Programmation de calculateur massivement parallèles : application à la factorisation d'entiers

Jean-Laurent Philippe

To cite this version:

Jean-Laurent Philippe. Programmation de calculateur massivement parallèles : application à la factorisation d'entiers. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1990. Français. NNT : . tel-00338193

HAL Id: tel-00338193

https://theses.hal.science/tel-00338193

Submitted on 12 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.

THÈSE

Présentée par Jean-Laurent PHILIPPE

pour obtenir le titre de **DOCTEUR**

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

(Spécialité: **Informatique**)

PROGRAMMATION DE CALCULATEURS MASSIVEMENT PARALLÈLES.

APPLICATION À LA FACTORISATION D'ENTIERS

Date de soutenance : 19 juin 199[illegible]

Composition du jury :

Président :	J. Della Dora
Rapporteurs :	J.L. Nicolas
	Y. Robert
Examinateurs :	M. Cosnard
	F. Cossec
	H.J.J. te Riele

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

THÈSE

Présentée par Jean-Laurent PHILIPPE

pour obtenir le titre de **DOCTEUR**

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

(Spécialité: **Informatique**)

PROGRAMMATION DE CALCULATEURS MASSIVEMENT PARALLÈLES.

APPLICATION À LA FACTORISATION D'ENTIERS

Date de soutenance : 19 juin 199

Composition du jury :

Président :	J. Della Dora
Rapporteurs :	J.L. Nicolas
	Y. Robert
Examinateurs :	M. Cosnard
	F. Cossec
	H.J.J. te Riele

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

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

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. NEMOZ Alain

Année Universitaire 1988 - 1989

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

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 ISN
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 Gerde	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 J.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	Thermodymamique 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. I.U.T.1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA. IUT 1

PROFESSEURS de 2ème classe

BEE Marc	Physique IUT 1
BOUTHINON Michel	EEA. 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	EEA.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	EEA.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 Therapeutique	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	Hygiéne 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 EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital 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	Hopital 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	Hopital 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	Hopital 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	Hopital 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.

A Christine, ma femme,

A mes parents,

A mes beaux-parents,

Remerciements

Je tiens à remercier Jean DELLA DORA pour avoir accepté de présider le jury de cette thèse.

Jean-Louis NICOLAS et Yves ROBERT, en rapportant sur cette thèse, m'ont permis d'améliorer la qualité du document manuscrit. Qu'ils trouvent ici la marque de ma gratitude pour leurs précieux conseils.

J'exprime mes plus vifs remerciements à François COSSEC de Matra MS2I Saint-Quentin en Yvelines et à Herman te RIELE de l'Université d'Amsterdam pour avoir accepté de se déplacer et de participer à ce jury.

C'est à Michel COSNARD que je dois tout. Il a accepté de diriger mes recherches dans l'équipe de Calcul Formel et Algorithmique Parallèle de feu le laboratoire TIM3, puis dans le nouveau Laboratoire de l'Informatique du Parallélisme de l'Ecole Normale Supérieure de Lyon, mais tout en me laissant effectuer mes recherches dans le nouveau Laboratoire de Modélisation et Calcul dirigé par Jean Della Dora à Grenoble. Dans l'équipe de Michel Cosnard à Grenoble, j'ai rencontré Christine au début de ma thèse. Nos chemins se sont croisés une fois. Depuis ils sont confondus. Michel m'a donné le goût de la recherche active et du travail bien fait. Son perfectionnisme ne souffre aucune exception. C'est souvent difficile, mais toujours récompensé. Avec lui, j'ai beaucoup appris. Pour tout cela, merci, Michel.

Jean-Louis Nicolas m'a initié à la théorie des nombres. François Morain et lui m'ont énormément aidé dans les domaines de l'arithmétique et de la cryptographie. Je les remercie pour leur patience et leur disponibilité extraordinaires.

Jean-Michel Muller a eu la gentillesse de relire le manuscrit très minutieusement, de corriger les erreurs, et de m'obliger à clarifier des points obscurs. Qu'il soit remercié jusqu'à la 42e génération !

Mais c'est surtout à l'ambiance et la camaraderie qui règnent aussi bien au LMC qu'au LIP que je dois d'avoir trouvé très agréables ces deux ans et demi de thèse. Mes remerciements grenoblois vont à Bernard T. et Afonso F., mes premiers copains de bureau, Yvan H., Isabelle G., avec qui j'ai partagé beaucoup et surtout un bureau, l'unique Michel G., Denis T., Jean-Louis R. (merci pour PaC), François S.-R., Guoting C., Pascale S., Denis D., Jean-Yves B., Ptit Gars, Brigitte P., et bien sûr JDD, notre chef bien aimé, pour l'accès à sa bibliothèque professionnelle ; de discussion en discussion, j'ai beaucoup appris à leur contact.

A Lyon bien sûr, je retrouve certains transfuges : Bernard T., Afonso F., Yvan H. et Michel G. Merci à eux et à tous ceux qui créent au LIP une extraordinaire ambiance de travail : Michel C. bien sûr, Yves R., Jean-Michel M., Serge M., Pierre F., Jean-Marc A. et tous les autres.

Je n'oublie pas non plus les gourous des Mac, Sun, et autres multiprocesseurs : Claire et Joëlle à Grenoble, Jean-Louis et Giles à Lyon. Bravo pour leur efficacité.

Il est impossible de trouver des mots assez puissants pour dire combien j'ai apprécié les efforts de Nejla et Antoinette qui ont tapé le texte de cette thèse, Angèle et Valérie, nos secrétaires charmantes et efficaces, Françoise Renzetti et toute l'équipe de la Médiathèque pour leur disponibilité et leur rapidité d'action.

Mais c'est à ma femme que je réserve le plus grand merci. Elle m'a continuellement aidé et m'a permis de mener à bien cette thèse dans de bonnes conditions. Merci, Kiki.

Enfin, je remercie les membres du service de scolarité de l'INPG et du service de reprographie de l'IMAG pour la perfection de leur travail.

Sommaire

Sommaire

Notation

Introduction 1

1. But de l'étude 1

2. Présentation du manuscrit 3

3. Présentation de la machine-cible : le FPS T40 4

Chapitre 1 : Génération de nombres premiers en parallèle 9

1. Introduction 9

2. Le crible d'Eratosthène 10
   2.1. Historique 10
   2.2. Présentation de l'algorithme du crible d'Eratosthène 11
   2.3. Le crible d'Eratosthène et ses parallélisations 12

3. Le crible d'Eratosthène sur un multiprocesseur à mémoire partagée 13
   3.1. L'algorithme 13
   3.2. Modification de l'algorithme de Bokhari 15
   3.3. Répartition des entiers par familles 15
   3.4. Répartition des entiers par sous-suites 16
      3.4.1. Exemple pratique 16
      3.4.2. Analyse mathématique 18
      3.4.3. Implantation algorithmique - Evaluation 20
   3.5. Conclusion 22

4. Le crible d'Eratosthène sur un multiprocesseur à mémoire distribuée 22
   4.1. Le crible classique sur le FPS T40 22
      4.1.1. Implantation 23
      4.1.2. Evaluation - Résultats 24
   4.2. Répartition des entiers par sous-suites 26
   4.3. Conclusion 28

5. Génération des nombres premiers par divisions successives 29
   5.1. Génération multipipelinée des nombres premiers sur un anneau 29
      5.1.1. Algorithme 30

5.1.2. Cet algorithme est incorrect 31
5.1.3. Modification de l'algorithme 31
5.1.4. Résultats 32
5.2. Un nouvel algorithme : communications par paquets 32
5.2.1. Algorithme 33
5.2.2. Evaluation de cette solution 34
5.3. Diminution du nombre de divisions 36
5.3.1. La modification 36
5.3.2. Résultats 37
5.4. Conclusion 39

6. Conclusion 39

Chapitre 2 : Détection de la terminaison distribuée et terminaison 41

1. Introduction 41

2. Etat de l'art 42
2.1. Le modèle de calcul distribué 42
2.2. Les premières solutions 43
2.2.1. Jeton valué 43
2.2.2. Estampillage 45
2.2.3. Les marqueurs 45
2.3. De nouvelles idées 46
2.3.1. Environnement asynchrone 46
2.3.2. Relâche des contraintes dans un environnement synchrone 47
2.4. Conclusion 47

3. Le problème réel à résoudre 48

4. Algorithme de détection de la terminaison et de terminaison 50
4.1. Mécanisme de gestion de la terminaison d'un processeur 51
4.2. Suppression de la bulle introduite par l'arrêt d'un processeur 51
4.3. Détection de la terminaison 51
4.4. Preuve de correction 51
4.4.1. La terminaison globale sera obligatoirement détectée (Propriété de vivacité) 52
4.4.2. La terminaison globale n'est pas détectée si elle n'est pas intervenue (Propriété de sûreté) 52
4.5. Algorithme de détection de la terminaison 52
4.5.1. Etats des processeurs 52
4.5.2. Types des messages 53
4.5.3. L'algorithme pour le processeur PE_i 53
4.6. Terminaison de l'application 54
4.7. Conclusion 54

5. Dérivation d'un algorithme de détection de la terminaison et de terminaison à l'aide d'un jeton valué 55
5.1. Mécanisme de gestion de la terminaison d'un processeur 55
5.2. Mécanisme de terminaison de l'application 55
5.3. Algorithme 56
5.4. Preuve de correction de l'algorithme de détection de la terminaison distribuée 57
5.4.1. La terminaison globale sera obligatoirement détectée (Propriété de vivacité) 57

5.4.2. La terminaison globale n'est pas détectée si elle n'est pas intervenue (Propriété de sûreté) 57
5.5. Evaluation du nombre de communications nécessaires pour détecter la terminaison, et pour terminer 57
5.5.1. Pire cas 57
5.5.2. Cas général 58
5.6. Conclusion sur l'algorithme avec jeton valué 59

6. Conclusion 60

Chapitre 3 : A propos d'une méthodologie de parallélisation d'applications Maître / Esclaves 63

1. Introduction 63

2. Etat de l'art 64
2.1. Environnements liés au parallélisme 64
2.2. Langages spécialisés et logiciels de simulation 65
2.3. Langages parallèles 65
2.4. Outils d'aide à l'analyse 66
2.5. Les problèmes théoriques et philosophiques 66

3. Méta-algorithme de parallélisation d'applications Maître / Esclaves sur un réseau linéaire 67
3.1. Le réseau linéaire 68
3.2. Le problème 68
3.3. La structure des données à communiquer 69
3.4. Le méta-algorithme 70
3.5. Applications 72
3.5.1. Le crible d'Eratosthène 73
3.5.2. Calcul de $\Omega(k)$ pour $1 \le k \le N$ 77
3.5.3. Calcul de N(N, k) 78
3.5.4. Conclusion 79

4. Méta-algorithme de parallélisation d'applications Maître / Esclaves sur une grille non torique à 2 dimensions 79
4.1. La topologie de communication 80
4.2. La structure des données à communiquer 80
4.3. Le choix du Maître suivant 80
4.4. Reconfiguration dynamique du réseau d'interconnexion 81
4.4.1. Processeur normal 81
4.4.2. Processeur de rotation 82
4.5. Le méta-algorithme 82
4.6. Application : le crible d'Eratosthène 83

5. Comparaison du réseau linéaire et de la grille 86
5.1. Comparaison des temps d'exécution 86
5.2. Comparaison des implantations d'une même application 86

6. Conclusion 87
6.1. Réseau linéaire et grille 87
6.2. Autres topologies 87
6.2.1. L'arbre 87
6.2.2. L'hypercube 88

Chapitre 4 : Le crible quadratique : Etude de la parallélisation sur un multiprocesseur hypercube à mémoire distribuée illimitée 91

1. Introduction 91
   1.1. Méthodes de factorisation par différence de carrés 92
   1.2. Méthodes de factorisation sur des groupes 93
   1.3. D'autres méthodes 93
   1.4. Les architectures dédiées 93

2. Présentation de l'algorithme du crible quadratique multipolynomial (MPQS) 95

3. Parallélisation en mémoire distribuée illimitée de l'étape d'initialisation de l'algorithme du MPQS 97
   3.1. Présentation de l'initialisation 98
   3.2. Le nombre k d'éléments de la base de facteurs 99
   3.3. Estimation du plus grand élément p_k de la base 100
   3.4. La borne B 100
   3.5. La longueur 2M de l'intervalle de crible 100
   3.6. Le calcul des k éléments de la base de facteurs 102
      3.6.1. Calcul des k éléments sur 2 processeurs 103
      3.6.2. Calcul des k éléments sur P processeurs 103
      3.6.3. Sommation sur un processeur de P nombres répartis sur P processeurs 104
      3.6.4. Diffusion d'un nombre à tous les processeurs 105
      3.6.5. Implantation mathématique du calcul des k éléments 106
      3.6.6. Conclusion 106
   3.7. Le calcul des racines carrées de N modulo les éléments de la base 106
      3.7.1. Répartition générale des k éléments en fonction du nombre d'équations à résoudre 107
      3.7.2. Répartition des k éléments sur P processeurs, le processeur PE_0 traitant tous les éléments ayant $\alpha_{max} > 1$ 109
      3.7.3. Calcul et répartition des k éléments en parallèle avec la résolution des équations 111
   3.8. Conclusion 113

4. La boucle de génération de la matrice des factorisations complètes sur la base 113
   4.1. Présentation de la parallélisation de cette boucle 113
   4.2. La génération des polynômes w(x), présentation 115
      4.2.1. Théorie 115
      4.2.2. Le calcul du coefficient a 115
      4.2.3. Le nombre de polynômes nécessaires 116
   4.3. Algorithmique de la génération des polynômes 117
      4.3.1. Génération des polynômes lors de l'initialisation 118
      4.3.2. Génération des polynômes un par un dans la boucle 118
         4.3.2.a. Un processeur dédié calcule les polynômes et les envoie à un processeur à la fois 119
         4.3.2.b. Un processeur dédié calcule les polynômes et les envoie à tous les processeurs en même temps 122
         4.3.2.c. Redondance du calcul des polynômes sur chaque processeur 122
      4.3.3. Conclusion 123
   4.4. Le crible de l'intervalle [-M, M[........ 123
      4.4.1. Distribution des polynômes 124
      4.4.2. Distribution de l'intervalle de crible 125
      4.4.3. Distribution de la base de facteurs 125
      4.4.4. Conclusion 127

4.5. Le choix des w(x) candidats à la factorisation et leur factorisation 127
4.5.1. Choix des candidats 127
4.5.2. Leur factorisation 128
4.5.2.a. Par divisions successives 128
4.5.2.b. Par égalité de congruences 128
4.5.2.c. Factorisation en fonction de la variable de boucle qui est distribuée 129
4.5.3. Conclusion 129
4.6. L'élimination de Gauss et la factorisation de N 129

5. Conclusion 130

Chapitre 5 : Le crible quadratique : Parallélisation et implantation sur l'hypercube FPS T40 131

1. Introduction 131

2. La boucle de génération de la matrice des factorisations complètes 132
2.1. Construction de la matrice des factorisations par distribution des polynômes 133
2.1.1. Découpage de l'intervalle de crible et de la base en sous-ensembles 134
2.1.2. Evaluation des coûts d'accès aux sous-ensembles de la base et de l'intervalle 134
2.1.2.a. Communications entre voisins 134
2.1.2.b. Lectures sur disques 134
2.1.3. Conclusion 137
2.2. Construction de la matrice des factorisations par distribution de l'intervalle de crible 137
2.3. Construction de la matrice des factorisations par distribution de la base de facteurs 138
2.3.1. Le processeur s'occupe de l'intervalle de crible complet 139
2.3.1.a. Taille des informations à acquérir 139
2.3.1.b. Echange des informations par communication entre voisins sur un 3-cube 140
2.3.1.c. Echange des informations sur un 2-cube 143
2.3.1.d. Echange des informations entre deux processeurs et un disque 145
2.3.1.e. Evaluation des solutions 146
2.3.1.f. Choix et conclusion 148
2.3.2. Le processeur s'occupe d'un sous-intervalle de l'intervalle de crible complet 148
2.3.3. Evaluation 150
2.3.3.a. Temps de calcul pour la génération d'un polynôme 150
2.3.3.b. Modification du travail des processeurs lors du crible par sous-intervalle 151

3. Conclusion de la pré-étude d'implantation sur l'hypercube FPS T40 152
3.1. Distribution des polynômes 152
3.2. Distribution de la base de facteurs 153
3.3. Distribution de l'intervalle à cribler 153
3.4. Evaluation 153

4. Les améliorations possibles pour l'algorithme du crible quadratique multipolynomial 154
4.1. Le multiplicateur 154
4.2. La variation des petits nombres premiers 154
4.3. La variation du grand nombre premier 155
4.4. La variation des deux grands nombres premiers 155
4.5. Conclusion 155

5. Implantation du crible quadratique multipolynominal sur l'hypercube FPS T40 156
5.1. Factorisation de nombres ayant entre 35 et 41 chiffres 156
5.2. Analyse des résultats et modification de l'algorithme 164
5.3. Résultats obtenus avec l'algorithme intégrant la nouvelle répartition des tâches 164

6. Conclusion 176

Conclusion 179

Bibliographie 181

Notation

Dans l'ensemble du manuscrit, les noms de variables suivants se réfèrent toujours au même type. Nous les présentons ici.

p_i est le $i^{ème}$ nombre premier (ordre croissant)
ou le $i^{ème}$ élément de la base de nombres premiers (crible quadratique).

P est le nombre de processeurs.

PE_i est le processeur de numéro i parmi P processeurs numérotés de 0 à P-1.

N est la valeur supérieure d'un intervalle d'entiers (crible d'Eratosthène)
ou la valeur d'un entier de grande taille (crible quadratique).

En ce qui concerne l'ordre d'une complexité :

$O(f(n))$: de l'ordre d'au plus f(n)

$\Omega(f(n))$: de l'ordre d'au moins f(n)

$\Theta(f(n))$: de l'ordre de f(n) exactement

$o(f(n))$: d'un ordre inférieur à f(n)

Introduction

La demande en puissance de calcul croît continuellement surtout en ce qui concerne les applications scientifiques, comme l'aérodynamique (aéronautique, automobile), la météorologie, la dynamique des fluides ... [Her 86], [Qui 87] et la simulation en général. Pour faire face à cette demande de puissance de calcul impressionnante, les chercheurs, voyant que la croissance de la densité d'intégration ne pourrait pas suffire, se sont tournés vers les ordinateurs multiprocesseurs. En effet, plusieurs processeurs travaillent théoriquement plus vite qu'un seul, puisqu'ils peuvent se répartir les tâches à effectuer.

Présentons rapidement les architectures parallèles. On distingue d'une part les architectures à mémoire partagée où les processeurs partagent des zones de mémoire commune, d'autre part les architectures à mémoire distribuée, où les processeurs disposent d'une mémoire locale propre [HwB 84].

Dans le cas d'une architecture à mémoire partagée, le coût de l'échange de données entre processeurs est faible puisqu'il suffit d'échanger l'adresse de la donnée. Le parallélisme pourra être fin, c'est-à-dire s'exprimer à un niveau bas, correspondant à l'exécution de différentes instructions en parallèle [Vin 88]. Les ordinateurs à mémoire partagée sont les plus utilisés des ordinateurs multiprocesseurs, car ils offrent des facilités de programmation parallèle : le compilateur Fortran de ces machines sait paralléliser des boucles imbriquées. Ces ordinateurs sont par exemple le Cray I, Cray II, Cray XMP, Cray YMP, l'IBM 3090 multiprocesseur, ...

Dans le cas d'une architecture à mémoire distribuée, l'échange de données se fait par la transmission d'un message contenant ces données, entre deux processeurs. Le coût est plus important que celui de l'échange d'informations dans un environnement à mémoire partagée ; et il dépend des caractéristiques physiques de la machine (débit des voies de communication, distance entre les protagonistes de la communication, topologie de connexion, ...). La philosophie de programmation est différente. Il n'existe pas encore de vrais outils d'aide à la parallélisation. Ce type de machines n'est pas encore très répandu. Il existe cependant maintenant de telles machines commerciales comme l'iPSC/2, le TNode ou le MégaNode, la Connection Machine 2, ... elles restent surtout des outils de recherche.

Dans cette thèse, nous nous intéressons principalement aux multiprocesseurs à mémoire distribuée. Toutes les implantations sont réalisées sur ce type de machines. Quelques digressions théoriques concernent aussi des ordinateurs à mémoire partagée, mais elles sont rares. Pour rendre accessible la puissance que les machines multiprocesseurs à mémoire distribuée peuvent fournir, il faut les habiller d'une couche rendant transparents certains aspects du parallélisme. Il faut donner aux utilisateurs la possibilité de les exploiter, et non pas réserver ces machines à quelques spécialistes.

1. BUT DE L'ÉTUDE

Le but de notre travail de recherche est double :

- montrer, d'une part, qu'il est possible de mettre au point des méthodologies d'implantation sur de telles machines. Pour cela, il faut analyser les algorithmes et les classer par types. Puis, pour chaque type, il faut concevoir des outils adaptés. Actuellement, les outils disponibles sont des environnements de programmation, restrictifs [PRG 83], [CCG 88], ou

des simulateurs, adaptés à certaines machines [JMG 88], [GiS 88] ou des langages [SRV 88], [Lec 85], [BaW 86] qui ne permettent pas à un non-spécialiste du parallélisme de tirer parti de sa machine.
C'est pourquoi nous pensons que classer les applications et les algorithmes par type (Maître / Esclaves, "Diviser pour Régner", fermes de processeurs, ...) et développer des outils d'aide à l'implantation de ces algorithmes sur des topologies précises (réseau linéaire, grille, arbre, hypercube, ...) à mémoire distribuée peut apporter beaucoup aux utilisateurs de ces machines.

- prouver que, grâce à une analyse de l'algorithme séquentiel afin d'extraire le parallélisme dans toutes les étapes, l'implantation produit une exécution plus efficace, plus rapide. Par conséquent, la puissance importante de calcul des machines à mémoire distribuée peut, de cette manière, être bien utilisée.

Outre le domaine de l'algorithmique parallèle distribuée, notre travail concerne aussi l'arithmétique, utilisée ici en tant que domaine d'applications pour l'étude de la parallélisation d'algorithmes de la théorie des nombres (crible d'Eratosthène pour la génération de nombres premiers, crible quadratique multipolynomial pour la factorisation de grands entiers).

Le crible d'Eratosthène est souvent utilisé pour évaluer les performances des ordinateurs parallèles [TYN 83], [Bok 87]. Bokhari ne se limite pas à cette évaluation et propose d'améliorer l'efficacité de cet algorithme sur une mémoire partagée [Bok 87]. Nous reprenons ses travaux, pour montrer qu'il est possible d'améliorer encore l'efficacité en modifiant la répartition des données à traiter. Nous atteignons une répartition asymptotiquement optimale des données sur les processeurs. Ainsi, les processeurs mettent le même temps pour cribler avec chaque nombre premier.

Le crible quadratique multipolynomial [Pom 82] reçoit de plus en plus d'attention depuis le milieu des années 80. Il a surtout été implanté en parallèle sur des ordinateurs vectoriels [DHS 85], [Ger 83], [RLW 88] et sur des réseaux de stations de travail [Sil 87], [CaS 88], [Sil 88], [LeM 89]. Dans les deux cas, seule la boucle principale est parallélisée. Les calculs d'initialisation et de factorisation terminale sont effectués séquentiellement. Une seule implantation de cet algorithme a été réalisée sur une machine multiprocesseur à mémoire distribuée, le Ncube, au National Sandia Laboratories à Albuquerque (Nouveau-Mexique - Etats-Unis) par Davis et Holdridge [DaH 88]. Cependant, aucune étude théorique de la parallélisation du crible quadratique multipolynomial n'a été menée. Les implantations parallèles sont triviales.

Nous proposons d'extraire le parallélisme de chaque étape de cet algorithme, pour en dériver une implantation efficace sur des multiprocesseurs à mémoire distribuée.

La théorie des nombres fournit aussi des exemples utilisés pour valider la puissance des méthodologies et des outils parallèles que nous avons développés.

Dans un but initial d'aide à la parallélisation distribuée d'algorithmes séquentiels, nous avons étudié la conception d'outils capables de prendre en charge tout ou partie de la gestion d'une exécution parallèle (communications, état des processeurs, ...). Nous nous sommes restreints à des applications de type Maître / Esclaves, et nous avons étudié l'influence de la topologie sur trois variables :

- la facilité de conception de l'outil logiciel de gestion du parallélisme,
- la facilité d'utilisation de cet outil par un développeur qui veut implanter un algorithme de type Maître / Esclaves sur une machine parallèle à mémoire distribuée,
- les performances de l'application ainsi implantée, et surtout les diminutions de la puissance ou de l'efficacité, compte tenu que l'outil logiciel que nous proposons ajoute une couche lors de l'exécution.

Cette étude conduit finalement à deux méta-algorithmes (outils logiciels à un haut niveau d'abstraction de la machine-cible). Nous les utilisons sur des applications de la théorie des nombres, comme le crible d'Eratosthène.
Mais avant d'en arriver à cette étape, nous construisons systématiquement une étude de la parallélisation de cet algorithme de génération de tous les nombres premiers inférieurs à une limite donnée.

2. PRÉSENTATION DU MANUSCRIT

Ainsi le chapitre 1 est consacré à l'étude et la conception d'algorithmes parallèles de génération de nombres premiers. Dans un premier temps, nous nous inspirons du travail de Bokhari [Bok 87], qui évalue les performances du Flex / 32, un multiprocesseur à mémoire partagée, grâce à une implantantation parallèle du crible d'Eratosthène. Il essaie d'améliorer les performances en modifiant la répartition des données sur les processeurs.

Nous pensons que sa stratégie n'est pas optimale. Nous étudions d'autres répartitions pour aboutir à une répartition par sous-suites asymptotiquement optimale sur un multiprocesseur à mémoire partagée. Dans un deuxième temps, nous transposons cette étude sur un multiprocesseur à mémoire distribuée. Nous en déduisons une implantation efficace. Nous montrons que la stratégie de répartition des entiers par sous-suites, asymptotiquement théoriquement la meilleure sur une mémoire partagée, est très coûteuse en temps sur une mémoire distribuée, à cause d'une gestion trop lourde.

Nous constatons sur cette étude, que le crible d'Eratosthène est un algorithme de type Maître / Esclaves. Donc les Esclaves perdent du temps en attente d'informations. Nous pensons qu'il est possible d'utiliser une technique de pipeline afin d'accélérer la génération des nombres premiers. C'est pourquoi nous étudions une méthode différente de génération des nombres premiers, où les nombres composés sont éliminés par des divisions successives. Cette méthode est coûteuse, mais nous utilisons un mode multipipeline qui permet de ne pas laisser inactifs les processeurs. Au vu des temps d'exécution, cette méthode n'est vraiment pas la meilleure. Mais nous l'améliorons en résolvant de nombreux problèmes, comme la répartition des tâches et des données sur les processeurs, les communications, la redondance d'informations, ...

Un dernier problème spécifique aux environnements à mémoire distribuée apparait. Il concerne le contrôle de l'exécution de l'algorithme précédent, et plus particulièrement la détection de sa terminaison. C'est au chapitre 2 que nous abordons ce problème. Nous le résolvons pour l'application qui nous intéresse, c'est-à-dire la génération des nombres premiers par divisions successives sur une topologie particulière, l'anneau. Nos solutions sont symétriques (tout processeur peut détecter la terminaison), et elles n'introduisent pas de message de contrôle pour un processeur encore actif. L'idée est de compter le nombre de processeurs devenus passifs. Par extension, nous étudions aussi la terminaison de l'algorithme. Celle-ci consiste à laisser les processeurs et les voies de communication dans un état cohérent. Ainsi, il faut vider les canaux et faire savoir à chaque processeur qu'il ne doit plus recevoir ni envoyer de nouveau message.

Nous pouvons maintenant analyser les résultats de nos études, afin d'en dériver des outils et des méthodologies de parallélisation d'algorithmes de type Maître / Esclaves. Au chapitre 3, nous présentons notre étude permettant de mettre au point ces outils sur les topologies de réseau linéaire et de grille. Le crible d'Eratosthène est une des applications choisies et permet de comparer, d'une part les temps d'exécution et donc la différence entre les temps de gestion pour le réseau linéaire et pour la grille, et d'autre part la facilité d'utilisation de ces deux outils. D'autres algorithmes sont aussi implantés. Puis nous présentons des idées pour porter ces méta-algorithmes sur d'autres topologies comme l'arbre ou l'hypercube par exemple.

Les chapitres 4 et 5 sont consacrés à l'étude de la parallélisation d'un algorithme de la théorie des nombres : le crible quadratique multipolynomial dans un environnement multiprocesseurs à mémoire distribuée. Cet algorithme permet de factoriser des grands entiers. Il est déterministe, par opposition aux courbes elliptiques, algorithme probabiliste. Les parallélisations existantes

ont été implantées principalement sur des réseaux de stations de travail ou sur des ordinateurs vectoriels. Une seule étude, à notre connaissance, a conduit à une implantation sur une machine à mémoire distribuée. Elle a été menée au National Sandia Laboratories à Albuquerque (Nouveau-Mexique - Etats-Unis) par Davis et Holdridge, et a abouti à une implantation sur le NCUBE [DaH 88]. Notre but, ici, est de montrer qu'une étude approfondie de toutes les étapes de l'algorithme, et une parallélisation de chacune d'elles, peuvent apporter des gains en performances et une efficacité accrue, si on la compare à celle des versions parallèles existantes.

Au chapitre 4, dans un environnement théorique où la mémoire distribuée de chaque processeur est infinie, nous étudions et évaluons les différentes possibilités de parallélisation de chaque étape de cet algorithme. L'étape d'initialisation ne peut pas se faire sans communications. Nous les réduisons au minimum en répartissant les données de sorte que les calculs soient indépendants sur deux processeurs différents. De même, l'étape finale de factorisation nécessite des communications. Mais l'étape principale, qui est aussi la plus coûteuse en temps d'exécution et en place mémoire utilisée, peut être parallélisée d'au moins trois manières, suivant la variable de boucle qui est distribuée entre les processeurs. Une seule de ces parallélisations théoriques conduit à un algorithme exécutable sans communication sur une ferme de processeurs. Les autres nécessitent des synchronisations et des communications très coûteuses.

Au chapitre 5, utilisant les résultats du chapitre précédent, nous étudions les parallélisations possibles sur des machines où la mémoire de chaque nœud est limitée à 1 Mo. Notre machine-cible est le FPS T40. Nous montrons que la stratégie de distribution des polynômes est optimale tant que la mémoire est suffisante pour stocker la base de facteurs et une partie de l'intervalle de crible. Pour la factorisation d'entiers très grands (de 100 chiffres environ), des problèmes de distribution d'autres variables, comme la base de facteurs, interviennent et introduisent des communications. Nous évitons cependant un problème rencontré par Caron et Silverman
[CaS 88] dans leur implantation : deux processeurs risquent de travailler sur les mêmes données et obtenir les mêmes résultats, d'où une redondance inutile. Nous avons réparti les données par familles, de manière à éviter une telle redondance.

Après l'étude théorique, nous donnons des résultats de l'implantation sur l'hypercube de FPS. Nous sommes capables de factoriser des entiers ayant jusqu'à 60 chiffres, mais en des temps importants, de l'ordre de 30 heures. Des améliorations algorithmiques peuvent être apportées pour rendre ces temps plus raisonnables. Nous obtenons des accélérations superlinéaires, grâce à l'augmentation de la taille mémoire lorsqu'on augmente le nombre de processeurs.

3. PRÉSENTATION DE LA MACHINE-CIBLE : LE FPS T40

Donnons tout d'abord une description de la machine-cible de toutes nos implantations : le FPS T20 devenu par la suite le FPS T40. Le FPS T20 est une machine multiprocesseur à mémoire distribuée de la société américaine FPS (Floating Point Systems), comportant $20_8 = 16_{10}$ processeurs. La configuration de cette machine a été portée à 40_8 soit 32_{10} processeurs.

Le FPS T20 ou T40 est basé sur le Transputer T414. Le Transputer IMS T414 est un microprocesseur 32 bits avec 2 Koctets de RAM interne, une interface mémoire configurable et 4 canaux de communication Inmos standards. Le jeu d'instructions permet une implantation efficace des langages de haut niveau et assure un support direct au mode de simultanéité d'OCCAM (sur un seul Transputer ou sur un réseau).

Le T414 inclut une arithmétique entière à hautes performances et les opérations flottantes sont micro-codées. Un Transputer fonctionnant à 20 MHz permet d'atteindre des performances de 10 Mips. Il peut accéder directement à 4 Goctets de mémoire. L'interface mémoire 32 bits permet un accès de 4 octets tous les 150 nanosecondes, soit 26,6 Moctets/s (pour un Transputer fonctionnant à 20 MHz).

Les canaux de communications externes permettent de construire des réseaux de Transputers par des connexions point par point, sans aucune logique externe. Les canaux sont prévus pour la vitesse standard de 10 Mbits par seconde.

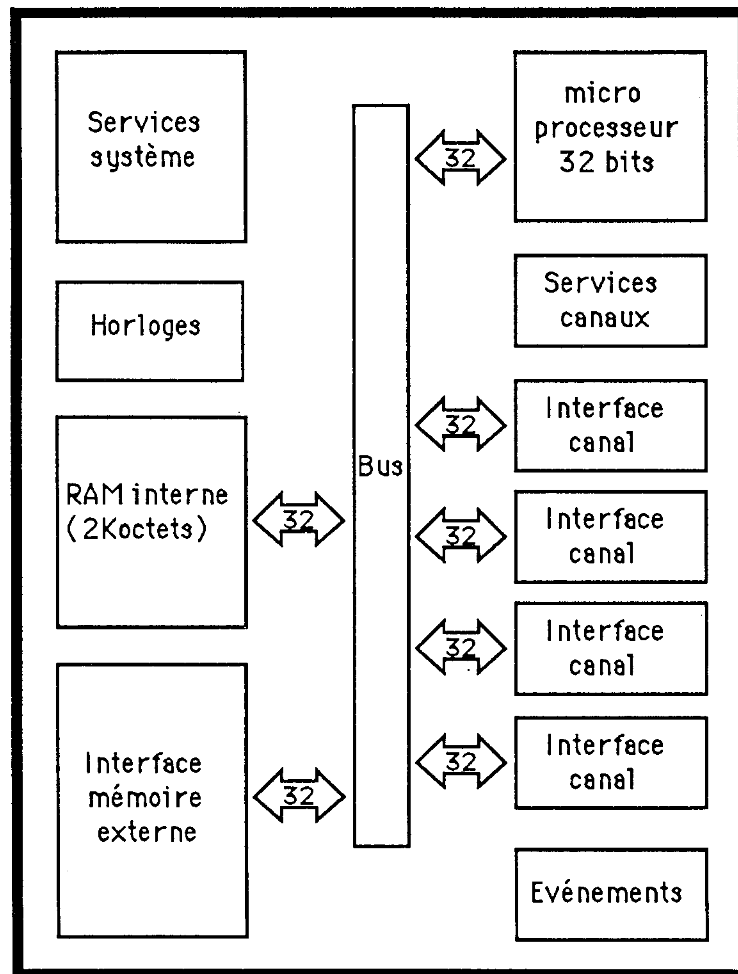


Figure 0.1. Le transputer IMS T414

Décrivons maintenant l'architecture de l'hypercube. Le "T20", respectivement "T40", est une machine capable d'effectuer des opérations vectorielles pipelinées. Cette machine possède une mémoire distribuée sur chacun des processeurs.

Le réseau de connexion entre processeurs est de type hypercube. L'intérêt de ce réseau est qu'il comporte peu de liens matériels entre les processeurs et que le plus long chemin entre ceux-ci est $\log_2$(nombre de processeurs), ce qui permet d'obtenir des machines massivement parallèles.

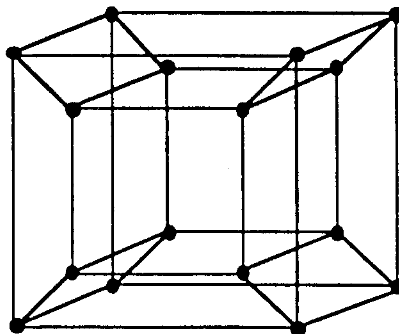


Figure 0.2. Hypercube de dimension 4

Il n'y a pas de synchronisation globale entre les processeurs des ordinateurs de la série T. Il peut donc être considéré, dans la classification de Flynn, comme une machine MIMD asynchrone (mais par contre Single Program Multiple Data), avec des processeurs qui communiquent par échanges de messages.

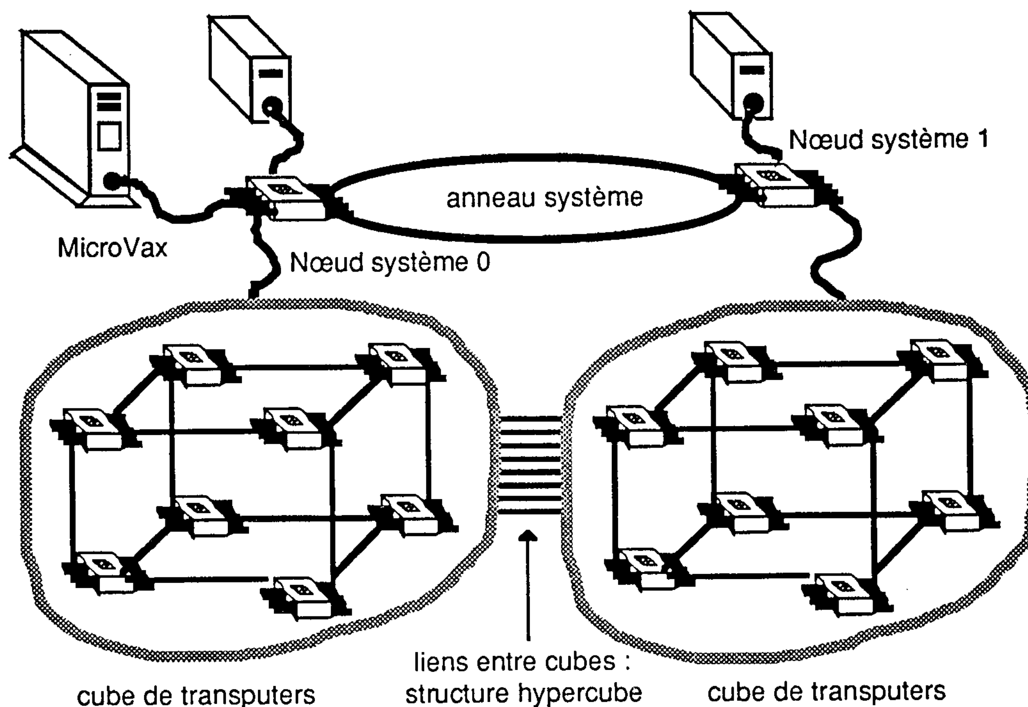


Figure 0.3. Organisation générale du T20

Les ordinateurs de la série T doivent être reliés à une machine hôte qui sert d'interface avec les utilisateurs : un MicroVax (de chez DEC). Le système d'exploitation utilisé est un dérivé d'Unix (Ultrix).

Les nœuds systèmes sont reliés entre eux suivant une structure d'anneau, et chacun d'entre eux possède un disque dur. Les entrées-sorties entre le T20 et le MicroVax s'effectuent par un bus connecté au nœud système 0 qui est le processeur privilégié pour les communications vers le calculateur hôte.

Un processeur comprend trois parties principales :
- un Transputer T414, qui assure les communications avec les processeurs voisins ou le nœud système, tient lieu d'unité arithmétique et logique et assure le contrôle du programme,
- une unité de calcul vectoriel (VPU, Vector Processing Unit) permet des calculs flottants très rapides sur des vecteurs de réels codés sur 64 bits (format IEEE double précision), grâce à un additionneur et un multiplieur pipelinés,
- une mémoire vidéo (VRAM) très performante à double accès (aléatoire vis à vis du Transputer et série vis à vis des registres vectoriels du VPU).

Un nœud système comprend quant à lui deux parties principales :
- un disque dur Winchester de 85 Moctets,
- un Transputer.

Le nœud système sert d'interface avec le MicroVax pour tout ce qui concerne les entrées-sorties et assure le contrôle des données sur son disque. Lors de l'exécution d'un programme, il charge le code reçu du microVax sur tous les processeurs de son cube et le transmet aux autres nœuds via l'anneau système. Ainsi tous les processeurs du T20 reçoivent le même code, mais ils connaissent cependant leur numéro absolu dans l'architecture, et ils peuvent donc n'exécuter que la partie du code les concernant.

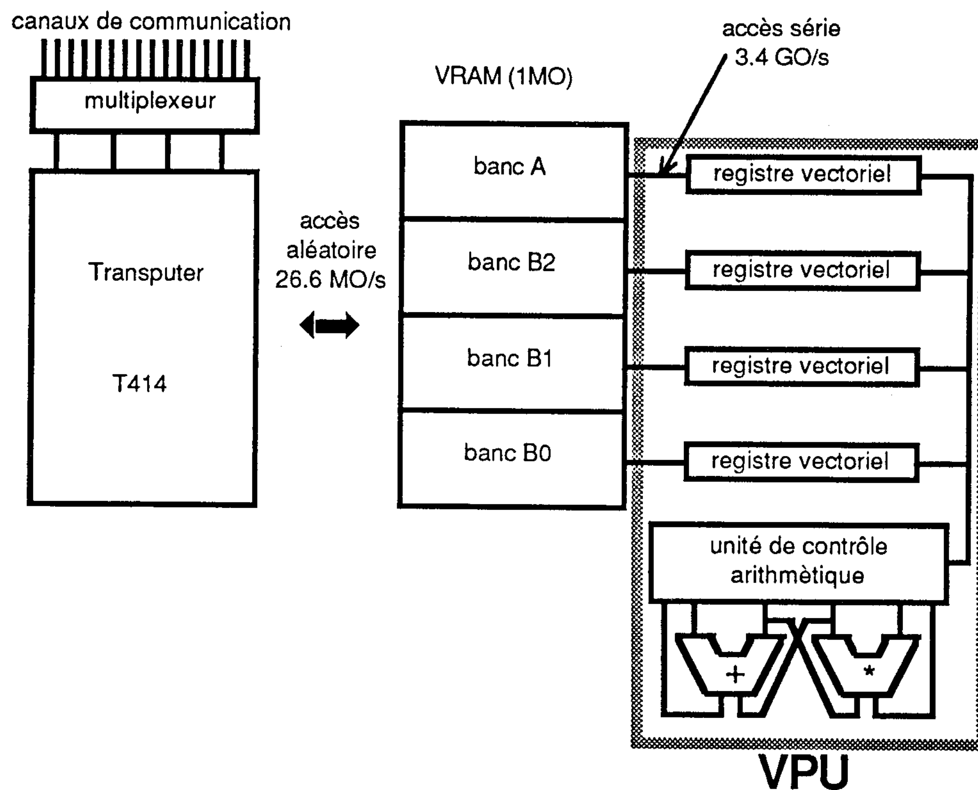


Figure 0.4. Structure d'un nœud

Sur chaque nœud la gestion des communications est assurée par le Transputer. Celui-ci possède 4 canaux de communications bi-directionnels (émission et réception). Afin d'augmenter la capacité du système, ces 4 sorties ont été multiplexées avec des éléments à 4 sorties. Ainsi chaque processeur dispose donc de 16 canaux de communications, cependant seuls 4 d'entre eux peuvent être utilisés simultanément (un multiplexeur ne pouvant avoir qu'une seule de ses entrées actives).

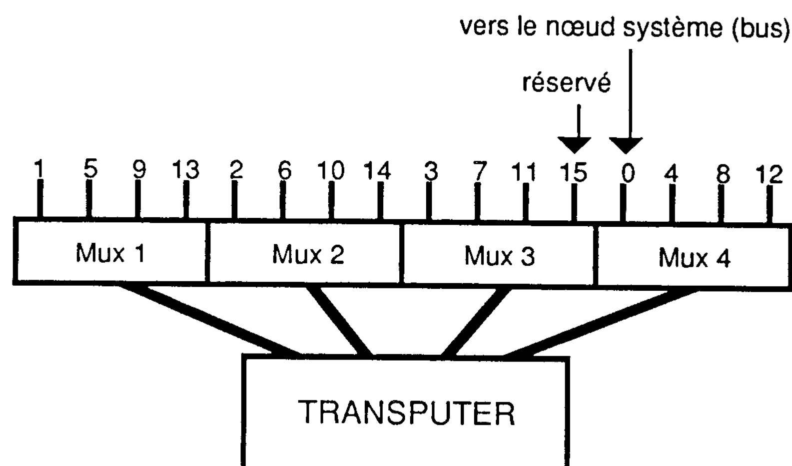


Figure 0.5. Multiplexage des liens du Transputer

Les multiplexeurs étant configurés à chaque communication, on obtient un temps d'initialisation élevé par rapport au temps de transfert des données. Si l'on suppose que le coût d'une

communication peut s'exprimer sous la forme : $\mu n + \beta$, alors par expérimentation, on obtient les valeurs μ=1,44 et β=860 microsecondes, où n est le nombre d'octets transférés.
Une communication entre processeurs s'effectue en trois étapes :

- choix de la topologie,
- définition des canaux de communication,
- appel des fonctions de communication.

Il existe différentes routines de communication permettant chacune d'atteindre un nombre divers de processeurs.

On peut ainsi communiquer :

- d'un processeur à un autre (one_to_one),
- d'un processeur à tous les autres (one_to_all),
- de tous les processeurs à un seul d'entre eux (all_to_one),
- de tous les processeurs à tous les autres (all_to_all).

La mémoire est distribuée sur chaque nœud de l'hypercube et chaque processeur dispose donc de 1 Moctet de mémoire vive, accessible depuis le Transputer, ainsi que de 2 Koctets internes au Transputer, qui lui permettent un accès aléatoire plus rapide. La mémoire principale est du type vidéo RAM et son accès est double : aléatoire depuis le Transputer et série depuis le VPU. La mémoire est adressable par mots de 32 bits.

C'est donc sur cette machine que nous avons programmé toutes les applications décrites dans la suite. Pour les trois premiers chapitres de cette thèse, nous avons travaillé sur un FPS T20 comportant 16 processeurs. Puis, la machine a été portée à 32 processeurs. C'est pourquoi, dans les chapitres 4 et 5, nous utilisons 32 processeurs sur le FPS T40.

Chapitre 1
Génération de nombres premiers en parallèle

1. INTRODUCTION

Les problèmes arithmétiques, même ceux qui s'expriment simplement, conduisent à des calculs de grande taille (stockage ou durée d'exécution). L'utilisation des ordinateurs, pour accélérer la résolution de ces problèmes ou permettre de résoudre des problèmes de plus grande taille, a profondément modifié l'Arithmétique, conduisant à l'émergence d'un nouveau domaine de recherches : concevoir des algorithmes efficaces de calculs arithmétiques, analyser ces algorithmes, les implémenter sur des ordinateurs cibles ... Cette nouvelle "discipline" (dont l'essor doit beaucoup aux recherches sur la cryptographie) est profondément sensible à l'évolution des concepts informatiques.

Dans la première partie des années 80, le développement des supercalculateurs a conduit à exploiter au mieux les techniques de pipeline, vectorisation, registres vectoriels. Actuellement, l'essor des calculateurs massivement parallèles remet en cause ces techniques pour privilégier la distribution des calculs et des données sur des processeurs différents en vue de la diminution du volume des communications.

En ce qui concerne les tests de primalité, le parallélisme apporte beaucoup. C'est en effectuant des calculs dans un environnement distribué de 12 stations SUN que François Morain a prouvé pour la première fois la primalité d'un nombre Titanic en moins d'un an CPU [Mor 90]. (Un nombre Titanic est un entier dont le nombre de chiffres dépasse 1000. Un nombre ordinaire est un entier d'aucune forme particulière : ni nombre de Fermat, ni nombre de Mersenne, ...). Dans le même ordre d'idée, à l'aide des ordinateurs actuels de plus en plus puissants, multiprocesseurs ou non, des équipes se sont attaqué à la preuve de non primalité de certains nombres particuliers (de Fermat, Mersenne, Fibonacci,...). Ce problème est un peu différent du test de primalité. Mais il nécessite lui aussi une somme considérable de calculs. Par exemple, le nombre de Fermat $F_{20} = 2^{2^{20}} + 1$ a été prouvé composé par la technique de Pépin [Pép 1877] sur un Cray [YoB 88]. Certains problèmes de factorisation sont étudiés dans les deux derniers chapitres de cette thèse.

