp,ai5i1
 inte, 6,tim,8,inp,bi1i5
 inte, 5,plu,9
 inte, 9,pre,3
 inte, 13,arr,4
 inte, 13,arr,14
 inte, 14,plu,15
 inte, 15,tim,7
```

Cet analyseur est écrite en C++ et fonctionne sur station SUN.

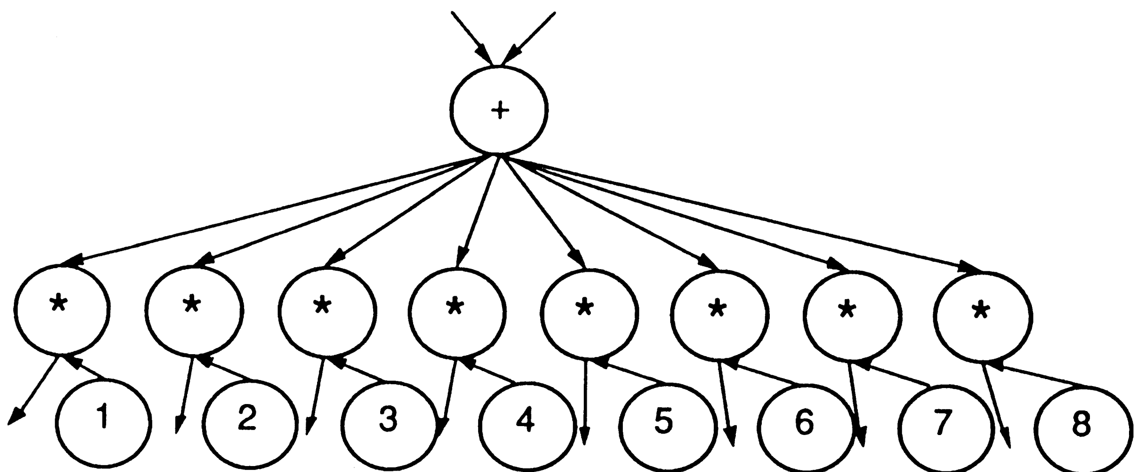
Limitation du degré de sortie des opérateurs

La sémantique du langage LUSTRE ne donne aucune limitation au degré d'entrées (nombre d'entrées) d'un nœuds. Ainsi le programme suivant :

```
let
Somme = entrée1 + entrée2 ;
S1 = Somme * 1;
S2 = Somme * 2;
S3 = Somme * 3;
S4 = Somme * 4;
S5 = Somme * 5;
S6 = Somme * 6;
S7 = Somme * 7;
S8 = Somme * 8;
tel .
```

nous donne le graphe suivant:

Exemple d'un noeud addition ayant un degré de 10



Une telle configuration pose deux problèmes :

- Le grand nombre de messages à envoyer du nœud addition vers les nœuds multiplications risque de saturer le mécanisme de

communication de la cellule émettrices.

- Les adresses de chacun de ces messages doivent être stockées dans la mémoire de la cellule, qui risque pour un très grand degré d'être saturée.

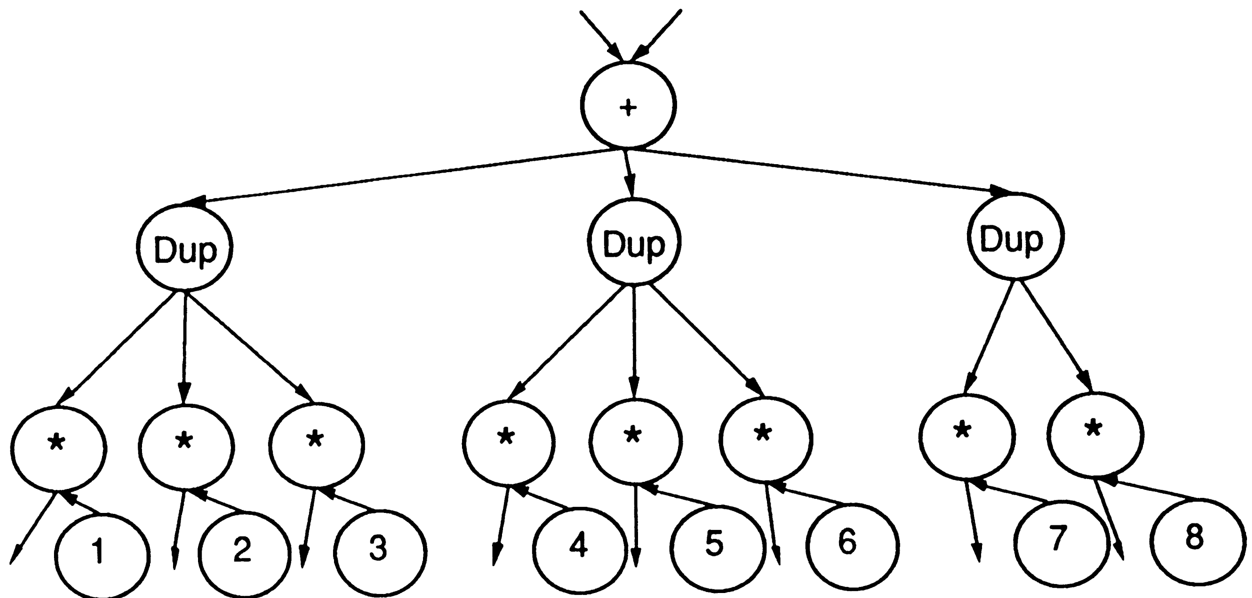
Pour éviter ces problèmes nous avons choisi de limiter le degré total (nombre d'entrées plus nombre de sorties) à 4.

Il nous faut donc prévoir d'intercaler des opérateurs de duplication (qui serons notés DUP) pour répartir entre plusieurs opérateurs la charge que représente l'envoi des messages. L'algorithme utilisé est le suivant :