Le parallélisme apporte un gain certain dans les problèmes concernant les nombres premiers. Par exemple, dans le calcul du nombre $\pi(x)$ de nombres premiers inférieurs ou égaux à x, Lagarias et al. montrent que l'algorithme peut être accéléré grâce au parallélisme [LMO 85]. On utilise les nombres premiers en cryptographie pour le codage ou pour générer des nombres pseudo-aléatoires [Haa 87] ou encore pour concevoir des algorithmes distribués [Ray 89]. On peut aussi s'en servir pour éliminer les petits facteurs d'un grand nombre, avant de le factoriser par une autre méthode...

Dans ce chapitre, nous nous attachons au problème de la génération de tous les nombres premiers inférieurs à une limite donnée. Pour ce faire, nous utilisons deux techniques

différentes : le crible d'Eratosthène d'une part, et une méthode liée aux divisions successives d'autre part.

Bokhari utilise le crible d'Eratosthène comme benchmark pour le Flex/32. Mais il propose aussi une modification de l'algorithme pour améliorer la répartition des tâches. Nous pensons que la répartition obtenue n'est pas bonne : nous voulons l'améliorer. Notre but est que chaque processeur reste inactif le moins longtemps possible.

Dans le deuxième paragraphe, nous présentons la version du crible d'Eratosthène que nous étudions sur une mémoire partagée, puis sur une mémoire distribuée. De manière à pouvoir comparer ultérieurement diverses répartitions théoriques des tâches (avec celle de Bokhari), nous travaillons avec l'algorithme qu'il a implanté.

Dans le troisième paragraphe, nous montrons que la répartition des tâches que Bokhari propose n'est pas optimale. Nous présentons les deux types de répartition de données non encore étudiées (par familles d'entiers ou par sous-suites) qui influent sur la répartition des tâches. Nous évaluons ces deux stratégies. Ainsi, tous les types de répartition théorique des données sont exhaustivement étudiés.

Le quatrième paragraphe est consacré à une étude similaire sur le FPS T20. Nous voulons aussi comparer les temps d'exécution avec ceux de Bokhari, c'est pourquoi nous proposons des implantations de l'algorithme. Nous souhaitons voir si le temps mesuré correspond au temps théorique.

Or le crible d'Eratosthène est un algorithme de type Maître / Esclaves. Dans nos implantations en mémoire distribuée, il y a donc des pertes de temps pour les Esclaves, car ils doivent attendre que le Maître trouve le nombre premier suivant, et des pertes de temps en synchronisation lors de l'envoi de ce nombre. Nous pensons qu'un algorithme pipeline ferait perdre moins de temps. Nous développons au cinquième paragraphe un nouvel algorithme de génération des nombres premiers, que nous espérons très rapide, car utilisant mieux les processeurs. En fait, l'algorithme est plus lent, car il utilise le principe des divisions successives, alors que crible d'Eratosthène n'a besoin que d'additions pour éliminer les nombres composés.

2. LE CRIBLE D'ÉRATOSTHÈNE

2.1. HISTORIQUE

Le crible d'Eratosthène est un algorithme inventé au 3ème siècle av. J.C. par le mathématicien géographe et astronome grec Eratosthène. L'algorithme est simple et par conséquent hautement pédagogique [Dew 88]. Dijkstra [Dij 72] le choisit comme son premier exemple de construction méthodique de programme. Wirth [Wir 73], Hoare [Hoa 72] et Pritchard [Pri 79] par exemple, l'utilisent pour expliquer les concepts de base de la programmation. Leur but n'én est pas l'implantation. Hoare [Hoa 78] l'utilise comme exemple de programmation en CSP.

A l'opposé, d'autres auteurs présentent des implantations, séquentielles ou parallèles [HiK 80], et des algorithmes dérivés plus performants soit en temps d'exécution (complexité arithmétique), soit en taille mémoire, soit les deux. Mairson, dans [Mai 77], rappelle que la complexité arithmétique du crible d'Eratosthène est O(N log log N) additions (nombres d'entiers éliminés), pour chercher les nombres premiers inférieurs à N. La complexité en est alors O(N log N log log N) bits. Cette technique n'est pas assez efficace, car chaque entier est éliminé par tous ses facteurs premiers. Elle requiert O(N) espace mémoire.

Une modification, employant des listes chaînées dans les deux sens, conduit à un algorithme dont la complexité arithmétique est O(N) sous la condition qu'une multiplication prenne une unité de temps. Ce n'est qu'en utilisant des techniques de prétraitement que cette complexité

arithmétique se réduit à O(N/log log N) et que la complexité en bits devient O(N log N log log log N). L'espace nécessaire est alors O(N log N/log log N) bits [Mai 77].

C'est en 1977 aussi que Bays et Hudson, dans [BaH 77], présentent un crible segmenté pour compter le nombre de nombres premiers dans des progressions arithmétiques.

En 1978, Gries et Misra, dans [GrM 78], présentent un autre crible multiplicatif linéaire, simple et élégant, et qui permet aussi de trouver la factorisation des entiers de l'ensemble criblé {2, ..., N} en un temps proportionnel à N.

Il faut attendre 1981 pour que Paul Pritchard propose une amélioration de l'algorithme du crible de Mairson. Il n'a plus besoin que de Θ(N/log log N) additions (pas de multiplications) [Pri 81]. Il en déduit un autre algorithme de même complexité arithmétique, mais dont la complexité en espace mémoire est Θ(N/log log N) bits ; cet algorithme est meilleur que celui d'Eratosthène. En 1982, le même auteur reprend cet algorithme et le présente avec son fondement mathématique [Pri 82]. En 1983, il propose un algorithme qui requiert Θ(N) additions, et un espace $o(\sqrt{N})$ bits, grâce à une méthode compacte de stockage. Il fait une synthèse de tous ces travaux, en montrant dans [Pri 87] que les algorithmes récents d'énumération des nombres premiers inférieurs à une limite donnée sont des cribles (les nombres premiers sont trouvés en éliminant les nombres composés) et que ces cribles sont linéaires (chaque entier composé est éliminé exactement une fois).

Bengelloun, en 1986, propose un algorithme de crible linéaire qui peut décider rapidement (espace et temps arithmétique linéaires) de la primalité de N+1 à partir du résultat du crible de {2, ..., N}. Il est possible d'appliquer certaines des techniques exposées ci-dessus pour augmenter l'efficacité de l'algorithme et la porter à O(N/log log N) temps et espace [Ben 86].

2.2. PRÉSENTATION DE L'ALGORITHME DU CRIBLE D'ÉRATOSTHÈNE

La procédure est la suivante. Les nombres 2 à N sont d'abord écrits. On recherche le premier nombre non encore éliminé de la liste (c'est-à-dire 2) et on parcourt la liste pour éliminer tous ses multiples (dans ce cas, 4, 6, 8, 10, ...). Une fois la liste parcourue de bout en bout, il faut revenir au point de départ, chercher le prochain nombre non éliminé (qui est 3), et éliminer ses multiples (6, 9, 12, ...). En revenant à nouveau au début de la liste, le prochain nombre non éliminé est 5 (car 4 a été supprimé lors du premier passage, en tant que multiple de 2). Il faut éliminer les multiples de 5, et ainsi de suite. Les entiers non éliminés à la fin de ce processus sont les nombres premiers.

Plusieurs variations permettent d'améliorer cet algorithme :

- n'écrire que les nombres impairs,
- ne balayer la liste à la recherche du prochain nombre non éliminé que jusqu'à $\sqrt{N}$. (Cette notion n'a été introduite que bien plus tard par Léonard de Pise, appelé aussi Fibonacci),
- ne commencer l'élimination des multiples du nombre premier p qu'à partir de p^2, car les multiples plus petits ont déjà été supprimés lors du crible par les nombres premiers plus petits que p.

Exemple : crible d'Eratosthène sur les entiers 2 à 49.

Ecriture de ces entiers :
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49

Suppression des multiples de 2 sauf 2 :
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25
~~26~~ 27 ~~28~~ 29 ~~30~~ 31 ~~32~~ 33 ~~34~~ 35 ~~36~~ 37 ~~38~~ 39 ~~40~~ 41 ~~42~~ 43 ~~44~~ 45 ~~46~~ 47 ~~48~~ 49

Suppression des multiples de 3 sauf 3 :
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25
~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ 35 ~~36~~ 37 ~~38~~ ~~39~~ ~~40~~ 41 ~~42~~ 43 ~~44~~ ~~45~~ ~~46~~ 47 ~~48~~ 49

Suppression des multiples de 5 sauf 5 :
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~
~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ ~~35~~ ~~36~~ 37 ~~38~~ ~~39~~ ~~40~~ 41 ~~42~~ 43 ~~44~~ ~~45~~ ~~46~~ 47 ~~48~~ 49

Suppression des multiples de 7 sauf 7 :
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~
~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ ~~35~~ ~~36~~ 37 ~~38~~ ~~39~~ ~~40~~ 41 ~~42~~ 43 ~~44~~ ~~45~~ ~~46~~ 47 ~~48~~ ~~49~~

Il ne reste que les nombres premiers inférieurs à 49 :
2 3 5 7 11 13 17 19 23
29 31 37 41 43 47

Figure 1.1. Exemple d'utilisation du crible d'Eratosthène

L'algorithme du crible d'Eratosthène peut être présenté sous la forme suivante :

Algorithme
  Ecrire les entiers de 2 à N
  Prendre le premier entier p non éliminé (p=2)
  Répéter
    Eliminer les multiples de p à partir de 2p (algorithme de base)
    A partir de p non compris, chercher le prochain entier non éliminé : c'est le nouveau p
  jusqu'à $p > \sqrt{N}$
Fin algorithme

Figure 1.2. Algorithme du crible d'Eratosthène

Ce crible est additif, puisque pour éliminer les multiples de p, on ajoute p à la valeur du multiple précédent (2p, 3p, 4p,...) et on élimine cet entier.

2.3. LE CRIBLE D'ÉRATOSTHÈNE ET SES PARALLÉLISATIONS

Plusieurs articles sont parus sur ce sujet. Parmi eux, certains utilisent le crible d'Eratosthène comme un benchmark pour de nouvelles machines parallèles. Dans le cas de Takahashi et al., il s'agit de la machine CORAL '83 [TYN 83]. C'est un arbre binaire de 15 processeurs à mémoire distribuée. Pour trouver tous les nombres premiers inférieurs à M, il faut cribler avec les nombres premiers inférieurs à $\sqrt{M}$, appelés ici les cribles. L'ensemble de ces cribles est divisé en L+1 sous-ensembles, où L est le nombre de niveaux de CORAL '83. Puis chaque sous-ensemble est alloué aux processeurs de chaque niveau, qui testent les nombres envoyés par les fils droit et gauche et ne renvoient à leur père que les nombres qui ne sont pas multiples des éléments du sous-ensemble qu'ils possèdent.

Ce sont évidemment les processeurs feuilles qui génèrent des suites différentes de nombres et qui les testent avant de les envoyer vers leur père. De cette manière, l'hôte (connecté à la racine de l'arbre binaire) reçoit une suite de nombres premiers. Le problème ici réside dans la répartition des cribles en sous-ensembles optimaux tels que la charge de travail des processeurs soit équilibrée. Une répartition quasi-optimale consiste à allouer deux fois plus de cribles à un processeur qu'à son père.

De même, Bokhari utilise le crible d'Eratosthène pour mesurer les performances de la machine Flex/32, un multiprocesseur de 20 nœuds, ayant chacun 1 Mo de mémoire locale, et pouvant accéder à 2 Mo de mémoire partagée par un un bus global [Bok 87].
Cet algorithme a aussi été utilisé pour évaluer les performances de l'Intel 80286 [Pat 82].

3. LE CRIBLE D'ÉRATOSTHÈNE SUR UN MULTIPROCESSEUR À MÉMOIRE PARTAGÉE

3.1. L'ALGORITHME

Bokhari implante le crible d'Eratosthène sur le Flex/32, un multiprocesseur ayant P nœuds, P=20. Parmi ces P processeurs, il existe un maître et P-1 esclaves. Son algorithme comprend deux étapes : l'écriture et le crible lui-même [Bok 87]. La figure 1.3 montre le timing de son exécution avec 7 processeurs, pour énumérer les nombres premiers inférieurs à 200.

Temps	PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	Commentaires	
1	M[1,i]	-	-	-	-	-	-	M lance PE_1	
2	M[2,i]	1:1	-	-	-	-	-	M lance PE_2; PE_1 commence	
3	M[3,i]	1:2	2:33	-	-	-	-	M lance PE_3; PE_2 commence	E
4	M[4,i]	1:3	2:34	3:64	-	-	-	M lance PE_4; PE_3 commence	C
5	M[5,i]	1:4	2:35	3:65	4:94	-	-	M lance PE_5; PE_4 commence	R
6	M[6,i]	1:5	2:36	3:66	4:95	5:123	-	M lance PE_6; PE_5 commence	I
7	0:178	1:6	2:37	3:67	4:96	5:124	6:151	PE_6 et PE_0 commencent	T
...	...	...	...	...	...	...	...		U
29	0:200	1:28	2:59	3:89	4:118	5:146	6:173		R
30	-	1:29	2:60	3:90	4:119	5:147	6:174		E
...	...	...	...	...	...	...	...		
33	-	1:32	2:63	3:93	4:122	5:150	6:177	Initialisation terminée	
34	-	-	-	-	-	-	-	M informé	
35	M[1,2]	-	-	-	-	-	-	M lance PE_1 : crible avec 2	
36	M[2,3]	2:4	-	-	-	-	-	M lance PE_2 : crible avec 3	
37	M[-,4]	2:6	3:6	-	-	-	-	M trouve 4 non premier	
38	M[3,5]	2:8	3:9	-	-	-	-	M lance PE_3 : crible avec 5	
39	M[-,6]	2:10	3:12	5:10	-	-	-	...	
40	M[4,7]	2:12	3:15	5:15	-	-	-		
41	M[-,8]	2:14	3:18	5:20	7:14	-	-		
42	M[-,9]	2:16	3:21	5:25	7:21	-	-		
43	M[-,10]	2:18	3:24	5:30	7:28	-	-		
44	M[5,11]	2:20	3:27	5:35	7:35	-	-		
45	M[-,12]	2:22	3:30	5:40	7:42	11:22	-		
46	M[6,13]	2:24	3:33	5:45	7:49	11:33	-		C
47	M[-,14]	2:26	3:36	5:50	7:56	11:44	13:26		R
...	-	...	...	...	...	...	...	15.15>200; M inactif	I
60	-	2:52	3:75	5:115	7:147	11:187	13:195	PE_6 termine avec 13	B
61	-	2:54	3:78	5:120	7:154	11:198	-	PE_5 termine avec 11	L
62	-	2:56	3:81	5:125	7:161	-	-		E
...	...	...	...	...	...	...	...		
67	-	2:66	3:96	5:150	7:196	-	-		
68	-	2:68	3:99	5:155	-	-	-		
...	...	...	...	...	...	...	...		
77	-	2:86	3:126	5:200	-	-	-		
78	-	2:88	3:129	-	-	-	-		
...	...	...	...	...	...	...	...		
101	-	2:134	3:198	-	-	-	-		
102	-	2:136	-	-	-	-	-		
...	...	...	...	...	...	...	...		
134	-	2:200	-	-	-	-	-		
135	-	-	-	-	-	-	-	Le programme se termine.	

M[x,i] : le maître ordonne à l'esclave x de commencer à écrire des entiers consécutifs.
M[x,y] : le maître ordonne à l'esclave x de supprimer les multiples de y.
M[-,y] : le maître trouve que le nombre y a été éliminé (y n'est pas premier).
x:y : écriture : le processeur x écrit le nombre y. crible : le multiple y de x est éliminé.
- : étape d'inactivité.

Figure 1.3. Timing de l'exécution de l'algorithme de Bokhari P=7, N=200

Première étape : l'écriture.
Le maître ordonne aux P processeurs, chacun leur tour, d'écrire une séquence d'environ N/P nombres consécutifs dans la mémoire commune. Chaque processeur connait (comment calculer) les bornes inférieure et supérieure de sa séquence. Dès que les P-1 esclaves ont reçu l'ordre d'écriture (et ont ainsi commencé à écrire), le maître se transforme en esclave et se met à écrire sa propre séquence d'entiers. Les séquences sont équilibrées au possible, de manière à minimiser la durée totale de la phase d'écriture.

Deuxième étape : le crible.
Le maître trouve le premier entier i non éliminé, dans la mémoire commune, et donne l'ordre à un processeur d'éliminer tous les multiples de i en commençant à 2i. Puis le maître trouve le nombre premier suivant i' (le prochain entier non éliminé) et donne l'ordre à un autre processeur d'éliminer les multiples de i', en commençant à 2i', et ainsi de suite jusqu'à $\sqrt{N}$. S'il n'a pas assez de processeurs, le maître se transforme en esclave et crible lui aussi.

Les caractéristiques de cet algorithme sont les suivantes :

- les esclaves accèdent de manière simultanée à la mémoire ;
- la synchronisation globale est faite par le maître, ce qui conduit à une perte de temps ;
- des vitesses de travail différentes entre les processeurs risquent de conduire à la détection par le maître de faux nombres premiers, et ainsi d'entraîner un travail inutile : en effet supposons que le maître ordonne au processeur PE_1 de cribler avec 2, que le maître ait le temps de lancer PE_2 avec 3 et de trouver 4 non éliminé avant que PE_1 ne commence, alors le maître lance PE_3 avec 4, ce qui induit un travail supplémentaire inutile ;
- la répartition des tâches est mauvaise : il y a plus de multiples de 2 que de multiples de 3, de 5 ou de 7... Le processeur qui élimine les multiples de 13, même s'il commence bien après celui qui supprime les multiples de 2, termine le premier et risque de rester inactif.

Le temps de travail des processeurs est schématisé sur la figure 1.4.

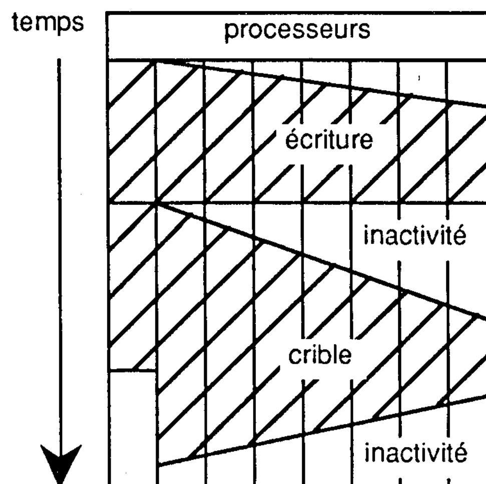


Figure 1.4. Temps d'inactivité et temps de travail des processeurs lors de l'exécution de l'algorithme de Bokhari.

Cette figure fait état d'une mauvaise répartition des tâches, lorsque le nombre de processeurs est élevé par rapport à la limite N de recherche des nombres premiers. L'aspect triangulaire de la répartition de la charge de travail n'est pas caractéristique de cette implantation. Mais l'inactivité de plusieurs processeurs et une mauvaise répartition des tâches peuvent être considérées comme typiques.

C'est surtout la phase de crible (la plus longue aussi) qui accuse un déséquilibre important. En effet la phase d'écriture est construite de manière à obtenir un équilibre optimal.

3.2. MODIFICATION DE L'ALGORITHME DE BOKHARI

Une des principales limitations de cet algorithme provient des accès simultanés à la mémoire commune. Supposons que cette restriction soit levée et que la mémoire puisse être divisée en plusieurs bancs disjoints auxquels les processeurs ont accès indépendamment [HwB 84]. Alors, chacun des P processeurs écrit ses N/P entiers dans son propre banc.

Le maître est le seul à pouvoir accéder à toute la mémoire. Il envoie le même nombre premier à tous les processeurs qui éliminent les multiples de ce nombre dans leur propre banc. Les caractéristiques sont :

- l'opération de base est toujours l'addition ;
- il n'y a pas d'accès simultanés à la mémoire pendant le processus d'élimination ;
- la synchronisation globale est toujours effectuée par le maître ;
- mais il ne reste pas, temporairement de faux nombres premiers ;
- cependant, à la fin de ce processus, la répartition des tâches est mauvaise, dans le cas où les processeurs de rang petit PE_0, PE_1, PE_2, ... sont inactifs, si les entiers à éliminer sont trop grands et ne sont présents que dans les bancs mémoires des processeurs de rang élevé PE_{P-3}, PE_{P-2}, PE_{P-1}.

3.3. RÉPARTITION DES ENTIERS PAR FAMILLES

Répartissons les entiers {2, ..., N} différemment : chaque processeur, dans sa phase d'écriture écrit, dans son propre banc de la mémoire partagée, N/P entiers non consécutifs, mais définis par une famille pour chaque processeur. Le processeur PE_i écrit les nombres i, i+P, i+2P, i+3P, ... Ainsi chaque processeur écrit et crible des entiers non consécutifs, et qui se répartissent uniformément entre 2 et N.

L'algorithme est toujours le même ; mais le maître, lors de sa recherche du nombre premier suivant, doit faire de nombreux accès à la mémoire : il doit rechercher dans toute la mémoire commune. Les entiers cherchés se trouvent dans des zones non contiguës, d'où une impossibilité d'utiliser les facilités des caches.

Mais dans ce cas, un autre problème apparaît. Il tient à la distribution des nombres premiers dans les progressions arithmétiques. Le théorème de Dirichlet [Sch 86] stipule que :

Chaque progression linéaire $a_k = mk+c$, $(c,m) = 1$, $k = 0, 1, 2, \ldots$ contient asymptotiquement le même nombre de nombres premiers.

Mais, si P est un nombre premier de processeurs et si c est l'identité du processeur PE_c, c = 0, 1, 2, ..., P-1, alors le processeur PE_0 qui traite la progression Pk+0 (m=P, c=0) a tous les multiples de P. De telle sorte que, lors du crible par P, ce processeur sera le seul à travailler alors que les autres processeurs resteront inactifs, car ils n'auront pas de multiples à éliminer.

Ce cas est inévitable, quelle que soit la valeur de P. Posons

$$P = \prod_{j=1}^{m} p_j^{\alpha_j}.$$

Le processeur PE_i traite les entiers i+kP, $k \in \mathbb{Z}$. Si i est divisible par un des facteurs premiers p_j de P, alors i+kP est divisible par ce même facteur premier p_j.

Et le processeur PE_{i+1} traite les entiers i+kP+1. Comme $i+kP \equiv 0\ [p_j]$, on a $i+kP+1 \equiv 1\ [p_j]$. Les nombres traités par ce processeur ne sont pas divisibles par p_j. Donc, lors du crible par p_j, le processeur PE_i a des multiples à éliminer, alors que le processeur PE_{i+1} n'en a aucun, et n'a donc rien à faire.

Il est donc impossible de répartir la charge de travail entre les processeurs avec cette stratégie.

3.4. RÉPARTITION DES ENTIERS PAR SOUS-SUITES

Il faut donc avoir recours à un compromis entre les deux solutions précédentes : chaque processeur écrit un sous-intervalle d'entiers consécutifs assez court, saute à un autre sous-intervalle et ainsi de suite.

Ainsi, un processeur PE_j, j = 0, ..., P-1, génère et écrit les nombres de la forme L.P.k + I + L.j + 2 avec :
j : identité de ce processeur,
I : ensemble de génération des éléments de sous-suite. I = {0, 1, ..., L-1},
k : variable de génération des sous-suites,
P : nombre de processeurs,
L : longueur d'une sous-suite.

3.4.1. Exemple pratique

Recherche des nombres premiers inférieurs à 49 sur 4 processeurs, avec des sous-suites de longueur 2.

Ecriture des entiers :

PE_0		PE_1		PE_2		PE_3	
2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41
42	43	44	45	46	47	48	49

Le maître trouve 2 chez le processeur PE_0. Suppression des multiples de 2 :

PE_0		PE_1		PE_2		PE_3	
2	3	~~4~~	5	~~6~~	7	~~8~~	9
~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17
~~18~~	19	~~20~~	21	~~22~~	23	~~24~~	25
~~26~~	27	~~28~~	29	~~30~~	31	~~32~~	33
~~34~~	35	~~36~~	37	~~38~~	39	~~40~~	41
~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49
5 éliminations		6 éliminations		6 éliminations		6 éliminations	

Le maître trouve 3 chez le processeur PE_0. Suppression des multiples de 3 :

PE_0		PE_1		PE_2		PE_3	
2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17
~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	25
~~26~~	~~27~~	~~28~~	29	~~30~~	31	~~32~~	~~33~~
~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~	41
~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49
3 éliminations		4 éliminations		4 éliminations		4 éliminations	

Le maître trouve 5 chez le processeur PE_1. Suppression des multiples de 5 :

PE_0		PE_1		PE_2		PE_3	
2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17
~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~
~~26~~	~~27~~	~~28~~	29	~~30~~	31	~~32~~	~~33~~
~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~	41
~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49
2 éliminations		2 éliminations		2 éliminations		2 éliminations	

Le maître trouve 7 chez le processeur PE_2. Suppression des multiples de 7 :

PE_0		PE_1		PE_2		PE_3	
2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17
~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~
~~26~~	~~27~~	~~28~~	29	~~30~~	31	~~32~~	~~33~~
~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~	41
~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~
2 éliminations		2 éliminations		1 élimination		1 élimination	

Il ne reste que les nombres premiers :

PE_0		PE_1	PE_2	PE_3
2	3	5	7	
	11	13		17
	19		23	
		29	31	
		37		41
	43		47	

Figure 1.5. Exemple d'exécution de l'algorithme du crible d'Eratosthène avec équirépartition des multiples entre les processeurs

L'algorithme est le suivant pour un processeur PE_j, parmi P processeurs pour cribler l'ensemble {2, ..., N} avec des sous-suites de longueur L. Soit S_j l'ensemble des entiers du processeur PE_j.

```
Algorithme
|       /* écrire les entiers */
|       k = 0
|       tant que (L.P.k + L.j + 2 ≤ N) faire
|       |       i = 0
|       |       répéter
|       |       |       écrire_élément_de_Sj (L.P.k + L.j + 2 + i)
|       |       |       i = i + 1
|       |       jusqu'à ((i > L) ou (L.P.k + L.j + 2 + i > N))
|       |       k = k + 1
|       fin tant que
|       /* cribler l'ensemble des entiers écrits */
|       recevoir_du_maître_nombre_premier (p)
|       répéter
|       |       /* cribler avec ce nombre p */
|       |       multiple = premier_multiple (p)          /* parcours des multiples de p */
|       |       élément = premier_élément_de_Sj          /* parcours des entiers de Sj */.
|       |       tant que (élément ≤ plus_grand_élément_de_Sj) faire
|       |       |       tant que (élément < multiple) faire
|       |       |       |       élément = élément_suivant_de_Sj
|       |       |       fin tant que
|       |       |       Si (élément == multiple) alors
|       |       |       |       éliminer_élément_de_Sj (élément)
|       |       |       fin si
|       |       |       multiple = multiple_suivant (multiple, p)
|       |       fin tant que
|       |       recevoir_du_maître_nombre_premier (p)
|       jusqu'à (p = marque_de_fin)
fin algorithme
```

Figure 1.6. Algorithme du crible d'Eratosthène avec équirépartition des multiples

Le travail est bien équilibré. On peut toujours réaliser le crible avec des additions ; chaque processeur parcourt deux ensembles :
- l'ensemble des entiers qu'il possède,
- l'ensemble des multiples du nombre premier avec lequel il crible.

Il parcourt les deux ensembles en parallèle de manière croissante. Il génère les multiples du nombre premier avec lequel il crible et regarde s'il l'a chez lui. Si oui, il l'élimine, sinon il passe au multiple suivant.

Prenons un exemple : la suppression des multiples de 5 sur le processeur PE_2.

Le maître envoie 5 à PE_2. PE_2 ajoute ce nombre au nombre reçu et cherche s'il l'a en mémoire. Dans sa recherche, il s'arrête soit sur ce nombre s'il le trouve, soit sur le premier entier plus grand. Ici PE_2 ne trouve pas 10 et s'arrête sur 14. Puis il ajoute 5 au dernier multiple calculé, soit 15. Il cherche ce nombre, le trouve et l'élimine. Il ajoute encore 5, soit 20 et cherche 20. Il ne le trouve pas...

Il y a alors un meilleur équilibrage des tâches que dans les stratégies précédentes.

3.4.2. Analyse mathématique

Nous allons montrer que, pour N fixé et P puissance de 2 donnée (nombre de processeurs), il existe des valeurs de L telles que la charge de travail soit asymptotiquement équirépartie entre les processeurs pour l'exécution du crible d'Eratosthène.

Lemme :

Soient $P = 2^a$, $a > 0$, le nombre de processeurs et $L = 2^b$, $b > 0$, la longueur des sous-suites.
Pour tout nombre premier $p_i \geq 2$, et pour tout j, $0 \leq j < P$, PE_j contient L représentants de chaque classe de $\mathbb{Z}/p_i\mathbb{Z}$, dans p_i sous-suites consécutives de longueur L.

Démonstration :

Une sous-suite générée par PE_j au rang k est définie par

$PLk + Lj + 2 + I$

avec : P : nombre de processeurs,
L : longueur de la sous-suite,
k : rang (indice) de la sous-suite,
j : identité du processeur,
$I = \{0, 1, \ldots, L-1\}$.

Ecrivons les p_i sous-suites consécutives, de rang y à $y+p_i-1$, sous la forme d'un tableau de p_i lignes et L colonnes :

y :	x	x+1	x+2	...	x+L-2	x+L-1
y+1 :	x+L.P	x+L.P+1	x+L.P+2	...	x+L.P+L-2	x+L.P+L-1
y+2 :	x+2L.P	x+2L.P+1	x+2L.P+2	...	x+2L.P+L-2	x+2L.P+L-1
:	:	:	:	:	:	:
$y+p_i-2$:	$x+(p_i-2)L.P$	$x+(p_i-2)L.P+1$	$x+(p_i-2)L.P+2$	...	$x+(p_i-2)L.P+L-2$	$x+(p_i-2)L.P+L-1$
$y+p_i-1$:	$x+(p_i-1)L.P$	$x+(p_i-1)L.P+1$	$x+(p_i-1)L.P+2$	...	$x+(p_i-1)L.P+L-2$	$x+(p_i-1)L.P+L-1$

Nous allons montrer que chaque colonne contient exactement un représentant de chacune des p_i classes de $\mathbb{Z}/p_i\mathbb{Z}$. Autrement dit, chaque colonne plongée dans $\mathbb{Z}/p_i\mathbb{Z}$ est une permutation de $\mathbb{Z}/p_i\mathbb{Z}$.

Prenons une colonne de ce tableau. Elle est de la forme :

x+r		s
x+L.P+r		s+L.P
x+2L.P+r	soit	s+2L.P
:		:
x+(p_i-2)L.P+r		s+(p_i-2)L.P
x+(p_i-1)L.P+r		s+(p_i-1)L.P

En plongeant cette colonne dans $\mathbb{Z}/p_i\mathbb{Z}$, l'écart entre deux lignes consécutives devient $d = (L.P \bmod p_i)$.

Et la colonne s'écrit :

s
s+d
s+2d
:
s+(p_i-2)d
s+(p_i-1)d

Montrons que la suite ($kd \bmod p_i$), pour $k \in \mathbb{Z}/p_i\mathbb{Z}$ et $d \in (\mathbb{Z}/p_i\mathbb{Z})^*$, est en fait une permutation de $\mathbb{Z}/p_i\mathbb{Z}$.
Alors, $\forall\ s \in \mathbb{Z}/p_i\mathbb{Z}$, $s + kd \bmod p_i$ est encore $\mathbb{Z}/p_i\mathbb{Z}$ (par translation).

Montrons $\{kd \bmod p_i / k \in \mathbb{Z}/p_i\mathbb{Z}, d \in (\mathbb{Z}/p_i\mathbb{Z})^*\} = \mathbb{Z}/p_i\mathbb{Z}$.
Ceci revient à montrer que $d \neq 0$ est un générateur multiplicatif de $\mathbb{Z}/p_i\mathbb{Z}$.
Comme $\mathbb{Z}/p_i\mathbb{Z}$ est un corps, l'équation $kd = y$ se résout en $k = d^{-1}y$.
Donc chaque colonne est une permutation de $\mathbb{Z}/p_i\mathbb{Z}$.

Donc, dans chaque ensemble de p_i sous-suites de longueur L consécutives, il y a L représentants de chaque classe d'équivalence de $\mathbb{Z}/p_i\mathbb{Z}$.

CQFD.

Théorème :

Soient $P = 2^a$ et $L = 2^b$, $a \geq 1$, $b \geq 1$, donnés, indépendants de N.
Si les entiers sont répartis sur les P processeurs par sous-suites de longueur L, alors, quel que soit le nombre premier $p_i \leq \sqrt{N}$, les multiples de p_i sont asymptotiquement équidistribués entre les processeurs.

Démonstration :

Le nombre de lignes (sous-suites) par processeur est

$$\left\lfloor \frac{N-1}{L.P} \right\rfloor \text{ ou } \left\lceil \frac{N-1}{L.P} \right\rceil$$

suivant l'identité du processeur. Le nombre de lignes par processeur est O(N).

Or on sait que $p_i \leq \sqrt{N}$. Donc le nombre de paquets consécutifs de p_i lignes consécutives est de l'ordre de $O(\sqrt{N})$ sur chaque processeur.

Par le lemme, on sait que dans chaque paquet de p_i lignes consécutives sur un processeur, il y a L multiples de p_i. Comme il y a plus de $\sqrt{N}$ paquets de p_i lignes sur chaque processeur, il y a $L.O(\sqrt{N})$ multiples de p_i sur chaque processeur.
Cependant, après ces paquets de p_i lignes, il peut rester des lignes qui ne forment pas un paquet complet de p_i lignes. Dans ces lignes restantes, il y a moins de L-1 multiples de p_i, donc O(1) multiples.

Ainsi le nombre de multiples diffère, entre deux processeurs, de L-1 au maximum, soit O(1).

Conclusion : les multiples de tous les p_i sont asymptotiquement équidistribués entre les processeurs.

CQFD.

Nous en déduisons le résultat suivant :

> Supposons qu'on souhaite réaliser le crible d'Eratosthène sur un ordinateur multiprocesseur à mémoire partagée. En utilisant $P = 2^a$ processeurs, et en écrivant les entiers sous forme de sous-suites de longueur $L = 2^b$ sur les processeurs, une fois que le maître a trouvé un nombre premier et l'a mis à disposition des esclaves, ceux-ci mettent asymptotiquement le même temps pour cribler avec chaque nombre premier, puisque les multiples sont équirépartis. Ils mettent au total asymptotiquement le même temps pour exécuter le crible d'Eratosthène.

De manière exacte, la différence entre le plus rapide et le plus lent des processeurs, pour le crible avec chaque p_i, est au plus L-1 opérations de crible.

3.4.3. Implantation algorithmique - Evaluation

Cette version du crible d'Eratosthène consiste en la recherche des éléments communs à deux listes ordonnées, les multiples d'un entier premier, et les éléments de l'ensemble des entiers.

L'algorithme est le suivant :

```
Algorithme
|       Ecrire les entiers de 2 à N.
|       Prendre p=2 comme nombre premier.
|       Répéter
|       |       /* Supprimer les multiples de p */
|       |       multiple = premier_multiple (p)
|       |       élément = premier_entier
|       |       tant que (multiple ≤ N) faire
|       |       |       tant que (élément < multiple) faire
|       |       |       |       élément = entier_suivant
|       |       |       fin tant que
|       |       |       Si (élément == multiple) alors
|       |       |       |       éliminer_entier (élément)
|       |       |       fin si
|       |       |       multiple = multiple_suivant
|       |       fin tant que
|       |       /* Chercher le nombre premier p suivant */
|       |       p = nombre_premier_suivant (p)
|       jusqu'à p > √N.
Fin algorithme
```

Figure 1.7. Algorithme du crible d'Eratosthène avec la répartition par sous-suites

Dans cet algorithme, on parcourt la liste des multiples en priorité, car elle contient moins d'éléments que la liste des entiers.

Caractérisons les entiers d'un processeur donné PE_j. Un élément PLk + Lj + 2 + i est défini par son rang k de sous-suite, et sa position i dans cette sous-suite, sur le processeur PE_j. Pour un processeur donné, un élément est donc déterminé par un couple (k, i). On peut donc expliciter les fonctions de parcours de la liste des entiers du processeur PE_j :

```
Fonction entier_suivant
|   Si (i == L-1) alors
|   |   k = k+1
|   |   i = 0
|   sinon
|   |   i = i+1
|   fin si
|   retourner (PLk + Lj + 2 + i)
fin fonction

Fonction premier_entier
|   k = 0
|   i = 0
|   retourner (PLk + Lj + 2 + i)
fin fonction
```

Figure 1.8. Les fonctions de parcours de la liste des entiers sur le processeur PE_j

De la même manière, on veut parcourir, sur le processeur PE_j, la liste des multiples d'un nombre premier donné p. Mais la caractérisation d'un multiple sur un processeur est coûteuse : elle dépend du multiple précédent et de k, et requiert un parcours de la liste des entiers de ce processeur, afin de vérifier que le multiple trouvé appartient à ce processeur. Or ce parcours serait redondant avec le parcours de cette liste dans la boucle secondaire de l'algorithme. Donc, pour minimiser le contrôle, le parcours des multiples est séquentiel, indépendant des multiples que traite un processeur.

Les fonctions sont donc simples :

```
Fonction premier_multiple
|   retourner (p + p)
fin fonction

Fonction multiple_suivant
|   retourner (multiple + p)
fin fonction
```

Figure 1.9. Les fonctions de parcours de la liste des multiples de p sur le processeur PE_j

Le crible d'Eratosthène est puissant et rapide car les fonctions qu'il met en oeuvre sont rapides :

- addition pour accéder au multiple suivant,
- accès à un tableau (addition et multiplication) pour localiser un entier,
- modification d'un bit en mémoire (et, ou, non) pour supprimer cet entier,
- test de boucle (addition).

Dans notre implantation actuelle, les opérations sont moins simples et plus nombreuses (plusieurs tests, additions et multiplications). De plus, chaque processeur criblant avec un nombre premier p, doit parcourir tous les multiples de p, alors qu'il n'en a que p/P.

Le nombre plus grand d'opérations et le parcours de tous les multiples coûtent cher. Donc l'opération de suppression d'un multiple nécessite un nombre plus important d'instructions que celle de suppression dans le crible de base. C'est pourquoi nous pensons que cette stratégie de répartition implique un temps de calcul plus grand que la stratégie, certes théoriquement moins bonne, de Bokhari.

Nous avons atteint le but : équirépartir les tâches sur les processeurs dans un environnement à mémoire partagée, mais au prix d'une gestion plus importante. Nous comparerons les temps d'exécution de ces stratégies sur le FPS T20 à mémoire distribuée.

3.5. CONCLUSION

L'algorithme de base de Bokhari, en mémoire partagée, souffre d'un sérieux handicap : la charge de travail n'est pas répartie entre les P processeurs. La modification qu'il propose réduit ce déséquilibre, mais sans le supprimer. Nous pensons que sa répartition des données est mauvaise : l'intervalle à cribler est découpé en P sous-intervalles. Le $i^{ème}$ processeur traite le $i^{ème}$ sous-intervalle. Ainsi, un processeur de rang petit traite des petits entiers, et un processeur de rang élevé traite des grands entiers. Lorsque le nombre premier avec lequel il faut cribler devient grand, un petit processeur n'a plus de travail.

Nous modifions donc la répartition des entiers sur les processeurs de manière que chaque processeur ait des petits et des grands entiers (stratégie de répartition par familles), et que les multiples de tous les nombres premiers p_i avec lesquels on va cribler soient équidistribués entre les processeurs. Ainsi, lorsque la valeur du nombre premier p (crible) croît, le processeur qui a les entiers 2,3,... a aussi des entiers plus grands que p. Donc il n'est pas inactif. Et les processeurs peuvent cribler en même temps avec le même nombre premier.

Cependant, l'équirépartition des tâches n'est pas effective. C'est pourquoi nous proposons une autre répartition des données (stratégie de répartition par sous-suites). Si le nombre P de processeurs vaut 2^a, $a > 0$, et si la longueur L des sous-suites vaut 2^b, $b > 0$, alors l'équirépartition des tâches est atteinte, car les multiples de chaque nombre premier sont équidistribués entre les processeurs. Cependant, l'implantation réclame une gestion plus importante. Et le coût d'une opération de base (accès à un entier et un multiple, suppression de ce multiple, calcul de l'entier suivant et du multiple suivant) est augmenté. L'implantation sur le FPS T20 permettra d'évaluer cette augmentation du coût des opérations.

4. LE CRIBLE D'ÉRATOSTHÈNE SUR UN MULTIPROCESSEUR À MÉMOIRE DISTRIBUÉE

Bokhari donne des temps d'exécution de ses deux algorithmes sur le Flex/32. Nous voulons comparer les temps d'exécution du crible d'Eratosthène sur l'hypercube de FPS, avec les temps sur le Flex/32. Pour ce faire, nous prenons les mêmes restrictions que Bokhari : le crible avec le nombre premier p commence à 2p, le crible total de $\{1, 2, ..., N\}$ ne se fait qu'avec les nombres premiers inférieurs à $\sqrt{N}$, on ne supprime a priori aucun entier (on écrit tous les entiers, même les pairs).

Nous présentons dans ce paragraphe les critères qui nous ont conduits à implanter l'algorithme sur un anneau. Puis nous essayons d'adapter la stratégie asymptotiquement optimale développée pour la répartition des données dans un environnement à mémoire partagée, au FPS T20 à mémoire distribuée.

Notre but est d'évaluer pour ces algorithmes les coûts de communication lors de l'envoi des nombres premiers, et d'analyser les performances de cet algorithme Maître / Esclaves sur un multiprocesseur à mémoire distribuée.

4.1. LE CRIBLE CLASSIQUE SUR LE FPS T20

Notre implantation dérive de celle de Bokhari sur le Flex/32 [Bok 87]. En fait, c'est une adaptation de la première version modifiant l'algorithme de Bokhari, où chaque processeur gère une suite d'entiers consécutifs dans une partie réservée de la mémoire commune.

Ici, bien sûr, pas de mémoire commune. Donc, chaque processeur peut écrire indépendamment dans sa mémoire. Les bornes minimum et maximum de l'intervalle dépendent, comme dans l'algorithme de Bokhari, de l'identité du processeur. Les P processeurs du FPS sont numérotés de 0 à P-1. Soit N la borne supérieure de l'intervalle de crible $\{1, ..., N\}$ duquel on veut

extraire tous les nombres premiers inférieurs à N. Chaque processeur PE_i gère les entiers de 1+iN/P à (1+i)N/P.

La différence avec l'algorithme de Bokhari réside dans le fait qu'il n'y a pas de maître qui accède à une mémoire globale. Cependant, un processeur particulier doit chercher le nombre premier suivant et l'envoyer à tous les processeurs. Alors chaque processeur supprime les multiples de ce nombre. C'est un algorithme de type Maître / Esclaves.

4.1.1. Implantation

Les entiers {1, ..., N} sont répartis entre les P processeurs. Un processeur PE_i traite les entiers {i.N/P + 1, ..., (i+1).N/P} consécutifs. A un instant donné, un processeur, le Maître, cherche le nombre premier p suivant. Tous les autres attendent de connaître p. Le Maître envoie ce nombre à tous les processeurs. Et tous suppriment les multiples de p dans leur ensemble d'entiers.

Les processeurs sont numérotés de 0 à P-1. Le processeur PE_0 a les entiers les plus petits. Le processeur PE_{P-1} a les entiers les plus grands. Initialement, c'est donc le processeur PE_0 qui cherche le nombre premier suivant. Mais ensuite, qui trouve les autres nombres premiers nécessaires au crible ? C'est le processeur PE_0, et lui seul, dans le cas où $N/P \geq \sqrt{N}$, c'est-à-dire $N \geq P^2$, soit, dans notre cas $N \geq 256$.

Avec 1 Mo par processeur, nous pouvons utiliser 800 Ko pour l'intervalle de crible. En prenant un tableau d'entiers (4 octets/entier), on peut coder 200 000 entiers par processeur. Or un entier est soit premier, soit non premier ; il suffit donc d'un bit par entier. 800 Ko permettent donc de coder 6 400 000 entiers. Et avec les 16 processeurs du FPS T20, $16*6\,400\,000 = 102\,400\,000 \approx 10^8$. C'est la valeur maximale de N pour le FPS T20. Donc, dans les exemples que nous traiterons, $N \geq 256$, ce qui implique que seul le processeur PE_0 cherchera les nombres premiers. Et tous les processeurs cribleront avec ces nombres.

Comment le processeur PE_0 envoie-t-il chaque nombre premier à ses esclaves ? Autrement dit, sur quelle topologie ? Le FPS est un hypercube. La technique de diffusion la plus rapide théoriquement consiste à utiliser un arbre de recouvrement minimal de l'hypercube. Cet arbre a une profondeur et un degré maximal égaux à la dimension de l'hypercube.

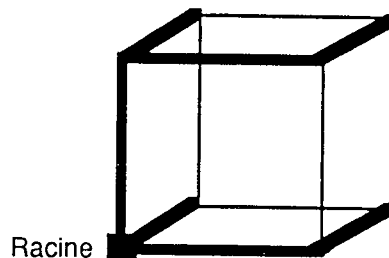


Figure 1.10. Hypercube de dimension 3 et un arbre de recouvrement

Comme une communication de n octets coûte $\beta + \mu.n$ [Saa 85], la diffusion d'un entier de la racine vers tous les processeurs coûte $d*(\beta + 4\mu)$ sur un hypercube de dimension d. Sur le FPS T20, cette diffusion coûte plus de 3 ms. Or la procédure de diffusion d'un processeur vers tous les autres (one_to_all) de FPS est plus rapide (moins de 2 ms) sur un anneau de 16 processeurs. Mais elle ne peut être utilisée que dans une seule dimension à la fois. C'est pourquoi nous configurons les 16 processeurs en anneau.

Nous donnons les algorithmes du Maître et des Esclaves. Ils sont différents, puisque le Maître est toujours le processeur PE_0. Et il est le seul à chercher les nombres premiers. C'est pourquoi nous séparons ces deux algorithmes.

Les algorithmes sont les suivants.

Pour le Maître :

Algorithme
    Ecrire les entiers de 2 à N/16.
    nombre_premier = 2
    Répéter
        envoyer (nombre_premier) à tous les processeurs
        supprimer les multiples de nombre_premier inférieurs à N/16
        chercher le nombre premier suivant (&nombre_premier)
    jusqu'à nombre_premier > $\sqrt{N}$
    envoyer (nombre_premier = marque_de_fin) à tous les processeurs
fin algorithme

Pour les Esclaves :

Algorithme
    Ecrire les entiers de i.N/16 + 1 à (i+1).N/16.
    recevoir (nombre_premier)
    Répéter
        supprimer les multiples de nombre_premier entre i.N/16 + 1 et (i+1).N/16
        recevoir (nombre_premier)
    jusqu'à (nombre_premier == marque_de_fin)
fin algorithme

Figure 1.11. Algorithme du Maître et des Esclaves en mémoire distribuée avec répartition des entiers selon Bokhari

4.1.2. Evaluation - Résultats

Comparons les temps avec ceux qu'obtient Bokhari [Bok 87] pour des valeurs de $N = 10^6$ et $N = 2.10^6$.

N	Temps Bokhari en s sur 16 processeurs	Temps FPS T20 en s sur 16 processeurs
10^6	5,8	3,26
2.10^6	11,4	6,54

Figure 1.12. Temps comparés de l'exécution sur Flex/32 et FPS T20

Le Flex/32 est construit à base de processeurs National 32032 [Mat 85]. Le FPS T20 est à base de Transputers T414. Le FPS semble légèrement plus rapide, malgré les communications.

Quelle proportion représentent les communications dans le temps total ?
Il y a $\sqrt{N}$ / $\log(\sqrt{N})$ communications environ. Le tableau suivant donne la part du temps de communication dans l'exécution de l'algorithme.

N	Temps total en s	Temps de communication en s	Part des communications
10^4	0,0548	0,042	77 %
10^6	3,26	0,288	9 %
10^8	350	2,172	1 %

Figure 1.13. Temps et proportion des communications dans le temps d'exécution

Lorsque les sous-intervalles sont très petits sur les processeurs, la suppression des multiples prend un temps négligeable. Ce sont les communications qui nécessitent le plus de temps. Par la suite, la suppression des multiples devient dominante, car les sous-intervalles sont plus grands, et ils contiennent donc plus de multiples d'un même nombre premier. Les communications deviennent négligeables.

Les temps sont-ils cohérents avec la théorie ?

En séquentiel, le temps d'exécution du crible d'Eratosthène est O(N log log N). En parallèle, on a la même complexité. Vérifions que la courbe des temps mesurés suit la courbe des temps théoriques.

N	Temps mesuré en s	Temps théorique en nombre d'opérations	Temps de communication en s
10^3	0,0148	1 930	0,013
10^4	0,0548	22 200	0,042
10^5	0,355	244 000	0,110
10^6	3,26	2 630 000	0,288
10^7	33,5	27 800 000	0,784
10^8	350	291 000 000	2,172

Figure 1.14. Temps mesuré et temps théorique d'exécution du crible d'Eratosthène

Le temps de communication est négligeable surtout pour les grandes valeurs de N. Mais pour les petites valeurs, les communications coûtent cher. La figure suivante donne une visualisation de ces valeurs.

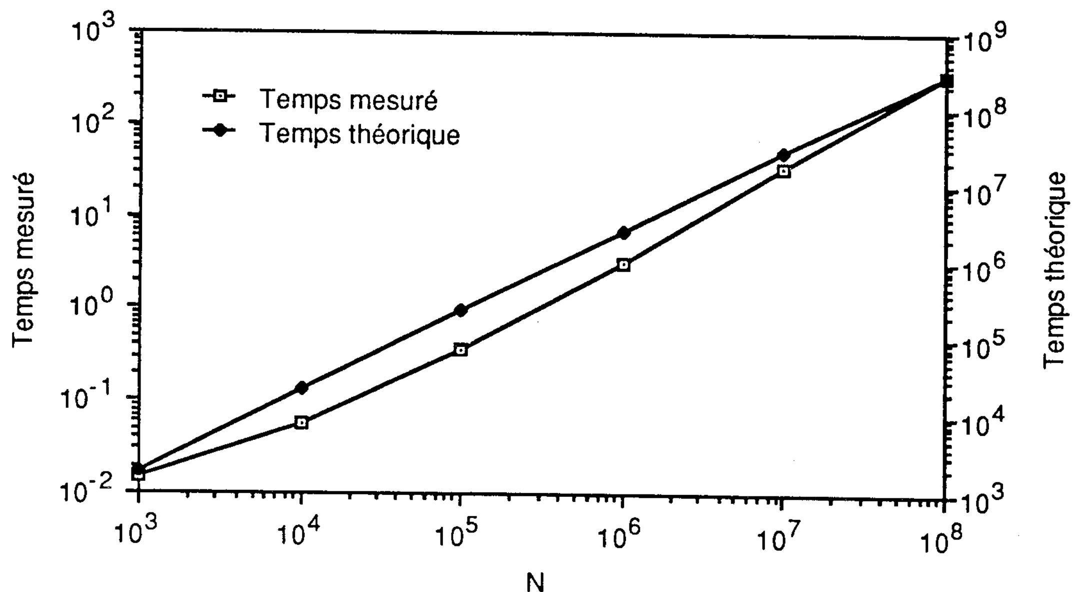


Figure 1.15. Courbes des temps mesurés et théoriques de l'exécution du crible d'Eratosthène

Mais les temps théoriques ne tiennent pas compte des temps de communications. Etudions les mêmes données, en faisant abstraction des temps de communication.

N	Temps mesuré en s	Temps de communication en s	Temps des calculs en s	Temps théorique en nombre d'opérations
10^3	0,0148	0,013	0,0018	1 930
10^4	0,0548	0,042	0,0128	22 200
10^5	0,355	0,110	0,245	244 000
10^6	3,26	0,288	2,972	2 630 000
10^7	33,5	0,784	32,7	27 800 000
10^8	350	2,172	348	291 000 000

Figure 1.16. Temps de calcul et de communication du crible d'Eratosthène

La figure suivante donne une visualisation de ces résultats.

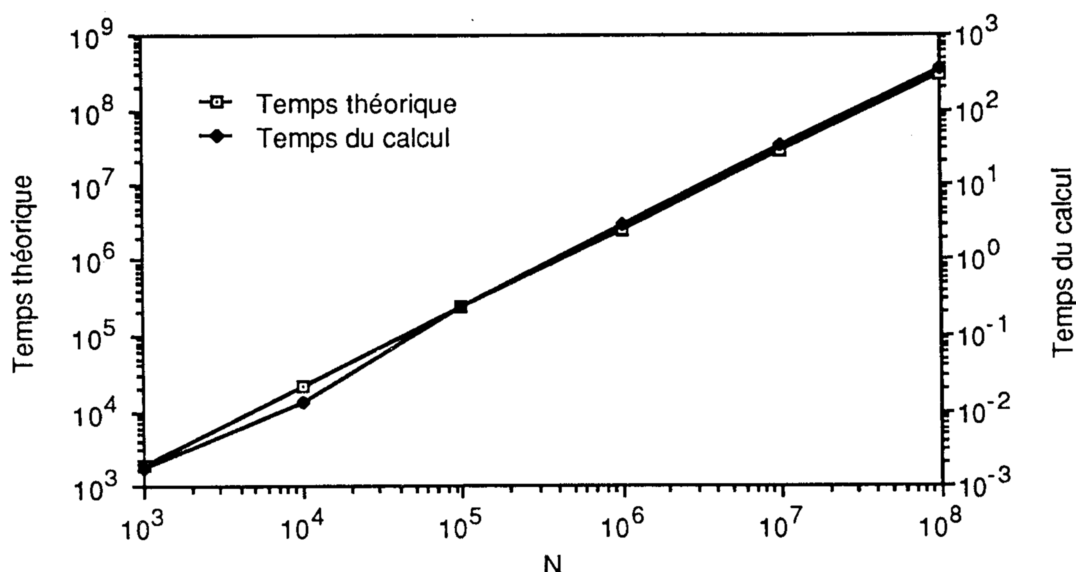


Figure 1.17. Courbes des temps de calcul mesurés et théoriques de l'exécution du crible d'Eratosthène

La courbe des temps de calcul (sans les communications) suit la courbe du temps théorique de calcul. La complexité n'est pas modifiée par l'introduction du parallélisme.

4.2. RÉPARTITION DES ENTIERS PAR SOUS-SUITES

Avec notre répartition des entiers sur les processeurs, lorsqu'on crible avec $p_i \approx \sqrt{N}$, le processeur PE_0 a moins de travail que les autres processeurs. Donc la répartition des tâches n'est pas équilibrée.

En adoptant la répartition des entiers par sous-suites, il est clair que l'équilibrage serait meilleur. Cependant, le nombre d'opérations est plus important par processeur, parce que ni les entiers, ni les multiples ne sont complètement consécutifs sur un processeur.

Dans la version du paragraphe 4.1, bien que les calculs ne soient pas complètement équirépartis entre tous les processeurs, ils le sont entre les processeurs autres que le processeur PE_0 (qui a moins de multiples à éliminer). Cependant, ce processeur a un travail supplémentaire que les autres processeurs n'ont pas : il doit chercher le nombre premier suivant, avec lequel tous les processeurs cribleront. Ceci permet donc de diminuer l'écart de charge de travail entre le processeur PE_0 et les autres processeurs.

Pour implanter l'algorithme par sous-suites sur le FPS, nous utilisons la même topologie d'anneau, afin de minimiser les temps de communication. Nous adaptons l'algorithme du § 3.4.3 à l'hypercube. Mais dans ce cas, le Maître change très souvent de localisation. Donc tous les processeurs peuvent devenir Maître. Il faut donc introduire un mécanisme de passage à l'état de Maître. Et un processeur qui perd son état de Maître redevient un Esclave. Le Maître suivant est toujours le successeur sur l'anneau, car les entiers suivent cet ordre sur l'anneau.

L'algorithme est le suivant. C'est le même pour tous les processeurs.

```
Algorithme
|       Ecrire les entiers en mémoire
|       Maître = Processeur PE0
|       Répéter
|       |       Si (Identité du Processeur == Maître) alors
|       |       |       chercher le nombre premier suivant (&trouvé, &p)
|       |       |       Si (trouvé) alors
|       |       |       |       envoi (p) à tous les processeurs
|       |       |       sinon
|       |       |       |       envoi (-1) à tous les processeurs
|       |       |       fin si
|       |       sinon
|       |       |       réception (p)
|       |       fin si
|       |       Si (p == -1) alors
|       |       |       Maitre = Maitre + 1
|       |       sinon
|       |       |       supprimer les multiples de p
|       |       fin si
|       jusqu'à p > √N
fin algorithme
```

Figure 1.18. Algorithme du crible d'Eratosthène avec répartition des entiers par sous-suites

Les résultats sont donnés dans le tableau suivant. Nous avons fait varier la longueur L des sous-suites, de manière à étudier l'impact de ce paramètre sur les temps d'exécution.

Temps en s	pour N	L = 2	L = 4	L = 8	L = 16	L = 32
total calcul communication	1 024	0,0655 0,0115 0,0540	0,0565 0,0185 0,0380	0,0510 0,0210 0,0300	0,0490 0,0230 0,0260	0,0480 0,0240 0,0240
total calcul communication	10 240	0,600 0,446 0,154	0,555 0,451 0,104	0,541 0,463 0,078	0,535 0,469 0,066	0,532 0,472 0,060
total calcul communication	102 400	9,33 8,88 0,45	9,07 8,77 0,30	9,01 8,80 0,21	9,01 8,82 0,19	9,00 8,85 0,15
total calcul communication	1 024 000	188,3 186,9 1,35	185,5 184,7 0,84	185,2 184,6 0,60	185,5 185,1 0,47	186,0 185,6 0,40

Figure 1.19. Temps de calcul et de communication pour le crible d'Eratosthène avec répartition des entiers par sous-suites sur le FPS T20

Le temps de communication diminue en fonction de L. En effet, le nombre de nombres premiers trouvés est le même quelle que soit la valeur de L. Mais le nombre de messages pour changement de Maître est plus important lorsque L est petit. Il diminue donc lorsque L croît. Pour les petites valeurs de N, ce sont les communications qui prennent le plus de temps. Pour

les valeurs plus grandes, les communications ne représentent qu'une toute petite partie du temps total. Et on voit que le temps d'exécution est presque constant en fonction des valeurs de L.

Cependant les temps d'exécution sont beaucoup plus importants qu'avec le crible de base. Par exemple, il faut 185 secondes pour trouver tous les nombres premiers inférieurs à 1 024 000. Or il ne faut que 3,26 secondes avec le crible de base (§ 4.1.2). Cette version n'est donc pas très puissante, car les opérations sont nombreuses.

4.3. CONCLUSION

L'algorithme de base du crible d'Eratosthène s'implante de manière efficace sur l'hypercube de FPS, bien que la répartition théorique des calculs ne soit pas la meilleure.

Lorsqu'on étudie une équirépartition théorique, celle préconisée pour des architectures à mémoire partagée, on se rend compte qu'il existe toujours un processeur particulier, celui qui cherche le nombre premier suivant, qui a une charge de travail supplémentaire que les autres processeurs n'ont pas. Dans notre première version sur le FPS, ce travail remplace les calculs moins nombreux sur le processeur PE_0. En effet, le processeur PE_0 a moins de travail pour cribler, car il a les petits entiers. Mais il doit chercher le nombre premier suivant, ce qui lui permet de combler la différence de charge avec les autres processeurs. Par conséquent, un équilibrage relatif est possible.

Dans l'algorithme qui équirépartit les entiers par sous-suites, les calculs sont eux aussi équirépartis entre les processeurs, et le surplus dû à la recherche du nombre premier suivant ne vient pas combler un manque de travail lors de l'élimination des multiples. L'équilibrage des tâches n'est, par suite, plus respecté, puisque le maître, qui cherche le nombre premier suivant, change lors du passage d'une sous-suite sur un processeur à la sous-suite sur le processeur voisin. Ce passage d'un processeur à un autre implique une synchronisation et une communication entre ces processeurs.

On le voit : le coût supplémentaire dû à cet algorithme est important pour une architecture à mémoire distribuée, alors qu'il l'est moins pour une architecture à mémoire partagée, où les communications se font par la mémoire globale et où un maître peut être complètement dévolu à la recherche du nombre premier suivant.

Dans l'implantation de l'algorithme de répartition des entiers par sous-suites (mémoire distribuée), il y a une communication de changement de Maître à la fin de chaque sous-suite de L entiers. Comme on ne cherche les nombres premiers que jusqu'à $\sqrt{N}$, il y a $\sqrt{N}/L$ communications de changement de Maître. Ceci incite à augmenter la valeur de L. Mais d'un autre côté, rappelons que la différence entre le nombre de multiples d'un nombre premier du processeur qui en a le plus et celui du processeur qui en a le moins, est au plus L-1. Donc, pour que les processeurs travaillent le même temps, il faut que cette différence reste faible, donc que L reste petit. Ainsi, il faut trouver un compromis pour L, en fonction de P et de N.

Mais le coût d'une communication est aussi un paramètre très important. Sur le FPS, les communications sont très chères par rapport aux calculs, et il faut une grande valeur de L pour diminuer le nombre de communications de changement de Maître, car la différence de L-1 opérations est négligeable entre les processeurs.

Les performances de cet algorithme sont très limitées si on s'attache aux temps d'exécution, car sa gestion est très lourde. Mais les processeurs se répartissent bien la charge de travail.

Nous n'avons atteint que la moitié de nos objectifs : la répartition des entiers par sous-suites est théoriquement intéressante. Elle permet de faire travailler tous les processeurs sans période d'inactivité. Mais son implantation nécessite une gestion coûteuse, induisant un temps d'exécution supplémentaire important, qui réduit à néant le temps gagné par l'équirépartition des tâches.

5. GÉNÉRATION DES NOMBRES PREMIERS PAR DIVISIONS SUCCESSIVES

Il ne s'agit plus ici de l'algorithme du crible d'Eratosthène dans le sens où, à partir d'un nombre premier, on élimine ses multiples. En effet, ici, on choisit un entier quelconque et on teste sa primalité en le divisant successivement par tous les nombres premiers inférieurs à sa racine carrée, s'il est premier, ou jusqu'à ce qu'une division rende un reste nul, auquel cas cet entier n'est pas premier.

Nous pensons que l'aspect Maître / Esclaves du crible d'Eratosthène est un élément important qui limite son efficacité sur un multiprocesseur à mémoire distribuée. L'utilisation d'un pipeline apportera, nous l'espérons, de meilleurs résultats en temps d'exécution, car les processeurs seront mieux utilisés. Cependant, l'opération de base devient la division, bien plus coûteuse que l'addition. Nous allons étudier si une meilleure gestion des communications et des temps de travail des processeurs peut combler le coût supplémentaire dû à l'utilisation des divisions.

Cependant, nous allons éviter les tests des entiers multiples de 2, 3 et 5. En effet, pour éviter les multiples des r premiers nombres premiers $p_1, \ldots, p_r$, ($p_1=2$), les entiers à générer sont de la forme :

$$\left(\prod_{i=1}^{r} p_i\right) x + y \text{ , avec } 0 < y < \prod_{i=1}^{r} p_i \text{ et } \left(y, \prod_{i=1}^{r} p_i\right) = 1.$$

Donc, pour ne générer que des entiers non multiples de 2, 3 et 5, nous produisons les entiers de la forme 30x+1, 30x+7, 30x+11, 30x+13, 30x+17, 30x+19, 30x+23, 30x+29. Il y a donc 8 familles disjointes qui génèrent tous les nombres premiers.

Nous pouvons aussi ne pas générer les multiples de 7, mais la gestion est beaucoup plus lourde. En effet, dans ce cas, il faut générer les entiers de la forme 210x+y avec (210, y) = 1. Alors y peut prendre 48 valeurs différentes.

Or le T20 ne dispose que de 16 processeurs. Donc pour l'étude de cet algorithme, nous nous restreignons aux entiers non multiples de 2, 3 et 5, de la forme 30x+y. y peut prendre 8 valeurs. Ainsi, nous affectons chacune de ces 8 familles à un processeur. La réalisation serait la même avec un plus grand nombre de processeurs, pour des nombres non multiples de 2, 3, 5 et 7, et éventuellement de 11, 13 ...

5.1. GÉNÉRATION MULTIPIPELINÉE DES NOMBRES PREMIERS SUR UN ANNEAU

Chacune des 8 familles est affectée à un processeur sur un anneau de 8 processeurs. Ce processeur a trois fonctions :

- générer les nombres de la famille qui lui est propre ;
- tester la primalité des entiers qu'il génère ou qu'il reçoit, relativement à un ensemble de nombres premiers qui lui est propre ;
- stocker les nombres premiers dans sa mémoire.

Le principe est le suivant. Chaque processeur génère des entiers de sa famille, et teste leur primalité relativement aux nombres premiers qu'il possède. Si l'entier généré passe ce test de primalité relative, il est envoyé au processeur suivant sur l'anneau. Sinon, il est supprimé et un autre entier de la même famille est généré, et suit la même procédure. Chaque processeur teste la primalité des nombres qu'il reçoit, relativement aux nombres premiers qu'il possède. Si l'entier reçu passe ce test, il est envoyé au processeur suivant, sinon il est remplacé par un entier généré par le processeur qui l'a supprimé. Les nombres premiers sont ainsi générés dynamiquement et stockés dans la mémoire locale de chaque processeur : lorsqu'un entier a effectué un tour complet sur l'anneau, et qu'il revient au processeur qui l'a généré, alors il est premier. Donc ce processeur stocke ce nombre premier dans sa mémoire locale et génère un autre entier à tester. Cette stratégie garantit que le pipeline ne contient jamais de bulle.

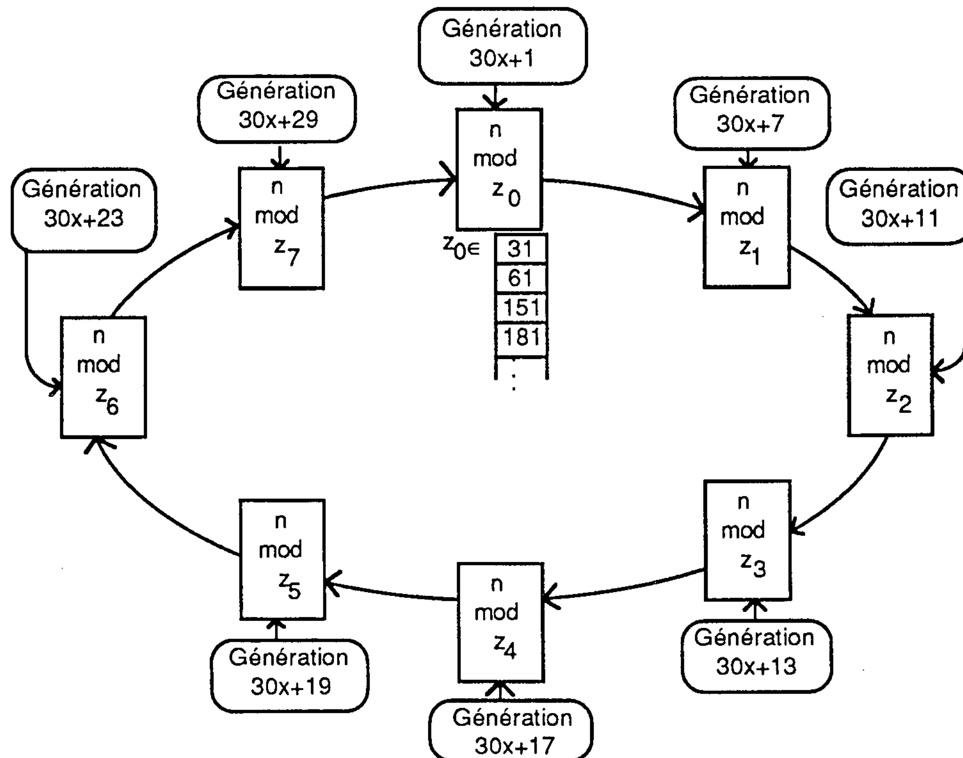


Figure 1.20. Génération multipipelinée des nombres premiers

Dans notre étude avec 8 familles, on utilise donc un processeur par famille.

5.1.1. Algorithme

Chaque processeur exécute le même programme sur des variables différentes : 30x+y. C'est y qui varie d'un processeur à l'autre.
Voici l'algorithme d'un processeur PE_i :

```
Algorithme
|       Générer m localement premier
|       Envoyer m
|       Répéter
|       |       Recevoir un entier n
|       |       Si n a été généré par moi-même alors
|       |       |       /* n est premier */
|       |       |       Stocker n dans la mémoire locale
|       |       |       Générer m localement premier
|       |       |       Envoyer m
|       |       sinon
|       |       |       Si n est localement premier alors
|       |       |       |       Envoyer n
|       |       |       sinon
|       |       |       |       /* n est composé ; il est supprimé */
|       |       |       |       Générer m localement premier
|       |       |       |       Envoyer m
|       |       |       Fin si
|       |       Fin si
|       jusqu'à fin                              /* n > valeur max, par exemple */
Fin algorithme
```

Figure 1.21. Algorithme de génération multipipelinée des nombres premiers

5.1.2. Cet algorithme est incorrect

Cet algorithme est incorrect. En effet, lors de l'initialisation de l'exécution de l'algorithme, les seuls nombres premiers que les processeurs connaissent sont :
- 31 pour le processeur 0 qui génère des entiers de la famille 30x+1,
- 7 pour le processeur 1 qui génère des entiers de la famille 30x+7,
- 11 pour le processeur 2 qui génère des entiers de la famille 30x+11,
- 13 pour le processeur 3 qui génère des entiers de la famille 30x+13,
- 17 pour le processeur 4 qui génère des entiers de la famille 30x+17,
- 19 pour le processeur 5 qui génère des entiers de la famille 30x+17,
- 23 pour le processeur 6 qui génère des entiers de la famille 30x+23,
- 29 pour le processeur 7 qui génère des entiers de la famille 30x+29.

Et la racine carrée du nombre reçu risque d'être plus grande que le plus grand nombre stocké sur ce processeur.

Pour assurer que l'algorithme se comporte de manière correcte, le nombre reçu doit être divisé par tous les nombres premiers inférieurs à sa racine carrée sur ce processeur. Mais ceci n'est pas toujours vérifié, et peut conduire à conserver des nombres composés. Comme nous le verrons plus loin, expérimentalement, ceci ne se produit jamais. Cependant, nous n'avons pas réussi à le démontrer.

5.1.3. Modification de l'algorithme

Pour faire face à ce problème, le test de primalité est modifié : il est dorénavant effectué relativement aux nombres premiers stockés dans la mémoire locale et relativement aux nombres qui ont été générés et qui n'ont pas encore été reçus après un tour complet. De cette manière, on repousse le risque précédent. Mais on ne l'élimine pas pour autant.

C'est pourquoi nous introduisons une FIFO sur chaque processeur, utilisée comme suit. Chaque fois qu'un entier est reçu, le processeur vérifie que ce nombre satisfait la règle de la racine carrée. Si oui, les tests de primalité locale sont effectués, et le nombre est soit envoyé au processeur voisin, soit éliminé. Si la règle n'est pas satisfaite, le nombre reçu est temporairement rangé dans la FIFO, alors qu'un nouvel entier est généré par ce processeur, en remplacement du nombre rangé dans la FIFO. Grâce à cette nouvelle génération, la valeur des nombres qui peuvent être testés sur ce processeur augmente.
Ceci est valable en mode continu. Mais il faut initialiser les pipelines. Nous montrons que chaque processeur peut toujours commencer à générer, grâce à la règle suivante :

Soit x le plus grand nombre généré sur ce processeur. Le plus grand entier qui puisse être testé est x^2. Le nombre suivant généré par ce processeur est x+30.
Or le blocage dans la phase de génération est possible si $x^2 < x+30$. Ceci est vrai pour tout x tel que $-5<x<6$. Ces valeurs ne sont jamais atteintes puisque $x \geq 7$. En effet, la phase de génération commence avec x=30m+y, et les entiers x stockés en mémoire sont plus grands que 7.

Par la suite, si la FIFO est vide ou si le premier nombre de la FIFO ne satisfait pas la condition sur la racine carrée, un nouvel entier est généré par la procédure de génération d'un entier dans la famille de ce processeur. Au contraire, si la FIFO n'est pas vide, le premier entier de cette file est sorti dès que possible, et ce processeur ne génère pas un nouvel entier. L'entier de la FIFO est alors testé et envoyé au processeur suivant ou éliminé.

Remarques :
- Nos résultats expérimentaux ont fait apparaître deux différences remarquables entre cet algorithme et le crible d'Eratosthène, à savoir que les nombres composés ne sont pas éliminés dans le même ordre et que chaque nombre non premier n'est éliminé qu'une seule fois, alors que dans le crible d'Eratosthène de base, un entier composé est éliminé par tous ses facteurs premiers.

- Et, de plus, quelle que soit la limite supérieure N de génération des entiers dans l'intervalle {100, ..., 20 000 000}, la FIFO n'est jamais utilisée.
- L'espace mémoire utilisé pour stocker les nombres premiers est différent de celui qui est nécessaire pour le crible d'Eratosthène puisqu'ici, les entiers premiers sont physiquement écrits en mémoire sur 32 bits, alors que dans le crible d'Eratosthène, ils ne requièrent qu'un bit.

5.1.4. Résultats

Lorsque la limite supérieure N de génération des entiers augmente, le temps de communication et le temps de calcul augmentent aussi. Mais le temps de communication croît plus lentement. Cependant, il reste toujours supérieur au temps de calcul.

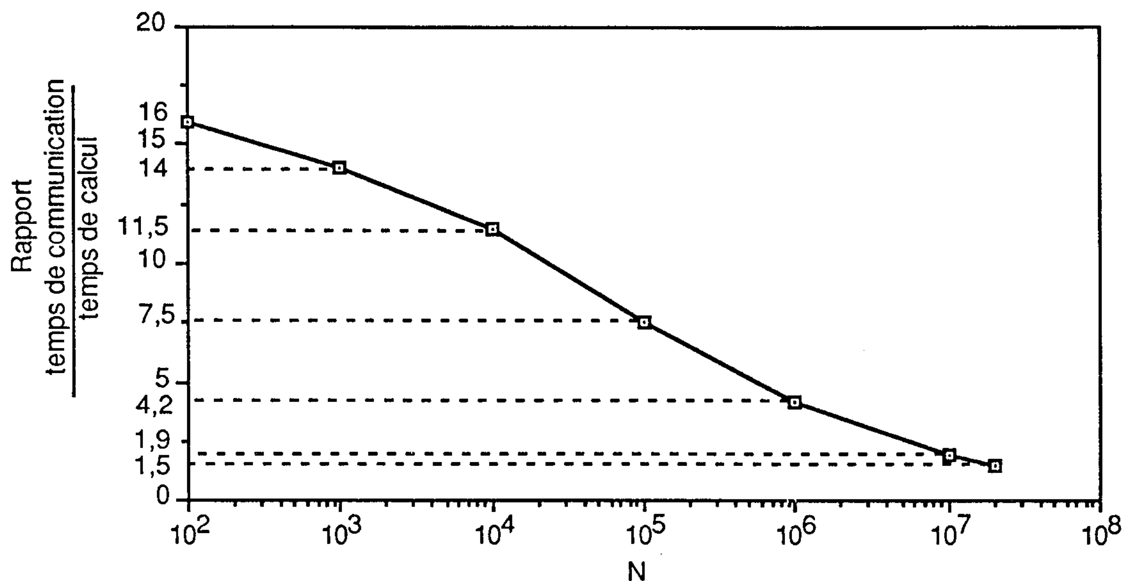


Figure 1.22. Rapport temps de communication sur temps de calcul pour la génération pipelinée des nombres premiers

Il faut diminuer ce temps de communication.

5.2. UN NOUVEL ALGORITHME : COMMUNICATIONS PAR PAQUETS

Y. Saad propose de modéliser le temps de communication par $t = \beta + \mu n$ [Saa 85] où β est le temps d'initialisation de la communication, μ le temps nécessaire à l'envoi d'un octet et n la taille du message. Cette modélisation est valide pour les communications sur l'hypercube FPS T20, pour lequel les valeurs de β et μ sont $\beta = 860$ µs et $\mu = 1{,}44$ µs/octet [CTV 87b]. L'envoi d'un entier (4 octets) coûte donc $860 + 4*1{,}44 = 865{,}76$ µs. Pour comparer, l'envoi de 10 entiers coûte 917,6 µs, l'envoi de 100 entiers coûte 1436 µs et l'envoi de 1000 entiers coûte 6620 µs.

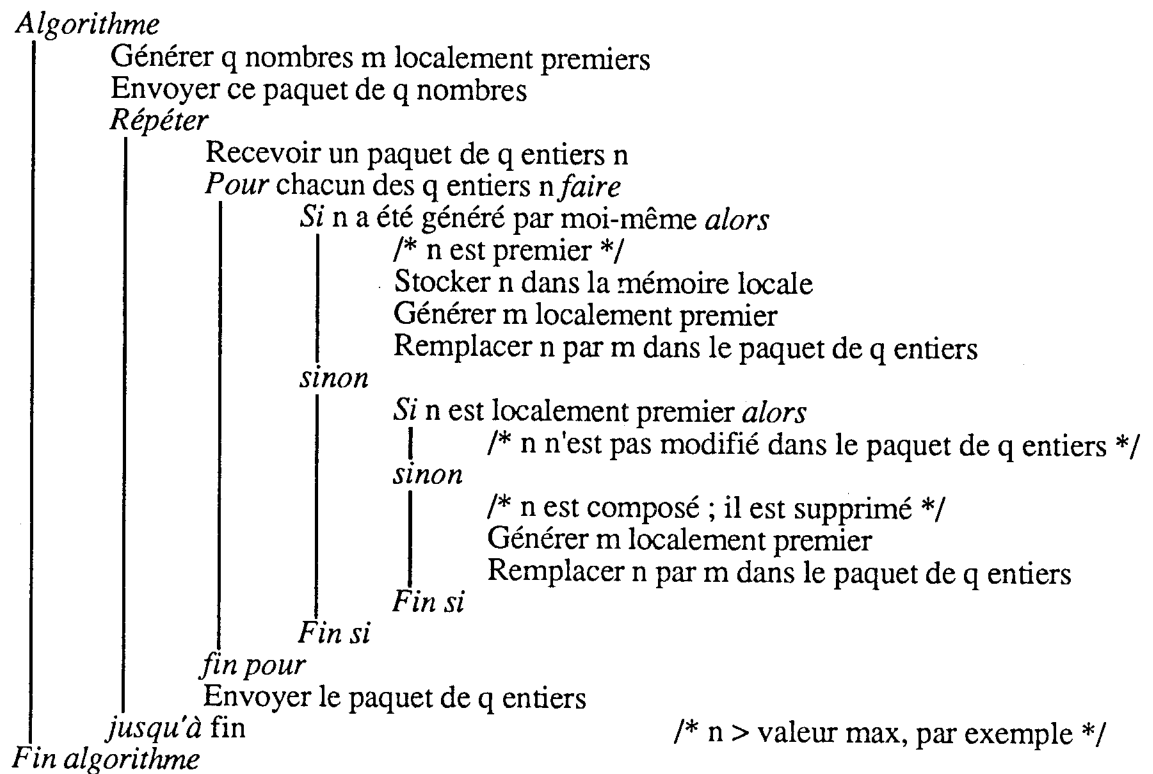
Nombre d'octets	Temps en µs	Coût par octet en µs
4	865,76	216,44
40	917,6	22,94
400	1436	3,59
4000	6620	1,66

Figure 1.23. Coût de l'envoi de messages en fonction de leur taille

Il faut diminuer le nombre de messages car le temps d'initialisation β est trop important. Il est plus efficace de transmettre des paquets d'entiers plutôt que d'envoyer les entiers un par un.

5.2.1. Algorithme

Groupons les entiers en paquets de q entiers sur chaque processeur et transmettons-les (envoi et réception) par paquets. L'algorithme est le suivant :



```
Algorithme
        Générer q nombres m localement premiers
        Envoyer ce paquet de q nombres
        Répéter
                Recevoir un paquet de q entiers n
                Pour chacun des q entiers n faire
                        Si n a été généré par moi-même alors
                                /* n est premier */
                                Stocker n dans la mémoire locale
                                Générer m localement premier
                                Remplacer n par m dans le paquet de q entiers
                        sinon
                                Si n est localement premier alors
                                        /* n n'est pas modifié dans le paquet de q entiers */
                                sinon
                                        /* n est composé ; il est supprimé */
                                        Générer m localement premier
                                        Remplacer n par m dans le paquet de q entiers
                                Fin si
                        Fin si
                fin pour
                Envoyer le paquet de q entiers
        jusqu'à fin                                     /* n > valeur max, par exemple */
Fin algorithme
```

Figure 1.24. Algorithme de génération multipipelinée des nombres premiers

Les résultats sont les suivants :

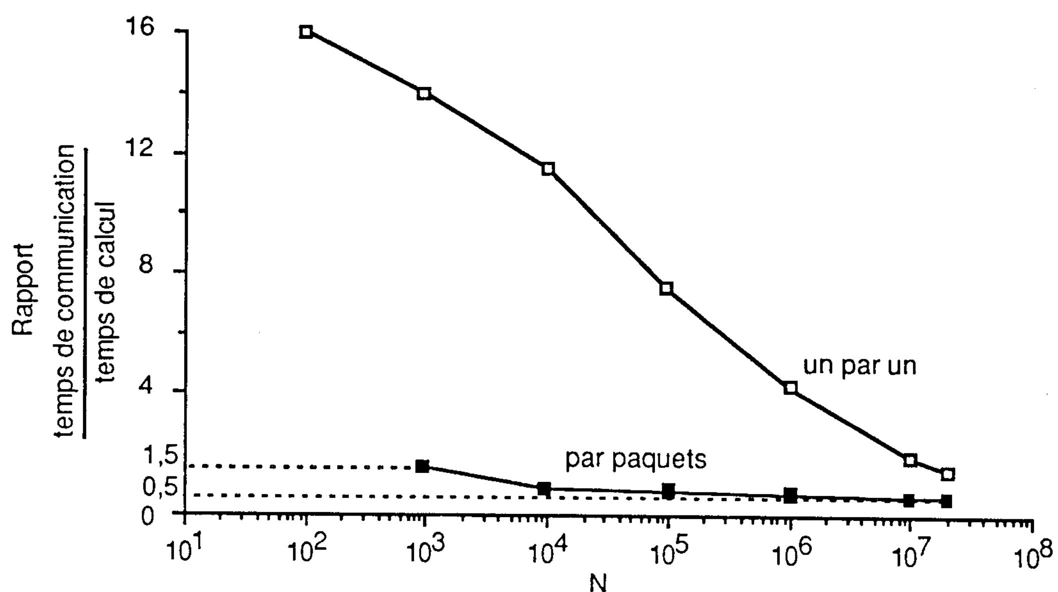


Figure 1.25. Rapports temps de communication / temps de calcul avec communications des entiers par paquets de 1024 entiers

Sur la figure 1.25, nous voyons que le rapport temps de communication / temps de calcul est plus petit avec le deuxième algorithme (utilisant des communications de paquets). Le rapport devient inférieur à 1, signifiant que la résolution du problème est maintenant dominée par les calculs.

Cette stratégie n'introduit pas de nouveau problème théorique, puisque la génération, les tests de primalité et le stockage se font toujours sur le même modèle. Cette stratégie permet un gain d'environ 50 % du temps pour $N = 10^7$ et $N = 2.10^7$, en utilisant des paquets de 1 024 entiers.

5.2.2. Evaluation de cette solution

Tout d'abord, le nouvel algorithme est plus rapide que le premier. Mais les processeurs n'éliminent pas les nombres composés à la même vitesse : il y a plus de multiples de 7 que de 19 ou de 29, et par conséquent, le processeur qui élimine les multiples de 7 élimine plus d'entiers, donc génère les nombres de sa famille (30x+7) plus vite que les autres processeurs.

Cependant, ceci n'est pas absolu : le processeur qui élimine les multiples de 7, n'élimine pas tous les multiples de 7. En effet, un multiple de 7 peut aussi être multiple d'un autre nombre premier, et être alors éliminé par un autre processeur.

Ceci se traduit par le fait que les processeurs n'effectuent pas le même nombre d'opérations de base (divisions).

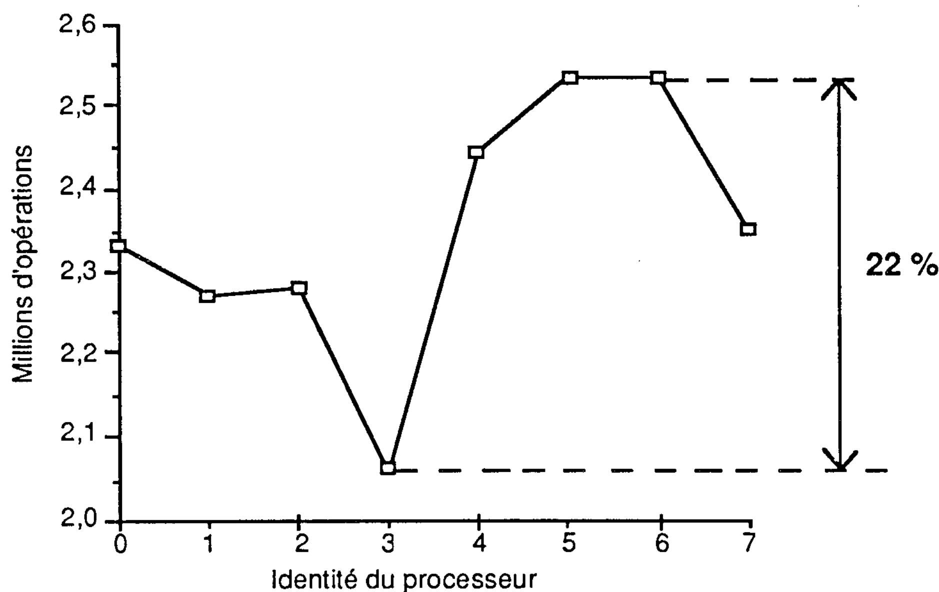


Figure 1.26. Nombre de divisions par processeur pour $N = 10^6$

C'est bien ce que nous pouvons constater sur la figure 1.26. Il y a une différence de plus de 400 000 opérations entre le processeur qui en effectue le plus, et le processeur qui en effectue le moins.

Le nombre d'opérations d'un processeur n'est pas proportionnel au nombre d'entiers que ce processeur élimine. C'est pourquoi, sur la figure 1.26, on ne peut pas dire que les processeurs PE_5 et PE_6 suppriment plus ou moins d'entiers que le processeur PE_3.

Et qu'en est-il du temps total d'exécution de chacun des 8 processeurs ? Terminent-ils tous en même temps ? Quelle est la différence entre le plus lent et le plus rapide ?

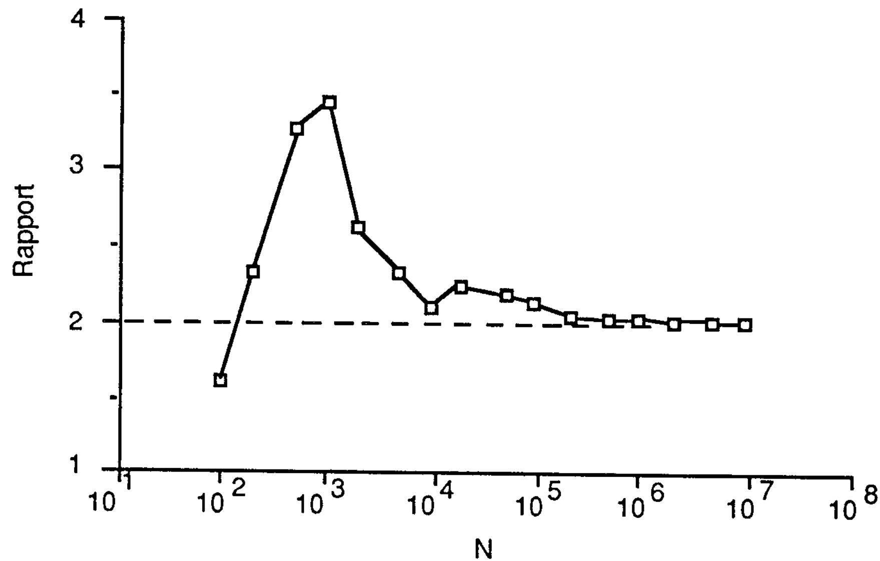


Figure 1.27. Rapport du temps d'exécution du processeur le plus lent au temps d'exécution du processeur le plus rapide

La figure 1.27 ci-dessus montre que le temps nécessaire à chaque processeur n'est pas le même, car ils n'effectuent pas le même nombre de divisions. De plus, le rapport du temps d'exécution du processeur le plus lent au temps d'exécution du processeur le plus rapide semble tendre vers 2. Nous ne savons pas pourquoi. Ce problème peut encore se poser sous la forme d'une mauvaise répartition des tâches. Par exemple, lors du calcul des nombres premiers inférieurs à 1 000, le plus grand entier généré par chaque processeur est donné sur la figure 1.28. Le processeur PE_1 vient de générer 907, alors que le processeur PE_5 est resté bloqué à 79 :

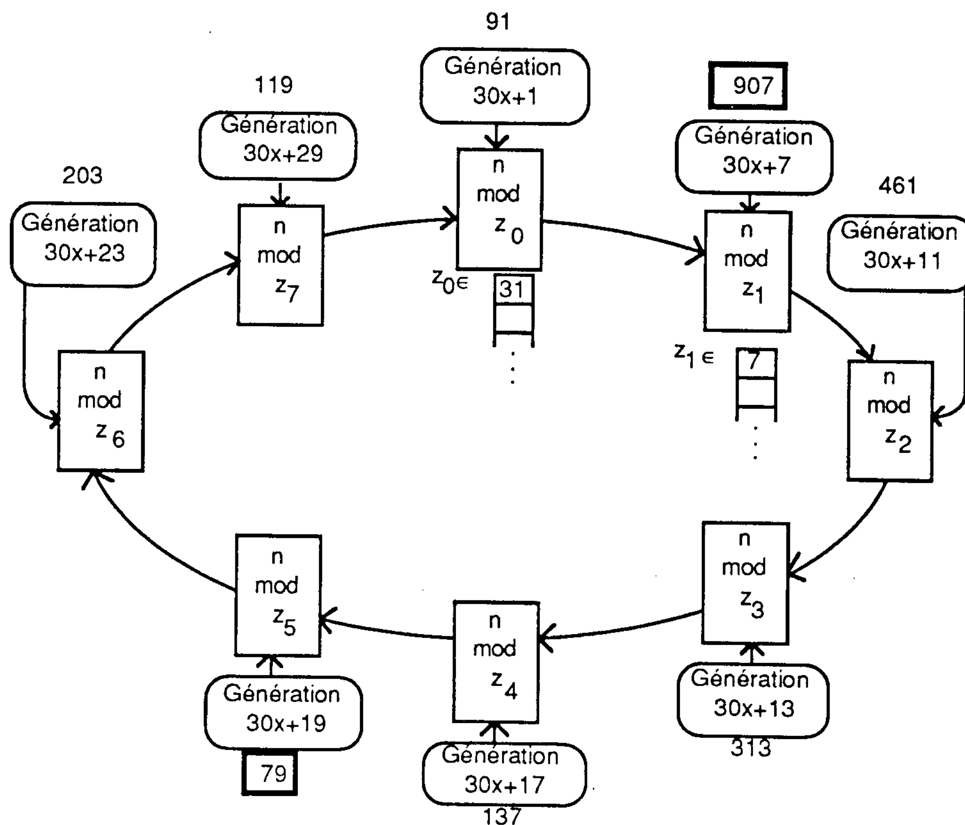


Figure 1.28. Plus grand entier généré par chaque processeur (état réel)

Ce problème de mauvaise répartition de la charge de travail est une conséquence du nombre différent d'opérations effectuées par chaque processeur. Il est difficile de répartir plus équitablement les divisions entre les processeurs puisque les différences tiennent à la répartition des nombres premiers dans les progressions arithmétiques. Asymptotiquement, les nombres premiers sont équidistribués entre les séries arithmétiques de la forme an+b, pour une valeur fixée de a, dans les séries contenant les nombres premiers [Rie 87 p. 62].

Nous allons essayer de diminuer le nombre de divisions.

5.3. DIMINUTION DU NOMBRE DE DIVISIONS

L'algorithme précédent effectue beaucoup de divisions inutiles. Montrons-le sur un exemple.

Supposons que nous énumérions les nombres premiers avec cet algorithme sur un anneau de 2 processeurs, générant respectivement des nombres de la forme 6x+1 (processeur PE_0) et 6x+5 (processeur PE_1). A un moment donné, le processeur PE_0 génère l'entier 1345 = 6*224+1. Or 1345 = 5*269. Le processeur PE_0 divise 1345 par 7, 13, 19 et 31. Comme $37^2 > 1345$, le processeur PE_0 envoie 1345 au processeur PE_1 qui le divise par 5, et élimine ce nombre dès la première division.

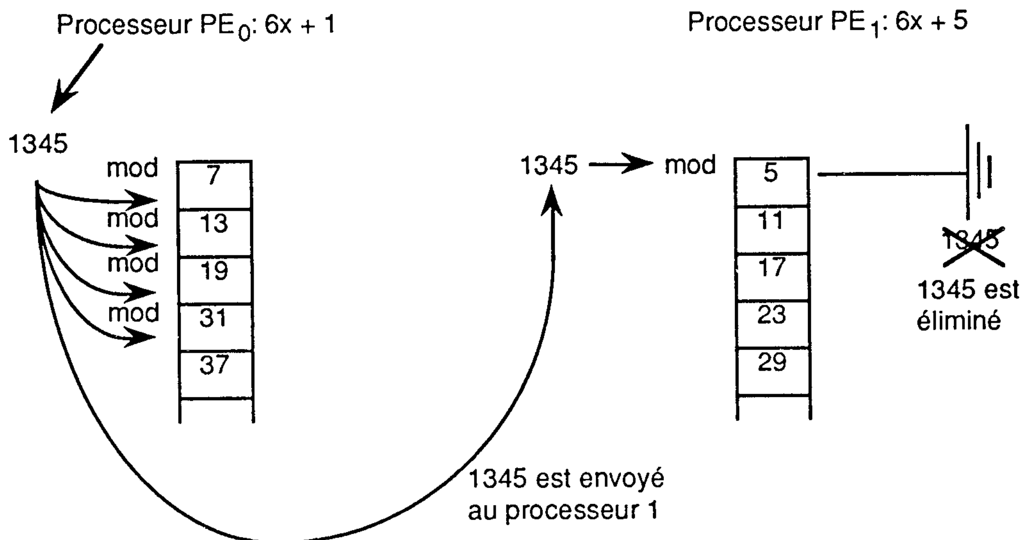


Figure 1.29. Exemple de divisions inutiles

On a effectué 4 divisions inutiles. Si le processeur PE_0 avait pu diviser par 5, il aurait immédiatement éliminé 1345.

5.3.1. La modification

Il semble intéressant de tester la primalité locale des entiers générés par les processeurs, non plus seulement relativement aux nombres premiers mémorisés (ou générés) par ce processeur, mais par tous les petits nombres premiers.