```
début
pour tout les nœuds faire
        si degré de sortie du nœud > 4 alors
                tant que degré de sortie du nœud > 4 faire
                        début
                        intercaler un nœud duplication entre
                                le nœud et 3 de ces sorties;
                        degré de sortie := degré de sortie -3;
                        fin tantque
fin
```

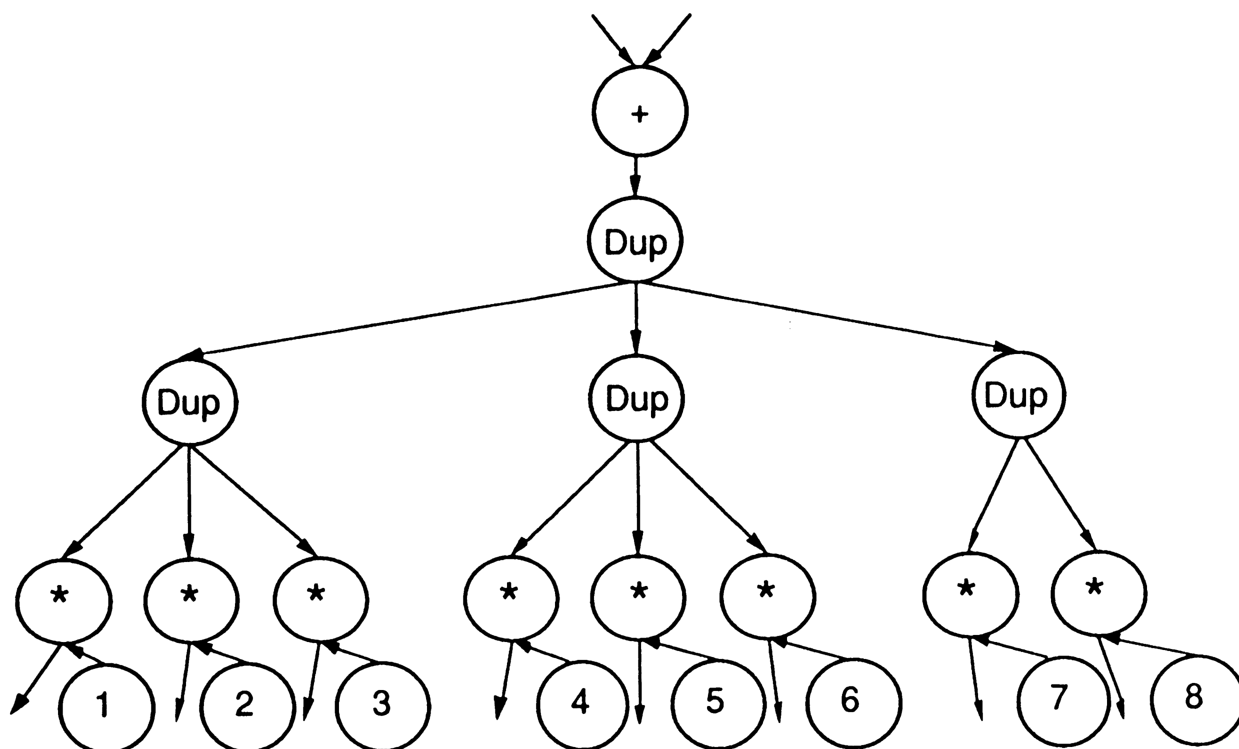
Ce programme va donc intercaler autant de nœuds duplication que nécessaire :

Réduction du degré à 5 par ajout de noeuds duplication



On voit sur cet exemple que cette opération doit être renouveler jusqu'à ce que plus aucun noeud n'aie son degré supérieur à 4.

Réduction du degré à 3 par ajout d'un deuxième
niveaux de noeuds duplication



Cet algorithme permet de construire une arborescence de Dup, cette structure est préférable à une file de Dup qui introduit trop de différences dans les niveaux de pipeline entre les nœuds du début de la file est ceux de la fin.

Le résultat de ce programme est un fichier texte utilisant la même syntaxe que le programme précédent.

Cet outil est écrit en Pascal sur Macintosh et comporte 320 lignes de programme.

Datation du graphe

Cette étape n'est nécessaire qu'à la méthode "séquentielle", elle consiste à mettre en place sur l'ensemble du graphe une numérotation des nœuds caractérisant les dépendances du graphe. Cette numérotation est

utilisée en deux occasions :

- Pendant le placement pour caractériser la position d'un opérateur dans le pipeline d'un programme (voir 3.4.8).

- Lors de la génération de code pour permettre, par la seule observation des opérateurs placés sur une cellule, de connaître l'ordre de leur exécution et donc l'ordre dans lequel il convient de générer leur code.

L'algorithme utilisé est un classique algorithme de parcours de graphe, d'abord en descendant pour donner à chaque nœud sa date "au plus tôt" puis en remontant pour les dates au plus tard. Cette méthode est connue dans la littérature sous le nom de méthode "PERTH". Seule la date "au plus tard" est utilisée par la suite, elle précise pour chaque opérateur la date à laquelle il devra avoir fini son calcul au plus tard pour ne pas ralentir l'ensemble du problème.

Le format de fichier utilisé est le même que précédemment, on ajoute simplement devant le code opération la date de l'opérateur.

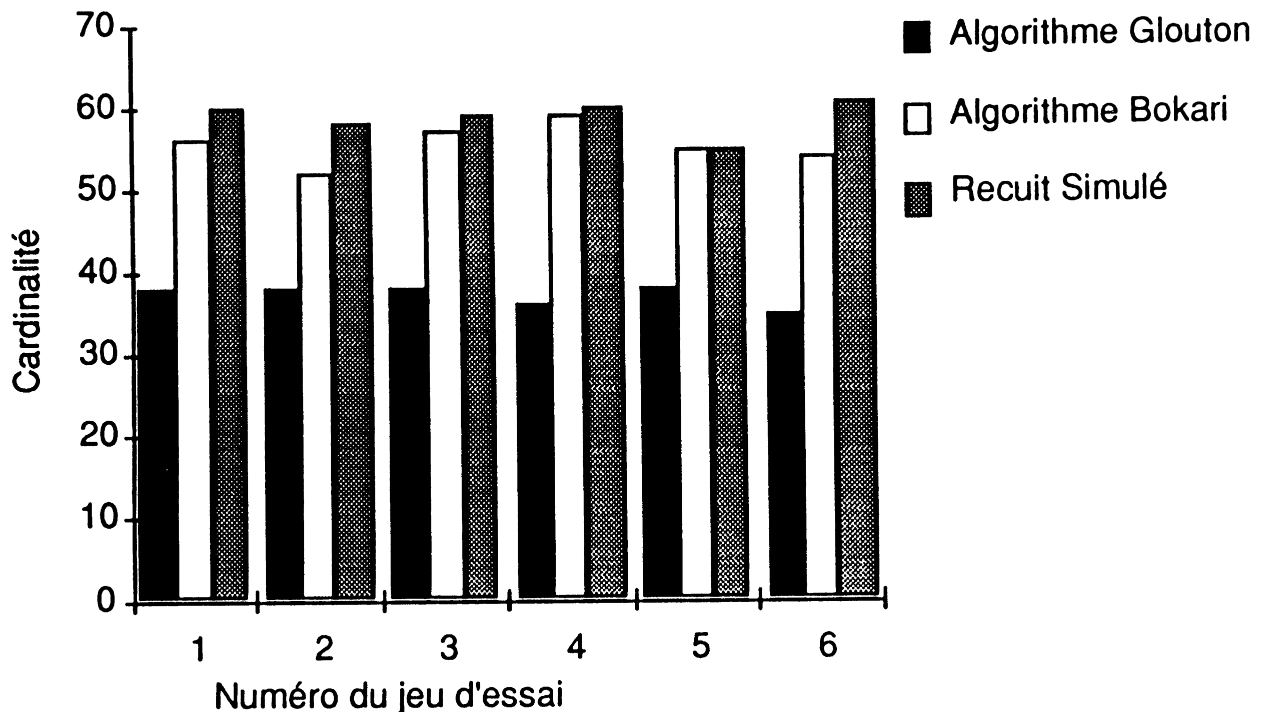
Cet outil est écrit en Pascal sur Macintosh et comporte 560 lignes de programme.

Placement

Le placement est connu pour être un problème de NP-complet; il est donc illusoire d'espérer trouver une solution optimale pour des exemples de tailles réalistes. De nombreux algorithme permettant d'approcher cette solution ont été proposés. Le problème du placement n'est pas l'objet de cette thèse, notre approche s'est limitée à choisir une méthode décrite dans la littérature. Nos critères de choix ont été les suivants :

Les performances

Une étude comparative des performances [RAY90] des trois algorithmes les plus classiques pour différents jeux d'essai (une architecture aléatoire est associée à un graphe de tâches aléatoire) nous donne le résultat suivant :



Le recuit apparaît donc comme la plus performante des méthodes de placement. Son principale défaut est son coût en temps de calcul : sa complexité est de 500 $O(n^3)$ contre 10 $O(n^3)$ pour Bokari.

Les possibilités d'implantation

Des études précédentes ont montré qu'il était possible d'implémenter de manière efficace le recuit simulé sur une architecture massivement parallèle. La complexité des méthodes de placement étant en $O(n^3)$, il est facile d'imaginer que sur de grands réseaux, le temps de calcul peut devenir rédhibitoire. La possibilité de faire exécuter le placement sur le réseau (et donc de le faire participer à la compilation de son propre programme) est un atout certain.

Un autre intérêt de cette méthode est sa relative facilité de programmation, ce qui a permis un développement rapide de cet outil et a simplifié ses évolutions.

La fonction de coût à minimiser lors du placement, dépend de la méthode de génération utilisé. L'expérience a montré que pour la méthode parallèle il convient de :

-réduire les longueurs des connexions entre les cellules

- équilibrer la charge de travail

Pour la méthode séquentielle, la fonction de coût doit principalement tenir compte de la date d'exécution des opérateurs et dans une moindre part réduire les longueurs des connexions entre les cellules.

Il faut aussi faire attention de ne pas placer sur une cellule plus d'opérateurs que sa mémoire ne peut en contenir. La simplicité du code généré pour la méthode parallèle, rend facile l'évaluation de l'occupation mémoire d'un opérateur lors du placement. Pour la version séquentielle, un calcul exact de l'espace mémoire nécessaire doit tenir compte des optimisations et notamment de la suppression de certain messages de synchronisation. Il est probable qu'un tel calcul ralentirait considérablement le placement. Nous avons préféré dans la version actuelle du placeur nous contenter d'un calcul approché beaucoup plus rapide.