Rappelons que dans notre implantation, les nombres premiers inférieurs à 31 sont tous connus, mais de manière distribuée, chaque processeur ne connaissant qu'un seul de ces nombres premiers. Cependant l'information est globalement présente. Nous décidons alors, au prix d'une légère redondance, de faire connaître à chaque processeur tous ces nombres premiers, c'est-à-dire les nombres premiers inférieurs à 31, nécessaires à l'algorithme : 7, 11, 13, 17, 19, 23, 29, 31. Donc, lors de la génération d'un nouvel entier, le processeur le testera relativement à ces 8 nombres premiers, puis relativement aux nombres qu'il a générés et stockés lui-même. L'algorithme est modifié, car les données initiales en mémoire sont modifiées.

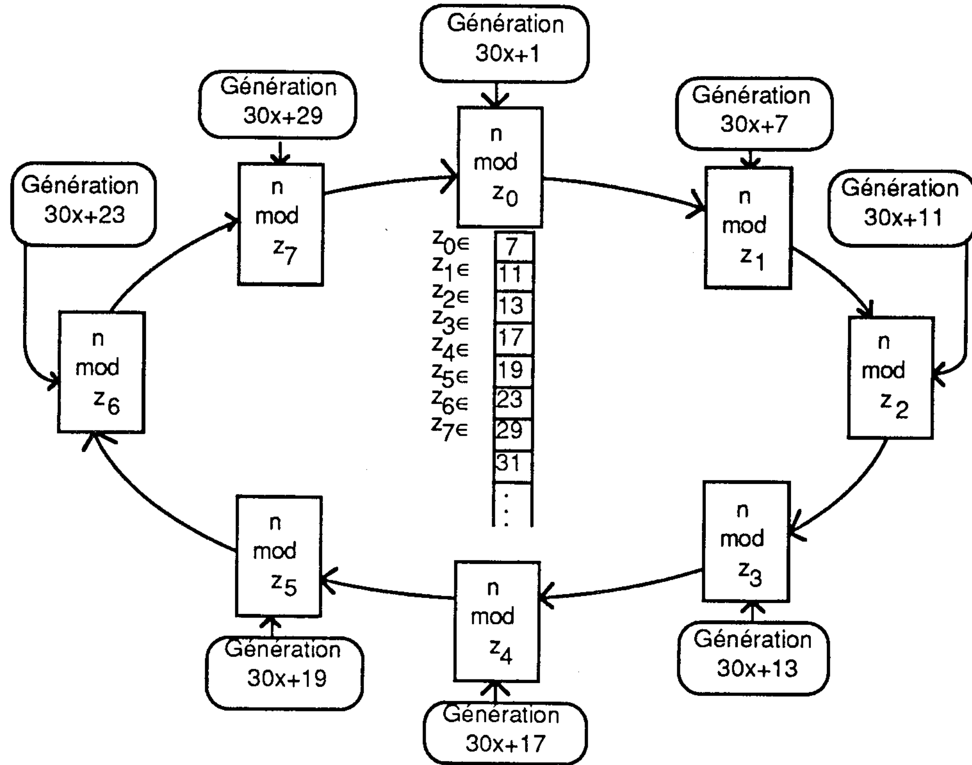


Figure 1.30. Les nouvelles données initiales en mémoire de chaque processeur

Ainsi tous les processeurs travaillent avec les mêmes données initiales.

5.3.2. Résultats

Le nombre d'opérations diminue d'environ 30%, ce qui constitue un gain appréciable pour une si faible redondance (8 entiers).

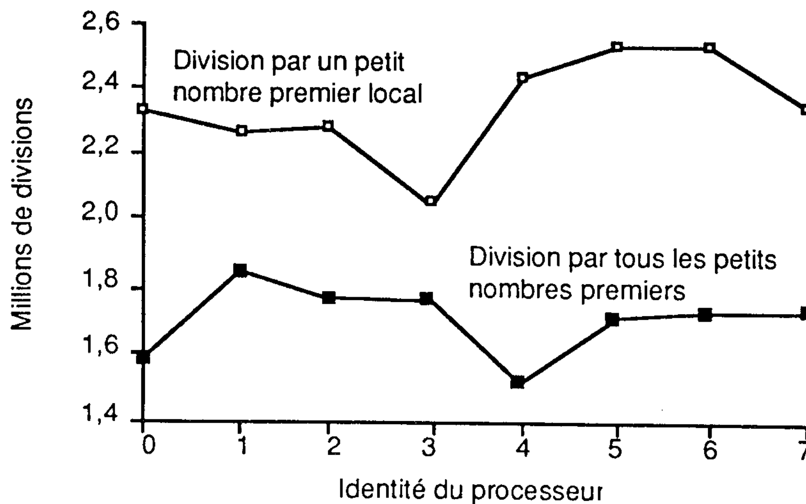


Figure 1.31. Nombre de divisions par processeur

De plus, cette stratégie permet un gain de près de 50% sur le temps d'exécution de l'algorithme présenté au paragraphe 5.2, toujours avec 8 processeurs. Ces résultats donnent envie d'augmenter encore un peu la redondance, en donnant à chaque processeur tous les nombres premiers inférieurs à p, mais p supérieur à 31. Des investigations futures ne pourraient-elles pas là trouver un compromis ?

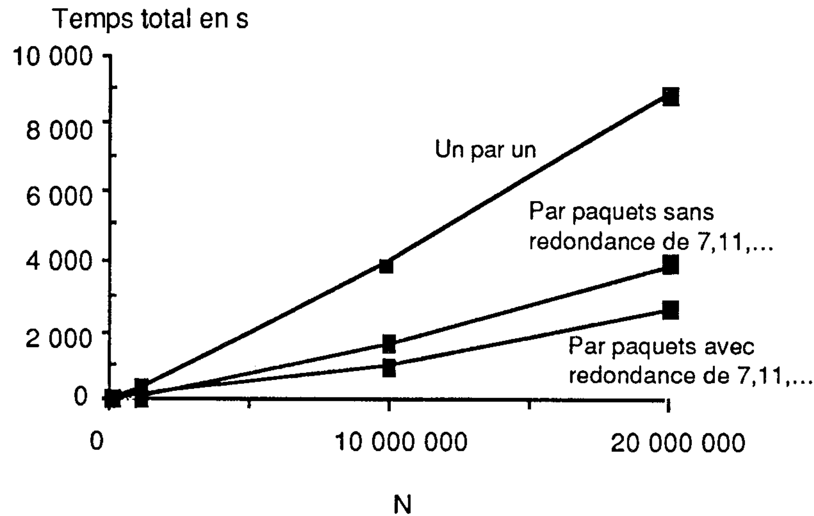


Figure 1.32. Comparaison entre les temps d'exécution des trois algorithmes multipipelinés

Pour étudier le gain (s'il existe) qu'apporte le parallélisme, comparons les temps d'exécution de cette stratégie algorithmique sur 1 processeur (pas de communication, uniquement du calcul), sur un anneau de 2 processeurs et sur l'anneau de 8 processeurs.

Sur un anneau de 2 processeurs, il faut moins de communications pour vérifier qu'un nombre est premier, que sur l'anneau de 8 processeurs.

Les temps sont donnés sur la figure suivante :

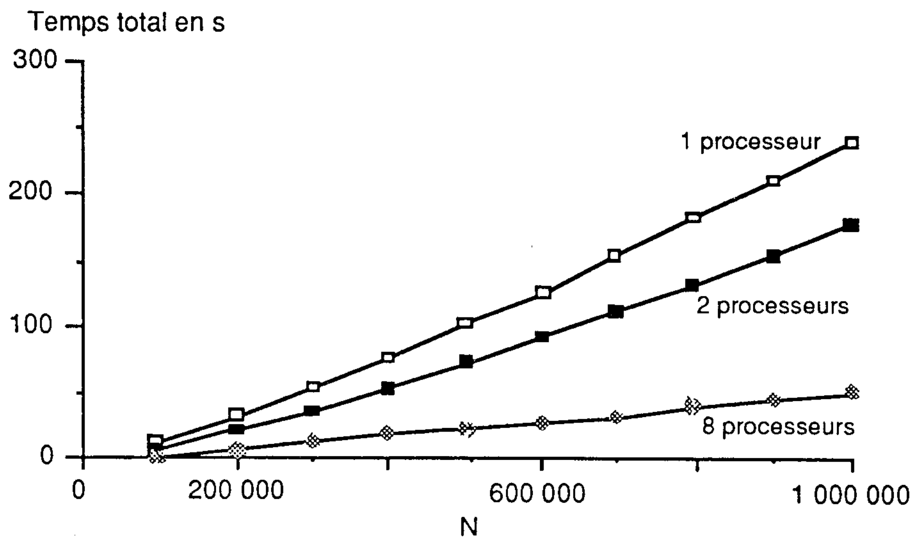


Figure 1.33. Comparaison des temps d'exécution de cet algorithme sur 1, 2 et 8 processeurs

Dans l'intervalle $N \in \{10^5, ..., 10^6\}$, faire exécuter l'algorithme sur 8 et même 2 processeurs fait gagner du temps par rapport à l'exécution sur un seul processeur. Le gain apporté par le parallélisme est suffisamment important pour compenser le temps perdu en communication.

Nous donnons une table des accélérations pour différentes valeurs de N. N est limité à 10^6, car sur un processeur, on ne peut stocker les nombres premiers que jusqu'à 1 000 000.

N	Temps en s 1 processeur	Temps en s 2 processeurs	Accélération 2 proc/1 proc	Temps en s 8 processeurs	Accélération 8 proc/1 proc
10^5	13	9	1,44	2,1	6,2
2.10^5	31	22	1,41	5,7	5,4
3.10^5	51	37	1,38	11	4,6
4.10^5	74	54	1,37	16	4,6
5.10^5	99	72	1,37	22	4,5
6.10^5	125	91	1,37	28	4,5
7.10^5	152	111	1,37	33,5	4,5
8.10^5	181	132	1,37	40	4,5
9.10^5	210	154	1,36	45	4,6
10^6	241	177	1,36	52	4,6

Figure 1.34. Temps comparés et accélérations sur 1, 2 et 8 processeurs

L'accélération mesurée est loin de l'accélération maximale théorique, égale au nombre de processeurs, car les communications sont importantes.

5.4. CONCLUSION

Cette stratégie de génération de tous les nombres premiers est efficace sur des multiprocesseurs à mémoire distribuée, si on utilise de façon correcte la mémoire (stockage des nombres premiers), et si on adapte l'algorithme (communications et calculs) à la machine-cible pour l'implantation. Cependant la répartition des données joue un rôle important qu'il ne faut pas négliger a priori. Et c'est au prix d'une faible redondance d'informations qu'on peut obtenir des accélérations intéressantes. Ces accélérations sont elles-mêmes limitées par les communications. Cependant, l'algorithme tel qu'il est présenté ci-dessus, ne fonctionne qu'en régime permanent, où N est illimité. Lorsque l'on fixe N, il faut arrêter l'exécution, ce qui n'est pas très simple. Nous présentons l'étude de la détection distribuée de la terminaison de cet algorithme dans le chapitre suivant.

6. CONCLUSION

On l'a vu, le crible d'Eratosthène, simple et efficace, peut poser de nombreux problèmes lors de sa parallélisation. Sur les multiprocesseurs à mémoire partagée, la répartition des entiers peut entraîner des déséquilibres importants lors de la phase d'élimination. Mais il est possible d'y faire face en distribuant les entiers inférieurs ou égaux à N de sorte que tous les multiples de tous les nombres premiers inférieurs à $\sqrt{N}$ soient en même proportion sur tous les processeurs.

De cette manière, les processeurs ne sont jamais inactifs pendant que d'autres travaillent. Nous avons atteint l'objectif d'équirépartition des tâches. Cependant, cette stratégie implique une gestion très importante et des opérations de base plus coûteuses. L'exécution est ralentie par rapport à l'algorithme de base du crible d'Eratosthène.

Sur les multiprocesseurs à mémoire distribuée, ce sont les communications qui jouent un rôle important et qui impliquent certaines restrictions sur les tâches des processeurs et la répartition des entiers. Mais le crible de base est très efficace sur ces machines, malgré des périodes d'inactivité (très courtes) des processeurs dues au type Maître / Esclaves de l'algorithme.

Quant à la génération multipipelinée, elle soulève de nombreux problèmes de coûts de communication qu'il faut réduire, de répartition des informations initiales sur les processeurs, sans compter la terminaison que nous étudions dans le chapitre suivant.

Les objectifs de cet algorithme étaient doubles. Nous voulions, en premier lieu, mieux utiliser les processeurs en installant un procédé de multipipeline. Ce but est atteint. Le deuxième but était, du fait de la meilleure utilisation des processeurs, de mettre au point un algorithme de génération des nombres premiers, qui soit plus rapide que le crible d'Eratosthène. Mais l'opération de base de ce nouvel algorithme est la division, beaucoup plus coûteuse que l'addition, opération de base du crible d'Eratosthène. Ce but n'a pas été atteint. Nous donnons dans le tableau suivant une comparaison des temps d'exécution de l'algorithme du crible d'Eratosthène et du générateur multipipeliné sur le FPS T20.

N	Temps d'exécution du générateur multipipeliné en s	Temps d'exécution du crible d'Eratosthène en s	Rapport
10^3	0,0316	0,0147	2,1
10^4	0,198	0,054	3,6
10^5	1,42	0,352	4,0
10^6	52,5	3,24	15
10^7	1580	33,4	47
2.10^7	4031	67,8	59

Figure 1.35. Temps comparés du générateur multipipeliné et du crible d'Eratosthène

Le crible "naturel" est beaucoup plus rapide.

Nous avons comparé les implantations de l'algorithme Maître / Esclaves du crible d'Eratosthène sur des multiprocesseurs à mémoire partagée et à mémoire distribuée. Nous avons remarqué, lors de l'écriture des algorithmes pour le FPS, un découpage des tâches du Maître et des Esclaves. De même les communications sont régulièrement localisées. Nous pensons qu'il est possible d'établir une méthodologie d'implantation d'algorithmes (de type Maître / Esclaves) sur des multiprocesseurs à mémoire distribuée. C'est ce que nous présentons au chapitre 3.

Chapitre 2

Détection de la terminaison distribuée et terminaison

1. INTRODUCTION

C'est à la fin des années 1970 que le problème de la détection de la terminaison distribuée est apparu. On s'est rendu compte que lors de calculs diffusants, le programme ne se terminait pas, bien que les calculs eux-mêmes soient tous effectués.

Rappelons le problème et sa formalisation tels que Dijkstra et Scholten les ont perçus à ce moment-là [DiS 80].

Considérons un graphe fini orienté. Si le graphe contient un arc d'un sommet A à un sommet B, appelons B "un successeur de A", et A "un prédécesseur de B". Supposons qu'il existe un unique sommet sans prédécesseur. Ce sommet sera appelé l'environnement (car il agit comme tel vis-à-vis du reste du graphe, et qu'il n'a pas de précédecesseur). Les autres sommets seront appelés les nœuds internes.

Pour chaque nœud, l'état initial est appelé "l'état neutre". Un "calcul diffusant" commence lorsque l'environnement envoie, de sa propre initiative, un message à un ou plusieurs de ses successeurs. On suppose qu'il ne fait ceci qu'une seule fois. A la réception de ce premier message, un nœud interne est libre d'envoyer des messages à ses successeurs. C'est cette caractéristique qui inspire l'appellation "calcul diffusant".

Supposons que chaque nœud interne envoie un nombre fini de messages. Pour un tel calcul, chaque nœud finira par atteindre la situation où il n'envoie ni ne reçoit plus aucun message. Lorsque tous les nœuds ont atteint cet état, le graphe est mort, et le calcul diffusant est alors terminé.

Dijkstra et Scholten, les premiers, cherchent à concevoir un procédé, superposé au calcul diffusant, qui signalerait la terminaison à l'environnement, lorsque celle-ci est atteinte [DiS 80].

Nous voulons, nous aussi, concevoir ce type de procédé, mais notre but est pratique : l'algorithme (chapitre 1 paragraphe 5) de génération des nombres premiers par divisions successives sur un anneau de 8 processeurs ne se termine pas, bien que les calculs soient tous complètement effectués sur chaque processeur.

Certains processeurs sont en attente de messages qu'il ne recevront jamais, puisque leur prédécesseur a terminé et n'a donc plus rien à leur envoyer, alors que d'autres processeurs cherchent à envoyer des messages à des processeurs qui n'en attendent plus. Nous cherchons donc à développer une technique qui détecte la terminaison distribuée sur un anneau de processeurs. Nous voulons aussi donner une preuve de sa correction.

Nous souhaitons que ce mécanisme de détection satisfasse certaines contraintes :

- il doit générer le plus petit nombre possible de messages,
- la gestion de ce mécanisme doit être indépendante de l'application sous-jacente,
- la gestion de ce mécanisme doit nécessiter le moins de puissance et de temps possible de la part du processeur.

Nous voulons aussi mettre en place, lorsque la terminaison est détectée, un mécanisme de terminaison.

Dans ce chapitre, nous donnons, au paragraphe 2, un état de l'art concernant les algorithmes pour la détection de la terminaison distribuée.

Puis nous exposons, dans le paragraphe 3, le problème qui se pose à nous : détecter la terminaison de l'algorithme de génération de nombres premiers sur un anneau par divisions successives, présenté au chapitre 1, paragraphe 5.

Le paragraphe 4 est ensuite consacré à la présentation d'une première solution à ce problème de détection de la terminaison et de terminaison elle-même, ainsi qu'à la preuve de l'algorithme. Cet algorithme utilise le principe du décompte de processeurs passifs. Chaque processeur en tient le compte.

Le paragraphe 5 donne une version de l'algorithme où les processeurs ne tiennent plus le compte des processeurs passifs. Ce sont des jetons valués circulant sur l'anneau qui tiennnent à jour une valeur minorant le nombre de processeurs passifs. Nous montrons que cet algorithme est correct. Puis nous donnons un évaluation de son coût, avant de présenter son exécution sur deux exemples.

Enfin, la conclusion donne les temps de terminaison de l'algorithme de génération des nombres premiers du chapitre 1, paragraphe 5.

2. ÉTAT DE L'ART

La détection de la terminaison se pose dans un environnement distribué synchrone ou asynchrone. Nous décrivons rapidement le modèle de calcul distribué sous-jacent à la détection de la terminaison distribuée. Puis nous présentons les premières solutions selon leurs caractéristiques :

- détection par jeton valué,
- détection par estampillage,
- détection à l'aide de marqueurs...

Ces solutions sont améliorées de diverses manières, et les études actuelles s'orientent vers la distinction synchrone / asynchrone, et vers un relâchement des contraintes posées sur les premiers algorithmes.

2.1. LE MODÈLE DE CALCUL DISTRIBUÉ

Un système distribué est composé d'un ensemble donné de processus qui ne communiquent que par échange de messages. Tous les messages sont reçus correctement, après un temps arbitraire mais fini.

Si le modèle est synchrone, l'émetteur et le récepteur doivent être prêts pour la communication. Si le modèle est asynchrone, les protagonistes de la communication n'ont pas besoin d'être prêts simultanément. On suppose qu'ils disposent de buffers illimités. Dans ce cas, les messages envoyés sur le même canal peuvent ne pas obéir à une règle FIFO.

Présentons les caractéristiques du modèle :

- un processus est soit actif, soit passif,
- seuls les processus actifs envoient des messages de travail,
- lors de la réception d'un message, un processus passif peut redevenir actif,
- un processus actif peut devenir inactif à tout instant.

Nous nous plaçons dans le cas où tout processus devient définitivement passif après un temps fini.

Nous essayons de détecter que l'application sous-jacente est terminée.

2.2. LES PREMIÈRES SOLUTIONS

Les premières techniques de détection de la terminaison distribuée consistaient à accuser réception de chaque message reçu, et à tenir le compte des messages envoyés et de leurs accusés de réception reçus, et des messages reçus et des accusés de réception renvoyés pour ces messages [DiS 80]. Le coût en est très élevé : cette technique double le nombre total de messages. De plus, il faut que le canal inverse existe pour chaque canal utilisé.

C'est Nissim Francez [Fra 80], qui formalise le premier le problème et le résout en termes de graphe acyclique à partir du langage CSP [Hoa 78].

A partir de là, les solutions sont aussi diverses que nombreuses. Cependant, ces algorithmes de détection de la terminaison peuvent se classer en plusieurs familles :

- les algorithmes utilisant des jetons values,
- les algorithmes de détection de la terminaison par estampillage,
- les algorithmes utilisant les marqueurs.

Nous allons donner les caractéristiques de chaque technique ainsi que les particularités des solutions basées sur ces techniques.

2.2.1. Jeton valué

Le principe est le suivant. Il s'agit de compter le nombre total de messages envoyés et de messages reçus. Si ces deux quantités sont égales (les canaux sont vides), et si les processus sont passifs, alors l'application est terminée.

On veut que le nombre total E(t) de messages envoyés à un instant t,

$$E(t) = \sum_i e_i(t)$$

soit égal au nombre total R(t) de messages reçus au même instant t,

$$R(t) = \sum_i r_i(t)$$

avec $e_i(t)$ et $r_i(t)$ le nombre de messages respectivement envoyés et reçus par le processus PE_i. Or un processus ne peut avoir qu'une vue approximative E* et R* de E(t) et R(t)

$$E^* = \sum_i e_i(t_i) \text{ et } R^* = \sum_i r_i(t_i)$$

car les valeurs e_i et r_i sont prises à différents instants sur les processeurs.

Donc, lorsque les processus sont passifs, et que E* = R*, le mécanisme de détection lance une deuxième vague de calcul pour donner deux autres valeurs E'* et R'*. La terminaison est détectée si

$$E^* = R^* = E'^* = R'^*.$$

Même si le système distribué a déjà terminé son application, l'algorithme de détection de la terminaison nécessite en général deux vagues pour conclure à la terminaison.

Cette idée de base a été développée par Dijkstra et al. [DFG 83] et Topor [Top 84].

Dijkstra et al. interprètent cette idée pour implanter l'algorithme utilisant un jeton unique circulant sur un anneau des processus. Le jeton circule dans un seul sens, du processus PE_i vers le processus PE_{i-1}. Ce jeton est initialement lancé par le processus PE_0. Le jeton reçu par un processus PE_i, n'est envoyé au processus successeur PE_{i-1} que si le processus PE_i est passif. Si ce processus est actif, il n'envoie le jeton que lorsqu'il devient passif.

Comme la topologie de communication de l'application sous-jacente est quelconque, un processus passif peut être rendu actif par un message provenant d'un processus actif non encore visité par le jeton. Lorsque le jeton aura effectué un tour sur l'anneau, il ne doit pas conclure que la terminaison est intervenue, même s'il a vu une fois tous les processus passifs.

Pour éviter ceci, le jeton est coloré en blanc ou en noir. Il est initialement blanc, et s'il revient blanc après un tour sur l'anneau, la terminaison est détectée.

Voyons comment la couleur du jeton est modifiée. Si un processus PE_i envoie un message à un processus PE_j $(j > i)$, il risque de rendre actif un processus déjà passif. PE_i, initialement blanc, devient noir. Lorsqu'un processus reçoit le jeton, il attend d'être passif pour le transmettre. Lorsqu'il envoie le jeton, un processus blanc ne change pas la couleur du jeton, mais un processus noir envoie un jeton noir.

Le processus PE_0, qui a envoyé le jeton blanc initialement, détecte la terminaison s'il est lui-même blanc et s'il reçoit un jeton blanc. Sinon, il envoie un nouveau jeton blanc en remplacement de celui qu'il vient de recevoir.

L'algorithme est simple. Un processus encore actif ne génère pas de message. En ce sens, il ne surcharge pas le système de communication. Cependant, il n'est pas symétrique, car un seul processus choisi avant l'exécution peut détecter la terminaison. Enfin, cet algorithme ne peut se dérouler que sur un anneau. Ceci peut éventuellement entrainer la création de nouvelles voies de communication.

Jean-Pierre Verjus [Ver 87] propose une preuve de cet algorithme de Dijkstra et al. [DFG 83].

Topor [Top 84] supprime la restriction d'exécution sur l'anneau, en le remplaçant par un arbre statique, un arbre de recouvrement du graphe des processus et des voies de communication. La racine joue encore un rôle privilégié. Cet algorithme n'est pas symétrique.

Francez en 1980, avait lui aussi utilisé le même type d'algorithme sur un arbre [Fra 80]. Apt prouve la correction de cet algorithme dans [Apt 84], [Apt 86].

Misra [Mis 83] utilise le même type d'algorithme avec un jeton coloré sur un circuit de processus contenant tous les arcs de communication. Il introduit la notion de symétrie dans sa nouvelle technique.

Le principe est le suivant. Le jeton visite les processus. De même que dans l'algorithme de Dijkstra et al. [DFG 83], la constatation par le jeton que tous les processus sont passifs ne permet pas de conclure que la terminaison est effective : un ou des processus ont pu être réactivés et les voies de communication peuvent contenir des messages en transit.

Le principe de l'algorithme de Misra repose sur la constatation que tous les processus sont restés en permanence passifs entre deux passages du jeton. (Il faut donc deux passages sur le circuit complet).

Le problème réside dans la détection de la passivité permanente d'un processus. De la même manière que précédemment, les processus sont noirs ou blancs. Le principe est le suivant :

- un processus devenant actif prend la couleur noire,
- le jeton peint un processus en blanc lorsqu'il le quitte,
- si le jeton retrouve blanc un processus qu'il a déjà visité, alors ce processus est resté continûment passif depuis le dernier passage du jeton,
- lorsque le jeton a visité tous les processus et les a trouvés tous blancs, alors il conclut que la terminaison a eu lieu.

L'avantage est qu'il n'y a pas de processus particulier. N'importe lequel des processus peut détecter la terminaison. Le seul inconvénient provient de la nécessité d'avoir, préalablement à l'exécution, défini un circuit qui incluse toutes les voies de communication.

2.2.2. Estampillage

L'idée de base pour estampiller les messages est la suivante : chaque processus est doté d'une horloge locale, incrémentée entre deux événements locaux et mise à jour lors des communications, pour que toutes les horloges soient synchronisées.

Rana [Ran 83] propose une méthode utilisant l'estampillage. Le principal avantage de son algorithme réside dans sa symétrie : il n'y a pas de processus fixé qui soit le seul à chercher à détecter la terminaison, mais tous les processus cherchent en même temps. Rana utilise l'environnement synchrone défini par Lamport [Lam 78]. Dans sa méthode, les processus sont connectés par un cycle hamiltonien et les communications de contrôle se font toujours sur ce cycle dans une seule et même direction.

Lorsqu'un processus devient passif, il écrit l'heure courante dans sa mémoire et envoie un message de détection, contenant l'heure courante et un compteur initialisé à la valeur 1, à son successeur sur le cycle.

Un processus actif détruit ce message de détection.
Si un processus passif reçoit ce message, il compare l'heure du message à l'heure qu'il a notée lorsqu'il est devenu passif. Si cette heure est supérieure à celle du message, le message est supprimé, sinon le compteur est incrémenté de 1 et transmis au successeur sur le cycle.
Si un processus passif reçoit un message de détection avec le compteur valant P (nombre total de processus), la terminaison distribuée est détectée. Ce processus lance un message de terminaison au successeur et termine.

Les messages de détection de la terminaison ne sont générés que lorsqu'un processus devient passif. Ainsi, ils n'encombrent pas les canaux de communication lors de l'exécution de l'application sous-jacente.

Michel Raynal [Ray 85] présente cet algorithme d'une façon élégante et donne une preuve de sa correction.

L'estampillage sert aussi à la détermination des états globaux de systèmes distribués, en obtenant des "instantanés" (snapshots) [ChL 85].

2.2.3. Les marqueurs

L'algorithme de Misra [Mis 83] peut être aussi considéré comme un algorithme utilisant les marqueurs. En fait, l'étude de Misra est à la base de cette idée d'utiliser des marqueurs. Dans

son algorithme, Misra utilise un seul marqueur et de façon séquentielle. Chandy et Misra [ChM 85] développent l'idée et utilisent plusieurs marqueurs en parallèle.

2.3. DE NOUVELLES IDÉES

A partir de 1986, on cherche des algorithmes plus généraux.

Arora et al. [ARG 86] proposent une solution complètement distribuée et symétrique, sans estampillage ni synchronisation d'horloges, et sur une topologie quelconque en ce qui concerne l'algorithme de base. Les messages de contrôle circulent sur un anneau unidirectionnel. Ils contiennent le numéro du processus qui les a envoyés. Un processus connaît l'état de ses voisins.

Lorsqu'un processus devient passif, il envoie cette information à tous ses voisins sur la topologie de l'application. Lorsqu'un processus reçoit cette information, il met à jour la variable sauvegardant l'état de ce voisin. Puis si tous ses voisins et lui-même sont passifs, il envoie un message de détection de la terminaison, à son successeur sur l'anneau. Un processus passif dont tous les voisins sont passifs transmet ce message sur l'anneau. Dans les autres cas, le processus récepteur détruit ce message. La terminaison est détectée lorsque le processus émetteur reçoit son propre message de détection de la terminaison (tout message contient l'identité de l'émetteur).

La restriction ici provient de la nécessité pour chaque processus de connaître l'état de ses voisins. Ceci implique des communications, même si elles sont locales.

Hazari et Zedan [HaZ 87] proposent, sur un anneau, de compter le nombre de processus passifs pour détecter la terminaison globale. Ainsi un processus devenant passif envoie un message avec un compteur égal à 1. Un processus passif ajoute 1 au compteur et transmet le message, tandis que tout processus actif détruit un tel message. Lorsque le compteur atteint le nombre de processus total, la terminaison est détectée.

Cet algorithme est simple, symétrique, et n'introduit pas de message de contrôle sur les processus actifs. Mais il ne fonctionne que dans des cas restreints, lorsque la topologie de communication est un anneau, et que les processus restent continûment passifs lorsqu'ils le sont devenus une fois. Cet algorithme n'est pas correct dans le cas général : Tel et van Leuwen [TeL 87] ont montré que cet algorithme pouvait détecter de fausses terminaisons.

Les recherches actuelles portent sur deux domaines :

- l'opposition synchronisme / asynchronisme,
- la suppression des contraintes sur les algorithmes existants en environnement synchrone (topologie figée, un seul processus détecte la terminaison, les informations globales que chaque processus doit connaître, ...).

2.3.1. Environnement asynchrone

Les algorithmes que nous venons de présenter sont essentiellement synchrones. Dans un environnement asynchrone, les délais d'acheminement des messages ne sont pas nuls, et les messages peuvent arriver au destinataire dans un ordre différent de celui dans lequel ils ont été émis.

Friedemann Mattern [Mat 87] analyse des algorithmes de détection de la terminaison dans un tel environnement. Il montre que la plupart des algorithme pour des environnements synchrones sont des cas particuliers (on a ajouté des contraintes) d'algorithmes de détection pour des environnements asynchrones.

Ferment et Rozoy [FeR 87] présentent eux aussi un algorithme pour un environnement asynchrone. Ils examinent le cas des processeurs en panne et l'impact sur la détection de la terminaison.

Dans un environnement asynchrone, Koo et Toueg [KoT 88] étudient l'impact de la perte de messages sur la détection de la terminaison.

Pascale Blanc [Bla 88] relâche les contraintes et donne plusieurs algorithmes dans le cas où les messages arrivent dans un ordre quelconque. Dans un environnement asynchrone, elle montre que la notion de symétrie n'est pas facile à mettre en œuvre. Son algorithme le plus puissant fait intervenir une notion de quasi-symétrie.

2.3.2. Relâche des contraintes dans un environnement synchrone

Certaines études portent sur d'anciens algorithmes. Friedemann Mattern [Mat 89] présente un algorithme similaire à celui de Dijkstra et al. [DFG 83], mais plus rapide. Cependant, l'algorithme respecte la règle de l'accusé de réception systématique pour tout message reçu. F. Mattern donne une solution de coût faible.

D'autres études essaient d'éviter des restrictions. Huang [Hua 88] présente un algorithme symétrique, sans restriction sur la topologie de connexion des processus. Lorsqu'un processus devient passif, il le fait savoir à tous les autres processus en spécifiant la date de son passage à l'état passif. Comme chaque processus connait la date la plus récente de passage à l'état passif de n'importe quel processus passif, il peut comparer la date qu'il connait à la date reçue.

Si tous les processus (sauf celui qui a envoyé cette date) sont unanimes pour dire que c'est la plus récente, alors la terminaison est détectée. Si un processus est encore actif, ou vient de devenir passif, il ne reconnait pas cette date comme la plus récente, et ne répond rien. Donc la terminaison n'est pas effective, et n'est pas détectée.

Dans ce contexte, il faut une horloge logique globale. De plus les communications se font d'un processus vers tous les autres, puis de tous vers un seul. Ce type de communication est très coûteux. En 1989, Huang modifie son algorithme et supprime le message d'accusé de réception pour chaque message [Hua 89]. En ce sens, il diminue de beaucoup le coût de l'algorithme.

Eriksen [Eri 88] propose un algorithme où les contraintes sont les plus faibles à ce jour : chaque processus ne connait qu'une borne supérieure de la taille du réseau. Mais l'algorithme nécessite de trop nombreuses communications. En effet, les processus mettent à jour systématiquement les valeurs de leurs compteurs chaque fois qu'elles changent.

Le dernier type de recherches concernant la détection de la terminaison distribuée se rapporte à la méthodologie de preuve de ces algorithmes. Si la preuve non formelle est aisée, une formalisation est parfois possible, même si elle est complexe. Certaines sont basées sur le modèle de CSP [Hoa 78] et utilisent sa modélisation, comme Rana [Ran 83]. Mais de plus en plus, de nouvelles techniques de preuve font leur apparition. Hehner et Malton [HeM 88] introduisent des formalismes sémantiques, en insistant sur la sémantique des prédicats. Hélary et Raynal [HeR 88] modélisent des trains de vagues pour réaliser des itérations distribuées. Ils construisent l'algorithme de détection de la terminaison avec sa preuve, par dérivation.

2.4. CONCLUSION

De 1980 à 1990, les recherches ont beaucoup évolué. D'un algorithme doublant le nombre de messages en 1980, on est parvenu, vers le milieu de la décennie, à des algorithmes n'impliquant pas de messages supplémentaires pour les processus encore actifs. D'algorithmes permettant seulement à un processus fixé de détecter la terminaison distribuée, on s'est acheminé vers des algorithmes symétriques où tout processus peut détecter cette terminaison, et même plusieurs processus peuvent la détecter en même temps. Des études sur des topologies précises (anneau, arbre) ont conduit à des solutions efficaces.

L'antinomie synchronisme/asynchronisme, la tolérance aux pannes, aux erreurs, la modification de l'ordre des messages, toutes ces notions nous permettent de nous rapprocher de la réalité et d'étudier des algorithmes adaptés aux problèmes réels.

Nous nous proposons de dériver dans ce chapitre un algorithme utilisable pour résoudre un problème réel [CoP 89] : celui de la détection de la terminaison et de la terminaison de l'algorithme de génération des nombres premiers par divisions successives sur un anneau de 8 processeurs, présenté au chapitre 1, paragraphe 5.

3. LE PROBLÈME RÉEL À RÉSOUDRE

Dans le chapitre 1, nous avons présenté une méthode de génération de nombres premiers par divisions successives, sur un anneau de 8 processeurs. Nous avons vu qu'en régime général, c'est-à-dire lorsque la valeur maximale du plus grand nombre premier à découvrir n'est pas fixée, le programme fonctionne correctement. On remarque cependant que des différences surviennent dans le degré d'avancement des processeurs. Certains ont généré des nombres bien plus grands que d'autres processeurs, alors qu'on s'attendait à ce que les processeurs travaillent à la même vitesse (ou à peu près la même).

Ils effectuent des nombres différents d'opérations, donc génèrent les nombres à des rythmes différents. Et la figure 2.1 le montre : l'état ici représenté est réellement atteint lorsqu'on cherche tous les nombres premiers inférieurs à 1000.

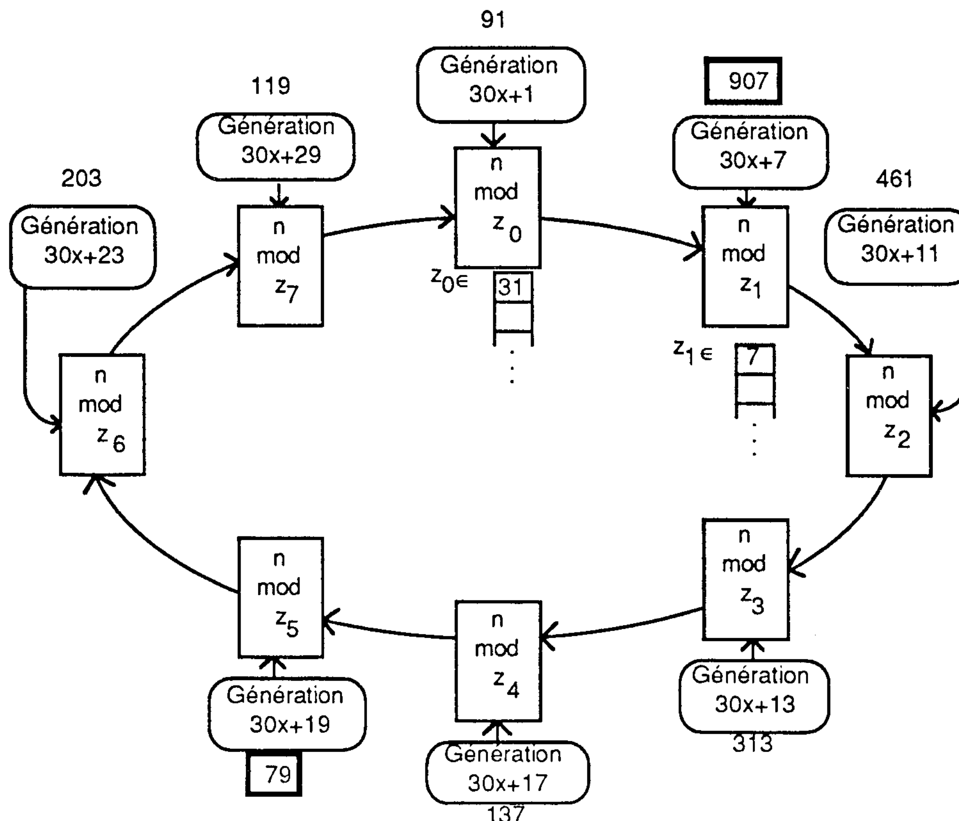


Figure 2.1. Plus grand entier généré par chaque processeur (état réel)

Le processeur PE_1 a déjà généré le nombre 907, alors que le processeur PE_5 est resté à l'entier 79. L'écart est très important : le processeur PE_1 a généré 90 % de ses nombres et le processeur PE_5 en a généré moins de 10 %.

Supposons que nous voulions obtenir tous les nombres premiers jusqu'à une limite supérieure N. Tous les processeurs PE_i, i = 0, ..., P-1, doivent donc s'arrêter de générer de nouveaux entiers de leur famille $30x+y_i$ lorsque le prochain élément de cette famille est plus grand que la limite supérieure N.

Mais chaque processeur a un autre processus qui s'exécute en même temps que celui de génération des entiers de la famille : c'est le processus de test de la primalité relative des entiers que ce processeur reçoit.

Un processeur s'arrête quand le prochain nombre qu'il devrait générer dépasse la limite supérieure, connue de chaque processeur. Lorsqu'un processeur s'arrête, il ne génère plus d'entiers. Mais il continue à tester la primalité des nombres qu'il reçoit. Si le nombre reçu est premier pour ce processeur, il est envoyé au successeur sur l'anneau. S'il n'est pas premier, il est éliminé. Le processeur recevra aussi des nombres qu'il a générés lui-même, et qu'il devra stocker dans sa mémoire locale.

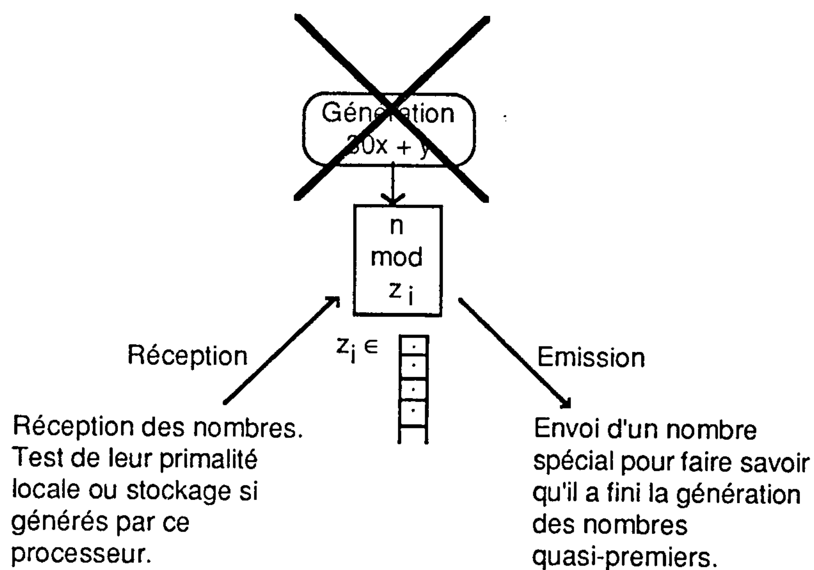


Figure 2.2. Un processeur arrête de générer

N'oublions pas que la vitesse d'élimination, même si elle a été améliorée, n'est pas encore parfaite. Les processeurs ne terminent pas la génération en même temps, car ils n'éliminent pas les nombres composés à la même vitesse les uns et les autres. C'est pourquoi nous devons détecter le moment où tous les processeurs atteignent ou viennent d'atteindre la limite supérieure de génération. Les processeurs qui ont arrêté de générer doivent le faire savoir aux autres. De plus, quand ils ont tous terminé, il faut que l'un d'eux sache que le calcul est fini, et que le programme distribué doit s'arrêter.

On doit détecter la terminaison et ensuite faire se terminer l'exécution.

Notre but est de concevoir un mécanisme de détection de la terminaison de cet algorithme de génération des nombres premiers. Nous le voudrions simple à mettre en œuvre, peu coûteux en nombre de communications qu'il induit et peu coûteux en temps pour détecter la terminaison une fois qu'elle est effective. Nous souhaitons pouvoir le prouver et éventuellement le porter sur d'autres topologies que l'anneau, tout en gardant les mêmes caractéristiques de simplicité et de coût faible.

La notion de simplicité de mise en œuvre peut se traduire par l'introduction de la symétrie dans l'algorithme, c'est-à-dire que le programme est le même pour tous les processeurs. Ainsi, n'importe quel processeur peut détecter le terminaison. Les algorithmes actuels sont

symétriques, par exemple [Ran 83], [Bla 88], [CoP 89]... Nous souhaitons donc concevoir un algorithme symétrique, pour des raisons de facilité d'écriture de l'algorithme.

La notion de coût peu élevé en nombre de communications que l'algorithme de détection induit, peut se traduire par : un processeur actif ne génère pas de message relatif à la détection de la terminaison. Seuls les processeurs passifs peuvent en générer. De cette manière, les canaux de communication ne sont pas surchargés par des messages de contrôle, et sont donc utilisables complètement par les messages de travail de l'application sous-jacente. Il faut que les messages de détection de la terminaison ne soient émis que par des processeurs passifs, comme dans l'algorithme de Dijkstra et al. [DFG 83], où le jeton n'est envoyé que par un processeur passif.

Nous pouvons donner des évaluations des coûts d'algorithmes de détection de la terminaison. L'algorithme de Topor [Top 84] utilisant un réseau de N machines, et où l'application génère M messages de travail nécessite au plus O(N*M) messages pour détecter la terminaison. S. Levi et Brigitte Plateau proposent un algorithme nécessitant moins de $O(N^2)$ communications pour détecter la terminaison sur N processus distincts [LeP 86]. Mais ces coûts sont beaucoup trop importants.

La notion de coût peu élevé en temps pour détecter la terminaison, lorsqu'elle est effective, peut se traduire par : une fois que la terminaison distribuée est atteinte, l'algorithme détecte cette terminaison en effectuant un petit nombre de communications. Nous estimons que deux passages sur chaque processeur et chaque voie de communication doivent suffire pour détecter la terminaison lorsqu'elle est effective. Nous souhaitons même aller vers un algorithme qui ne demande qu'un seul passage sur chaque processeur et chaque voie de communication. Par exemple, l'algorithme de Dijkstra et al. [DFG 83] nécessite deux tours sur l'anneau pour détecter la terminaison lorsqu'elle est intervenue.

En résumé, un bon algorithme de détection de la terminaison serait, à notre avis, un algorithme qui ne génère pas de message de contrôle sur les processeurs actifs, qui soit symétrique, et qui détecte la terminaison lorsque celle-ci est effective, en ne parcourant qu'une seule fois l'ensemble des processeurs et des canaux. S'il pouvait ensuite terminer l'application en un seul autre parcours des processeurs et des canaux, il aurait un attrait supplémentaire.

Nous allons essayer de nous rapprocher de cette solution. Il faut dire que nous partons avec des hypothèses assez fortes, puisque la topologie de connexion de l'application sous-jacente est un anneau unidirectionnel, ce qui simplifie la construction d'un circuit hamiltonien.

4. ALGORITHME DE DÉTECTION DE LA TERMINAISON ET DE TERMINAISON

L'idée de base est de compter les processus qui ne sont plus actifs, c'est-à-dire qui ne génèrent plus d'entiers. Lorsque ce nombre est égal au nombre de processeurs du réseau, alors la terminaison est détectée. Nous utilisons en partie les idées de Hazari et Zedan [HaZ 87], mais leur algorithme n'est pas correct dans tous les cas [TeM 89].

Nous voulons que notre algorithme soit symétrique. Pour que chaque processeur puisse détecter la terminaison, il faut que chacun d'eux compte les processeurs passifs. Il faut aussi qu'il connaisse le nombre total de processeurs du réseau.

N'oublions pas (voir chapitre 1, paragraphe 5) que d'autres messages circulent aussi sur l'anneau : les nombres à tester. Nous ne voulons pas que les messages de contrôle (détection de la terminaison) influent sur les communications de l'application sous-jacente.

Nous allons mettre en place un mécanisme pour faire savoir à tous les processeurs qu'un des leurs arrête de générer des entiers. Ce mécanisme permet aussi d'accélérer l'application sous-

jacente lorsqu'un processeur s'arrête, en réutilisant la place laissée par ce processeur dans les communications. Par la suite, nous montrons que cet algorithme est correct. Puis nous proposons un algorithme de terminaison de l'application sous-jacente.

4.1. MÉCANISME DE GESTION DE LA TERMINAISON D'UN PROCESSEUR

Rappelons le contexte topologique. L'application s'exécute sur un anneau unidirectionnel. Les messages circulant sur cet anneau sont des entiers strictement positifs. La gestion de l'application sur une topologie théorique de multipipelines imbriqués permet de ne jamais introduire de bulle dans les pipelines, en régime général.

Lorsqu'un processeur arrête de générer des entiers, il crée une bulle dans les pipelines. Nous utilisons cette bulle pour faire savoir à tous les processeurs que ce processeur n'est plus actif. Le message envoyé est un message que les autres processeurs savent reconnaître : l'opposé de l'identité de ce processeur - 1 (car les P processeurs sont numérotés de 0 à P-1). Donc, lorsqu'un processeur reçoit un message dont la valeur est comprise entre -1 et -P, il ajoute 1 au compteur, initialisé à 0, de processeurs ne générant plus. Lorsque ce message a effectué un tour complet sur l'anneau, l'émetteur sait reconnaître le message qu'il a émis et il le supprime. Nous avons atteint le premier but : pas de message de contrôle émis par un processeur encore actif.

4.2. SUPPRESSION DE LA BULLE INTRODUITE PAR L'ARRÊT D'UN PROCESSEUR

Pendant un tour sur l'anneau, le message de contrôle, stipulant qu'un processeur arrête de générer des entiers, utilise les canaux de communication de l'application sous-jacente. Et ce message de contrôle est un message inutile pour l'application. Mais après que ce message a effectué un tour, le processeur le supprime et crée une bulle qu'il lance dans les pipelines.