Le placeur à été écrit en Pascal, il fonctionne sur un Macintosh IIfx (68030 à 40 Mhz, Coprocesseur arithmétique) à la vitesse de 4300 itérations à la minute soit pour obtenir un placement complet environ 1 heures. Il est écrit en Pascal et comporte environ 1750 lignes de programme.

Simulations fonctionnelles

Pour permettre la simulation du réseau plusieurs solutions ont successivement été étudiées et essayées. Par ordre chronologique nous avons successivement étudié des simulations en OCCAM, avec HELIX, et le développement d'un simulateur fonctionnel spécifique.

Les simulations OCCAM

De part sa sémantique le langage OCCAM [INM84] permet une description aisée de processus parallèles communicants. Il était donc naturel d'utiliser ce langage pour simuler notre réseau cellulaire [ANS88]. Les premiers travaux de l'équipe portant sur la réalisation d'un réseau cellulaire dédié à la simulation logique ont ainsi été simulés en OCCAM d'abord sur un VAX puis sur une carte Transputer.

A l'usage ce langage pose plusieurs problèmes :

- L'idée simple qui consiste à associer à chaque cellule du réseau un processus OCCAM ne permet pas de prendre en compte les différences de temps de calcul entre le calcul dans les cellules et le routage des messages. Les résultats de ces simulations ne peuvent donc qu'être imprécis du point de vue temporel.

- Le manque d'outils de mise au point de ce langage rend très difficile la mise au point des simulateurs et l'observation du réseau simulé.

Ces difficultés de mise au point ajoutées aux modestes performances de la carte Transputer dont nous disposions (un T414 avec 2 méga-octets de mémoire) nous ont fait abandonner l'idée de développer avec ce langage un simulateur plus performant du point de vue temporel (du type à échéancier distribué).

Les simulations en HELIX

HELIX [SIL86] est un outil de simulation faisant partie du programme de conception assisté par ordinateur Silvard-Lisco. Cet outil permet la description aisée (sous forme graphique) de processus communiquants. Il permet de plus de fixer précisément le temps de calcul respectif des processus. Grâce à ses outils graphiques et à ses notions temporelles, HELIX semble bien adapté à la simulation de notre réseau. C'est donc logiquement qu'il a été utilisé pour l'évaluation du projet de réseau dédié à la reconstruction d'image [LAT89]. Les premières études pour l'implantation de LUSTRE sur une architecture massivement parallèle ont été elles aussi menées grâce à ce système.

Malgré toutes ces qualités, la très grande lenteur de ce système (plusieurs heures CPU sur un VAX pour simuler un réseau de 25 cellules dédiées à la reconstruction d'image) nous a conduit à l'abandonner.

Le simulateur fonctionnel

Pour résoudre ces problèmes nous avons développé un simulateur adapté au réseau. Le cahier des charges initial de cet outil était :

- Une simulation complète du mécanisme de routage

- Une simulation des opérateurs prenant en compte leurs différents temps de calcul.

- Des performances permettant de simuler en un temps raisonnable des réseaux de plusieurs centaines de cellules.

Lors du développement de ce simulateur, le jeu d'instructions de la cellule n'était pas encore fixé. Nous avons donc choisi de ne pas simuler les instructions qui composent les opérateurs, mais de choisir le niveau opérateur comme niveau de simulation le plus bas.

Le principe de fonctionnement adopté pour cet outil correspond à la génération de code "pseudo parallèle" dans lequel chaque opérateur est

un processus indépendant des autres opérateurs placés sur la même cellule.

Cet outil fonctionne suivant le principe classique de la simulation à échéancier, Cet outil est écrit en Pascal sur Macintosh et comporte 1150 lignes de programme. Il à permis de valider les bases de notre compilateur avant que le jeu d'instruction de la cellule ne soit définitivement choisie.

Génération de code

Pour permettre une confrontation précise des deux schémas d'exécution de LUSTRE sur notre réseau, ce n'est pas un mais deux générateurs de code qu'il nous a fallu développer :

Le générateur de code "pseudo parallèle"

Ce programme utilise le résultat du placement pour générer un fichier texte contenant le programme en langage assembleur. Cette méthode ne permettant pas d'optimisation du code, le fonctionnement de cet outil est très simple.

Les types de données actuellement implémentés sont :

- Les entiers codés sur 16 bits

- Les booléens codés sur 8 bits

Les nombres en virgule flottante ne sont pas implémentés dans la version actuelle.

Exemple d'une portion de programme généré:

```
size 7,7            <-- Taille du réseau cible
dest 0,0,0,0        <-- Coordonnés du rectangle de cellules, cible de cette portion de programme
org 16
1/ send msg7
1/ send msg10       ==> Envoie des demandes
1/ send msg11
; Generation d un operateur :dup
op3:
6/ lda #op4
6/ sta 0
6/ try 254          ==> Test de présence des opérandes
6/ try 253
6/ try 252
6/ try 251
4/ get 253
4/ get 252
4/ get 251
2/ get 254
1/ send msg7
5/ sta msg1
5/ sta msg3         ==>
5/ sta msg5
2/ get 255
5/ sta msg2
5/ sta msg4
5/ sta msg6
5/ send msg1
5/ send msg3
5/ send msg5
5/ send msg2
5/ send msg4
5/ send msg6
; Generation d un operateur :tim
op4:
6/ lda #op3
6/ sta 0
6/ try 249
6/ try 247
```

Numéro de zonne indiquant au simulateur le rôle de ces instructions :
1 : Envoie des demandes
2 : Récéption des opérandes
3 : Calcul de l'opérateur
4 : Récéption des demandes
5 : Envoie des résultats
6 : Attente des opérateurs

Ces valeurs sont utilisé par le simulateur pour permetre la réalisation de graphe donnant à chaque instant, le pourcentage de cellule efféctuant une de ces opérations.