Ainsi tout processeur ayant reçu et supprimé le message de diffusion de son passage à l'état passif, envoie un message de valeur 0. C'est la bulle. Cette bulle est supprimée par le premier processeur actif qui la reçoit. Ce processeur génère un nouveau nombre à tester qui remplace cette bulle. Tout processeur passif transmet cette bulle à son successeur. De cette manière, les pipelines sont toujours pleins de messages de travail, sauf lorsqu'un processeur devient passif.

Nous avons atteint un but non fixé : le passage à l'état passif d'un processeur ne ralentit pas l'exécution de l'application sous-jacente, puisqu'un processeur encore actif prend la place d'un processeur devenant passif. En effet, initialement, chaque processeur avait un message dans les pipelines. Maintenant, un processeur peut en avoir plusieurs. Cette technique permet à l'application de nécessiter moins de temps pour s'exécuter.

4.3. DÉTECTION DE LA TERMINAISON

Le principe de détection de la terminaison distribuée est le suivant. A un certain instant, il ne reste plus qu'un processeur actif. Et il y a donc au moins un processeur dont le compteur de processeurs passifs vaut P-1. Le dernier processeur actif atteint sa limite, et envoie le message de diffusion de son passage à l'état passif. Ce message fait un tour sur l'anneau. Donc le processeur dont le compteur valait P-1 a ajouté 1 à ce compteur. Il vaut P maintenant. Et ce processeur détecte la terminaison.

4.4. PREUVE DE CORRECTION

Il existe plusieurs types de preuve. Nous allons utiliser celle qui prouve les propriétés de vivacité et de sûreté. Elles correspondent respectivement à la propriété de détecter la terminaison lorsqu'elle est effective, et à la propriété de ne pas détecter de fausse terminaison. Rana [Ran 83] utilise ce type de preuve.

4.4.1. La terminaison globale sera obligatoirement détectée. (Propriété de vivacité).

Démonstration :
Quand un processeur devient passif, il envoie un message contenant - (son identité) - 1. Ce message effectue un tour complet sur l'anneau, et rencontre donc tous les processeurs. Chaque processeur possède un compteur, initialisé à 0, de processeurs passifs. Tout processeur, lorsqu'il reçoit un message dont la valeur est comprise entre -1 et -P, ajoute 1 à son compteur de processeurs passifs.

A un instant précis, il ne reste qu'un processeur actif. Il y a donc au moins un processeur dont le compteur de processeurs passifs vaut P-1 après un certain temps. Le dernier processeur devient passif et envoie le message de diffusion de son passage à l'état passif. Ce message fait un tour sur l'anneau. Il est reçu et transmis au suivant par tous les processeurs, et en particulier par celui dont le compteur vaut P-1. Ce processeur ajoute 1 à son compteur qui atteint alors la valeur P. Ce processeur détecte la terminaison globale.
La terminaison globale distribuée est par conséquent détectée lorsqu'elle survient.

CQFD.

4.4.2. La terminaison globale n'est pas détectée si elle n'est pas intervenue. (Propriété de sûreté).

Démonstration :
La terminaison globale n'est détectée que si un processeur a une valeur de son compteur de processeurs passifs égale à P. Prouvons qu'une telle valeur ne peut pas être atteinte par un quelconque compteur, tant qu'un processeur, au moins, est encore actif. Lorsqu'un processeur devient passif, il envoie un message le stipulant, à tous les processeurs. Les processeurs comptent les messages de ce type qu'ils reçoivent. Or, chaque processeur qui devient passif n'envoie ce message qu'une seule fois. Donc s'il reste encore au moins un processeur actif, ce processeur n'a pas encore envoyé un tel message. Le compteur de processeurs passifs sur chaque processeur ne peut donc pas avoir atteint P : il vaut au plus P-1.
Donc une fausse terminaison globale ne peut pas être détectée.

CQFD.

4.5. ALGORITHME DE DÉTECTION DE LA TERMINAISON

A partir des idées et de la preuve qui précèdent, nous donnons l'algorithme de détection de la terminaison, après avoir décrit les états des processeurs et les types de messages possibles.

4.5.1. Etats des processeurs

- Un processeur qui exécute le programme de l'application sous-jacente est *actif*.
- Un processeur qui ne peut plus exécuter le programme de l'application sous-jacente devient *passif*. A l'instant où il devient *passif*, il envoie le message de diffusion de son passage à l'état *passif*. Ce message contient la valeur : - (identité du processeur) - 1. Il reste dans cet état *passif* jusqu'à ce qu'il reçoive le message de diffusion de son passage à l'état *passif* qu'il a émis. Ce message effectue un tour sur l'anneau.
- Un processeur *passif*, qui reçoit son propre message de diffusion de son passage à l'état *passif*, ajoute 1 à son compteur de processeurs passifs, et passe de l'état *passif* à l'état *bulle*, si son compteur est inférieur au nombre total P de processeurs. Il passe de l'état *passif* à l'état *détection* si son compteur vaut P (il a détecté la terminaison).
- Un processeur *passif* qui reçoit un message de passage à l'état *passif* d'un autre processeur, incrémente de 1 son compteur de processeurs passifs. Si son compteur est inférieur à P, le processeur reste *passif*, sinon il passe à l'état *détection* (il a détecté la terminaison).
- Un processeur à l'état *bulle* qui reçoit un message de passage à l'état *passif* d'un autre processeur, incrémente de 1 son compteur de processeurs passifs. Si son compteur est inférieur à P, le processeur reste dans l'état *bulle*, sinon il passe à l'état *détection* (il a détecté la terminaison).

Il y a donc 4 états possibles pour un processeur :

- *actif* : exécution de l'application sous-jacente,
- *passif* : le processeur ne peut plus exécuter l'application, et le fait savoir,
- *bulle* : le processeur *passif* a reçu son message de diffusion, et son compteur est inférieur à P,
- *détection* : le compteur de ce processeur vaut P.

4.5.2. Types des messages

Les messages de l'application et les messages de contrôle (détection de la terminaison) circulent en même temps sur l'anneau. Les messages possibles sont :

- des messages à valeur positive : les messages de l'application sous-jacente,
- des messages à valeur comprise entre -1 et -P, pour faire savoir qu'un processeur devient *passif*,
- un message à valeur nulle, la bulle, qu'un processeur *passif* transmet sur l'anneau, et que le premier processeur *actif* remplace par un message de travail.

4.5.3. L'algorithme pour le processeur PE_i, $0 \leq i \leq P-1$.

```
Algorithme
|   état = actif; compteur = 0
|   tant que (état ≠ détection) faire
|   |   réception (message)
|   |   choix état parmi :
|   |   |   état==actif:  choix message parmi
|   |   |                 |  message > 0 :          si impossible_générer_msg_appl alors
|   |   |                 |                         |   état = passif; message = -i - 1
|   |   |                 |                         fin si
|   |   |                 |  -P ≤ message ≤ -1 :    compteur = compteur + 1
|   |   |                 |  message == 0 :         supprimer_message
|   |   |                 |                         générer_msg_appl (&message)
|   |   |                 fin choix
|   |   |   état==passif :choix message parmi
|   |   |                 |  message ≥ 0 :
|   |   |                 |  -P ≤ message ≤ -1 :    compteur = compteur + 1
|   |   |                 |                         si (compteur == -P) alors
|   |   |                 |                         |   état = détection
|   |   |                 |                         fin si
|   |   |                 |  message == -i-1 :      compteur = compteur + 1
|   |   |                 |                         si (compteur == -P) alors
|   |   |                 |                         |   état = détection
|   |   |                 |                         sinon
|   |   |                 |                         |   état = bulle; message = 0
|   |   |                 |                         fin si
|   |   |                 fin choix
|   |   |   état==bulle : choix message parmi
|   |   |                 |  message ≥ 0 :
|   |   |                 |  (-P ≤ message ≤ -1) &&
|   |   |                 |  (message ≠ -i-1) :     compteur = compteur + 1
|   |   |                 |                         si (compteur == -P) alors
|   |   |                 |                         |   état = détection
|   |   |                 |                         fin si
|   |   |                 fin choix
|   |   fin choix
|   |   envoyer (message)
|   fin tant que
fin algorithme
```

Figure 2.3. Algorithme de détection de la terminaison (chaque processeur a un compteur)

Les procédures de traitement de l'application sous-jacente s'insérent dans cet algorithme aux points de choix où le message est positif ou nul.

4.6. TERMINAISON DE L'APPLICATION

Lorsque la terminaison est détectée (par un processeur ou par plusieurs en même temps), il faut mettre en œuvre un mécanisme qui soit apte à vider les voies de communication et mettre les processeurs dans un état cohérent appelé fin, où ils n'attendent plus de message et n'en envoient plus.

Pour ce faire, l'idée est que tout processeur qui détecte la terminaison bloque les messages qu'il reçoit et les détruise. Cependant, il envoie encore un seul et dernier message vers son successeur sur l'anneau. Ce message a pour but de faire savoir que la terminaison est détectée, et que les processeurs doivent terminer proprement. Donnons à ce message la valeur -F, avec -F < -P. Le récepteur de ce message le transmet à son successeur, et sait qu'il ne doit plus rien envoyer ni recevoir. Ce message est arrêté et détruit par tout processeur qui a détecté la terminaison.

La figure 2.4 montre comment deux processeurs (ou plus) peuvent terminer en même temps :

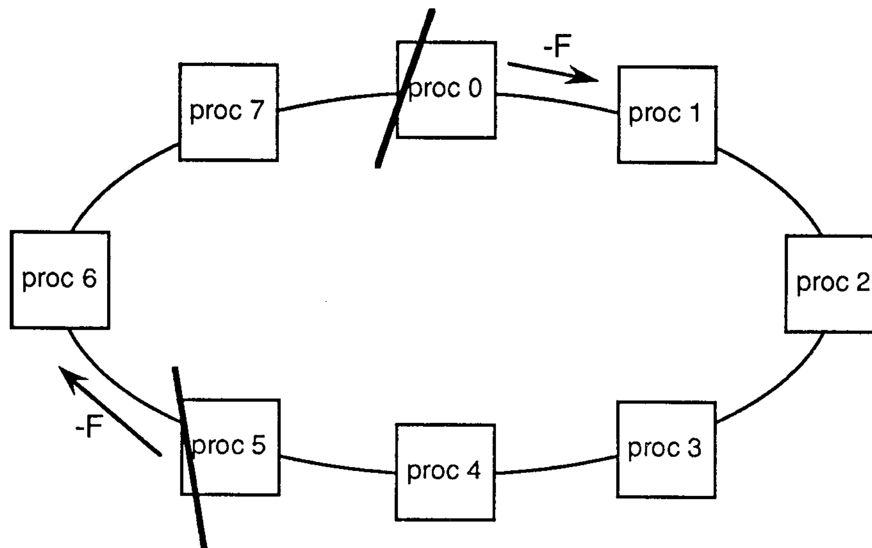


Figure 2.4. Passage simultané de deux processeurs en mode terminaison

Deux processeurs peuvent donc terminer dans le même tour. Le processeur PE_5 recevra la marque -F émise par le processeur PE_0, et tous les processeurs entre le PE_0 et le PE_5 auront terminé. En parallèle, le processeur PE_0 recevra la marque -F émise par PE_5, et tous les processeurs entre le PE_5 et le PE_0 auront terminé.

4.7. CONCLUSION

Nous avons conçu et prouvé un algorithme de détection de la terminaison distribuée. Il satisfait en partie les contraintes que nous lui avions assignées :
- symétrie,
- détection rapide de la terminaison lorsqu'elle est intervenue.

Cependant, un processeur qui devient passif génère un message qui utilise du temps de l'application sous-jacente, pour faire savoir à tous les processeurs qu'il devient passif.

Il est possible de supprimer cette étape. Pour ce faire, il faut que le compteur ne soit pas présent sur chaque processeur, mais sur un jeton valué. Nous utilisons une partie des idées de Misra [Mis 83] pour intégrer la notion de jeton valué à notre algorithme actuel.

5. DÉRIVATION D'UN ALGORITHME DE DÉTECTION DE LA TERMINAISON ET DE TERMINAISON À L'AIDE D'UN JETON VALUÉ

L'algorithme précédent de détection de la terminaison perd du temps lorsqu'un processeur devient passif, car il envoie cette information aux autres processeurs. Il faut supprimer cette étape.

Misra [Mis 83] propose la technique suivante : il fait circuler sur un anneau un jeton comptant le nombre de processeurs passifs. Et ce jeton n'est émis par un processeur que lorsqu'il est ou devient passif. Donc cette technique ne génère pas de communications supplémentaires. Cependant, un seul processeur peut détecter la terminaison, à savoir celui qui a émis le jeton. En effet, il faut que le jeton revienne à l'émetteur avec le compteur valant P pour que ce processeur puisse détecter la terminaison. En fait, il pourrait s'arrêter sur le prédécesseur de l'émetteur du jeton, et détecter la terminaison dès que ce prédécesseur devient passif.

Nous allons modifier l'algorithme du paragraphe précédent et intégrer la technique des jetons valués comptant les processeurs passifs. Nous lui ajoutons la notion de symétrie. Nous autorisons que plusieurs jetons valués circulent dans le réseau. Ainsi, n'importe quel processeur pourra détecter la terminaison. De plus, plusieurs processeurs pourront simultanément détecter cette terminaison.

Nous donnons l'algorithme correspondant, puis nous le prouvons. Nous ajoutons à cet algorithme, la terminaison de l'application sous-jacente qui permet de terminer l'application avec des processeurs dans un état cohérent et des canaux de communication vides. Nous donnons une évaluation du coût de cet algorithme, et nous présentons son exécution dans deux cas : le cas où les processeurs s'arrêtent l'un après l'autre, et le cas où les processeurs s'arrêtent en même temps. Il faut moins d'un tour sur l'anneau, après que le dernier processeur est devenu passif, pour détecter la terminaison. Il faut moins d'un tour sur l'anneau pour terminer l'exécution de l'application dès lors que la terminaison a été détectée.

5.1. MÉCANISME DE GESTION DE LA TERMINAISON D'UN PROCESSEUR

Lorsqu'un processeur devient passif, il génère un jeton contenant un compteur dont la valeur initiale est -1 et l'envoie à son successeur sur l'anneau. Le compteur de ce message est décrémenté de 1 par chaque processeur passif qu'il rencontre. Il est absorbé par un processeur actif, qui le remplace par un message de travail de l'application sous-jacente. Lorsqu'un processeur passif reçoit un message de travail, lui demandant d'émettre un message de travail, ce processeur ne peut pas le faire. En remplacement de ce message de travail, il envoie un jeton initialisé à -1. Lorsqu'un processeur reçoit un jeton de valeur -P, alors il détecte la terminaison.

5.2. MÉCANISME DE TERMINAISON DE L'APPLICATION

Lorsque la terminaison est détectée, il faut terminer l'application et rendre les processeurs dans des états propres et les canaux de communication vides. A cet instant, tous les processeurs sont passifs (puisque la terminaison a été détectée). De la même manière que pour l'algorithme du paragraphe 4, un processeur qui détecte la terminaison bloque les messages qui lui parviennent.

- Si un processeur *passif* reçoit un message dont la valeur du compteur est -P, il détecte la terminaison, passe à l'état *blocage* et envoie un dernier message avec la valeur -P-1. Il détruit les messages qui lui parviennent par la suite.
- Si un processeur *passif* reçoit un message dont la valeur du compteur est strictement inférieure à -P, alors il termine après avoir envoyé le message décrémenté de 1. Il ne reçoit ni n'envoie plus aucun message. Il passe à l'état *fini*.
- Si un processeur *passif* reçoit un message dont la valeur du compteur est strictement supérieure à -P, alors il décrémente ce message et l'envoie.

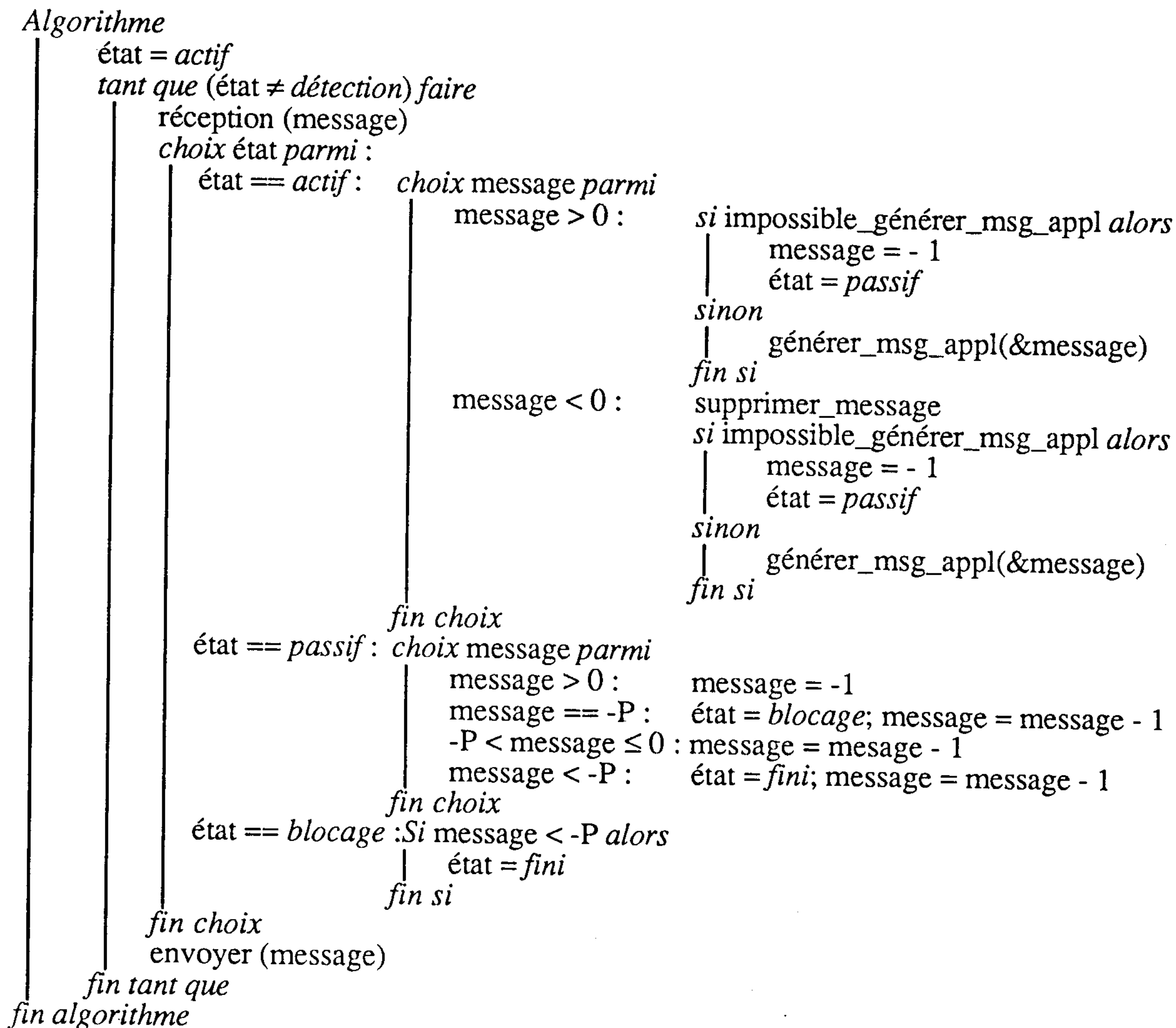
- Si un processeur à l'état *blocage* reçoit un message dont la valeur du compteur est strictement inférieure à -P, il termine (état = *fini*) et ne reçoit ni n'envoie plus rien.
- Si un processeur à l'état *blocage* reçoit un message dont la valeur du compteur n'est pas strictement inférieure à -P, il reçoit des messages qu'il détruit, mais n'envoie plus rien.

5.3. ALGORITHME

Chaque processeur peut être dans un état parmi 4 :
- *actif* : il exécute l'application sous-jacente,
- *passif* : il n'exécute plus l'application sous-jacente, mais n'a pas détecté la terminaison,
- *blocage* : il a détecté la terminaison et est rentré dans la phase de terminaison propre de l'application sous-jacente,
- *fini* : il ne fait plus rien et les voies de communication adjacentes sont vides.

Il existe maintenant deux types de messages :
- les messages positifs : les messages de l'application sous-jacente,
- les messages négatifs, comportant un compteur des processeurs *passifs* :
  - si $-P \leq$ compteur ≤ -1, alors le message est un message de détection de la terminaison,
  - si compteur $\leq -P$, alors le message est un message de terminaison. (compteur $\geq -2P+1$).

L'algorithme de détection de la terminaison et de terminaison est le suivant.



```
Algorithme
|   état = actif
|   tant que (état ≠ détection) faire
|   |   réception (message)
|   |   choix état parmi :
|   |   |   état == actif :  choix message parmi
|   |   |                    |   message > 0 :   si impossible_générer_msg_appl alors
|   |   |                    |                   |   message = - 1
|   |   |                    |                   |   état = passif
|   |   |                    |                   sinon
|   |   |                    |                   |   générer_msg_appl(&message)
|   |   |                    |                   fin si
|   |   |                    |   message < 0 :   supprimer_message
|   |   |                    |                   si impossible_générer_msg_appl alors
|   |   |                    |                   |   message = - 1
|   |   |                    |                   |   état = passif
|   |   |                    |                   sinon
|   |   |                    |                   |   générer_msg_appl(&message)
|   |   |                    |                   fin si
|   |   |                    fin choix
|   |   |   état == passif : choix message parmi
|   |   |                    |   message > 0 :       message = -1
|   |   |                    |   message == -P :     état = blocage; message = message - 1
|   |   |                    |   -P < message ≤ 0 :  message = mesage - 1
|   |   |                    |   message < -P :      état = fini; message = message - 1
|   |   |                    fin choix
|   |   |   état == blocage :Si message < -P alors
|   |   |                    |   état = fini
|   |   |                    fin si
|   |   fin choix
|   |   envoyer (message)
|   fin tant que
fin algorithme
```

Figure 2.5. Algorithme de détection de la terminaison (chaque processeur a un compteur)

5.4. PREUVE DE CORRECTION DE L'ALGORITHME DE DÉTECTION DE LA TERMINAISON DISTRIBUÉE

Nous montrons les deux propriétés de vivacité et de sûreté.

5.4.1. La terminaison globale sera obligatoirement détectée. (Propriété de vivacité).

Démonstration :
Quand un processeur atteint la limite supérieure N, il envoie un message. Ce message contient un compteur négatif initialisé à -1. Il est envoyé au processeur suivant sur l'anneau :
- si le processeur qui le reçoit est actif, il détruit ce message, et le remplace par un message de l'application,
- s'il est déjà passif, il ajoute -1 au compteur et l'envoie au processeur suivant sur l'anneau.

Pendant l'exécution du programme, tous les processeurs lancent cette détection de terminaison une fois et une seule, au moment où ils deviennent passifs. Mais leur message peut être arrêté par un processeur actif.

A un instant précis, il ne reste plus qu'un seul processeur actif. Il atteindra la limite supérieure N et enverra alors -1. Ce compteur fera alors un tour sur l'anneau tout en étant régulièrement décrémenté par chaque processeur. Le compteur reviendra au processeur qui l'aura émis avec la valeur -P.

La terminaison distribuée sera par conséquent détectée de façon globale.

CQFD.

5.4.2. La terminaison globale n'est pas détectée si elle n'est pas intervenue. (Propriété de sûreté).

Démonstration :
La terminaison globale n'est détectée que si un processeur reçoit un compteur dont la valeur est inférieure à -P. Prouvons qu'une telle valeur ne peut pas se rencontrer tant que tous les processeurs n'ont pas terminé.

Quand un processeur termine localement, il envoie un compteur initialisé à -1. Ce compteur est envoyé au successeur sur l'anneau :
- si le processeur est *actif*, le compteur est éliminé,
- si le processeur n'est plus *actif*, il décrémente la valeur du compteur et l'envoie.

Il n'est pas possible qu'un processeur reçoive un compteur dont la valeur est inférieure ou égale à -P si un processeur est encore *actif*, car ce processeur aura détruit le compteur.

Donc, la terminaison globale ne peut pas être détectée si elle n'est pas intervenue.

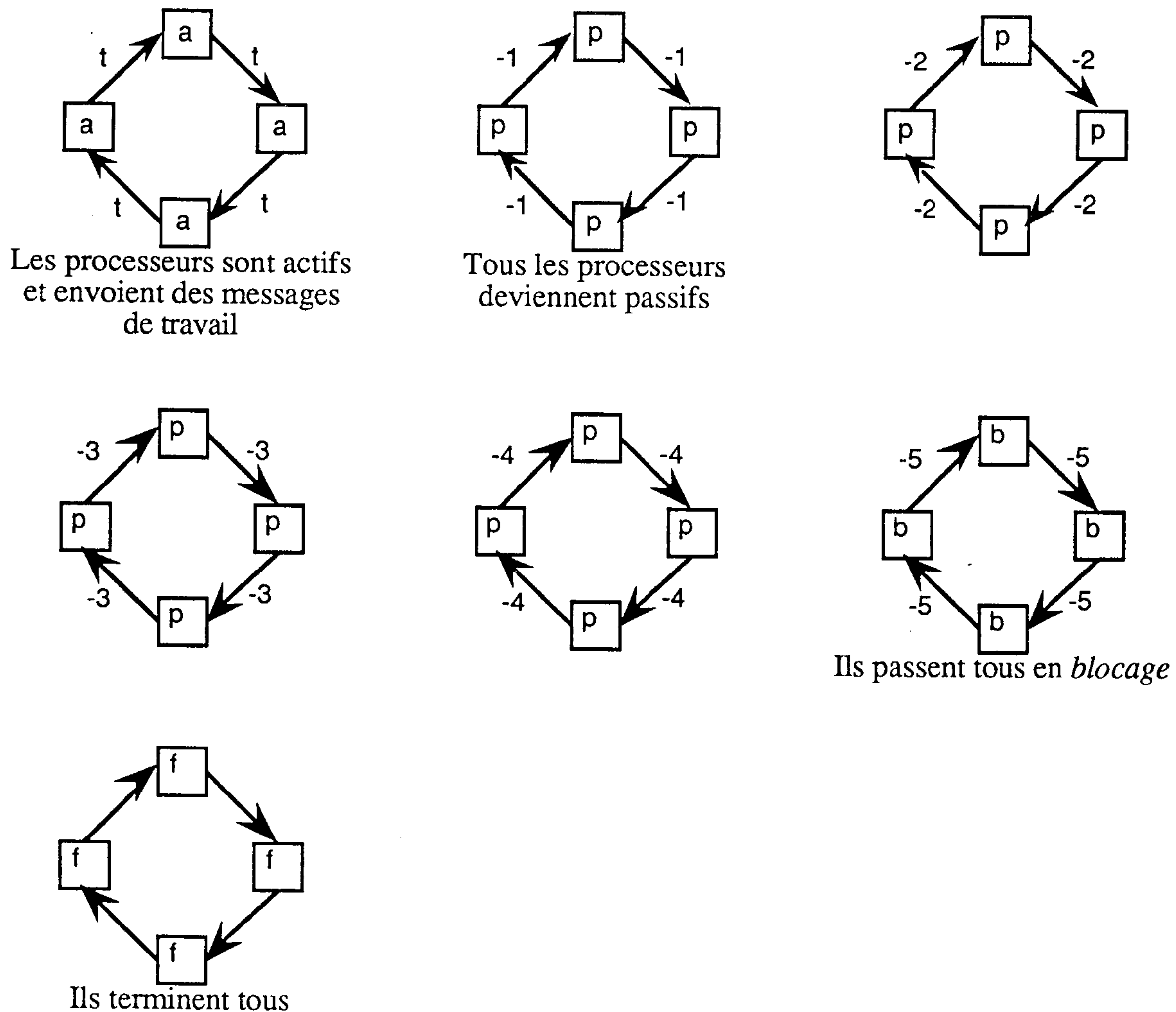
CQFD.

5.5. ÉVALUATION DU NOMBRE DE COMMUNICATIONS NÉCESSAIRES POUR DÉTECTER LA TERMINAISON, ET POUR TERMINER

5.5.1. Pire cas

Si tous les processeurs terminent en même temps, il faut P communications pour que les compteurs des jetons soient décrémentés jusqu'à -P. A cet instant-là, tous les processeurs détectent la terminaison. C'est le pire cas : il faut P communications pour détecter la terminaison. En effet, il n'y avait aucun autre compteur dont la valeur aurait pu être négative, auquel cas il aurait fallu moins de P communications. Puis il faut une communication pour passer à l'état *blocage* et une communication pour passer à l'état *fini*, car tous les messages sont les mêmes.

La figure 2.6 illustre ceci sur un anneau de 4 processeurs.



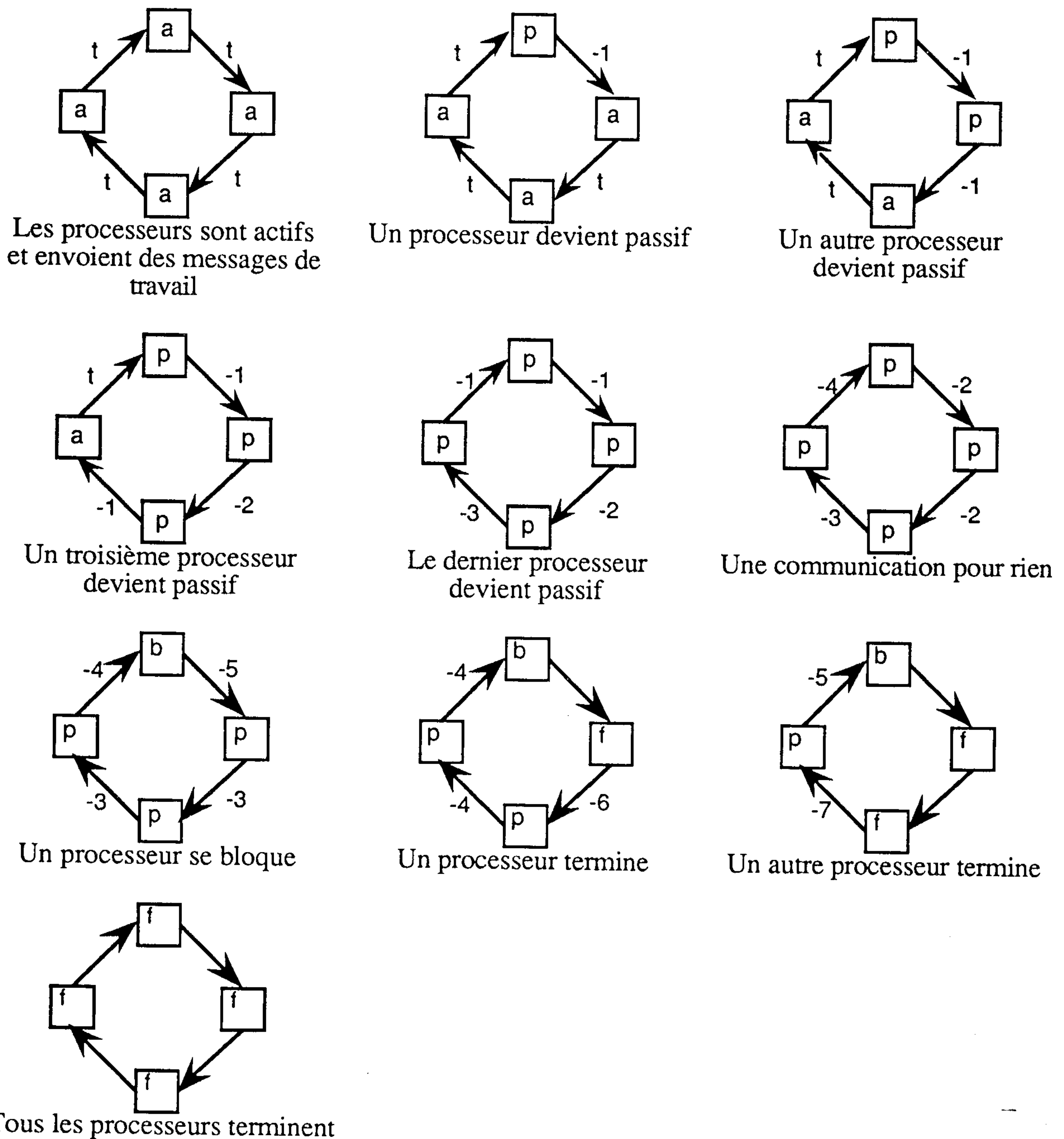
Légende :
- messages sur les voies de comunication :
  - t : message de travail de l'application sous-jacente
  - -x : valeur du compteur du message
- état des processeurs :
  - a : *actif*
  - p : *passif*
  - b : *blocage*
  - f : *fini.*

Figure 2.6. Chronogramme de détection de la terminaison et de terminaison (pire cas)

5.5.2. Cas général

Les processeurs terminent à différentes dates. Dans ce cas, considérons le dernier processeur à devenir passif. Son prédécesseur a reçu le jeton comptant les processeurs passifs. Le compteur vaut -P+1, puisqu'il y a P-1 processeurs passifs. Le dernier processeur devient passif et envoie -1. Il reçoit -P+1. A l'étape suivante, comme il est passif, il décrémente de 1 le compteur reçu et envoie donc un message dont le compteur vaut -P. Son successeur reçoit ce message et détecte la terminaison. Il faut donc deux communications pour détecter la terminaison. Puis il faut P-1 communications pour terminer, puisqu'un seul processeur a détecté la terminaison.

La figure 2.7 illustre de cas sur un anneau de 4 processeurs.



La légende est celle de la figure précédente.

Figure 2.7. Chronogramme de détection de la terminaison et de terminaison (pire cas)

Remarque :

Des cas intermédiaires sont possibles (deux processeurs ou plus deviennent passifs en même temps). Alors le nombre de communications nécessaires est intermédiaire entre le cas général et le pire cas.

5.6. CONCLUSION SUR L'ALGORITHME AVEC JETON VALUÉ

Cet algorithme est plus simple que le premier. Il ne génère pas de message de détection pour un processeur encore actif. De plus, et contrairement à l'algorithme précédent, il n'utilise pas les voies de communication de l'application sous-jacente, tant qu'un processeur est actif. Dans

l'algorithme précédent, un processeur devenant passif envoyait un message spécial qui effectuait un tour d'anneau et revenait au processeur émetteur. Ici, ce message n'existe pas, car les processeurs n'ont pas besoin de compter les processeurs passifs. Ce compteur (ou un minorant) est sur le jeton (ou les jetons).

Cet algorithme est symétrique : tout processeur peut détecter la terminaison (selon la valeur du compteur du message qu'il reçoit). Et même, plusieurs processeurs peuvent détecter la terminaison en même temps.

Enfin, il comprend aussi une étape de terminaison de l'application qui est tout aussi rapide.

Il est efficace sur un anneau.

6. CONCLUSION

Nous avons conçu deux algorithmes, chacun basé sur une technique propre, mais décomptant tous les deux le nombre de processeurs devenus passifs. Dans le premier, chaque processeur compte les processeurs passifs. Dans le deuxième, le compteur est placé sur un jeton valué qui parcourt l'anneau.

Le deuxième algorithme est plus simple, et plus efficace, car il ne génère pas de messages inutiles, interférant avec l'application sous-jacente (et la ralentissant), comme le fait le premier algorithme.

Ils sont tous deux symétriques, ce qui permet à n'importe quel processeur de détecter la terminaison. Et même, ces deux algorithmes autorisent plusieurs processeurs à détecter cette terminaison simultanément.

Nous leur ajoutons un algorithme de terminaison de l'application sous-jacente, pour que les processeurs et les voies de communications soient rendus dans un état cohérent.

Enfin, nous ne créons pas de nouveaux canaux physiques pour exécuter l'algorithme de détection de la terminaison.

En nous replaçant dans le contexte du chapitre 1 (génération de nombres premiers), nous donnons ici le temps nécessaire à la terminaison de l'application en fonction de la taille des paquets d'entiers qui sont transmis.

La figure 2.8 donne ces temps. La courbe est linéaire. Le temps augmente avec la taille des paquets, car le temps de traitement des paquets est important : il est séquentiel à l'intérieur de chaque paquet. De plus les messages d'un paquet n'ont pas tous la même valeur : ce peuvent être à la fois des messages de travail de l'application sous-jacente, et des messages de détection de la terminaison distribuée.

On peut interpréter ce cas comme une technique de détection de la terminaison et de terminaison d'un algorithme affectant plusieurs processus à chaque processeur. Et ces processus ne terminent pas tous en même temps. Donc la détection et la terminaison sont plus coûteuses, car chaque processeur doit gérer plusieurs processus.

Cependant, la terminaison prend 4 secondes. Rappelons que la génération des nombres premiers inférieurs à 20.10^6 prend 4 031 secondes. La terminaison coûte environ 1 ‰ du temps total. Elle est négligeable.

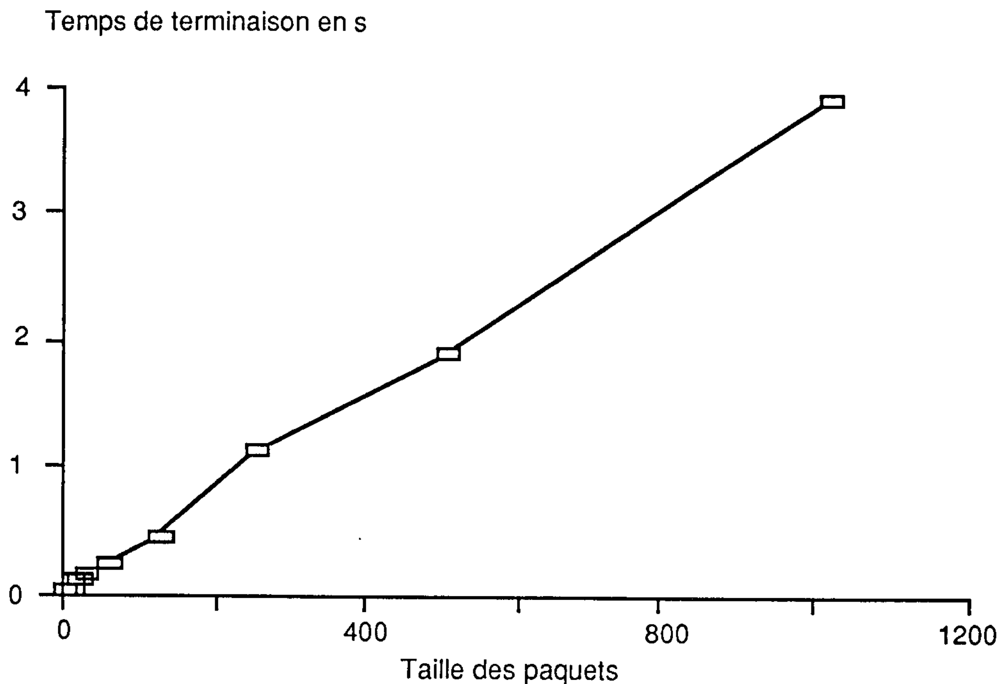


Figure 2.8 : Temps de terminaison en fonction de la taille des paquets

Il est envisageable d'étudier ces types d'algorithme de détection de la terminaison (et éventuellement de terminaison) sur d'autres topologies que l'anneau unidirectionnel. Il y a deux grandes familles d'algorithmes pour les topologies différentes de l'anneau : les algorithmes qui utilisent un arbre de recouvrement et font circuler des vagues de détection, et ceux qui construisent un anneau hamiltonien pour surveiller tous les processeurs et tous les canaux. Ces derniers permettent l'utilisation d'un jeton valué. En effet, on crée un anneau virtuel, donc les techniques présentées dans ce chapitre s'adaptent à leur topologie.

Il reste une restriction à lever pour obtenir un algorithme général. Dans nos hypothèses, nous avons supposé qu'un processeur passait de l'état actif à l'état passif une fois et une seule, et ne pouvait donc pas redevenir actif. Or, dans la littérature, le passage de l'état actif à l'état passif et de l'état passif à l'état actif sont possibles aussi souvent que l'algorithme le nécessite. Si on introduit cette hypothèse, il faut ajouter un mécanisme qui dise si un processeur est redevenu actif depuis la dernière fois que le jeton l'a vu passif. Et on se ramène alors à des algorithmes existants comme celui de Misra [Mis 83], par exemple.

Chapitre 3

A propos d'une méthodologie de parallélisation d'applications Maître / Esclaves

1. INTRODUCTION

L'algorithme séquentiel d'une application Maître / Esclaves est composé de deux parties : le travail du Maître et le travail des Esclaves. Lorsqu'on essaie de porter une telle application sur une machine parallèle à mémoire distribuée, en affectant un processeur au travail du Maître, et d'autres processeurs au travail des Esclaves, la structure de l'algorithme reste la même. Les différences proviennent de la nécessité de gérer les envois d'informations du Maître vers les Esclaves.

Le parcours exhaustif de ces données dans l'algorithme séquentiel est évident, car le Maître a accès à toutes les données. Mais dans un environnement parallèle à mémoire distribuée, les processeurs se partagent les données, et le Maître n'a accès qu'à un sous-ensemble de données.

Lorsqu'il doit accéder aux données disponibles sur un autre processeur, un Esclave, il faut que ce processeur Esclave devienne Maître. On suppose que l'application est telle que lorsque le Maître sur le processeur PE_i doit changer et passer sur le processeur PE_j, le processeur PE_i n'a plus de travail, et ne devient pas Esclave. Il est simplement retiré de l'ensemble des processeurs utiles. Le processeur PE_j devient le nouveau et seul Maître. Et le nombre d'Esclaves diminue.

Ainsi, on voit que porter une application séquentielle de ce type sur une machine multiprocesseur à mémoire distribuée peut poser des problèmes, dépendant de la topologie de la machine-cible, et concernant surtout la gestion des communications et de l'état Maître / Esclaves des processeurs.

Le crible d'Eratosthène est une application de type Maître / Esclaves. Nous l'avons étudié dans le chapitre 1 et nous l'avons implanté. A partir de cette implantation, nous remarquons que les modifications à apporter à l'algorithme séquentiel concernent la gestion des communications et de l'état des processeurs, dès l'instant où les données sont réparties entre les processeurs.

Pour ce type d'applications, nous proposons de mettre en œuvre une méthodologie pour l'implantation parallèle sur machines multiprocesseurs à mémoire distribuée. Cette méthodologie doit conduire à un algorithme de prise en charge des communications et des changements d'état des processeurs, indépendamment de l'application sous-jacente.

Or la gestion des communications est différente selon les topologies. C'est pourquoi nous allons étudier la méthodologie d'implantation parallèle sur un réseau linéaire, puis sur une grille non torique à deux dimensions.

Les études de parallélisation ou d'aide à l'implantation parallèle tendent à proposer des compilateurs, des langages, des simulateurs, mais peu ou pas d'outils adaptés à certains types d'algorithmes. C'est ce que nous montrons dans le paragraphe 2, en donnant un état de l'art des outils d'aide à la parallélisation.

Le paragraphe 3 est consacré à la dérivation et la mise en œuvre d'une méthodologie d'implantation d'applications de type Maître/Esclaves sur un réseau linéaire, car cette topologie est simple. A partir des programmes réalisés pour les applications du chapitre 1, nous dérivons une méthodologie d'aide à l'implantation de ces algorithmes sur un réseau linéaire. L'utilisateur doit choisir la structure des messages transmis du Maître vers les Esclaves. Il doit aussi déterminer la répartition des données sur les processeurs. La méthodologie est concrétisée par un méta-algorithme capable de gérer les communications et les changements d'état des processeurs. En comparant les temps d'exécution du crible d'Eratosthène ainsi géré, et ceux donnés au chapitre 1, sur le même multiprocesseur FPS T20, on constate que l'ajout d'une couche logicielle (gestion des communications et de l'état des processeurs) augmente de beaucoup le temps d'exécution.

Nous essayons de voir si le type de topologie intervient dans le coût de gestion. Et dans le paragraphe 4, nous étudions une méthodologie d'implantation de ces algorithmes sur une grille non torique. Nous définissons une stratégie de choix du Maître suivant. Mais cette stratégie et les communications plus complexes alourdissent la gestion de l'algorithme. Les performances sont moins bonnes.

Nous envisageons enfin d'utiliser un arbre de recouvrement de l'hypercube, de manière à mieux utiliser les canaux physiques disponibles sur l'hypercube.

2. ÉTAT DE L'ART

Sur les multiprocesseurs à mémoire partagée, certains compilateurs proposent la parallélisation automatique de programmes (Fortran). Les programmeurs pensent et écrivent leurs programmes de manière séquentielle et le compilateur essaie de reconnaître des boucles parallélisables.

Mais, pour les multiprocesseurs à mémoire distribuée, il n'existe pas de compilateur-paralléliseur actuellement disponible. En effet, de nombreux paramètres interviennent : l'allocation des tâches, la répartition de la charge de travail, les communications, l'efficacité, le partage de variables entre plusieurs processeurs...

Plusieurs approches sont envisagées pour aider le programmeur à tirer le meilleur parti de sa machine. Et de nombreuses équipes de recherche (du domaine public ou du domaine privé) travaillent dans ce sens :
- environnements adaptés au parallélisme,
- langages spécialisés et logiciels de simulation,
- langages parallèles,
- outils d'aide à l'analyse.

D'autres prennent du recul et tentent une analyse théorique et philosophique des problèmes liés à la programmation parallèle.

2.1. ENVIRONNEMENTS LIÉS AU PARALLÉLISME

Purtilo et al. [PRG 88] proposent un système de conception d'environnement où la spécification de conception est séparée de l'implantation. Car il est vrai que la conception d'algorithmes et

d'applications parallèles est fortement conditionnée, et par là même biaisée, par la machine-cible, c'est-à-dire les contraintes de l'implantation.

Cameron et al. [CCG 88] présentent le projet IC*. Il intègre un environnement pour la conception, la spécification et le développement de systèmes complexes. Un système est alors caractérisé par un ensemble d'expressions invariantes qui décrivent son comportement temporel. Les caractéristiques novatrices de ce système comprennent les contraintes temporelles et structurelles, le parallélisme inhérent, la modélisation explicite du temps, l'évolution non déterministe et l'activation dynamique.

Ces environnements sont trop théoriques. C'est pourquoi d'autres chercheurs ont orienté leurs travaux vers des langages spécialisés plus pragmatiques ou des logiciels de simulation permettant de mettre au point, plus facilement, les programmes ou les prototypes.

2.2. LANGAGES SPÉCIALISÉS ET LOGICIELS DE SIMULATION

Ainsi, C. Jard et al. [JMG 88] ont développé Véda, un outil logiciel, en vue d'aider les concepteurs dans la modélisation et la validation de protocoles. Véda utilise la description d'Estelle. Il est orienté vers le prototypage rapide d'algorithmes distribués. Véda permet la trace pour l'observation de l'exécution de programmes distribués. Il se situe donc au niveau de la simulation.

Toujours pour permettre la simulation, mais de manière plus conviviale, Giacalone et Smolka [GiS 88] offrent un outil comportant des possibilités de visualisation graphique. Mais leur logiciel semble peu souple, car trop intégré.

L'expérimentation sur une machine parallèle distribuée contribue à valider les algorithmes parallèles dans un environnement réel, en explorant le plus grand nombre de leurs comportements possibles. Sans pour autant aller jusqu'à l'implantation, l'expérimentation permet de remplacer des variables de la simulation par des constantes d'une machine parallèle. Les paramètres sont réels, et ne sont plus biaisés par l'interprétation du simulateur. L'algorithme ainsi validé peut être porté sur une autre machine, et seuls quelques paramètres seront modifiés. De cette manière, on tire parti de la puissance de calcul de la machine parallèle dans des conditions réelles.

C'est ce que proposent C. Jard et J.M. Jézéquel [JaJ 88] avec leur compilateur Estelle, qu'ils ont d'ailleurs porté sur le FPS T20.

De la même manière, Schwan et al. [SRV 88] proposent un langage, intégré dans un système, qui permet au programmeur de réaliser des expérimentations de programmes à un niveau d'abstraction très élevé.

D'autres simulateurs existent. Citons par exemple SIGLE de F. André et A. Joubert [AnJ 87].