```
6/ try 246
4/ get 246
2/ get 250
2/ get 249
2/ get 248
2/ get 247
3/ mul 250,248
3/ staw datach
3/ mul 249,248
3/ add datach
3/ sta datach
3/ mul 250,247
1/ send msg11
1/ send msg10
3/ add datach
5/ sta msg8
3/ lda datacl
5/ sta msg9
5/ send msg8
5/ send msg9
3/ bra op3
msg1: msg 0,254,[2,0]
msg2: msg 0,255,[2,0]
msg3: msg 0,254,[4,3]
msg4: msg 0,255,[4,3]
msg5: msg 0,246,[0,5]
msg6: msg 0,247,[0,5]
msg7: msg 0,248,[0,6]
msg8: msg 0,246,[3,1]
msg9: msg 0,247,[3,1]
msg10: msg 0,247,[6,6]
msg11: msg 0,253,[4,1]
org 3
datach: ds 1
datacl: ds 1
```

Déscription des messages

Label donnant le nom du message

Valeur initiale du champ donné du message

Tag du message

Dx et Dy du message

Ce générateur a été écrit en Pascal et comporte environ 1150 lignes de programme.

Le générateur de code séquentiel

Comme le précédent, ce générateur accepte en entrée un fichier issu du placeur, et fournit en sortie dans un fichier texte le programme correspondant. Le programme généré doit incorporer les optimisations précédemment décrites (3.4.8) concernant les messages de synchronisation. Si il est facile de supprimer toute synchronisation entre opérateurs placés sur la même cellule, il est plus complexe de savoir lesquelles sont nécessaires entre cellules.

La version actuelle ne permet pas le passage de paramètres entre opérateurs par les registres, ils sont systématiquement passés par la mémoire. Les types de données actuellement implémentés sont :

- Les entiers codés sur 16 bits

- Les booléens codés sur 8 bits

Les nombres en virgule flottante n'ont pas été encore implémentés.

Exemple de programme généré:

```
 dest  4,0,4,0
 org 16
1/ send msg3
1/ send msg4
; Generation d un operateur :dup num:    175
boucle:
2/ get 249
5/ sta msg5
5/ sta msg7
```

```
2/ get 250
5/ sta msg6
5/ sta msg8
5/ send msg5
4/ get 248
5/ send msg7
5/ send msg6
5/ send msg8
; Generation d un operateur :tim num:      93
2/ get 254
2/ get 252
3/ mul 255,253
3/ staw datach
3/ mul 254,253
3/ add datach
3/ sta datach
3/ mul 255,252
1/ send msg4
1/ send msg3
3/ add datach
5/ sta msg1
3/ lda datacl
5/ sta msg2
4/ get 251
5/ send msg1
5/ send msg2
3/ bra boucle
msg1: msg 0,244,[-4,4]
msg2: msg 0,245,[-4,4]
msg3: msg 0,247,[-3,5]
msg4: msg 0,250,[-2,4]
msg5: msg 0,252,[-4,3]
msg6: msg 0,253,[-4,3]
msg7: msg 0,243,[-3,6]
msg8: msg 0,244,[-3,6]
 org 3
datach: ds 1
datacl: ds 1
```

Cet outil à été écrit en pascal est comporte environ 1480 lignes de programme.

Simulation du réseau

P. Rubini a développé pour notre architecture un ensemble d'outil comprenant :

- Un assembleur capable de générer un programme binaire exécutable à partir de programmes sources. Ce programme à été développé en C sur macintosh, il fonctionne à la vitesse d'environ 175 lignes à la seconde.

- Un simulateur capable de simuler totalement le réseau. Le programme se déroulant sur une cellule est simulé au niveau des instructions assembleur avec respect du nombre de cycle nécessaire à leur exécution. Le routage des messages est simulé avec précision, il est possible de simuler plusieurs stratégies de routage et même de tenir compte des différences de routage entre les cellules situées sur un même circuits et celles situées sur deux circuits différents. Cet outil à été écrit sur macintosh en C et en assembleur 68000 ; ces performances sur un Mac II fx sont d'environ 6000 cycles par seconde pour une cellule simulée.

Des outils permettant la mise au point des programmes ont été ajoutés au simulateur :

- exécution en mode pas à pas

- dump mémoire d'une cellule

- trace de l'exécution d'une cellule

- possibilité de visualiser le contenu d'un message

Un affichage graphique du réseau permet de visualiser facilement les messages ainsi que les cellules bloquées (en attente d'un message).

Malgré toute ces qualités, il est apparu nécessaire de disposer d'un outil plus puissant. Une deuxième version de l'assembleur et du simulateur est en cours de développement, elle intégrera des possibilités supplémentaires :

- Assemblage conditionnel permettant de différencier plus facilement les programmes de certaines cellules (notamment celles du bord du réseau).

- Détection à la compilation d'un certain nombre d'erreurs (messages envoyés en dehors du réseau...).

- Détection à l'exécution de certaines autres erreurs (exécution de données, écrasement de code ou de messages non lus...).

- Affichage amélioré par l'utilisation de multi-fenêtrage permettant par exemple de tracer plusieurs cellules simultanément.

Annexe 1

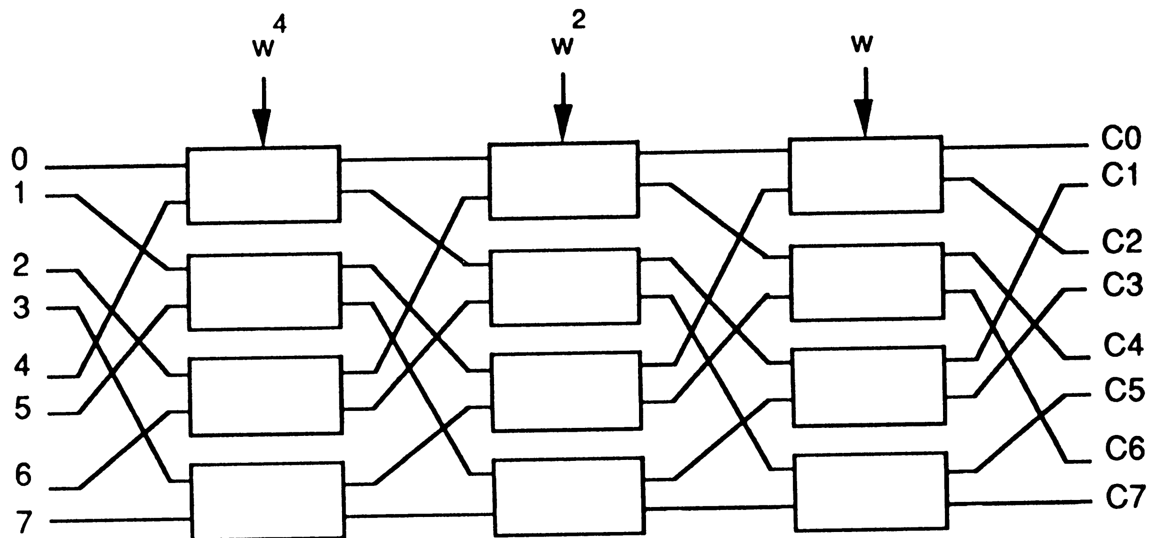
ANNEXE 2 :

Exemples de programmes

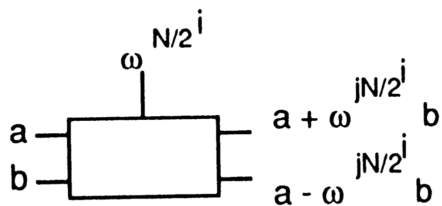
Programmes LUSTRE

La transformée de Fourier rapide (FFT)

La transformée de Fourier rapide [MUL88] suit le schéma suivant :



dans lequel les cellules élémentaires sont du type :



Il est facile de déduire d'une telle structure un programme LUSTRE équivalent :

```
node fft (
for k in 0..7
     let
     (reel[0][k]:int) when true
     (img[0][k]:int) when true
     tel
```

```
for l in 1..3
      let
      (wreel[l]:int) when true
      (wimg[l]:int) when true
      tel
)
returns
(
for k in 0..7
      let
      (reel[3][k]:int) when true
      (img[3][k]:int) when true
      tel
)

var
for k in 0..7
      let
      for l in 1..2
            let
            (reel[l][k]:int) when true;
            (img[l][k]:int) when true;
            tel
      tel

let
for l in 1..3
      let
      for k in 0..3
            let
            reel[l][2*k] = reel[l-1][k]
                        +(wreel[l]*reel[l-1][k+4] - wimg[l]*img[l-1][k+4]);
            img[l][2*k] = img[l-1][k]
                        +(wreel[l]*imgl[l-1][k+4] + wimg[l]*reel[l-1][k+4]);
            reel[l][2*k+1] = reel[l-1][k]
                        -(wreel[l]*reel[l-1][k+4] - wimg[l]*img[l-1][k+4]);
            img[l][2*k+1] = img[l-1][k]
                        -(wreel[l]*img[l-1][k+4] + wimg[l]*reel[l-1][k+4]);
```