2.3. LANGAGES PARALLÈLES

Mais c'est surtout au niveau des langages parallèles que des efforts énormes sont actuellement faits.

Certains langages ont été créés spécialement pour le parallélisme, comme OCCAM [Hoa 88], LC3 [Lec 85] ou Durra [BaW 86], lequel permet le développement d'applications logicielles parallèles à gros grain, grâce à une description de l'interaction entre les processus à un niveau logique uniquement. C'est sans doute OCCAM qui tire le mieux son épingle du jeu avec le succès commercial des Transputers, microprocesseurs pour lesquels il est conçu.

Il existe d'autres langages parallèles, issus de langages séquentiels, les C et Fortran parallèles, par exemple. Ils intègrent des primitives spéciales de gestion des communications par exemple. Mais, à l'opposé d'OCCAM, ils ne permettent pas toujours une exécution parallèle de plusieurs

processus sur le même processeur, comme un recouvrement de l'exécution des calculs et des communications sur le Transputer.

2.4. OUTILS D'AIDE À L'ANALYSE

Les programmeurs parallèles peuvent aussi obtenir de l'aide pour choisir les structures de données adaptées à leur problème [SBB 87], pour utiliser des architectures particulières [Sal 87] ou même des ordinateurs parallèles donnés [GaS 87]...

Il est tout aussi important d'effectuer des analyses indépendantes des machines-cibles. Adam et al. [AIR 88] partent de l'idée que la conception de certains algorithmes distribués et parallèles est fondée sur le concept de synchronisme. Les auteurs examinent les problèmes de synchronisation posés par la mise en œuvre des synchroniseurs. Certaines solutions sont examinées. Ils en proposent des implantations sur l'hypercube d'Intel.

Kruatachue et Lewis [KrL 88] proposent une technique automatique de détermination de la meilleure taille des tâches élémentaires pour un programme donné. Pour ce faire, le programme doit être présenté sous la forme d'un graphe de tâches et des communications entre les tâches. Le graphe est orienté acyclique. Les nœuds sont pondérés en fonction du temps d'exécution de la tâche (qui représente d'ailleurs la taille de la tâche). Les arcs sont pondérés par les temps de communication.

La technique proposée consiste à regrouper les tâches en entités plus importantes à affecter aux processeurs. Ce principe est intéressant car les paramètres de la machine-cible sont sous-jacents par l'intermédiaire des pondérations des nœuds et des arcs (temps de calcul et de communication). Cependant, une analyse préliminaire importante doit être effectuée pour mettre le programme (et son exécution) sous forme d'un graphe acyclique.

Ceci est statique, alors que Millot et Vautherin [MiV 88] permettent à des processus, lorsqu'un Transputer a atteint sa charge de travail maximale, de migrer sur un Transputer voisin, et ceci de manière automatique. C'est de l'allocation dynamique de tâches. Cette technique est très puissante, mais chaque Transputer doit connaître une topologie de migration des processus, et surtout, il doit avoir, résident dans sa mémoire et s'exécutant en continu, un processus d'attente de réception d'un processus migrant, ainsi qu'un processus de gestion des ressources, de manière à faire migrer lui aussi des processus.

2.5. LES PROBLÈMES THÉORIQUES ET PHILOSOPHIQUES

Enfin, au-delà de toutes les recherches même théoriques, Kai Hwang [Hwa 87] étudie les problèmes que pose l'utilisation de supercalculateurs :
• choix architecturaux,
• langages parallèles,
• techniques de compilation,
• gestion des ressources,
• contrôle de l'exécution simultanée de plusieurs processus,
• environnements de programmation,
• algorithmes parallèles,
• méthodes d'amélioration des performances...

Gelernter [Gel 86], encore plus philosophiquement, se demande si les chercheurs travaillant sur les langages de programmation parallèles pourront rendre accessible aux programmeurs la puissance qu'offre le parallélisme. En effet, les multiprocesseurs sont des ordinateurs proposant des puissances maximales élevées. Mais il est difficile d'obtenir même une petite partie de cette puissance pour des applications réelles. Ceci ne provient certes pas que de la machine, le programmeur a une certaine responsabilité dans les performances minimes obtenues. Mais, avec un environnement convivial et des outils adaptés, il pourrait certainement atteindre des résultats meilleurs...

En plus de ces différents outils, nous pensons qu'il est nécessaire de proposer des outils qui déchargent le programmeur des particularités d'implantation liées au parallélisme distribué des applications de type Maître / Esclaves. Dans son algorithme séquentiel, il définit la (ou les) tâche(s) du Maître et celle(s) des Esclaves. Lors du portage sur une machine parallèle distribuée, les tâches restent les mêmes, mais elles interagissent via des variables que les processus doivent se communiquer explicitement, alors que dans l'exécution séquentielle, cette partie était cachée car implicite.

Notre expérience et l'analyse que nous en avons faite nous permettent d'affirmer que même les applications les plus simples (soit-disant) à paralléliser (parce qu'intrinsèquement parallèles) posent des problèmes. En effet, il faut gérer des communications, des synchronisations, des structures de données pour les messages, l'état des processus, ...

C'est pourquoi nous avons développé un outil d'aide à la gestion du parallélisme dans des applications de type Maître / Esclaves, sur des topologies de réseau linéaire et de grille à 2 dimensions.

De nombreux algorithmes utilisent des interactions Maître / Esclaves entre les processeurs, tels la recherche du chemin critique en algèbre linéaire, le parcours de graphes, des problèmes arithmétiques, [ChM 88] ... L'algorithme général est toujours du même type, même si le Maître peut changer lorsqu'il n'a plus de travail à donner aux Esclaves, et donc, passer la main à un de ses successeurs.

3. MÉTA-ALGORITHME DE PARALLÉLISATION D'APPLICATIONS MAITRE / ESCLAVES SUR UN RÉSEAU LINÉAIRE

Soient P processeurs connectés comme sur la figure 3.1 (réseau linéaire) :

Figure 3.1. P processeurs connectés en réseau linéaire

Initialement, pour une application de type Maître / Esclaves, le Maître est le processeur PE_0, les Esclaves sont les processeurs PE_1 à PE_{P-1}. Pour ces derniers, le traitement est toujours le même, les communications se faisant en pipeline.

Lorsque le processeur PE_0 n'a plus de travail à envoyer aux autres processeurs, il s'arrête, et son successeur devient le Maître d'un nouveau réseau linéaire restreint. Ceci se perpétue jusqu'à ce que le dernier processeur devienne le Maître, termine son travail, et ne puisse pas passer son état de Maître à un autre processeur.

Lors de l'exécution, des messages sont communiqués entre les processeurs, mais toujours dans la même direction, du Maître vers les Esclaves. De plus, les processeurs changent d'état. Le Maître deviendra passif. Les autres processeurs seront les Esclaves, puis deviendront tour à tour Maître pour finalement terminer et être passifs.

Notre but est de définir un méta-algorithme qui gère ces connaissances et ces changements d'état. Alors le programmeur n'aurait qu'à fournir les conditions de transition de l'automate de changement d'état pour les processeurs, le corps des tâches du Maître et des Esclaves, et la structure de données des informations à communiquer du Maître vers les Esclaves.

3.1. LE RÉSEAU LINÉAIRE

Un réseau linéaire est un ensemble de processeurs connectés suivant une seule dimension, en une séquence acyclique. Chaque processeur, sauf les deux extrémités, est connecté à deux voisins. Ce réseau linéaire est unidirectionnel. Les extrémités sont respectivement appelées la *tête* et la *queue* du réseau linéaire. Les autres processeurs sont appelés les processeurs-*milieu.*

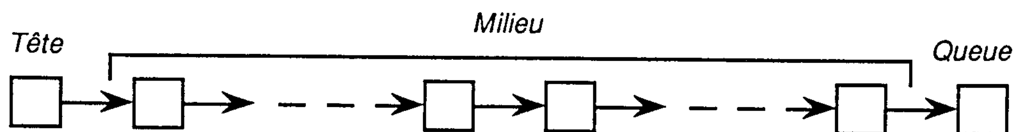


Figure 3.2. Topologie de réseau linéaire.

Si l'application n'a pas besoin de tous les processeurs disponibles, les processeurs inutilisés sont groupés après la *queue* et sont *passifs.*

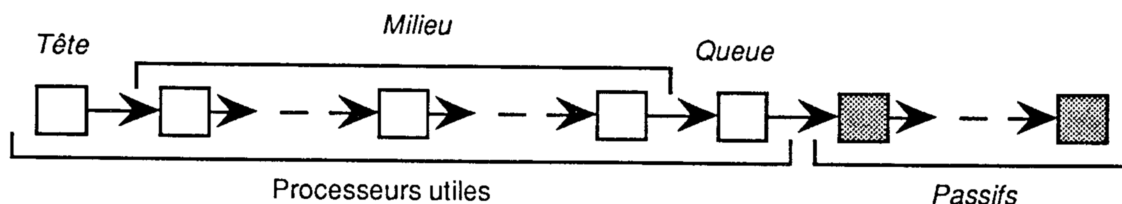


Figure 3.3. Utilisation d'un sous-ensemble des processeurs du réseau linéaire.

Outre les états *tête*, *milieu*, *queue* et *passif*, un processeur peut se trouver dans l'état *tête+queue*, lorsque le réseau linéaire est réduit à un seul processeur. Le processeur PE_0 *tête* est le Maître. Les autres processeurs PE_1 à PE_{P-1} (non *passifs*) sont les Esclaves.

3.2. LE PROBLÈME

Supposons une application de type Maître / Esclaves nécessitant P processeurs, et s'exécutant sur un réseau linéaire dans les conditions suivantes :

- Au début, le processeur PE_0 est le Maître, et les processeurs PE_1 à PE_{P-1} sont les Esclaves.
- Le processeur PE_0 calcule des informations que tous les Esclaves attendent pour travailler.
- Dès que le processeur PE_0 a envoyé ces informations à son successeur sur le réseau linéaire (le processeur PE_1), il commence à travailler en utilisant ces informations.
- Son successeur (le processeur PE_1) reçoit les informations, en transmet une copie à son successeur et commence à travailler.
- Et ainsi de suite jusqu'à ce que les informations atteignent la *queue*.
- Lorsque la *tête* (le Maître) a terminé de travailler, elle effectue d'autres calculs et obtient d'autres informations qu'elle transmet, via son successeur, à tous les Esclaves.
- A une certaine date, le Maître ne trouve plus d'informations. Il devient *passif* après avoir notifié à son successeur de devenir le Maître.
- Et ainsi de suite jusqu'à ce que la *queue* devienne Maître puis termine à son tour.

La figure 3.5 montre le chronogramme de ces opérations sur un réseau linéaire. La première partie (dans le temps) consiste à initialiser le pipeline du réseau linéaire. Lorsqu'il est plein, il travaille en régime stationnaire et reste continuellement plein.

Lorsque la *tête* ne trouve pas de nouvelles informations, elle envoie un message spécial à son successeur et devient *passive*. Le successeur prend la *tête* du réseau (figures 3.4 et 3.5).

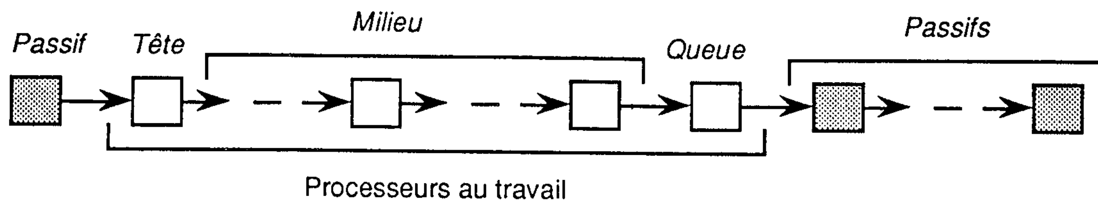


Figure 3.4. Modification dynamique de l'état des processeurs

Le message spécial envoyé par la *tête* qui devient *passive* à l'esclave qui devient la nouvelle *tête* est purement local. Il n'est pas transmis aux autres esclaves.

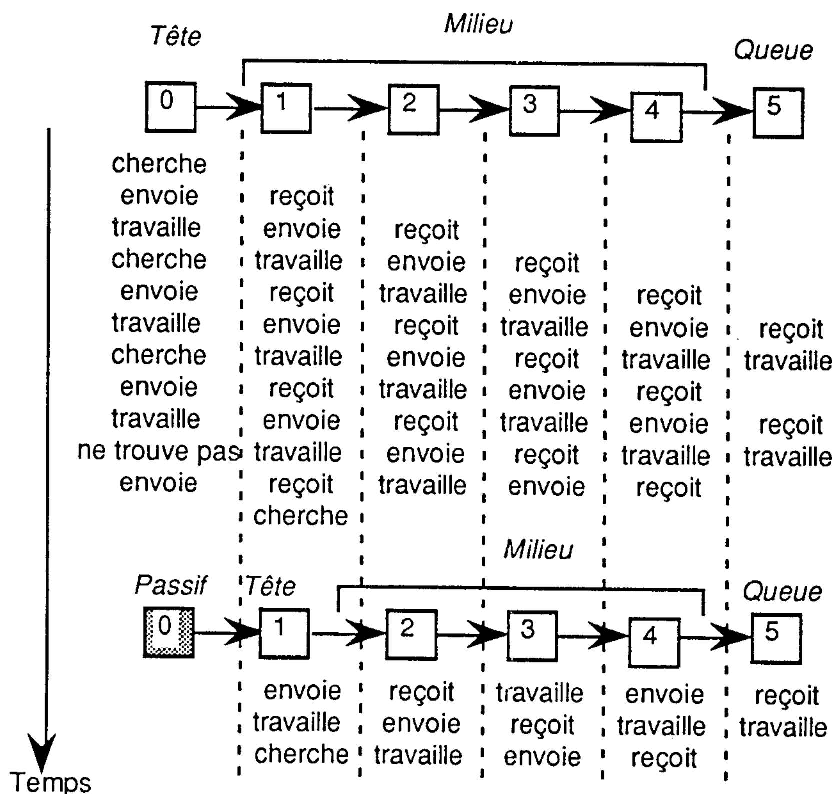


Figure 3.5. Chronogramme des opérations sur un réseau linéaire.

Ainsi, il suffit à un processeur de connaître ses deux voisins et son propre état. Il n'a besoin d'aucune connaissance globale, sauf éventuellement, le nombre total de processeurs, mais cette information n'est pas nécessaire pour toutes les applications.

Il nous faut construire un méta-algorithme qui gère toutes ces opérations. Donc, le méta-algorithme doit s'occuper de gérer les communications, de modifier l'état des processeurs et faire se terminer l'exécution lorsque les calculs sous-jacents de l'application sont effectués.

Nous allons étudier maintenant la structure des données à communiquer pour le contrôle de l'exécution par le méta-algorithme.

3.3. LA STRUCTURE DES DONNÉES À COMMUNIQUER

Les données de contrôle à communiquer sont simplement les messages de notification de changement de *tête* (ou de Maître) entre deux processeurs voisins. Cette donnée de contrôle doit être indépendante de l'application.

Si l'application n'utilise, pour ses messages de travail, qu'un sous-ensemble de l'ensemble des messages possibles, il faut coder le message de contrôle comme un message détectable par le méta-algorithme, c'est-à-dire comportant des données qui ne peuvent pas être destinés à l'application sous-jacente. De cette manière, il n'y a pas d'augmentation du nombre de messages.

Si l'application ne permet pas de définir un message semblable aux messages de travail de l'application (de même structure, de même taille, ...) il faut ajouter un en-tête à tous les messages, qui spécifiera s'il s'agit d'un message de travail ou d'un message de contrôle. Théoriquement, cet en-tête peut être un booléen. Pratiquement, il ne peut pas être moins important qu'un octet ou un entier suivant la structure du message de travail, ceci sur le FPS, à cause des langages de programmation utilisés.

Mais on peut envisager aussi d'accoler à un message de travail normal, l'information de contrôle, uniquement lorsqu'elle est nécessaire. Le coût supplémentaire est alors inférieur à celui de la solution avec en-tête. Mais sur le FPS T20, le récepteur doit connaître la taille du message envoyé. Donc il n'est pas possible d'envoyer un message de taille et de structure différentes de celles des messages habituels, et ceci, uniquement de temps en temps, lorsque le Maître change de processeur.
Sur le FPS T20, nous devons choisir la première solution (coût de communication supplémentaire dû au méta-algorithme = 0) lorsque cela est possible. Sinon, la seconde solution. Son coût dépend de la taille de l'en-tête et de la taille du message de travail. Envoyer un message de q octets entre deux voisins coûte $\beta + \mu q$.

Donc si l'en-tête occupe r octets, envoyer le même message de travail avec cet en-tête coûte $\beta + \mu(q+r)$. Le coût supplémentaire est donc μr. Le coût absolu et le coût relatif sont donnés dans le tableau suivant, avec $\beta = 860$ µs et $\mu = 1,44$ µs/octet.

Taille du message de travail (q)	Coût absolu supplémentaire (µr)		Coût relatif supplémentaire (µr / (β + µq))	
	r = 1 octet	r = 4 octets	r = 1 octet	r = 4 octets
q = 1 octet	1,44 µs	5,76µs	0,167 %	0,669 %
q = 4 octets	1,44 µs	5,76µs	0,166 %	0,665 %
q = 1 024 octets	1,44 µs	5,76µs	0,057 %	0,227 %

Figure 3.6. Coûts absolus et relatifs de la communication d'un message avec en-tête.

Le tableau 3.6 montre clairement que le coût supplémentaire induit par l'en-tête est négligeable, à cause de la valeur très grande de β.

Désormais, nous ne spécifierons plus le principe utilisé pour communiquer un message de contrôle : c'est la moins chère des solutions possibles, avec une priorité à la solution qui code un message de contrôle dans un sous-ensemble des messages de travail, et n'augmente pas le coût des communications.

3.4. LE MÉTA-ALGORITHME

Définissons une variable appelée 'message', dont la structure est celle d'une donnée de l'application sous-jacente à communiquer (voir § 2.3). Elle représente les informations de travail et de contrôle qui sont envoyées par le Maître.

Définissons les procédures de l'application sous-jacente :
- 'initialisation' : construit le réseau linéaire ; l'utilisateur donne le nombre de processeurs requis et les variables globales de l'application ;

- 'cherche' : cette procédure, exécutée seulement par la *tête* du réseau est le programme du Maître. Son but est de localement chercher une information. Cette information ('message') est gérée ensuite par le méta-algorithme qui l'envoie à tous les Esclaves s'il s'agit d'un message de travail, ou qui la garde pour le processeur suivant seulement, s'il s'agit d'un message de contrôle pour le changement de tête ;
- 'traitement_général' : tous les processeurs non *passifs* exécutent cette procédure, le Maître aussi éventuellement ;
- 'traitement_final' : impression, par exemple ;
- 'automate' : cette fonction gère l'état des processeurs en fonction de la valeur de 'message' (message de travail ou de contrôle) et leur état actuel.

Les communications nécessitent une procédure 'envoi' d'envoi d'une variable 'message' au successeur, et de réception 'réception' de 'message' en provenance du prédécesseur.

L'algorithme est donné en pseudo-langage C de manière à visualiser les procédures qui ont le droit de modifier la variable 'message'.

```
Méta-algorithme
|   initialisation (&message);        /* Construit le réseau linéaire et acquiert les paramètres */
|   tant que (état ≠ passif) faire
|   |   Si (état == tête) alors
|   |   |   cherche (&message);
|   |   |   envoi (message);
|   |   |   état = automate (message);                         /* tête ou passif */
|   |   sinon
|   |   |   Si (état == milieu) alors
|   |   |   |   réception (message);
|   |   |   |   état = automate (message);                     /* milieu ou tête */
|   |   |   |   Si (état == tête) alors
|   |   |   |   |   cherche (&message);
|   |   |   |   |   état = automate (message);                 /* tête ou passif */
|   |   |   |   fin si
|   |   |   |   envoi (message);
|   |   |   sinon
|   |   |   |   Si (état == queue) alors
|   |   |   |   |   réception (message);
|   |   |   |   fin si
|   |   |   |   état = automate (message);                     /* queue ou tête+queue */
|   |   |   |   Si (état == tête+queue) alors
|   |   |   |   |   cherche (&message);
|   |   |   |   |   état = automate (message);       /* état = tête+queue ou passif */
|   |   |   |   fin si
|   |   |   fin si
|   |   fin si
|   |   Si (état ≠ passif) alors
|   |   |   traitement_général (message);
|   |   fin si
|   fin tant que
|   traitement_final ();
fin méta-algorithme
```

Figure 3.7. Méta-algorithme pour réseau linéaire.

Plaçons-nous maintenant du point de vue utilisateur. Que faut-il faire pour utiliser ce méta-algorithme sur une application de type Maître / Esclaves ?

L'algorithme de l'application doit comporter deux parties :
- le corps du programme du Maître qui consiste à extraire séquentiellement de sa mémoire locale des informations nécessaires au travail des Esclaves ;
- le corps du programme des Esclaves qui consiste à utiliser l'information envoyée par le Maître pour modifier les données que traitent les Esclaves.

Le programme du Maître correspond à la procédure 'cherche', et celui des Esclaves à la procédure 'traitement_général', de l'algorithme précédent. Ces deux procédures portent obligatoirement ce nom, et sont stockées dans un fichier séparé qui est compilé en même temps que le méta-algorithme.

De plus, il faut savoir comment les données à traiter sont réparties entre les processeurs. Il faut que le premier processeur du réseau traite les premiers éléments, le processeur suivant les éléments suivants, ... Ceci est effectué dans la procédure 'initialisation', compilé avec le méta-algorithme.

Enfin, l'utilisateur doit déterminer la structure de données des informations transmises du Maître aux Esclaves. Cette étape consiste à définir les types des informations communiquées et leur nombre d'octets. Ces caractéristiques sont elles aussi stockées dans un fichier séparé, compilé avec le méta-algorithme. La taille (nombre d'octets) du message est nécessaire pour effectuer une communication sur le FPS T20.

Ainsi, le code n'est pratiquement pas modifié.

3.5. APPLICATIONS

Le méta-algorithme est implanté sur l'hypercube FPS T20, duquel on extrait une sous-topologie de réseau linéaire (voir figure 3.8).

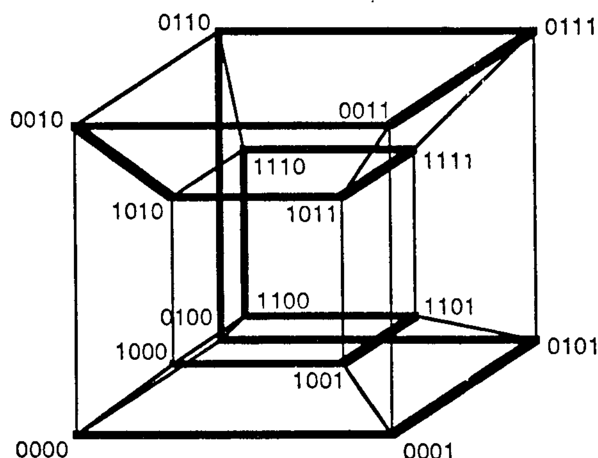


Figure 3.8. Un réseau linéaire de 16 processeurs dans un hypercube de dimension 4.

Les applications visées sont du type Maître / Esclaves. Le Maître, une fois qu'il a épuisé les informations qu'il possède, devient passif. C'est-à-dire qu'on ne peut pas traiter une application pour laquelle le Maître devient Esclave et attend des informations pour traiter les éléments qu'il possède.

Ainsi, nous proposons d'implanter le crible d'Eratosthène. Mais il n'est pas possible d'implanter la version où les entiers sont répartis par sous-suites, puisqu'un processeur a toujours des éléments à modifier même s'il a déjà été Maître. Car il doit redevenir Esclave après chaque passage à l'état Maître. Ceci provient de la répartition des entiers par sous-suites. En revanche, nous pouvons implanter la version de base du crible dans laquelle les petits entiers sont sur le premier processeur, puis les entiers un peu plus grands sont sur le processeur

suivant ... Ainsi, un processeur Maître qui devient passif n'a plus de multiples à supprimer car les nombres premiers trouvés par les autres processeurs sont plus grands que les entiers qu'il possède.

A partir des nombres premiers ainsi générés, nous pouvons envisager une application en théorie des nombres : le calcul de $\Omega(k)$ qui donne dans la décomposition de k en un produit de facteurs premiers, la multiplicité des facteurs premiers. De la même manière que précédemment, les entiers sont en ordre croissant sur les processeurs. Et cette application convient. Cette fonction permet de calculer les valeurs de N(n, k), le nombre d'entiers inférieurs à n ayant exactement k facteurs premiers. Cette application répond elle aussi aux normes. Nous l'implantons de la même façon que les deux applications précédentes.

Les messages de travail ne contiennent que des entiers positifs. Donc les messages de contrôle seront codés sur des entiers négatifs. Ce qui implique un coût supplémentaire nul pour les communications, car il n'y a pas besoin d'en-tête ou de message spécial.

3.5.1. Le crible d'Eratosthène

Pour une présentation de l'algorithme, prière de se reporter au chapitre 1.

Chaque processeur dispose d'un sous-intervalle de crible. L'intervalle global d'entiers [1, ..., N] est distribué entre les P processeurs PE_i, $0 \leq i \leq P-1$. Le processeur i traite l'intervalle

$$\left[i\frac{N}{P} + 1, (i+1)\frac{N}{P}\right].$$

Le processeur PE_0 est la *tête*, le Maître. Il génère les nombres premiers en ordre croissant. Les autres processeurs sont les Esclaves. Dès réception d'un nombre premier, un Esclave le transmet à son successeur et supprime les multiples de cet entier dans son sous-intervalle. Remarquons que le Maître aussi crible son sous-intervalle. Si on veut équilibrer les charges, il faut diminuer la taille de l'intervalle de crible du Maître. Mais la recherche du prochain nombre premier est d'un coût dérisoire.

La programmation du crible d'Eratosthène sur le réseau linéaire piloté par le méta-algorithme donné consiste à définir les procédures de l'application et la structure des données de communication :

- 'initialisation' : chaque processeur définit les bornes inférieure et supérieure du sous-intervalle qu'il gère, et 2 est le premier nombre premier ;

```
procédure initialisation
|       borne_inf = i*N/P + 1
|       borne_sup = (i+1)*N/P
fin procédure
```

- 'cherche' : la *tête* cherche le prochain nombre premier ; c'est le premier entier suivant non éliminé. 'message' prend cette valeur. Si la *tête* ne le trouve pas dans son sous-intervalle, 'message' prend une valeur négative, car il faut changer de Maître ;

```
procédure cherche
|       répéter
|       |       nombre_premier = nombre_premier + 1          /* initialisé à 2 */
|       jusqu'à (nombre_premier non éliminé ou nombre_premier > borne_sup)
|       si (nombre_premier > borne_sup) alors
|       |       message = -1
|       sinon
|       |       message = nombre_premier
|       fin si
fin procédure
```

- 'traitement_général' : tous les processeurs non *passifs* ont reçu un nombre premier. Ils éliminent ses multiples dans leur sous-intervalle ;

```
procédure traitement_général
|       si (état ≠ passif) alors
|       |       nombre_composé =  plus petit multiple de nombre_premier, différent
|       |                         de 2*nombre_premier, et supérieur à borne_inf
|       |       tant que (nombre_composé ≤ borne_sup) faire
|       |       |       éliminer (nombre_composé)
|       |       |       nombre_composé = nombre_composé + nombre_premier
|       |       fin tant que
|       fin si
fin procédure
```

- 'traitement_final' : impression des nombres premiers dans l'ordre croissant. Pour ce faire, nous utilisons ce méta-algorithme où 'cherche' consiste à imprimer le prochain nombre premier, où 'traitement_général' est vide, et où 'message' négatif signifie que le processeur *tête* a imprimé tous ses nombres, et que c'est au successeur d'imprimer ses propres nombres premiers ;

- 'automate' : si 'message' est positif, l'état de ce processeur n'est pas modifié. Or chaque processeur n'a le choix qu'entre deux états à chaque instant. Donc si 'message' est négatif, l'état prend pour valeur l'autre état possible (voir méta-algorithme, figure 3.7).

```
fonction automate (message) : état
|       si (message < 0) alors
|       |       choix état parmi
|       |       |       état == tête :          retourner (passif)
|       |       |       état == milieu :        retourner (tête)
|       |       |       état == queue :         retourner (tête+queue)
|       |       |       état == tête+queue :    retourner (passif)
|       |       fin choix
|       fin si
fin fonction
```

Voici les temps d'exécution pour différentes valeurs de N :

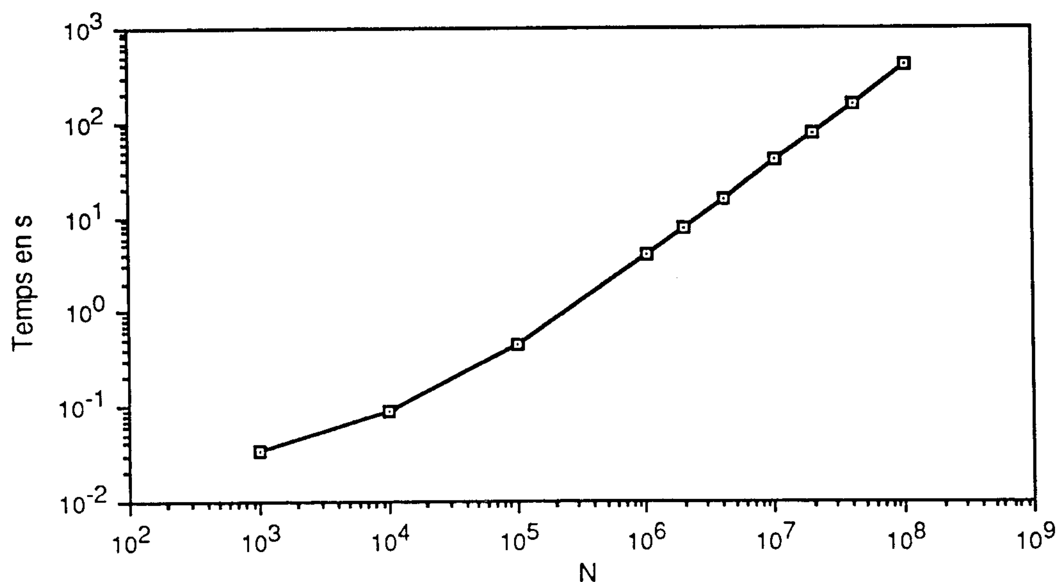


Figure 3.9. Temps d'exécution du crible d'Eratosthène sur un réseau linéaire de 16 processeurs

Notre implantation sur le FPS utilise un intervalle de 6 400 000 entiers (800 Koctets) sur chaque processeur. Donc 16 processeurs permettent de cribler pour toute valeur de N inférieure à 102 400 000. La figure 3.9 donne le temps de recherche de tous les nombres premiers inférieurs à N, avec N compris entre 10^3 et 10^8.

Pour de petites valeurs de N, le temps d'exécution est important à cause du contrôle.

Faisons s'exécuter l'algorithme sur un processeur, avec la valeur de N maximum, N = 6 400 000. Puis cherchons les nombres premiers dans le même intervalle avec différents nombres de processeurs (figure 3.10). Ainsi nous pourrons étudier l'accélération produite par l'exécution du même problème avec différents nombres de processeurs.

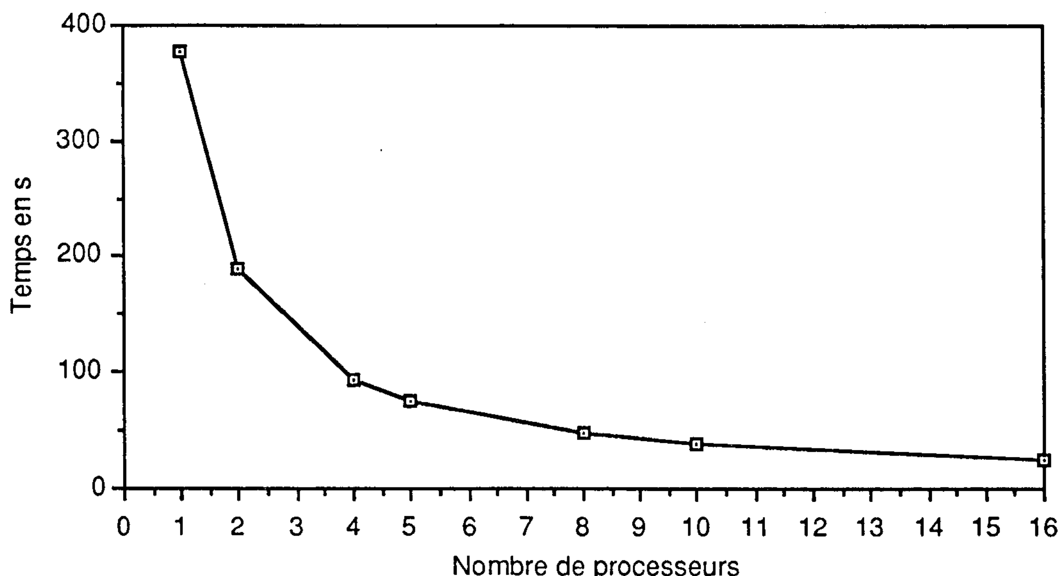


Figure 3.10. Temps d'exécution du crible d'Eratosthène sur l'intervalle [1, ..., 6 400 000] sur un réseau linéaire.

Le temps d'exécution décroît. Mais pour mieux apprécier la décroissance, calculons l'accélération et l'efficacité correspondant à la figure 3.10 avec N = 6 400 000. L'accélération obtenue avec i processeurs pour un même problème est le quotient du temps de résolution sur un processeur par le temps de résolution sur i processeurs. L'efficacité pour i processeurs est le quotient de l'accélération pour i processeurs par i.

Nombre de processeurs	Temps en s	Accélération	Efficacité
1	377	1,00	1,00
2	189	1,99	1,00
4	94,8	3,98	0,99
5	76,0	4,96	0,99
8	47,8	7,89	0,99
10	38,3	9,84	0,98
16	24,2	15,57	0,97

Figure 3.11. N = 6 400 000 : temps d'exécution, accélération, efficacité.

La figure 3.12 visualise ces valeurs sur des courbes. L'accélération obtenue est proche de l'accélération maximale, égale au nombre de processeurs. L'efficacité est elle aussi proche de son maximum : 1.

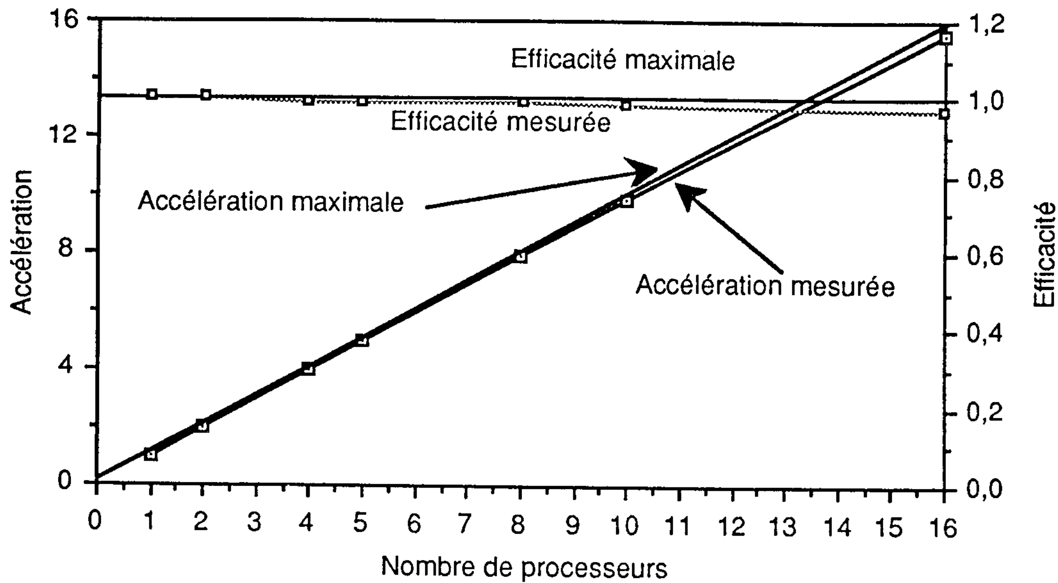


Figure 3.12. N = 6 400 000 : accélération et efficacité.

Etudions les temps de crible lorsque chaque processeur utilise toute sa mémoire disponible :

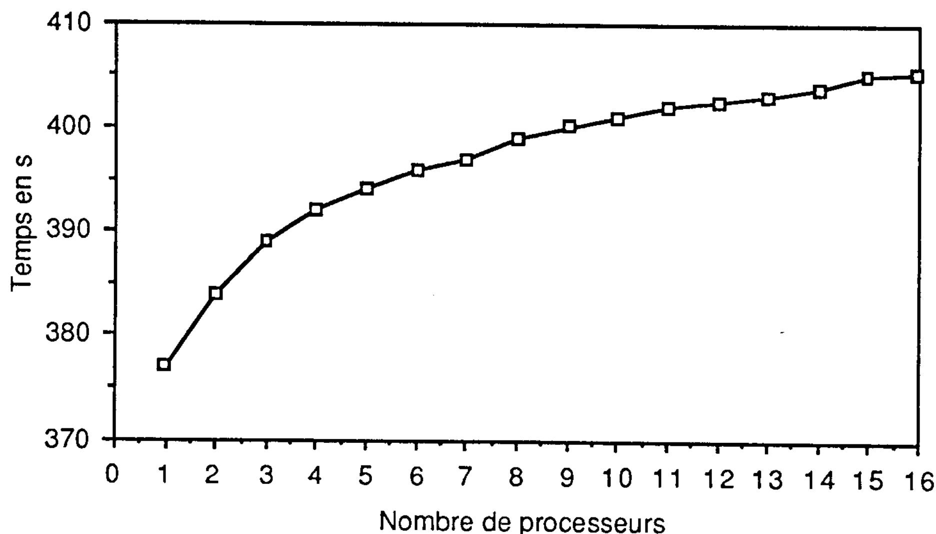


Figure 3.13. Temps d'exécution du crible d'Eratosthène sur un réseau linéaire (mémoire maximale).

Lorsqu'on crible avec y processeurs, chacun utilisant toute sa mémoire (800 Koctets) on peut générer tous les nombres premiers inférieurs à N = y * 6 400 000. Donc N croît rapidement. Pourtant le temps croît peu avec N. Ceci provient du fait que le pipeline est bien utilisé.

A partir des résultats, on peut calculer une autre accélération, celle de Gustafson [Gus 88] modifiée par Cosnard et al. [CRT 88].

Pour chaque nombre de processeurs, considérons la plus grande instance du problème qui puisse être résolue avec ce nombre de processeurs, et calculons le temps qu'il aurait fallu sur un seul processeur. Le rapport entre ces temps est l'accélération de Gustafson.

En effet, résoudre un problème de taille fixée, indépendamment du nombre de processeurs n'a que peu de sens sur une machine à mémoire distribuée. Avec le nombre maximum de

processeurs disponibles, on aimerait évaluer les performances obtenues lors de la résolution d'un gros problème.

Pour le calcul, on définit le temps moyen $\tau(i)$ d'une opération sur i processeurs. Alors l'accélération est $S_i = \tau(1) / \tau(i)$ et l'efficacité devient $E_i = S_i / i$.

Dans notre application, le crible de [1, ..., N] coûte environ N log log N opérations. Nous en déduisons $\tau(i)$, puis S_i et E_i.

Nombre de processeurs P	N (* 10^6)	Nombre d'opérations (* 10^6)	$\tau(P)$	S_P	E_P
1	6,4	17,6	21,41	1	1
2	12,8	35,8	10,73	2,00	1,00
3	19,2	54,1	7,19	2,98	0,99
4	25,6	72,6	5,40	3,96	0,99
5	32	91,2	4,32	4,96	0,99
6	38,4	109,8	3,61	5,93	0,99
7	44,8	128,5	3,09	6,93	0,99
8	51,2	147,3	2,71	7,90	0,99
9	57,6	166,1	2,41	8,88	0,99
10	64	184,9	2,17	9,87	0,99
11	70,4	203,8	1,97	10,87	0,99
12	76,8	222,6	1,81	11,83	0,99
13	93,2	241,6	1,67	12,82	0,99
14	89,6	260,5	1,55	13,81	0,99
15	96	279,5	1,45	14,77	0,98
16	102,4	298,5	1,36	15,74	0,98

Figure 3.14. Accélération et efficacité (méthode [CRT 88]).

L'accélération et l'efficacité sont proches du maximum.

Le temps total est moins bon qu'avec l'algorithme de base (voir chapitre 1). Mais la mise au point est très rapide, car l'algorithme est mieux analysé et plus structuré. Les problèmes de communication n'interviennent plus, car ils sont pris en charge automatiquement par le méta-algorithme. Le gain en temps de développement est très important.

Du point de vue de l'utilisateur, l'algorithme séquentiel une fois mis au point et les structures de données définies, l'implantation est très rapide. Si le gain en temps d'exécution est inexistant par rapport à une version de base, la mise au point pour l'exécution parallèle est grandement facilitée. C'est à ce niveau que se situe le gain.

3.5.2. Calcul de $\Omega(k)$ pour $1 \leq k \leq N$

Le but de cette application est de calculer rapidement et sans avoir besoin de factoriser les entiers, le nombre de facteurs premiers de chacun d'eux, sur un intervalle. Ce calcul peut être vu comme une étape intermédiaire du calcul de N(n,k), l'application du paragraphe 3.5.3.

$$\Omega(k) = \sum_{i=1}^{m} a_i, \text{ pour } k = \prod_{i=1}^{m} p_i^{a_i}$$

où p_i est un nombre premier. $\Omega(k)$ est le nombre de facteurs premiers de k, avec leur multiplicité.

Cet algorithme rentre dans la classe des algorithmes Maître / Esclaves, car le Maître doit chercher des nombres premiers, et les Esclaves doivent effectuer un traitement sur les données qu'ils possèdent en fonction du nombre premier qu'ils reçoivent. Globalement, on traite l'intervalle [1, ..., N]. Si cet intervalle est réparti de manière croissante pour les entiers sur les processeurs, il est implantable grâce au méta-algorithme présenté ci-avant.

Pour calculer $\Omega(k)$ sur [1, ..., N], plutôt que de factoriser chaque entier, cherchons tous les nombres premiers dans cet intervalle. Pour chaque nombre premier p de cet intervalle, calculons α tel que $p^{\alpha} \leq k < p^{\alpha+1}$. Pour chaque α_i de $[1, ..., \alpha]$, criblons les entiers jusqu'à N : pour chaque multiple r de p^{α_i}, ajoutons 1 à $\Omega(r)$. Pour cette méthode, il faut tous les nombres premiers inférieurs à N.

On peut s'affranchir de cette obligation et ne générer les nombres premiers que jusqu'à $\sqrt{N}$, pourvu qu'on puisse récupérer les diviseurs premiers de chaque entier. Une fois que l'intervalle [1, ..., N] est criblé avec tous les nombres premiers inférieurs à $\sqrt{N}$, on compare le produit des diviseurs trouvés pour chaque entier avec l'entier lui-même. Si ces deux nombres sont égaux, le nombre de facteurs premiers calculé est exact. Sinon, on a oublié un facteur premier plus grand que $\sqrt{N}$, et un seul. Dans ce cas, on ajoute 1 au nombre de facteurs premiers calculé.

Cette deuxième méthode nécessite 5 fois plus de place mémoire que la première. Mais elle est beaucoup plus rapide. Avec la première méthode, on peut aller jusqu'à N = 12 800 000, alors qu'avec cette version modifiée, on ne peut pas dépasser N = 2 400 000. Les temps d'exécution sont visualisés sur la figure 3.15.

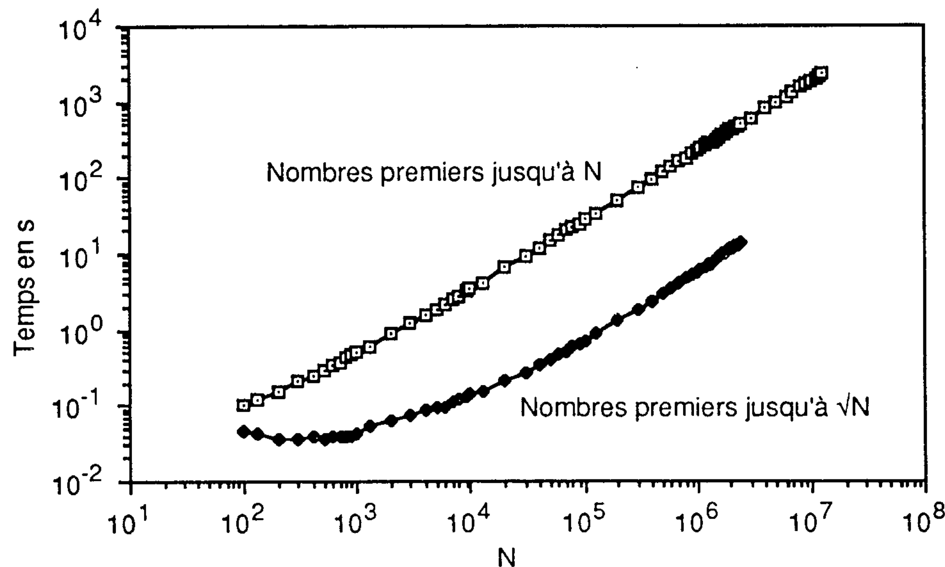


Figure 3.15. Temps d'exécution des deux méthodes de calcul de $\Omega(k)$.

Pour calculer toutes les valeurs de $\Omega(k)$, $k \leq N = 2\,400\,000$, il faut 498 s avec la première méthode et seulement 13,6 s avec la deuxième méthode, soit un gain d'un facteur proche de 37.

3.5.3. Calcul de N(N, k)

D'un point de vue algorithmique et méthodologique, nous donnons une autre application qui s'implante facilement et rapidement en parallèle grâce au méta-algorithme sur réseau linéaire.

Le but de cette implantation, d'un point de vue arithmétique, est de contribuer à la preuve que la suite N(N, k) est croissante puis décroissante en k pour N fixé, $N \geq N_0$, N_0 assez grand. Il est conjecturé que $N_0 = 0$.

N(N, k) est le nombre d'entiers inférieurs à N, ayant exactement k facteurs premiers. $\Omega(N)$ permet de calculer $N(N, k) = Card \{ x \leq N, \Omega(x) = k \}$ [Nic 84].

Pour calculer N(N, k) pour toutes les valeurs de k, nous calculons d'abord $\Omega(x)$ pour toutes les valeurs de x, $0 \leq x \leq N$. Ensuite, nous calculons N(N, k) en ajoutant 1 à la valeur de N(N, j) si $\Omega(x) = j$.

Cet algorithme est de type Maître / Esclaves, car il utilise principalement le calcul de $\Omega(j)$ qui rentre dans cette classe d'applications.

Le temps d'exécution est sensiblement le même que celui de $\Omega(N)$, car c'est le calcul des valeurs de $\Omega(k)$ qui prennent du temps.

Le FPS T20 a permis de calculer N(N, k) pour $N \leq 12\,800\,000$. Les résultats montrent que N(N, k) est croissante puis décroissante en k pour $0 \leq N \leq 12\,800\,000$.

3.5.4. Conclusion

Avec un investissement négligeable en temps de développement et d'implantation, nous avons parallélisé des applications de type Maître / Esclaves très facilement.

Le méta-algorithme présenté ci-avant est intéressant. Mais il faut le porter sur d'autres topologies, afin d'élargir le champ des applications qui peuvent être implantées en parallèle grâce à lui.

4. MÉTA-ALGORITHME DE PARALLÉLISATION D'APPLICATIONS MAITRE / ESCLAVES SUR UNE GRILLE NON TORIQUE À 2 DIMENSIONS

L'utilisation d'un réseau linéaire peut être coûteuse, car le message émis par le Maître doit traverser l'ensemble des Esclaves pour être reçu par la queue du réseau. Le diamètre d'un réseau linéaire de P processeurs vaut P-1. Nous souhaitons établir le même type de méthodologie, débouchant sur un méta-algorithme, mais pour une topologie plus compacte, où le diamètre a une valeur inférieure : la grille non torique à deux dimensions.