```
        tel
    tel
tel.
```

Le produit de matrice

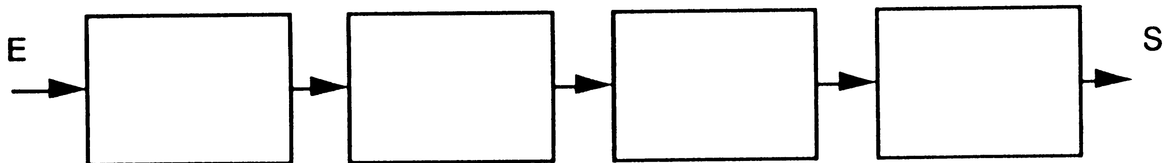
L'algorithme du produit de matrice à été décrit (5.10.2). Il est intéressant de comparer sont écriture en LUSTRE et celle du même algorithme en assembleur cellule (voir ci-dessous). La version LUSTRE est la suivante :

```
node matrice (
for x in 1..5
    let
    (a[x][1]:int) when true;
    (c[1][x]:int) when true;
    tel
)
returns ();

var
for x in 1..5
    let
    for y in 1..5
        let
        (r[x][y]:int) when true;
        tel
    tel

let
for x in 1..5
    let
    for y in 1..5
        let
        r[x][y] = 0 -> (a[x][1]*c[1][y] + pre(r[x][y]));
        tel
    tel
```

tel.

Programme de tri

Notre algorithme de tri est organisé en un tableau à une dimension de cellules élémentaires :



Chacune suit l'algorithme suivant :

```
Valeur interne:=0;
Si Entrée > Valeur interne alors
    début
    Sortie:=Valeur Interne;
    Valeur Interne:=Entrée;
    fin
sinon
    début
    Sortie:= Entrée;
    fin
```

Ce qui en LUSTRE s'exprime comme suit :

```
node tri (
(x:int) when true
)
returns (
 (y:int) when true;
 );

var
```

```
for k in 0..40
      let
      (i[k]:int) when true;
      (s[k]:int) when true;
      (c[k]:bool) when true;
      tel


let
for k in 1..40
      let
      c[k] = s[k-1]>pre(i[k]);
      i[k] = 0 -> if c[k] then s[k-1] else pre(i[k]);
      s[k] = if c[k] then pre(i[k]) else s[k-1];
      tel
s[0] = x;
y=s[40];
tel.
```

Produit de convolution

Le produit de convolution suit exactement l'algorithme décrit dans [Hal86].

```
node convolution (
(x:int) when true;
for k in 0..10
      let
      (c[k]:int) when true;
      tel
)
returns (
(y:int) when true;
);
```

```
var
for k in -1..10
      let
      (x[k]:int) when true;
      (y[k]:int) when true;
      tel

let
for k in 0..10
      let
      y[k] = pre (y[k-1]) + c[k]*x[k];
      x[k] = pre (pre (x[k-1]));
      tel;
x[-1] = x;
y[-1] = y;
y = y[10];
tel.
```

Exemple de programme assembleur : le produit de matrice

Ce programme calcul un produit de matrice 5*5, les opérations sont effectuées avec des données sur 16 bits. Pour simplifier sont écriture, l'extraction des résultats n'est pas prévue. Sa structure est nettement compliquée par les cellules des bords inférieur et droit, qui ne doivent pas réémettrent leurs opérandes (car elles sont sur le bord du réseau).

```
size 5,5

dest 0,0,3,3
org 16
3/ lda #0
3/ sta dataph
3/ sta datapl
boucle:
1/ send syncv
```

```
1/ send synch
2/ get 254
5/ sta msg1
4/ get 250
5/ send msg1
2/ get 255
5/ sta msg2
5/ send msg2
2/ get 252
5/ sta msg3
4/ get 249
5/ send msg3
2/ get 253
5/ sta msg4
5/ send msg4
3/ mul msg2,msg4
3/ staw datach
3/ mul msg1,msg4
3/ add datach
3/ sta datach
3/ mul msg2,msg3
3/ add datach
3/ sta datach
3/ ldaw dataph
3/ adcw datach
3/ staw dataph
3/ bra boucle

msg1: msg 0,254,[1,0]
msg2: msg 0,255,[1,0]
msg3: msg 0,252,[0,1]
msg4: msg 0,253,[0,1]
synch: msg 0,249,[0,-1]
syncv: msg 0,250,[-1,0]

org3
dataph: ds 1
datapl: ds 1
```

```
datach: ds 1
datacl: ds 1

dest 1,4,3,4
org 16
3/ lda #0
3/ sta dataph
3/ sta datapl
boucle:
1/ send synch
1/ send syncv
2/ get 254
5/ sta msg1
4/ get 250
5/ send msg1
2/ get 255
5/ sta msg2
5/ send msg2
2/ get 252
5/ sta msg3
2/ get 253
5/ sta msg4
3/ mul msg2,msg4
3/ staw datach
3/ mul msg1,msg4
3/ add datach
3/ sta datach
3/ mul msg2,msg3
3/ add datach
3/ sta datach
3/ ldaw dataph
3/ adcw datach
3/ staw dataph
3/ bra boucle
msg1: msg 0,254,[1,0]
msg2: msg 0,255,[1,0]
synch: msg 0,249,[0,-1]
syncv: msg 0,250,[-1,0]
```