Notre but est de minimiser le travail du programmeur tout en préservant certains aspects à l'exécution : temps de contrôle maintenu assez faible, nombre de messages supplémentaires réduit, choix automatique du processeur Maître suivant, plus long chemin depuis le Maître restant assez court, ...

Sur un réseau linéaire, il est facile de décider du processeur qui devient le Maître suivant, car chaque processeur Maître successif n'a qu'un successeur possible. Sur la grille, un processeur a en général deux voisins qui peuvent devenir Maître.

Ainsi, sur la grille, il y a deux topologies utilisées simultanément :
- la grille pour les communications (travail ou contrôle),
- un réseau linéaire pour le choix du prochain Maître.

4.1. LA TOPOLOGIE DE COMMUNICATION

Dans un hypercube, on peut immerger une grille $2^a.2^b$. Mais l'analyse que nous faisons est valable pour tout type de grille. Une topologie de communication sur la grille est la suivante :

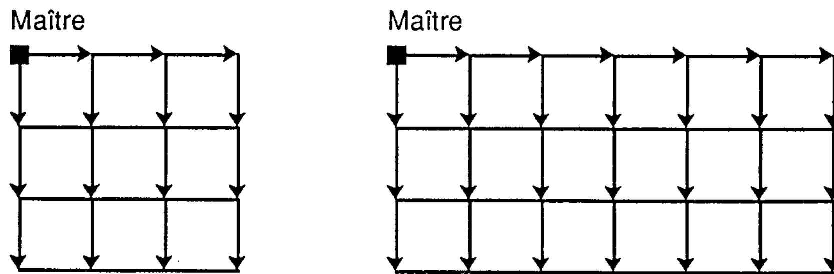


Figure 3.16. Grilles pour des hypercubes 2^{2n} et 2^{2n+1}.

Cette topologie permet d'avoir un diamètre initial minimal valant 2n-2 pour un hypercube ayant 2^{2n} sommets et 3n-2 pour un hypercube de 2^{2n+1} sommets.

Cependant, lorsque le Maître change, la longueur du plus long chemin issu de la *tête* peut être affectée. Et nous voulons la garder proche de sa valeur optimale, voire minimale.

4.2. LA STRUCTURE DES DONNÉES À COMMUNIQUER

La topologie n'a aucune influence sur le type de messages de contrôle et leur structure. L'analyse effectuée pour le réseau linéaire (§ 2.3) est valable pour la grille.

Lorsque cela est possible, il est important de ne pas augmenter le nombre de messages. Sinon, il faut utiliser un en-tête dans chaque message.

4.3. LE CHOIX DU MAITRE SUIVANT

Dans une grille, on peut immerger plusieurs réseaux linéaires :

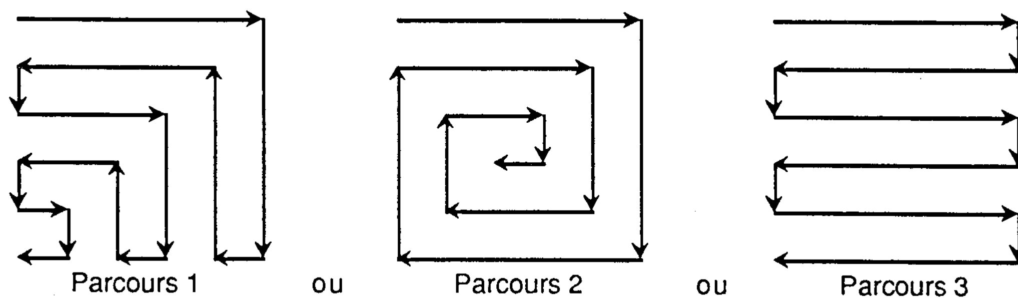


Figure 3.17. Quelques parcours linéaires d'une grille.

Pour choisir, il faut évaluer les retombées de ces parcours sur l'évolution de la longueur du plus long chemin issu du Maître, et sur d'autres paramètres éventuellement.

Parcours 1 : On remarque que, après avoir utilisé les processeurs de la première ligne, puis de la dernière colonne, de l'avant-dernière colonne et de la deuxième ligne, on revient dans un état semblable à l'état initial, avec une grille comportant 2 lignes et 2 colonnes de moins. Mais les modifications des liens de communication sont importants dans le parcours de ces 2 lignes et 2 colonnes. La longueur du plus long chemin issu du Maître, après quelques modifications lors des changements de Maître, peut être maintenu minimal.

Parcours 2 : C'est après le parcours d'une ligne qu'on se retrouve dans une situation semblable à la situation initiale, à une rotation près des liens. La longueur du plus long chemin issu du Maître est conservée à sa valeur minimale.

Parcours 3 : La longueur du plus long chemin issu du Maître n'est pas conservée à sa valeur minimale.

On choisit la solution du parcours 2, qui est plus régulier que le parcours 1, et plus puissant que le parcours 3. Il est de plus symétrique. Le problème maintenant concerne la reconfiguration locale des liens lorsque le Maître change.

4.4. RECONFIGURATION DYNAMIQUE DU RÉSEAU D'INTERCONNEXION

Il y a deux types de processeurs dans un tel réseau, en ce qui concerne la reconfiguration dynamique :

- ceux qui induisent une modification très locale des liens, (nous les appelons les processeurs normaux),
- ceux qui induisent une modification plus profonde, par la rotation des liens (nous les appelons les processeurs de rotation). Ces processeurs sont chaque premier processeur après qu'une ligne ou une colonne a été totalement utilisée.

4.4.1. Processeur normal

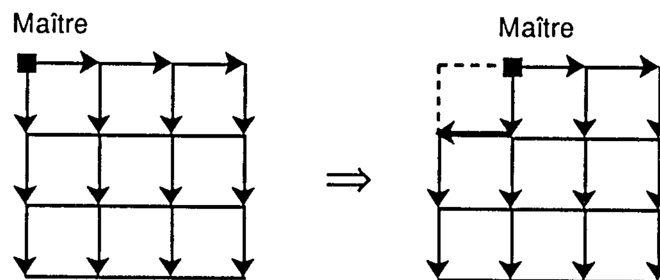


Figure 3.18. Nouveau processeur = processeur normal.

En fait la modification est plus profonde que la simple création d'un nouveau lien de communication.

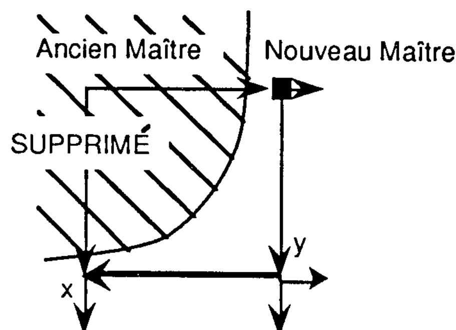


Figure 3.19. Les modifications locales des liens logiques de communication.

- Le Maître précédent avait deux liens en émission : ils sont supprimés.
- Le nouveau Maître avait un lien en réception : il est supprimé.
- Le processeur x avait un lien en réception : il est remplacé par un lien en réception sur un autre canal physique.
- Le processeur y se voit ajouter un lien en émission.

4.4.2. Processeur de rotation

Lorsqu'un processeur de rotation devient Maître, la longueur du plus long chemin issu du Maître est encore minimum. Mais le prochain Maître apporte des modifications locales qui impliqueront une augmentation de cette longueur, ce qui est très mauvais (voir fig. 3.20).

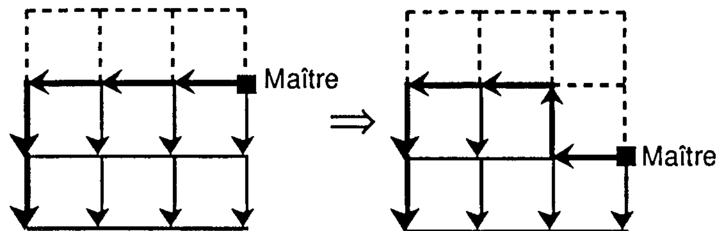


Figure 3.20. Augmentation de la longueur du plus long chemin issu du Maître.

De façon à maintenir la décroissance de la longueur du plus long chemin issu du Maître, un processeur de rotation effectue une rotation de -90° de certains liens de communication : ce sont tous les liens présents dans le sous-domaine du Maître (voir fig. 3.21).

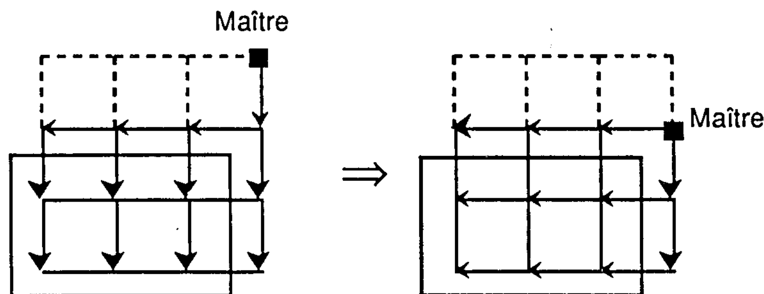


Figure 3.21. Rotation des liens.

On a donc toujours un arbre couvrant de hauteur minimum sur la topologie disponible.

Ces modifications n'affectent pas que les processeurs du sous-domaine, mais tous les processeurs restants dans la grille, sauf le Maître. Ceci provient du fait qu'un lien est en émission pour un processeur et en réception pour un autre processeur.

Après cette rotation des liens, la longueur du plus long chemin issu du Maître est minimale et le nouveau réseau est identique (en forme) au réseau initial. Cette propriété peut apporter une grande souplesse pour la mise en oeuvre du méta-algorithme pour la grille.

4.5. LE MÉTA-ALGORITHME

Il n'y a que deux types de processeurs non *passifs* :
- la *tête* (le Maître),
- les processeurs-*milieu* (les Esclaves).

En effet, la *queue* est la *queue* seulement pour le réseau linéaire de parcours des Maîtres successifs. Il n'y a pas de *queue* pour la grille.

On retrouve les mêmes procédures que pour le réseau linéaire. Elles jouent les mêmes rôles. Une nouvelle procédure est introduite :

- 'traitement_milieu' : cette procédure peut modifier localement la variable 'message', lorsque, par exemple, le Maître change et qu'il faut modifier localement quelques liens. Dans ce cas, le Maître envoie un message spécial. Le récepteur obéit aux ordres du message spécial, mais ne transmet pas automatiquement ce message à ses successeurs. En ce sens, il le modifie.

Méta-algorithme

```
Méta-algorithme
|   initialisation (&message);                  /* Construit la grille et acquiert les paramètres */
|   tant que (état ≠ passif) faire
|   |   Si (état == tête) alors
|   |   |   cherche (&message);
|   |   |   envoi (message);
|   |   |   état = automate (message);                                          /* tête ou passif */
|   |   sinon
|   |   |   réception (message);
|   |   |   traitement_milieu (&message);
|   |   |   envoi (message);
|   |   |   état = automate (message);                                          /* milieu ou tête */
|   |   fin si
|   |   Si (état ≠ passif) alors
|   |   |   traitement_général (message);
|   |   fin si
|   fin tant que
|   traitement_final ();
fin méta-algorithme
```

Figure 3.22. Méta-algorithme pour grille.

Dorénavant, les procédures de communication doivent vérifier que l'émetteur a des successeurs (0, 1 ou 2) et des prédécesseurs (0 ou 1).

4.6. APPLICATION : LE CRIBLE D'ÉRATOSTHÈNE

Reprenons le crible d'Eratosthène. Les caractéristiques de l'algorithme, les valeurs des variables sont les mêmes que celles présentées au § 3.5.1 pour le réseau linéaire. En ce sens, le programmeur n'a pas de modifications à apporter à son algorithme. C'est, par exemple, l'automate qui est modifié.

La figure 3.23 donne le temps de recherche de tous les nombres premiers inférieurs à N, avec N compris entre 10^3 et 10^8.

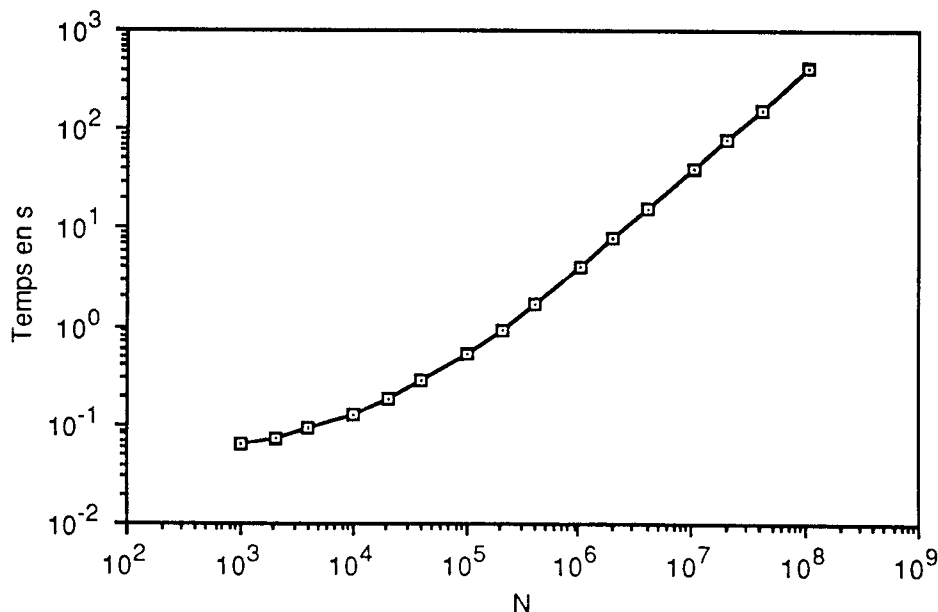


Figure 3.23. Temps d'exécution du crible d'Eratosthène sur une grille de 16 processeurs

Pour de petites valeurs de N, le temps d'exécution est important à cause du contrôle, tout comme pour le réseau linéaire.

Faisons s'exécuter l'algorithme sur un processeur, avec la valeur de N maximum, N = 6 400 000. Puis cherchons les nombres premiers dans le même intervalle avec différents nombres de processeurs (figure 3.24).

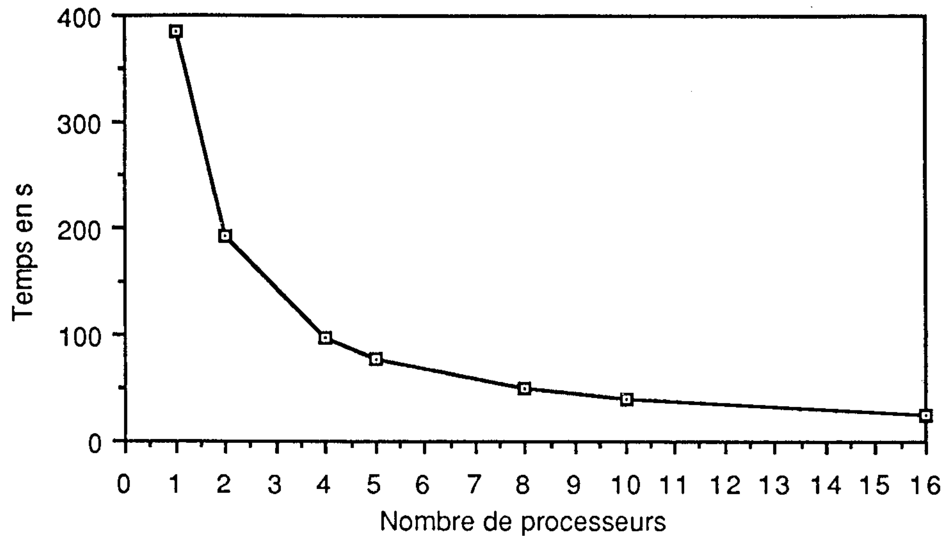


Figure 3.24. Temps d'exécution du crible d'Eratosthène sur l'intervalle [1, ..., 6 400 000] sur une grille.

Le temps décroît. Mais pour mieux apprécier la décroissance, calculons l'accélération et l'efficacité correspondant à cette figure 3.24 avec N = 6 400 000. Nous utilisons les mêmes formules que pour le calcul des accélérations et des efficacités sur le réseau linéaire.

Nombre de processeurs	Temps en s	Accélération	Efficacité
1	385	1	1
2	193,2	2,00	1,00
4	97,1	3,96	0,99
5	77,9	4,94	0,99
8	49,0	7,86	0,98
10	39,4	9,77	0,98
16	25,0	15,40	0,96

Figure 3.25. N = 6 400 000 : temps d'exécution, accélération, efficacité.

La figure 3.26 visualise ces valeurs sur des courbes. L'accélération obtenue est proche de l'accélération maximale, égale au nombre de processeurs.

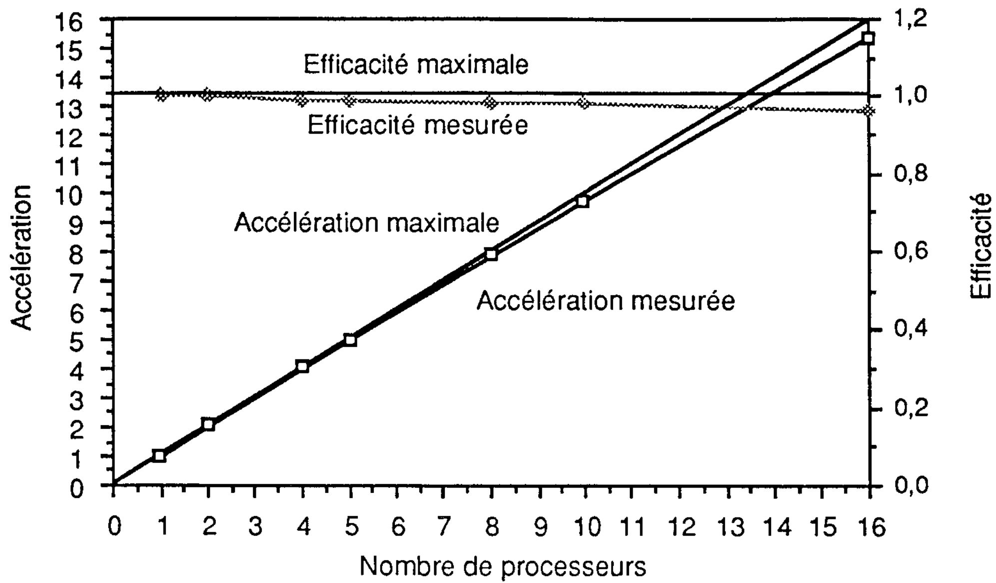


Figure 3.26. N = 6 400 000 : accélération et efficacité.

Il est aussi intéressant d'étudier les temps de crible lorsque chaque processeur utilise toute sa mémoire disponible.

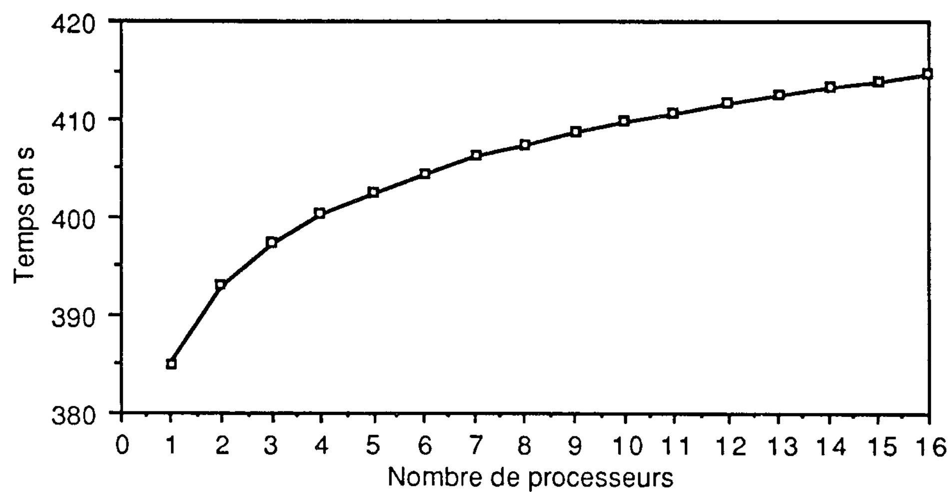


Figure 3.27. Temps d'exécution du crible d'Eratosthène sur une grille (mémoire maximale).

Lorsqu'on crible avec un nombre croissant de processeurs, chacun utilisant toute sa mémoire (800 Koctets), le temps croît peu avec N.

Calculons l'accélération de Cosnard et al. [CRT 88].

Nombre de processeurs P	N (* 10^6)	Nombre d'opérations (* 10^6)	$\tau(P)$	S_P	E_P
1	6,4	17,6	21,860	1	1
2	12,8	35,8	10,984	1,990	0,995
3	19,2	54,1	7,339	2,979	0,993
4	25,6	72,6	5,512	3,966	0,991
5	32	91,2	4,415	4,951	0,990
6	38,4	109,8	3,682	5,937	0,989

7	44,8	128,5	3,160	6,918	0,988
8	51,2	147,3	2,766	7,903	0,988
9	57,6	166,1	2,461	8,883	0,987
10	64	184,9	2,216	9,865	0,986
11	70,4	203,8	2,016	10,843	0,986
12	76,8	222,6	1,849	11,823	0,985
13	93,2	241,6	1,707	12,806	0,985
14	89,6	260,5	1,586	13,483	0,985
15	96	279,5	1,481	14,760	0,984
16	102,4	298,5	1,389	15,738	0,984

Figure 3.28. Accélération et efficacité (méthode [CRT 88]).

Les accélérations sont importantes. L'efficacité aussi. Comparons le réseau linéaire et la grille sur ce même exemple.

5. COMPARAISON DU RÉSEAU LINÉAIRE ET DE LA GRILLE

Nous pouvons comparer le réseau linéaire et la grille sur deux critères :
- le temps d'exécution d'une même application,
- la facilité d'implantation d'une application,

sur ces deux topologies.

5.1. COMPARAISON DES TEMPS D'EXÉCUTION

Nous faisons s'exécuter le même programme sur le réseau linéaire et sur la grille. La grille doit avoir une puissance de 2 comme nombre de processeurs. C'est pourquoi nous construisons des grilles de 1, 2, 4, 8 et 16 processeurs, et des réseaux linéaires de mêmes tailles.

Applications sur 16 processeurs :

Taille de l'exemple	Temps en s Réseau linéaire	Temps en s Grille	Rapport Grille / Réseau linéaire
1	0,301	0,415	1,38
10	2,19	2,97	1,36
100	17,19	23,32	1,36
1000	143,2	193,4	1,35

Figure 3.29. Comparaisons Réseau linéaire / Grille.

La grille est beaucoup plus coûteuse que le réseau linéaire. C'est somme toute assez normal. Les communications sont plus difficiles à gérer. Et bien sûr, le contrôle est plus important, surtout en ce qui concerne la rotation des liens, les modifications des voies de communication, ...

En fait, le temps se perd principalement en attente de synchronisation, car les communications se font sur rendez-vous.

5.2. COMPARAISON DES IMPLANTATIONS D'UNE MÊME APPLICATION

Nous avons vu que le découpage de l'algorithme séquentiel en deux parties, code du Maître et code des Esclaves, la définition de la structure des données à communiquer, et la répartition des informations entre les processeurs représentaient les étapes nécessaires à l'implantation d'une application sur le réseau linéaire.

Il en est de même avec la grille. Une fois l'algorithme découpé en deux tâches et les informations réparties entre les processeurs (attention au parcours de la tête du réseau), le travail de l'implanteur n'est pas plus difficile sur l'une ou l'autre topologie.

6. CONCLUSION

6.1. RÉSEAU LINÉAIRE ET GRILLE

Nous avons présenté un méta-algorithme pour implanter des applications de type Maître / Esclaves sur un réseau linéaire. L'utilisateur-programmeur de l'application parallèle n'a pas à se soucier des communications, des synchronisations, de l'état des processeurs, de leur activité / passivité dynamique. Le méta-algorithme gère ces informations.

Ainsi, le programmeur, qui a déjà dû découper son algorithme en tâches Maître et Esclaves pour son algorithme séquentiel, n'a pas de problème pour implanter son programme sur une machine parallèle. Toute la gestion du parallélisme lui est cachée. Son travail est ainsi grandement facilité.

Sur une grille de processeurs, c'est un peu plus difficile, car deux topologies sont utilisées en parallèle : un réseau linéaire pour la succession des Maîtres et une grille 2D pour les communications. La difficulté dans le méta-algorithme réside dans les modifications locales, mais surtout globales des liens de communication logiques (les liens physiques sont figés). Mais ainsi, toute la gestion du parallélisme est transparent pour le programmeur parallèle.

On remarque sur les résultats que la grille est beaucoup plus lente (plus de 30 %) que le réseau linéaire car sa gestion est lourde et, surtout, les synchronisations sont importantes. Il serait beaucoup plus intéressant d'avoir des communications asynchrones plutôt que synchrones pour les communications.

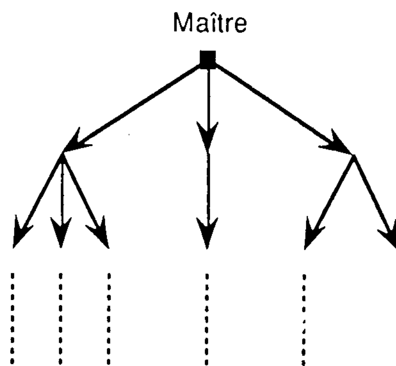
6.2. AUTRES TOPOLOGIES

Nous pouvons envisager la conception de méthodologies et de méta-algorithmes pour d'autres topologies comme l'arbre ou l'hypercube.

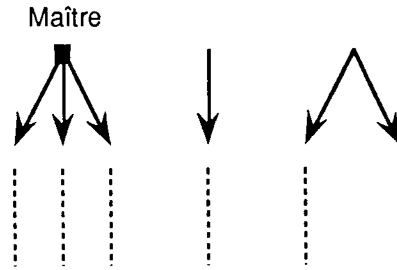
6.2.1. L'arbre

Initialement, le Maître est à la racine. Lorsque ce Maître devient passif, il est supprimé de l'ensemble des processeurs utilisables. Avec lui, sont aussi supprimées les voies de communication adjacentes.

L'arbre initial



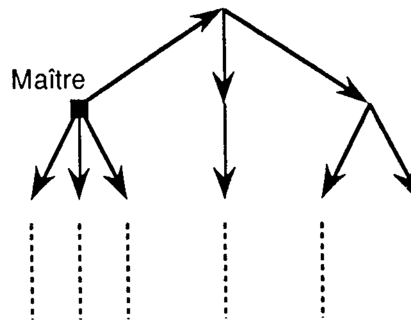
devient



c'est-à-dire une forêt. Le graphe n'est plus connexe. Quel que soit le Maître suivant, il ne peut pas envoyer les informations à tous les Esclaves, puisque les voies de communication n'existent plus.

Donc, il ne faut pas supprimer physiquement le Maître précédent, mais le supprimer logiquement de la liste des processeurs exécutant l'application sous-jacente, tout en le laissant comme nœud de communication.

L'arbre pourrait alors devenir :

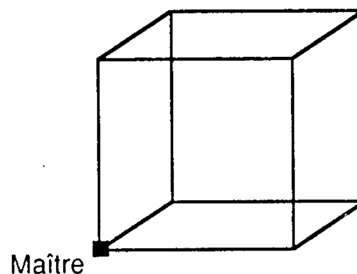


Ainsi, de la même manière que pour le réseau linéaire et la grille, on pourrait concevoir un méta-algorithme. Et c'est le parcours séquentiel des Maîtres qui serait modifié. Lorsqu'un sous-arbre est parcouru complètement, il est supprimé et on passe au sous-arbre suivant.

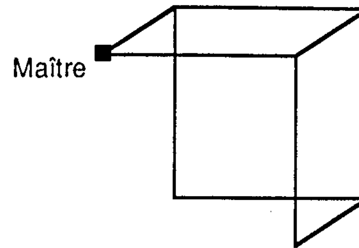
6.2.2. L'hypercube

Sur le réseau linéaire et sur la grille, le parcours des Maîtres se fait de proche en proche, et permet de garder le plus court possible, le plus long chemin du Maître à chaque Esclave. Ici, comme dans l'arbre, ce n'est pas le cas.

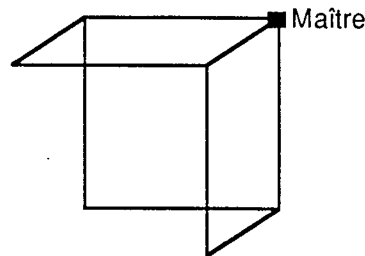
Choisissons un Maître initial :



Puis, lorsque ce Maître devient passif, il choisit un de ses voisins comme nouveau Maître (ici nous prenons un voisin quelconque puisque les voisins jouent un rôle symétrique) :



Pour avoir un arbre de recouvrement de hauteur minimale, il faudrait que sa racine soit située sur le sommet suivant :



Ainsi, on prévoit un parcours des Maîtres, non plus par voisinage, mais de manière à placer la tête du réseau à la racine d'un arbre de recouvrement de hauteur minimale.

Il est sans doute possible d'établir un parcours des Maîtres sur un hypercube, sans avoir besoin de réutiliser des processeurs déjà passifs pour les communications. C'est-à-dire qu'il faut définir un parcours qui conserve la connexité du graphe de communication.

Ainsi, pour des applications de type Maître / Esclaves, la conception d'outils de gestion du parallélisme peut être envisagée de manière à libérer le programmeur de certaines tâches ingrates de mise au point des synchronisations et des communications. Et ce sont souvent ces étapes qui posent des problèmes sur les machines à mémoire distribuée.

Mais il serait bon aussi d'ouvrir le domaine des applications à d'autres types d'algorithmes, par exemple les algorithmes de type "Diviser pour Régner", les fermes de processeurs, ..., eux aussi sur différentes topologies de réseaux de processeurs à mémoire distribuée.

Chapitre 4
Le crible quadratique : Etude de la parallélisation sur un multiprocesseur hypercube à mémoire distribuée illimitée

1. INTRODUCTION

Certaines parmi les méthodes de décodage actuelles reposent sur la connaissance des grands facteurs premiers, deux en général, d'un entier N de cent chiffres et plus. N permet le cryptage, mais le décryptage reste impossible sans ses deux facteurs premiers. C'est ce qu'on appelle un code à clé publique [RSA 78] où le nombre N est connu de tous, contrairement à ses facteurs qui sont gardés secrets.

Le principe du système de codage / décodage RSA est le suivant. Soient p et q deux nombres premiers impairs distincts, gardés secrets. Leur produit N est rendu public. Pour utiliser ce système, chaque utilisateur choisit deux entiers c, d < $\phi(N)$ (fontion d'Euler) tels que

$$\text{pgcd}(c, \phi(N)) = 1$$
$$c.d \equiv 1 \bmod \phi(N).$$

c est rendu public, mais d est gardé secret. Les fonctions C de codage est D de décodage sont respectivement :

$$C(x) = x^c \bmod N$$
$$D(x) = x^d \bmod N.$$

On peut calculer les facteurs de N par des méthodes de factorisation. Mais actuellement, elles nécessitent des moyens de calcul hors de portée des ordinateurs les plus puissants. En effet le plus grand entier ordinaire factorisé comprend 106 chiffres décimaux [LeM 89]. Il a été factorisé par la méthode du crible quadratique, en utilisant les temps de cycle d'inactivité de plusieurs centaines de machines de par le monde, et le temps total avoisine les 25 ans CPU.

Des nombres particuliers plus grands sont factorisés par d'autres méthodes. Par exemple François Morain a achevé la factorisation de $F_{11} = 2^{2^{11}} + 1$ [Mor 88]. A. Cunningham a trouvé 2 de ses facteurs en 1899. En 1980, S. Wagstaff a prouvé que le cofacteur restant était composé. En 1988, P. Brent a calculé sa décomposition en un produit de 3 entiers dont le plus

grand comportait 564 chiffres. F. Morain a prouvé que ce cofacteur était premier. Cet exemple montre que méthodes de factorisation et tests de primalité sont étroitement liés. Un autre exemple de factorisation d'un nombre de 138 chiffres de la forme $r^e \pm s$ où r et s sont de petits nombres positifs, est dû à Lenstra et al. par la méthode du crible dans un corps de nombres [LLM 90].

Il est cependant assez clair que les créateurs de codes de cryptographie vont orienter leurs recherches vers des nombres (les codes secrets) ordinaires. Pour factoriser ces nombres, il existe six algorithmes asymptotiquement les plus rapides : ces méthodes ont toutes expérimentalement le même temps d'exécution :

$$L(N) = \exp\left\{(1 + o(1)) \sqrt{\log N \log \log N}\right\}$$

pour factoriser N [PST 88].

Les six méthodes sont les suivantes :

- l'algorithme des courbes elliptiques de Lenstra [Len 87],
- l'algorithme Monte-Carlo de Schnorr-Lenstra [ScL 84],
- l'algorithme du crible linéaire de Schroeppel [COS 86], [Pom 82],
- l'algorithme du crible quadratique de Pomerance [Pom 82], [Pom 85],
- l'algorithme du crible de liste de résidus de Coppersmith, Odlyzko et Schroeppel, [COS 86],
- l'algorithme des fractions continues de Morrison-Brillhart [MoB 75].

Ces méthodes ont permis de compléter les tables de factorisation de nombres particuliers ou du projet Cunningham [BMS 88], [BLS 83], [DaH 84]. Le projet Cunningham a pour but de construire la plus grande compilation de factorisations d'entiers de la forme $b^n \pm 1$ [BLS 83].

Ces méthodes se regroupent en deux familles présentées ci-dessous.

1.1. MÉTHODES DE FACTORISATION PAR DIFFÉRENCE DE CARRÉS

Le but de ces méthodes est de trouver 2 entiers x et y tels que $x^2 - y^2 = (x-y)(x+y) \equiv 0 \bmod N$. L'idée sous-jacente aux méthodes des fractions continues et du crible quadratique, qui font partie de cette famille, est la suivante. Déterminons un ensemble de nombres premiers appelée "base de facteurs". Générons alors une liste de couples (A_i, Q_i) tels que $A_i^2 \equiv Q_i \bmod N$ et tels que les Q_i se factorisent complètement sur la base de facteurs. Construisons ensuite une matrice, avec une ligne par Q_i et une colonne par élément p_j de la base, dans laquelle l'élément (i,j) est l'exposant, modulo 2, de p_j dans la factorisation de Q_i. Les Q_i doivent être aussi "petits" que possible pour que la probabilité qu'ils soient factorisables sur la base soit aussi grande que possible. Si on trouve une combinaison linéaire nulle de certaines lignes $(R_{i_1}, \ldots, R_{i_r})$ de cette matrice, alors le produit des Q_{i_m} correspondants est un carré, et par conséquent

$$\prod_{m=1}^{r} A_{i_m}^2 \equiv \left(\left(\prod_{m=1}^{r} Q_{i_m}\right)^{1/2}\right)^2 \bmod N.$$

Alors

$$x = \prod_{m=1}^{r} A_{i_m} \text{ et } y = \left(\prod_{m=1}^{r} Q_{i_m}\right)^{1/2}$$

conviennent. Avec une probabilité d'au moins 1/2, pgcd(x-y, N) est un facteur non trivial de N.

La méthode des fractions continues a été implantée dans des versions parallèles sur le MPP (Massively Parallel Processor) de la NASA [Wun 84], [Wun 85] [Wun 87], et sur une machine dédiée, EPOC [PoW 83] [SmW 83], avec des résultats montrant que l'étape de factorisation des Q_i est très coûteuse.

Le crible quadratique est sans doute la méthode la plus puissante à ce jour pour factoriser des entiers N ordinaires (ordinaire signifiant "d'aucune forme particulière", c'est à dire que N n'est pas un nombre de Fermat ou de Mersenne...). Cette méthode a été implantée en parallèle sur des multiprocesseurs vectoriels (Cray I et XMP, [DHS 85] [Ger 83], Cyber 205 et NEC SX-2 [RLW 88], sur des réseaux de stations de travail [Sil 87] [CaS 88] [Sil 88] [LeM 89], et sur des machines parallèles distribuées, le Ncube [DaH 88] et le FPS T40 [CoP 90]. Les résultats sont bien supérieurs à ceux obtenus par la méthode des fractions continues.

1.2. MÉTHODES DE FACTORISATION SUR DES GROUPES [Bue 87]

La méthode p-1 de Pollard fait partie de cette famille.

Un fait élémentaire de la théorie des nombres est que les entiers modulo un nombre premier forment un groupe abélien cyclique fini pour la multiplication. C'est ce groupe qui est utilisé ici. Ainsi, le petit théorème de Fermat (p premier et a ne divise pas p, alors $a^{p-1} \equiv 1 \bmod p$) est vu comme le théorème de Lagrange pour les groupes finis, stipulant que tout élément d'un groupe multiplicatif, exposant l'ordre du groupe, est l'identité. Etant donné un facteur premier p de N, si on peut calculer $b \equiv a^{p-1} \bmod N$, alors on a p = pgcd (N, b-1). Le problème, bien entendu, est que p est inconnu. C'est lui que ces méthodes permettent de trouver.
La méthode des courbes elliptiques utilise le groupe des points d'une courbe de la forme suivante : $y^2 = x^3 + ax + b \bmod p$. Elle est bien adaptée à un traitement parallèle, en affectant à chaque processeur des courbes différentes [Lee 89]. Cette méthode donne les facteurs de 30 chiffres de nombres ayant jusqu'à plus de 200 chiffres.
La méthode de factorisation de Schnorr et Lenstra utilise le groupe des classes de formes quadratiques à discriminant négatif. Elle aussi est bien adaptée à une implantation parallèle [ScL 84]. Mais celle-ci n'a pas été réalisée.

1.3. D'AUTRES MÉTHODES

La plus simple est sans aucun doute celle des divisions successives. Mais elle ne sert plus, actuellement, qu'à éliminer les petits facteurs des grands entiers, avant qu'ils ne soient factorisés par une des méthodes présentées ci-dessus. M. Leeder [Lee 89] l'a implantée de façon efficace sur la Queen's Tree Machine, un multiprocesseur à mémoire distribuée dont la topologie est un arbre binaire de 15 processeurs.

Enfin, la méthode la plus récente, mais qui ne permet de factoriser que des nombres de la forme $r^e \pm s$ où r et s sont de petits nombres positifs, est la plus puissante à ce jour [LLM 90]. C'est la méthode du crible dans un corps de nombres. Elle est très parallèle et a été implantée sur un réseau de plusieurs centaines de processeurs CVAX, piloté par une machine centrale qui distribue les tâches et récupère les résultats.

1.4. LES ARCHITECTURES DÉDIÉES

Nombreuses sont les équipes de recherche qui ont étudié, développé, et pour certaines, réalisé des machines, parallèles ou non, dédiées à la factorisation. Ce ne furent parfois que des processeurs spécialisés dans le traitement des entiers en précision entière illimitée [CRB 85], [CCD 88] ou pour le crible quadratique [Pom 85 p. 180]. Mais des machines totalement dédiées virent aussi le jour, comme EPOC [SmW 83], une machine construite autour de 128 diviseurs 128 bits par 32 bits qui rendent 1 ou 0 selon que le mot de 32 bits divise ou non le mot de 128 bits. La Queen's Tree Machine [Lee 89] est en grande partie dédiée à la factorisation.

Mais c'est surtout l'étude de Pomerance, Smith et Tuler qui tire le maximum de la forme de l'algorithme du crible quadratique pour construire une machine dédiée à la factorisation par cet algorithme [PST 88]. Il s'agit d'une architecture pipeline qui permet d'accélérer la phase de crible de l'intervalle avec les éléments de la base. A ma connaissance, cette architecture est restée théorique.

L'étude de la parallélisation du crible quadratique est présentée sur deux chapitres. Le premier est consacré à la théorie de la parallélisation de l'algorithme sur un hypercube à mémoire distribuée illimitée. Le suivant est consacré à son implantation sur le FPS T40.

Les auteurs d'articles sur la parallélisation du crible quadratique s'attachent à paralléliser l'étape la plus coûteuse : le crible de l'intervalle par les éléments de la base. Or le crible quadratique est composé de plusieurs étapes. Nous souhaitons étudier théoriquement la parallélisation de ces étapes, indépendamment les unes des autres si cela est possible, de manière à tirer parti du parallélisme de l'algorithme complet, et non pas d'une seule de ses étapes. Mais Herman te Riele a vectorisé toutes les boucles [Rie 88] sur NEC SX2 et Cyber 205.

Nous montrons que lorsque les données tiennent en mémoire, la parallélisation est très efficace. Dans le cas contraire, les échanges de données sont très coûteux.

Le paragraphe 2 est consacré à une présentation plus précise de l'algorithme du crible quadratique multipolynomial que nous avons choisi d'implanter sur l'hypercube FPS T40. Puis nous étudions des parallélisations possibles de cet algorithme sur hypercube multiprocesseur à mémoire distribuée sans limitation de la taille mémoire. Etudier la parallélisation théorique de l'algorithme du crible quadratique multipolynomial consiste à étudier chaque étape séquentielle de cet algorithme, et à essayer d'en extraire le maximum de parallélisme possible. Dans le paragraphe 3, nous détaillons ces étapes, l'initialisation, le crible-factorisation et l'élimination de Gauss. Pour chacune d'elles, nous donnons divers critères qui permettent d'étudier le parallélisme : l'allocation des données, l'allocation des tâches, les communications...

Ce chapitre permet de poser les bases de l'implantation sur le FPS T40, que nous présentons au chapitre 5.

Nous donnons aussi des informations concernant l'analyse pour d'autres machines distribuées, hypercubes ou non, comme l'iPSC/2, le TNode ou le MégaNode, la Connection Machine. Nous nous plaçons encore dans l'hypothèse où la mémoire de ces machines est illimitée. Voici les caractéristiques de ces multiprocesseurs :

- Le FPS T40 est présenté dans le chapitre d'introduction de cette thèse.
- Le TNode est un multiprocesseur bâti sur une architecture reconfigurable de Transputers T800. Le principe de communication est celui d'OCCAM, c'est-à-dire sur rendez-vous et en parallèle sur les liens d'un même Transputer. De même que pour le FPS T40, une communication coûte $\beta + \mu n$, avec $\beta = 15$ µs et $\mu = 1{,}43$ µs/octet. Il n'y a pas de routage automatique.
- L'iPSC/2 est un multiprocesseur bâti sur une architecture hypercube. Le mode de communication est différent de celui des deux machines précédentes, car il existe un routeur automatique qui permet d'acheminer un message d'un processeur à un autre processeur même s'ils ne sont pas directement reliés. Un coprocesseur est dédié au routage. Le principe de communication est le suivant : un premier message d'un octet met en place les connexions, puis le message lui-même est envoyé en utilisant la pleine puissance des liens.
- Quant à la Connection Machine, la première différence avec les trois ordinateurs précédents est son appartenance à la classe SIMD : tous les processeurs sont synchrones et exécutent la même instruction sur des données différentes. De plus la mémoire par nœud (16 processeurs 1 bit) est assez limitée, 64 Ko (extensible à 128 Ko et 256 Ko). Son architecture est hypercubique. Son principe de communication est basé sur un routeur automatique en ce qui concerne les échanges point à point très rapides (chaque nœud a son routeur, un buffer de réception et un buffer d'émission). Quant aux communications sur des architectures régulières, comme la grille (NEWS), elles sont modélisables par $\beta + \mu n$ avec des valeurs de β et de μ dépendant du nombre de processeurs virtuels [PoM 89] (exemple : $\beta = 600$ µs et $\mu = 1\,560$ µs/octet pour une grille virtuelle 1024 x 1024 sur une CM2 à 64 K nœuds).

Dans notre étude, la mémoire de chaque nœud est illimitée.

2. PRÉSENTATION DE L'ALGORITHME DU CRIBLE QUADRATIQUE MULTIPOLYNOMIAL (MPQS)

Le crible quadratique a des origines qui remontent à Maurice Kraïtchik [Kra 26]. C'est seulement en 1975 que Morrison et Brillhart [MoB 75 p. 199] font référence à une méthode due à Daniel Shanks [Sha 71], utilisant un crible pour trouver un couple (X, Y) tel que $X^2 - Y^2 = N$. A cette époque elle n'était pas encore programmée. En 1982, Carl Pomerance présente et analyse le crible quadratique [Pom 82] comme un cas particulier du crible linéaire de Schroeppel. Mais il faut attendre Gerver en 1983 [Ger 83] pour avoir les premiers résultats expérimentaux. Bien d'autres ont suivi jusqu'à Silverman en 1987 qui implante le crible quadratique multipolynomial [Sil 87]. Entre temps des améliorations algorithmiques sont introduites, qui permettent des gains importants en ce qui concerne les temps d'exécution. Mais tout d'abord, présentons l'algorithme du crible quadratique multipolynomial.
L'algorithme de base du crible quadratique, permettant de factoriser N, cherche deux entiers X et Y tels $X^2 \equiv Y^2 \bmod N$. Lorsqu'on trouve X et Y alors pgcd(X-Y, N) est un facteur de N. Ce facteur est un facteur non trivial (différent de 1 ou N) avec une probabilité supérieure ou égale à 0,5 [PST 88]. Les deux entiers X^2 et Y^2 ne sont pas faciles à obtenir. Ils sont construits à partir de congruences auxiliaires de la forme $u_i^2 \equiv v_i^2\, w_i \bmod N$. Si on peut trouver un ensemble I d'indices tels que le produit des $w_{i\in I}$ soit un carré, alors on a

$$\prod_{i\in I} u_i^2 \equiv \left(\prod_{i\in I} v_i^2\right)\left(\prod_{i\in I} w_i\right) \bmod N$$

soit $$\prod_{i\in I} u_i^2 \equiv \prod_{i\in I} (v_i\sqrt{w_i})^2 \bmod N$$

et $$X = \prod_{i\in I} u_i \text{ et } Y = \left(\prod_{i\in I} v_i\right)\left(\prod_{i\in I} w_i\right)^{1/2}$$

conviennent. Le problème est de factoriser le produit des $w_{i\in I}$, c'est-à-dire chaque w_i . Pour cela, on utilise un ensemble de petits nombres premiers appelé la base de facteurs. Et on construit des entiers w_i qui se factorisent complètement sur cette base.

Posons

$$u_i = u(x) = a^2x + b$$
$$v_i = v(x) = a$$
$$w_i = w(x) = a^2x^2 + 2bx + c, \text{ avec } b^2 \equiv N \bmod a^2,\ |b| < a^2/2,\ b^2 - N = a^2c.$$

On a
$$\begin{aligned} u^2(x) &= (a^2x + b)^2 \\ &= a^4x^2 + 2a^2bx + b^2 \\ &= a^4x^2 + 2a^2bx + a^2c + N \\ &= a^2(a^2x^2 + 2bx + c) + N \\ &= v^2(x)\, w(x) + N. \end{aligned}$$

Donc $u^2(x) \equiv v^2(x)\, w(x) \bmod N,\ \forall x \in \mathbb{R}$.

Notre problème est de générer des valeurs de w(x) qui se factorisent complètement et assez facilement sur la base de facteurs, même si les valeurs de a varient, c'est-à-dire si on utilise plusieurs polynômes w(x). Donc il faut trouver des valeurs du paramètre a telles que w(x) reste à peu près constant sur un intervalle de la forme [-M, M[.

On veut donc que $|w(-M)| \approx |w(0)| \approx |w(M)|$. Ceci est vérifié si l'on choisit a proche de $\sqrt{\frac{\sqrt{2N}}{M}}$,

car $$w(-M) \approx w(M) \approx a^2M^2 = \frac{\sqrt{2N}}{M} M^2 = M\sqrt{2N},$$

et $$w(0) = c = \frac{b^2 - N}{a^2} \approx \frac{-N}{a^2} = -\frac{NM}{\sqrt{2N}} = -M\sqrt{\frac{N}{2}}.$$

Cette idée de Carl Pomerance [Pom 85] permet d'utiliser plusieurs polynômes dont la taille est à peu près constante en fonction de a. Avec chaque polynôme, nous générons un ensemble de valeurs de w(x). Parmi ces valeurs, certaines se factorisent complètement sur la base de facteurs.