```
org3
dataph: ds 1
datapl: ds 1
datach: ds 1
datacl: ds 1
msg3: ds 1
msg4: ds 1

dest 4,4,4,4
org 16
3/ lda #0
3/ sta dataph
3/ sta datapl
boucle:
1/ send synch
1/ send syncv
2/ get 254
5/ sta msg1
2/ get 255
5/ sta msg2
2/ get 252
5/ sta msg3
2/ get 253
5/ sta msg4
3/ mul msg2,msg4
3/ staw datach
3/ mul msg1,msg4
3/ add datach
3/ sta datach
3/ mul msg2,msg3
3/ add datach
3/ sta datach
3/ ldaw dataph
3/ adcw datach
3/ staw dataph
3/ bra boucle
synch: msg 0,249,[0,-1]
```

```
syncv: msg 0,250,[-1,0]

org3
dataph: ds 1
datapl: ds 1
datach: ds 1
datacl: ds 1
msg1: ds 1
msg2: ds 1
msg3: ds 1
msg4: ds 1

dest 4,1,4,3
org 16
3/ lda #0
3/ sta dataph
3/ sta datapl
boucle:
1/ send synch
1/ send syncv
2/ get 254
5/ sta msg1
2/ get 255
5/ sta msg2
2/ get 252
5/ sta msg3
4/ get 249
5/ send msg3
2/ get 253
5/ sta msg4
5/ send msg4
3/ mul msg2,msg4
3/ staw datach
3/ mul msg1,msg4
3/ add datach
3/ sta datach
3/ mul msg2,msg3
3/ add datach
```

```
3/ sta datach
3/ ldaw dataph
3/ adcw datach
3/ staw dataph
3/ bra boucle

synch: msg 0,249,[0,-1]
syncv: msg 0,250,[-1,0]
msg3: msg 0,252,[0,1]
msg4: msg 0,253,[0,1]

org3
dataph: ds 1
datapl: ds 1
datach: ds 1
datacl: ds 1
msg1: ds 1
msg2: ds 1
```

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Grenoble, le 24 Mai 1991

DÉPARTEMENT DES ÉTUDES DOCTORALES

Affaire suivie par
Tél : 76.57.

N/Réf. :

Objet :

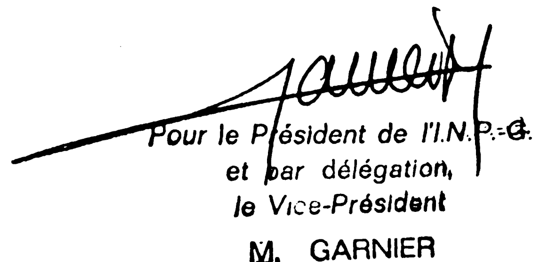
AUTORISATION de SOUTENANCE

Vu les dispositions de l'arrêté du 23 Novembre 1988 relatif aux Etudes Doctorales
Vu les rapports de présentation de :

- Monsieur SANSONNET
- Monsieur MERIAUX

Monsieur PAYAN Eric

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de **DOCTEUR de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE** spécialité :
"*Microélectronique*"



Pour le Président de l'I.N.P.G.
et par délégation,
le Vice-Président
M. GARNIER

Résumé

Pour répondre à des besoins toujours croissants en puissance de calcul, on a vu se multiplier depuis quelques années les études concernant les architectures parallèles. Malgré la variété des solutions proposées il existe encore une classe d'applications difficiles à exécuter en parallèle. Nous proposons dans cette thèse une architecture massivement parallèle basée sur un réseau régulier de cellules, qui ont la particularité d'être totalement asynchrones et de pouvoir communiquer entre elles grâce à un mécanisme d'acheminement de messages. Chaque cellule comprend une partie de traitement qui comprend un petit microprocesseur 8 bits et sa mémoire (donnée plus programme), et une partie routage permettant d'acheminer les messages. Notre second objectif consistait à imaginer puis développer une méthode de programmation adaptée à la fois à notre nouvelle architecture et à la classe d'algorithmes visée. La solution étudiée consiste à placer un graphe data flow obtenu à partir du langage LUSTRE sur notre réseau cellulaire. Un premier prototype de ce compilateur a été réalisé, il a permis d'étudier l'importance de paramètres comme la répartition de la charge de calcul entre les cellules ou l'enchaînement de l'exécution de plusieurs noeuds du graphe placés sur la même cellule.

Abstract

To satisfy the increasing demand of computing power, new computer architectures are clearly required. Several parallel architectures have been proposed. In spite of the number of proposed solutions, one class of applications the cannot be executed in parallel on existing computers still remains. We propose in this thesis a new architecture based on a 2 dimensions grid of asynchronous processing units. Each of them include a processing part made of a small 8 bits microprocessor and its memory (data plus program), and a routing part witch allow message transmission. Our second goal was to design and develop a new programming tool for this architecture. The solution we have chosen is to map a data flow graph, extracted from a LUSTRE program, onto the asynchronous cellular network. A first prototype of the compiler have been realized. It allows us to study such things as load balancing between cells, execution of several nodes mapped onto a single cell.

Mots clé :

Architecture parallèle. Langage Data-flow. Parallélisme massif. Réseau cellulaire. Placement de tâches.