Construisons d'abord la base de facteurs p_i.

On veut que $w(x) \equiv 0 \bmod p_i$

$\Leftrightarrow a^2 w(x) \equiv 0 \bmod p_i$

$\Leftrightarrow (a^2x + b)^2 \equiv N \bmod p_i$.

Il faut donc que [Mor 87] $(a, p_i) = 1$ pour que l'équivalence soit vraie et que N soit un résidu quadratique de p_i. Donc la base de facteurs contient des nombres premiers p_i tels que N soit un résidu quadratique de p_i. On ajoute à cet ensemble les nombres -1 et 2. On choisit les p_i les plus petits. C'est avec ces éléments que nous allons réaliser le crible de l'intervalle [-M, M[. Or

$$w(x) \equiv 0 \bmod p_i \Leftrightarrow (a^2x + b)^2 \equiv N \bmod p_i.$$

On calcule donc les solutions z_1 et z_2 de $z^2 \equiv N \bmod p_i$ pour chaque élément p_i de la base de facteurs. Alors

$$x_1 = \frac{z_1 - b}{a^2} \bmod p_i.$$

Idem pour x_2. On voit ici que x_1 et x_2 dépendent du polynôme w(x), contrairement à z_1 et z_2. Par conséquent z_1 et z_2 peuvent être calculés une fois et une seule en début de programme et stockés, alors que x_1 et x_2 sont recalculés pour chaque nouveau polynôme.

On sait alors que $w(x_1) \equiv w(x_2) \equiv 0 \bmod p_i$, pour tout élément p_i de la base. On cherche alors le plus petit élément x de l'intervalle [-M, M[congru à x_1. On sait que si

$$w(x) \equiv 0 \bmod p_i$$

alors
$$w(x + jp_i) = a^2(x+jp_i)^2 + 2b(x+jp_i) + c, \forall j \in \mathbb{Z}$$
$$w(x + jp_i) = a^2x^2 + a^2p_i (2xj + j^2p_i) + 2bx + 2bjp_i + c, \forall j \in \mathbb{Z}$$
$$w(x + jp_i) = w(x) + p_i (2a^2xj + a^2j^2p_i + 2bj), \forall j \in \mathbb{Z}$$
$$w(x + jp_i) \equiv w(x) \bmod p_i, \forall j \in \mathbb{Z}$$

et
$$w(x + jp_i) \equiv 0 \bmod p_i, \forall j \in \mathbb{Z}.$$

Donc en partant de x et en lui ajoutant p_i, $2p_i$, ..., on "tombe" sur des $w(x+jp_i)$ qui sont divisibles par p_i. A chacun de ces indices $x + jp_i$, on cumule les logarithmes de p_i qui ont permis d'"atteindre" cet indice. On fait de même avec les puissances α de p_i telles que $p_i^\alpha \leq B$.

Lorsqu'on a terminé l'opération de crible avec tous les éléments de la base de facteurs, on choisit parmi les valeurs de ce tableau celles qui sont proches de log |w(x)|. Les valeurs des w(x) correspondants se factorisent complètement sur la base de facteurs. Nous obtenons, avec un certain nombre de polynômes, un ensemble de valeurs w_i de w(x) qui se factorisent complètement sur la base. Nous les factorisons par un deuxième crible :

$$w_i = \prod_{j=1}^{k} p_i^{\alpha_{ij}}$$

et nous stockons ces factorisations dans une matrice. Chaque ligne i de cette matrice correspond à un w_i. Chaque colonne j correspond à un élément p_j de la base de facteurs. L'élément (i, j) de la matrice est l'exposant α_{ij} de p_j dans la factorisation de w_i.

Nous cherchons maintenant un ensemble I d'indices tels que le produit des w_i soit un carré. Cette opération peut se faire par une élimination de Gauss sur la matrice des factorisations dont les coefficients ont été transformés en leur classe dans $\mathbb{Z}/2\mathbb{Z}$. En effet si dans cette nouvelle matrice, on trouve une ligne qui soit combinaison linéaire (dans $\mathbb{Z}/2\mathbb{Z}$) d'autres lignes, alors la somme de cette ligne et des autres lignes est nulle.

Par conséquent, tous les exposants dans la factorisation du produit de toutes ces lignes sont des multiples de 2. Donc ce produit est un carré. Il ne reste alors plus qu'à calculer X et Y et enfin pgcd (X-Y, N) pour trouver un facteur (qu'on espère non trivial) de N.

Le nombre k d'éléments de la base de facteurs, la valeur de M (ou la longueur de l'intervalle de crible), le nombre de polynômes et d'autres paramètres seront présentés après l'algorithme formel, et étudiés plus tard.

Algorithme

Déterminer k (nombre d'éléments de la base),
p_k (le plus grand élément de la base),
B ($B = 2^s$ et $2^{s-1} < p_k \leq 2^s$),
M (la demi-longueur de l'intervalle de crible).
Calculer les k éléments p_i de la base de facteurs tels que p_i premier et N résidu quadratique de p_i : $(N/p_i) = 1$.
Calculer et stocker les solutions de $x^2 \equiv N \bmod p_i^{\alpha}$, pour tout α et p_i avec $p_i^{\alpha} \leq B$.
Tant que le nombre de w_i complètement factorisés est insuffisant *faire*
  Générer un polynôme $w(x) = a^2x^2 + 2bx + c$.
  Cribler [-M, M[avec chaque p_i et α tels que $p_i^{\alpha} \leq B$.
  Chercher les indices y de [-M, M[dont la valeur correspondante dans le tableau de crible soit proche de log $|w(x)|$.
  Factoriser les w(y) correspondants.
Fin tant que
Effectuer une élimination de Gauss sur la matrice des w_i factorisés
Calculer le pgcd des lignes dépendantes.
Factoriser N.

Fin algorithme.

Figure 4.1. Algorithme du crible quadratique

C'est l'algorithme de base du crible quadratique multipolynomial.

3. PARALLÉLISATION EN MÉMOIRE DISTRIBUÉE ILLIMITÉE DE L'INITIALISATION DE L'ALGORITHME DU MPQS

Nous allons présenter l'étude de la parallélisation théorique de chacune des étapes de l'algorithme du crible quadratique multipolynomial, sur un multiprocesseur hypercube dont la mémoire distribuée est illimitée. Notre but est d'extraire le parallélisme de chaque étape de l'algorithme. Cette étude permettra une implantation pratique, après adaptation aux paramètres de la machine-cible.

Nous essayons d'extraire le parallélisme dans chacune des étapes de l'algorithme :

- l'initialisation comportant
  - le calcul de k : le nombre d'éléments de la base,

- le calcul de p_k : une évaluation du plus grand élément de la base,
- le calcul de B : une borne supérieure du plus grand élément pour le crible,
- le calcul de M : la demi-longueur de l'intervalle de crible,
- le calcul des k éléments de la base,
- les racines carrées de N par rapport aux éléments de la base,

• la génération des polynômes w(x),
• le crible de l'intervalle [-M, M[,
• la factorisation des w(x) candidats à une factorisation complète sur la base,
• l'élimination de Gauss et le calcul des pgcd pour l'extraction des facteurs de N.

La machine-cible théorique de notre étude de parallélisation est un FPS T40 où la mémoire est illimitée sur chaque nœud.

Le crible quadratique nécessite un système de multiprécision entière. Jean-Louis Roch a développé, pour le FPS T40, un système de calcul formel parallèle, PaC [Roc 89], incluant les opérations en multiprécision entière, nécessaires à l'implantation de l'algorithme du crible quadratique. Nous supposons donc que la machine-cible théorique est pourvue de ce système de calcul en multiprécision entière.

3.1. PRÉSENTATION DE L'INITIALISATION

Cette étape permet de déterminer les paramètres de l'algorithme, c'est-à-dire le nombre k d'éléments de la base de facteurs, une estimation du plus grand élément p_k de cette base, la borne B qui limite la taille des éléments p_i avec lesquels on crible et la longueur 2M de l'intervalle de crible. Ce sont les paramètres de base. Puis cette étape génère les k éléments de la base et les solutions des équations $x^2 \equiv N \bmod p_i^\alpha$, avec $p_i^\alpha \leq B$, c'est-à-dire les racines carrées de N.

Comment les paramètres N, k, B et M sont-ils connus de tous les processeurs ?

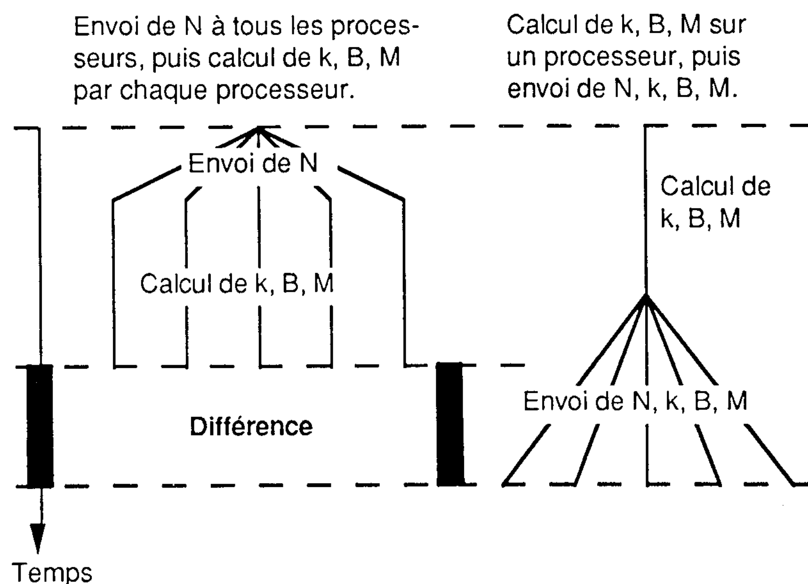


Figure 4.2. Calcul de k, B et M et diffusion de ces informations.

Rappelons que le calcul de k, B et M ne dépend que de la valeur de N. Nous avons donc deux stratégies pour que tous les processeurs connaissent les valeurs de k, B et M :

• soit un processeur diffuse à tous les autres la valeur de N, et tous les processeurs calculent ensuite k, B et M,
• soit un processeur calcule k, B et M à partir de N, et diffuse ensuite à tous les processeurs les valeurs de k, B et M.

Or, le système de calcul en multiprécision entière de PaC [Roc 89] ne permet pas le calcul du logarithme d'un entier écrit en précision entière illimitée et qu'il faut, lorsqu'on parle de l'envoi de N, comprendre l'envoi de N et de son logarithme.

Le coût du calcul de k, B et M est le même dans les deux stratégies. Quant aux coûts des communications, ils sont inférieurs dans le cas où on diffuse N (puisque dans l'autre cas, on diffuse N, k, B et M), car le coût des communications croît avec la taille des messages.

Le schéma de la figure 4.2 est certes caricatural, car l'envoi de N et 3 entiers n'est que très légèrement plus coûteux en temps que l'envoi de N. Mais nous voulons faire remarquer que des calculs distribués peuvent faire perdre du temps par rapport à ces mêmes calculs centralisés. Ici, le calcul réparti est plus coûteux que le calcul centralisé.

3.2. LE NOMBRE k D'ÉLÉMENTS DE LA BASE DE FACTEURS

Les avis sont partagés sur la valeur de k. Il est évident que plus k est grand, plus les w(x) ont une probabilité élevée de se factoriser complètement. En revanche lorsque k augmente, le temps nécessaire au crible augmente, ainsi que le temps de factorisation des valeurs de w(x), car il faut obtenir plus de lignes pour la matrice des factorisations.

F. Morain préconise $k = L^{1/\sqrt{8}}$ avec $L = L(x) = \exp\sqrt{\log x \log\log x}$. Cette valeur de k est réaliste pour des entiers N assez petits ($< 10^{40}$). Lorsque N croît vers 10^{100}, on a alors k = 270 000, ce qui entraîne des temps trop importants de crible et de factorisation des valeurs de w(x), et réclame un espace mémoire trop grand. C. Pomerance et al., dans [PST 88], sont plus raisonnables et estiment qu'une base de 100 000 nombres est suffisante. Pour ma part, je suivrai Caron et Silverman, dans [CaS 88], qui donnent des valeurs de k pour $N \leq 10^{87}$. En extrapolant pour $N \in \{10^{87}, \ldots, 10^{100}\}$, on trouve que k = 35 000 est une valeur cohérente pour $N = 10^{100}$. Ceci revient à poser $k = L^{0,2989}$ environ.

En fait ce paramètre dépend de la taille du nombre N, mais aussi de l'architecture et des performances (puissance, espace mémoire...) de la machine sur laquelle le MPQS est implanté.

Le tableau suivant donne des valeurs de k en fonction de la taille de N (nombre de chiffres). Les valeurs pour $N \in \{10^{21}, \ldots, 10^{60}\}$ sont issues de nos expériences. Celles pour $N \in \{10^{60}, \ldots, 10^{87}\}$ proviennent de [Cas 88], et les valeurs pour $N \in \{10^{90}, \ldots, 10^{100}\}$ en sont une extrapolation.

Nombre de chiffres de N	k	Nombre de chiffres de N	k
21	110	63	3 500
24	160	66	4 000
27	230	69	5 000
30	330	72	6 000
33	500	75	7 000
36	725	78	8 300
39	1 050	81	10 000
42	1 200	84	12 000
45	1 400	87	15 000
48	1 600	90	18 500
51	1 900	93	22 500
54	2 200	96	27 000
57	2 550	100	35 000
60	3 000		

Figure 4.3. Evaluation de k en fonction de la taille de N

3.3. ESTIMATION DU PLUS GRAND ÉLÉMENT p_k DE LA BASE

Chaque élément p_i de la base de facteur doit être premier, et N doit être un résidu quadratique de p_i. Or, statistiquement, on sait que pour obtenir les k premiers nombres premiers, il faut en moyenne tester les k log k premiers entiers. On sait aussi, toujours statistiquement, qu'un entier est un résidu quadratique d'un autre entier avec une probabilité 1/2. Donc, il faudra en moyenne tester 2k log (2k) entiers pour espérer trouver les k éléments de la base de facteurs.

3.4. LA BORNE B

Si, dans l'opération de crible, on ne crible qu'avec les éléments p_i de la base, on n'obtient que les w_i dont les exposants des p_i dans la factorisation sont 1 ou 0. Or certains w(x) ont des exposants de certains p_i égaux à 2, 3, 4..., ceci pour les petites valeurs des p_i. Il serait dommage d'oublier ces w_i.

Pour ce faire, on définit une borne B, plus grande que le plus grand élément de la base de facteurs. Cette borne sert à limiter l'exposant des éléments p_i de la base lors des étapes de crible et factorisation des w_i. Ainsi l'opération de crible est réalisée avec toutes les puissances α de tous les éléments p_i de la base tels que $p_i^\alpha \leq B$. Ainsi on oublie moins de w_i complètement factorisés.

Ceci implique bien évidemment un léger surcroît de travail lors de la résolution des équations $x^2 \equiv N \bmod p_i^\alpha$, car pour les petites valeurs de p_i, il faut résoudre α équations, alors que pour les plus grandes valeurs, il n'y en a qu'une à résoudre.

3.5. LA LONGUEUR 2M DE L'INTERVALLE DE CRIBLE

De la même manière que pour le nombre k d'éléments de la base, les avis sont partagés sur la longueur de l'intervalle de crible. Cependant, il est certain que plus l'intervalle est grand, plus on a de chances de trouver des w_i complètement factorisés, pour un coût très légèrement supérieur. En effet, tous les calculs des solutions ont été faits, le seul surcroît de travail concerne le parcours de l'intervalle [-M, M[qui est plus grand.

Cependant, il ne faut pas oublier que plus l'intervalle est grand, plus l'écart entre les valeurs extrêmales de w(x) sur cet intervalle est important. Et donc, on ne pourra plus considérer le polynôme w(x) comme donnant des valeurs à peu près constantes en fonction de a sur l'intervalle [-M, M[. On doit alors introduire des évaluations de w(x) différentes selon les sous-intervalles de [-M, M[.

F. Morain donne pour M la valeur suivante dans [Mor 87]

$$M = \begin{cases} 2^m \text{ avec } M \leq \dfrac{\sqrt{2N}}{p_k^2} < 2M \text{ si } M < 5B \\ 5B \text{ sinon, pour éviter de passer trop de temps avec le même polynôme.} \end{cases}$$

Les valeurs de M produites par cette formule sont bonnes pour des $N < 10^{40}$ environ.

Caron et Silverman [CaS 88] donnent des valeurs pour $N \in \{10^{60}, \ldots, 10^{87}\}$. On extrapole pour $N \in \{10^{90}, \ldots, 10^{100}\}$.

Et on remarque que le produit 2M.k est une fonction exponentielle du nombre de chiffres de N. Ceci signifie que, expérimentalement, l'espace mémoire occupé par les variables principales de

l'algorithme (base de facteurs et intervalle de crible) croît exponentiellement en fonction du nombre de chiffres de N.

Nous présentons, dans le tableau suivant, la taille de l'intervalle de crible en fonction du nombre de chiffres de N.

Nombre de chiffres de N	Longueur 2M de l'intervalle	Nombre de chiffres de N	Longueur 2M de l'intervalle
21	6 000	63	200 000
24	7 500	66	250 000
27	9 000	69	300 000
30	10 500	72	350 000
33	13 500	75	425 000
36	16 000	78	550 000
39	20 000	81	750 000
42	27 000	84	1 000 000
45	35 000	87	1 300 000
48	48 000	90	1 700 000
51	63 000	93	2 200 000
54	85 000	96	2 800 000
57	115 000	100	3 900 000
60	150 000		

Figure 4.4. Evaluation de la longueur de l'intervalle de crible en fonction de la taille de N

La figure suivante donne la courbe du produit 2Mk en fonction du nombre de chiffres de N.

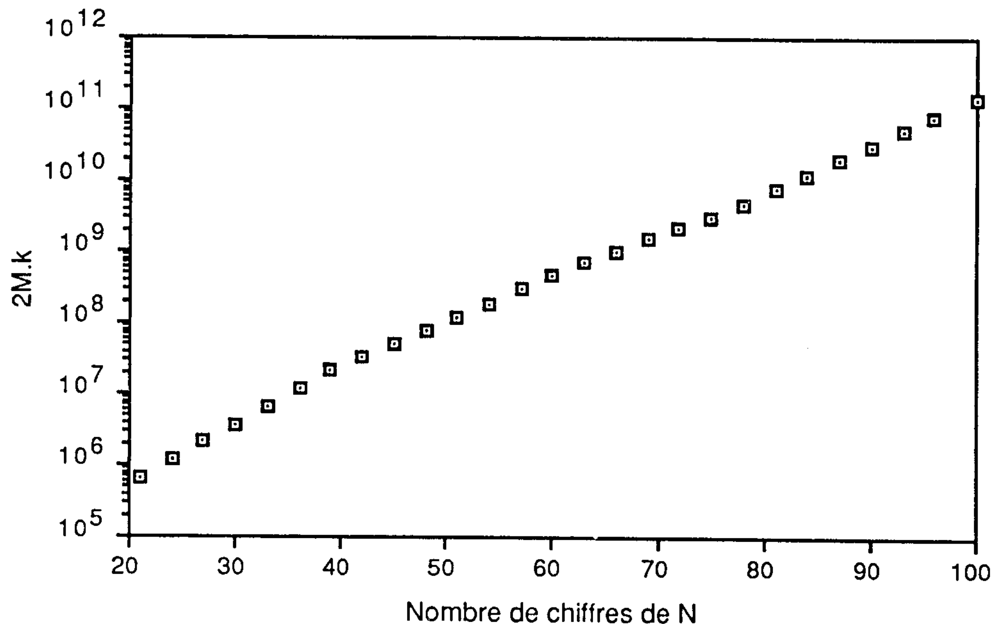


Figure 4.5. Produit 2M.k en fonction du nombre de chiffres de N

De cette courbe, on déduit que $2M.k \approx 59\,900\ N^{0,0636}$. Mais l'intervalle et la base occupent une place mémoire importante. Trouvons un compromis. Minimisons l'espace occupé par la base et l'intervalle. Chaque élément de la base nécessite environ 13 octets. Chaque élément de l'intervalle de crible occupe 1 octet. On veut donc minimiser la fonction $2M + 13k$ sous la contrainte $Mk = 29\,950\ N^{0,0636}$ pour $N = 10^{100}$:

$$\begin{cases} \min (2M + 13k) \\ M.k = 29950.N^{0,0636} \text{ avec } N = 10^{100} \end{cases}$$

c'est-à-dire

$$\begin{cases} \min (2M + 13k) \\ M.k = 6,86114862.10^{10} \end{cases}$$

Ce qui revient à minimiser la fonction de k :

$$\frac{1,372229724.10^{11}}{k} + 13k$$

Le minimum est atteint pour

$$k = \sqrt{\frac{1,372229724.10^{11}}{13}} = 102\ 740,5142 \approx 100\ 000$$

La valeur de M découle immédiatement :

$$M = \frac{6,86114862.10^{10}}{102\ 740,5142} = 667\ 813,3424 \approx 670\ 000.$$

Cette configuration utilise environ 2,6 Mo, alors que la configuration de Caron et Silverman dépasse les 4 Mo. Cependant, aucune des 2 solutions n'est meilleure dans l'absolu. Il faut relativiser par rapport aux machines utilisées.

En supposant que le temps nécessaire au crible d'un sous-intervalle (de longueur fixée) de l'intervalle de crible [-M, M[avec une puissance α d'un élément p_i de la base soit indépendant de p_i et de α, alors le temps de crible est proportionnel au produit 2M.k. Il est donc exponentiel en fonction du nombre de chiffres de N.

3.6. LE CALCUL DES k ÉLÉMENTS DE LA BASE DE FACTEURS

Ces éléments sont des nombres premiers tels que N soit un résidu quadratique de chacun d'eux. Ces nombres vont servir à factoriser complètement des valeurs de polynômes w(x). Il faut que ces nombres soient les plus petits possibles pour que les w(x) se factorisent complètement sur l'ensemble de ces k nombres. L'algorithme séquentiel pour les générer est le suivant :

```
Algorithme
|     base_de_facteurs = {-1, 2}; p = 3; nb_élts_dans_base = 2;
|     tant que nb_élts_dans_base < k faire
|     |     Si p premier et N résidu quadratique de p alors
|     |     |     base_de_facteurs = base_de_facteurs ∪ {p};
|     |     |     nb_élts_dans_base = nb_élts_dans_base + 1;
|     |     fin si
|     |     p = p + 2;
|     fin tant que
fin algorithme
```

Figure 4.6. Algorithme séquentiel de génération des k éléments de la base de facteurs

Il y a plusieurs possibilités pour paralléliser cet algorithme dans un environnement distribué. Nous proposons deux solutions :

- comme le calcul est constitué de deux phases (test de primalité et calcul de résidu quadratique), nous affectons chacune de ces deux phases à un processeur,
- les P processeurs de l'hypercube cherchent et trouvent k/P éléments chacun environ, en répartissant les données le plus équitablement possible de manière à terminer en même temps.

Cette deuxième solution est meilleure, car les processeurs sont mieux utilisés. Mais elle pose des problèmes de calcul d'un résultat global à partir de résultats locaux, ce qui entraine des communications.

Nous donnons à la fin de ce paragraphe un aperçu de l'implantation mathématique du calcul de ces éléments.

3.6.1. Calcul des k éléments sur 2 processeurs

Le calcul des k éléments de la base se décompose en deux parties : le test de primalité et la valeur du résidu quadratique (N/p) (symbole de Jacobi). Comme le test de primalité est plus coûteux que le calcul du symbole de Jacobi, on effectue d'abord ce dernier, et on ne teste la primalité de p que si le symbole de Jacobi (N/p) vaut 1. Mais cette répartition est mauvaise : d'une part on n'utilise que deux processeurs, d'autre part il y beaucoup de communications à effectuer, si on veut que les deux processeurs travaillent en même temps. En effet, le premier processeur calcule un ou des symboles de Jacobi, et dès qu'il trouve un p tel que (N/p) = 1, il l'envoie à l'autre processeur qui teste sa primalité. Le nombre de communications est donc au minimum k, dans le cas où tous les p envoyés sont premiers, et tend plutôt vers k log k (en effet, il y a environ x/log x nombres premiers inférieurs à x, et donc il faut en moyenne k log k entiers pour trouver environ k nombres premiers).

Etudions les communications entre ces deux processeurs. Le premier processeur cherche un nombre p tel que N soit un de ses résidus quadratiques. Pour ce faire, il risque d'être obligé de tester plusieurs p. Ensuite seulement, il envoie un p convenable, i.e. (N/p) =1, au deuxième processeur. Après cette initialisation s'instaure un pipeline [HwB 84]. Si on veut que ce pipeline fonctionne au mieux, il faut que les temps des opérations dans le pipeline soient sensiblement égaux, afin qu'il n'y ait pas d'attente sur un des opérateurs. Le problème provient de la répartition des résidus quadratiques et des nombres premiers, et du temps de calcul pour chacune de ces deux phases. Un entier a une probabilité 1/2 en moyenne d'être un résidu quadratique, ce qui est bien supérieur à sa probabilité d'être premier. D'autre part, le temps de calcul d'un résidu quadratique est court devant celui d'un test de primalité. Donc le premier processeur aura un temps de cycle plus court que le deuxième processeur. Il n'y aura donc pas d'équilibre, car les opérations du pipeline sont de durées différentes.

3.6.2. Calcul des k éléments sur P processeurs

La deuxième solution de parallélisation du calcul des k éléments de la base consiste à répartir les calculs entre les P processeurs. On veut alors que chaque processeur calcule environ k/P éléments de la base en évitant que deux processeurs ne trouvent les mêmes éléments. Il faut donc répartir les données afin d'éviter toute redondance dans les calculs et dans les résultats. Or les données sont l'intervalle d'entiers $\{2, \ldots, p_k\}$ dans lequel on espère trouver les k éléments de la base. Le processeur 0 a le sous-intervalle $\{2, \ldots, p_k/P\}$. Le processeur PE_i, $0 \leq i \leq P-1$, a le sous-intervalle

$$\left[i\frac{p_k}{P} + 1, (i+1)\frac{p_k}{P}\right].$$

Chaque processeur cherche tous les éléments de la base dans son sous-intervalle. En moyenne, chaque processeur PE_i trouvera k/P éléments. En réalité, il en trouvera un nombre voisin, k_i. A la fin de cette étape, on a $\sum_{i=0}^{P-1} k_i$, que l'on note $\sum k_i$, éléments dans la base.

- Si $\sum k_i = k$, alors le calcul est terminé.
- Si $\sum k_i > k$, alors les processeurs ont calculé trop d'éléments, et on ne prend que les k premiers. On supprime donc les $\sum k_i$ - k supplémentaires en commençant par les plus

grands, sur le processeur P-1, puis si nécessaire sur les processeurs P-2, P-3, ... On ne garde que les k plus petits éléments calculés pour la base de facteurs.

- Si $\sum k_i < k$, alors il manque des éléments.

La suppression d'éléments est facile (en modifiant la valeur maximale d'un indice). Mais la génération de nouveaux éléments l'est beaucoup moins.

Si le nombre d'éléments manquants est petit, un seul processeur cherche ces éléments. Sinon, les processeurs se définissent tous un nouveau sous-intervalle pour trouver les éléments manquants. Le nouvel intervalle débute en p_k+1 et sa longueur dépend du nombre d'éléments manquants. Il est réparti en sous-intervalles et chaque processeur cherche dans son propre sous-intervalle tous les éléments appartenant à la base de facteurs. L'estimation de p_k est mise à jour. Pour être certain que cet algorithme s'arrête après un nombre fini de boucles de ce type, on admet que si l'intervalle ou le nombre d'éléments manquants est trop petit, un processeur s'occupe seul de la recherche des éléments manquants. Il l'effectue sans se définir d'intervalle de recherche, mais seulement en comptant le nombre d'éléments qu'il trouve, et en s'arrêtant lorsqu'il atteint un total de k.

Après cette étape, on calcule une nouvelle fois le nombre total d'éléments trouvés, et on recommence soit l'étape de destruction si on en a trop générés, soit cette même étape de génération s'il en manque.

Détaillons les étapes de communication qui constituent ce calcul :
- regroupement et sommation sur le processeur PE_0 de P nombres provenant chacun d'un des P processeurs pour le calcul de $\sum k_i$,
- diffusion par ce processeur PE_0 de ce nombre $\sum k_i$, afin que chaque processeur puisse définir le nouvel intervalle et le sous-intervalle propre de cet intervalle où chercher les éléments manquants.

3.6.3. Sommation sur un processeur de P nombres répartis sur P processeurs

La sommation $\sum k_i$ du nombre d'éléments effectivement calculés se fait en log P étapes sur un hypercube à P processeurs, en utilisant des communications et des réductions (sommes) sur chaque dimension.

Exemple sur un hypercube de dimension 4 en obtenant la somme sur le processeur 0.

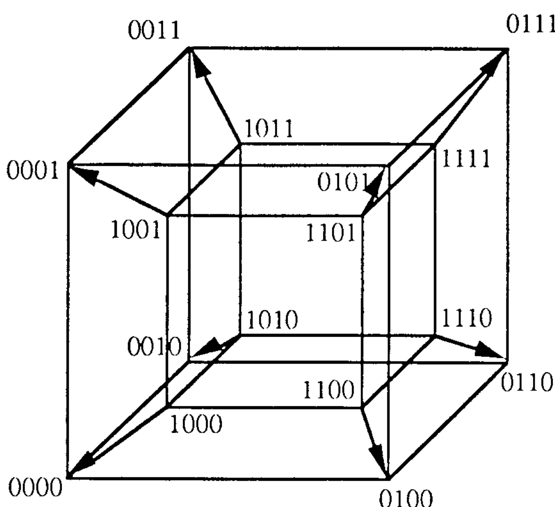


Figure 4.7. Première étape : dimension 4

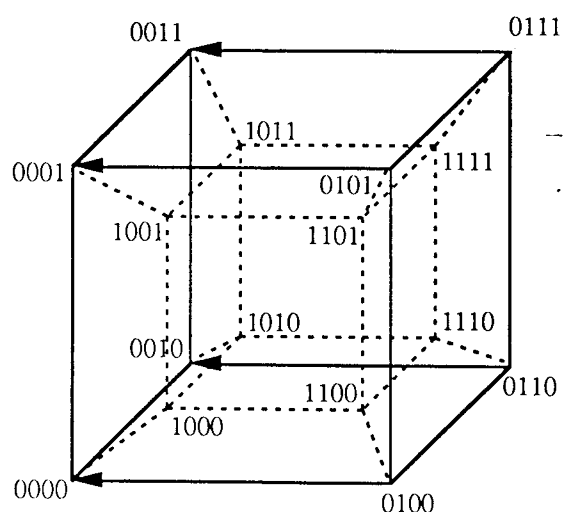


Figure 4.8. Deuxième étape : dimension 3

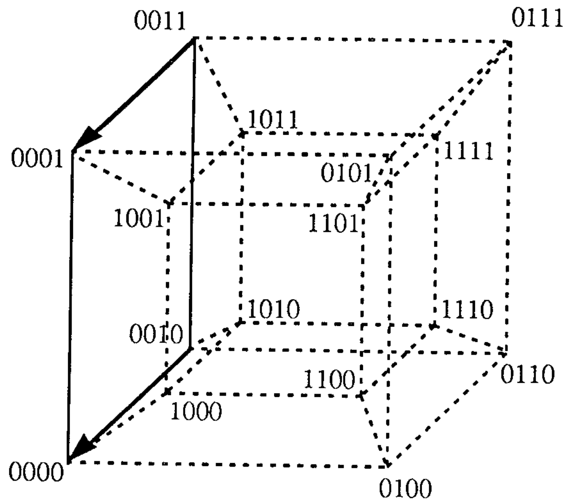


Figure 4.9. Troisième étape : dimension 2

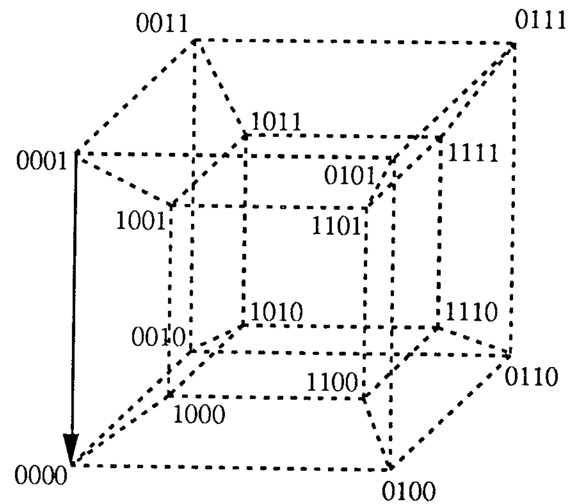


Figure 4.10. Quatrième étape : dimension 1

Ceci est en fait un regroupement d'informations (gathering) sur un arbre de recouvrement d'un hypercube de dimension 4. On voit qu'à chaque étape, on diminue de 1 la taille de l'hypercube dans lequel on somme. Après un nombre d'étapes égal à la dimension initiale de l'hypercube, le processeur 0 a le nombre total d'éléments calculés dans le réseau.

Cette technique est la plus rapide.

3.6.4. Diffusion d'un nombre à tous les processeurs

Puis comment les processeurs peuvent-ils calculer le nouvel intervalle de recherche ? Il faut qu'ils connaissent $\sum k_i$. Le processeur 0 diffuse ce total à tous les processeurs en suivant un arbre binaire de diffusion de racine lui-même. Cette étape se fait encore en log P communications, comme les figures 4.11 et 4.12 le montrent, respectivement dans le cas où un processeur peut envoyer des informations à un seul voisin à la fois (programmation en C sur le FPS T40) et dans le cas où un processeur peut communiquer avec plusieurs voisins simultanément (programmation en OCCAM sur le FPS T40) :

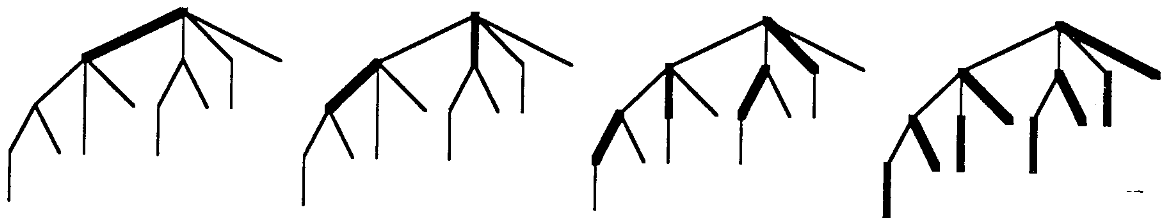


Figure 4.11. Diffusion sur un arbre de recouvrement d'un 4-cube (communication possible avec un seul voisin à la fois)

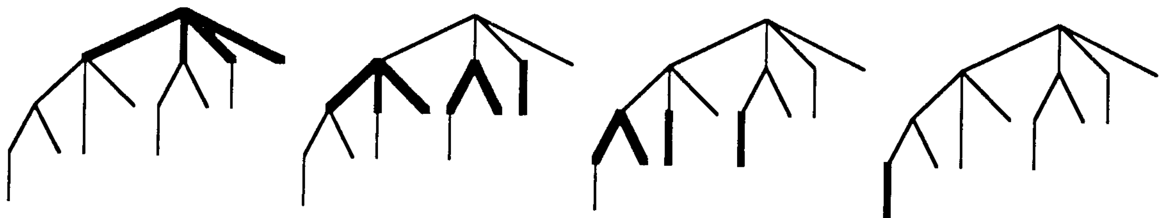


Figure 4.12. Diffusion sur un arbre de recouvrement d'un 4-cube (communication possible avec tous les voisins à la fois)

Une modélisation de la diffusion est présentée dans [Fra 89]. Là encore, il n'est pas possible de communiquer cette information plus rapidement.

3.6.5. Implantation mathématique du calcul des k éléments

Nous présentons maintenant l'implantation de la partie mathématique du calcul des k éléments p_i de la base.

Le calcul du symbole de Jacobi (N/p) n'est pas effectué en précision entière infinie, bien que N soit codé en précision entière infinie, car $p < 2^{31}-1$ et tient donc sur 32 bits. En effet,

$$\left(\frac{N}{p}\right) = \left(\frac{N \bmod p}{p}\right)$$

et par conséquent, une seule opération modulaire est effectuée en précision entière infinie : N mod p. Le résultat de cette opération est inférieur à p. Donc la suite des calculs peut être effectuée en précision 32 bits.

Quant au test de primalité, la méthode la plus rapide consiste à utiliser le crible d'Eratosthène qui est très rapide en parallèle. Chaque processeur sait qu'il ne criblera pas au-delà de $p_k + \varepsilon$. (On prend une marge de sécurité ε). Et en fait, on effectue d'abord l'étape de crible, puis on calcule le symbole de Jacobi. Il est vrai que si on teste individuellement la primalité de chaque candidat, le test de primalité prend plus de temps que le calcul du symbole de Jacobi, mais avec le crible d'Eratosthène, les rôles sont inversés.

3.6.6. Conclusion

En découpant l'intervalle de recherche en sous-intervalles, en itérant sur de nouveaux sous-intervalles pour trouver les k éléments et en utilisant les liens de l'hypercube pour communiquer, on obtient un algorithme efficace.

3.7. LE CALCUL DES RACINES CARRÉES DE N MODULO LES ÉLÉMENTS DE LA BASE

Le but de cette étape est de résoudre les équations $x^2 \equiv N \bmod p_i^\alpha$ pour toutes les puissances α des k éléments p_i de la base tels que $p_i^\alpha \leq B$. On se rend compte alors que certains éléments p_i ont une valeur maximale $\alpha_{max}(p_i)$ de α égale à 1, alors que cette valeur est strictement supérieure à 1 pour d'autres, les plus petits éléments en l'occurrence.

Chaque élément p_i induit $\alpha_{max}(p_i)$ équations à résoudre. Or, pour résoudre $x^2 \equiv N \bmod p_i^{\alpha+1}$, il faut connaître une solution de $x^2 \equiv N \bmod p_i^\alpha$. Ceci implique que toutes les équations concernant un même p_i doivent être résolues sur un seul processeur, afin d'éviter des communications.

Avec un total de k éléments dans la base, combien y a-t-il d'équations à résoudre ? Pour chaque élément p_i, il y a $\lfloor \log_{p_i} B \rfloor$ équations à résoudre. Dès que $p_i > \sqrt{B}$, il n'y a qu'une seule équation à résoudre pour chaque p_i.

Ce qui fait un total de

$$T = \lfloor \log_2 B \rfloor . \left(\frac{\left(\frac{N}{2}\right) + 1}{2}\right) + \sum_{i=2}^{k} \lfloor \log_{p_i} B \rfloor .$$

Ceci peut aussi s'écrire

$$T = \begin{cases} \sum_{i=2}^{k} \lfloor \log_{p_i} B \rfloor \text{ si } \left(\frac{N}{2}\right) = -1 \\ \sum_{i=1}^{k} \lfloor \log_{p_i} B \rfloor \text{ si } \left(\frac{N}{2}\right) = +1 \end{cases}$$

En ce qui concerne la mathématique de cette partie, il existe plusieurs techniques provenant de diverses sources pour la recherche des racines carrées modulo un nombre premier :

- les suites de Lucas [Rie 87],
- l'algorithme de Shanks [Knu 81],
- les idées données par Hardy and Wright [HaW 79].

En ce qui concerne l'algorithmique, on rencontre plusieurs problèmes dont la répartition des données, la répartition des calculs, le stockage des racines.

Notre but est de réaliser cette sous-étape de manière efficace, c'est-à-dire de sorte que chaque processeur ait la même somme de travail à effectuer : le même nombre d'équations. Nous avons donc deux étapes :

- la répartition des k éléments de la base pour que chaque processeur ait le même nombre d'équations à résoudre :
  - elle peut se préparer en cours de calcul des k éléments, en spécifiant sur chaque processeur, le nombre d'équations induites par les éléments qu'il a calculés. Lorsque le processeur PE_0 calcule le nombre total d'éléments générés, il calcule en même temps le nombre total d'équations à résoudre. Lorsque ce processeur décrète que le nombre d'éléments est suffisant, on peut immédiatement répartir ces éléments entre les processeurs, de sorte que tous les processeurs aient le même nombre d'équations à résoudre ;
  - elle peut se faire en parallèle avec d'autres opérations : après avoir extrait les éléments de $\{2, \ldots, p_k\}$, et avoir compté le nombre d'éléments trouvés, on désigne un processeur pour trouver les éléments manquants, pendant que d'autres se répartissent les éléments trouvés ;
- la résolution de ces équations, indépendamment sur chaque processeur.

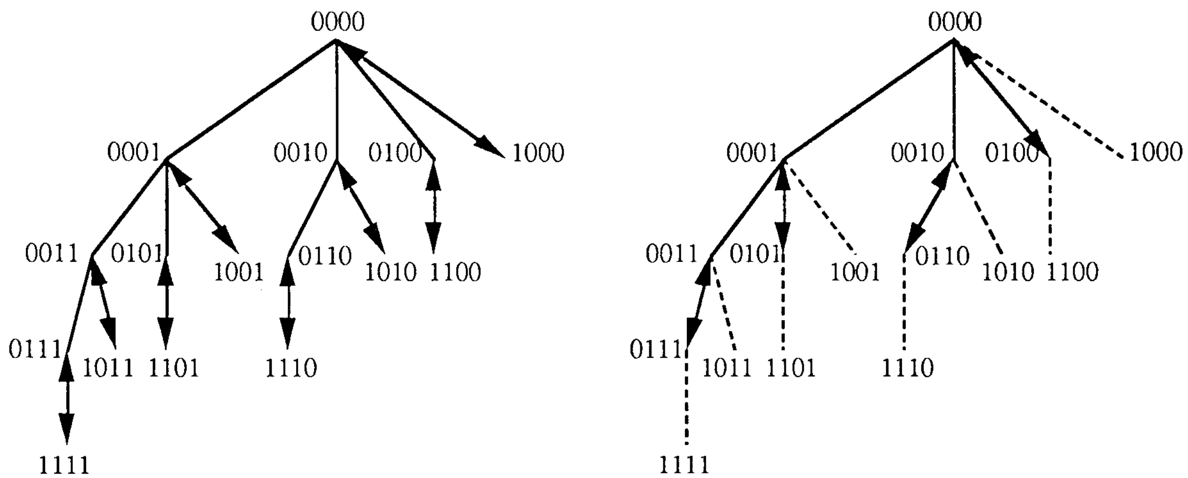
3.7.1. Répartition générale des k éléments en fonction du nombre d'équations à résoudre

Le nombre total d'équations dépend du nombre d'éléments de la base et de ces éléments. Pour le calculer, on ajoute une information aux messages envoyés par les processeurs vers le processeur PE_0, stipulant le nombre d'éléments générés (voir § 2.1.5) : on accole au nombre d'éléments calculés, le nombre d'équations que ces éléments impliquent en fonction de la plus grande valeur de α pour chacun d'eux.

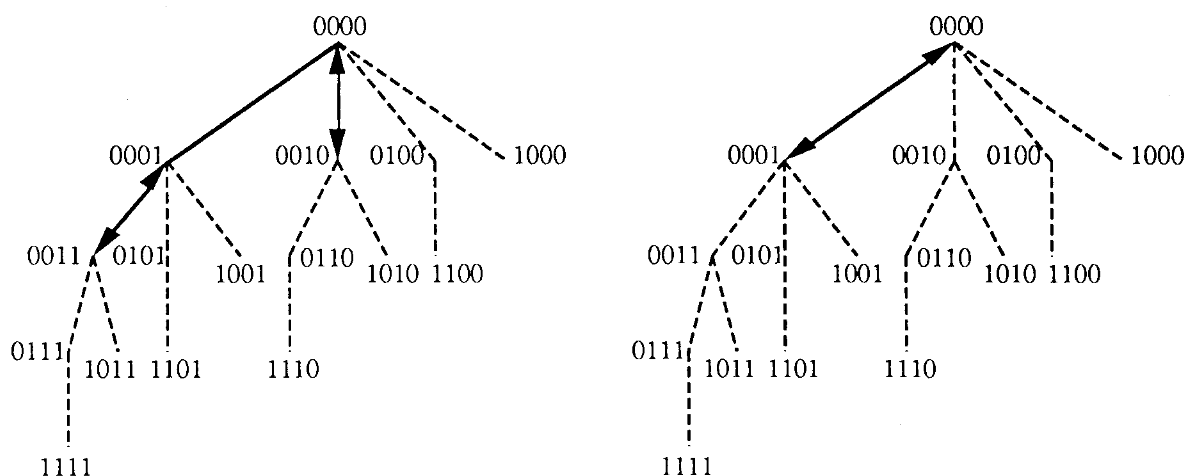
Lorsque le processeur 0 reçoit ces deux nombres et qu'il somme ses propres valeurs à celles-ci, il connaît le nombre total d'éléments de la base à cet instant, et le nombre total d'équations induites. Une fois que la base est complètement générée (par morceau sur chaque processeur), on connait tout de suite le nombre d'équations à résoudre.
On répartit donc les éléments de la base de sorte que tous les processeurs aient le même nombre d'équations à résoudre. Chaque processeur sait combien d'équations il doit résoudre et il en déduit le nombre d'éléments de la base qu'il doit garder.

Cependant, il doit connaître pour chaque élément la valeur maximale α_{max} de α et donc tester élément par élément cette valeur avant la communication. De plus, cette communication est un échange et peut se faire en envoi ou en réception suivant que ce processeur a trop ou trop peu d'éléments.

Ainsi, dans l'hypercube, les processeurs se répartissent les éléments en utilisant un arbre de diffusion, où, lors de la première communication, les feuilles s'attribuent le nombre correct d'éléments en effectuant des échanges avec leur père, puis se retirent des tractations. On diminue de 1 la dimension de l'hypercube en supprimant les feuilles, et on recommence les échanges sur le nouvel arbre. Sur un hypercube de P processeurs, cette étape nécessite log P communications.



Figures 4.13 et 4.14. Echanges d'éléments de la base sur un arbre de recouvrement d'un 4-cube (étapes 1 et 2).



Figures 4.15 et 4.16. Echanges d'éléments de la base sur un arbre de recouvrement d'un 4-cube (étapes 3 et 4).

Il est long et fastidieux, et surtout coûteux, de tester la valeur maximale de α avant l'envoi. Ne pourrait-on pas regrouper les éléments ayant $\alpha_{max} > 1$ sur un seul processeur, le processeur 0 par exemple, puisque c'est lui qui a les plus petits éléments de la base ?

3.7.2. Répartition des k éléments sur P processeurs, le processeur PE_0 traitant tous les éléments ayant $\alpha_{max} > 1$

Nous allons montrer que cela est possible selon les valeurs de N et du nombre de processeurs : plus le nombre de processeurs est élevé, plus N doit être grand pour que le processeur PE_0 ait tous les éléments ayant $\alpha_{max} > 1$. En effet, plus N est grand, plus le nombre d'éléments de la base de facteurs est élevé.

Si cela est possible, il faut donc que le plus grand élément ayant $\alpha_{max} > 1$ soit sur le processeur PE_0. Maisil est possible aussi que le processeur PE_0 ait des éléments ayant $\alpha_{max} = 1$. Or le plus grand élément du processeur PE_0 est le $(k/P)^{ème}$ élément de la base triée en ordre croissant. Soit K cet élément. Et il faut que K, ou un élément plus petit, ait $\alpha_{max} > 1$. Il faut surtout que tout élément supérieur à K ait $\alpha_{max} = 1$.

Or $K \approx \frac{k}{P}\log\left(\frac{k}{P}\right)$.

Prenons cette valeur comme valeur de K : $K = \frac{k}{P}\log\left(\frac{k}{P}\right)$.

On veut que

$$\lfloor \log_K B \rfloor = 2$$

$$\Leftrightarrow \left\lfloor \frac{\log B}{\log K} \right\rfloor = 2$$

Or $B \approx p_k \approx 2k\log(2k)$.

Donc
$$\left\lfloor \frac{\log\,(2k\log(2k))}{\log\left(\frac{P}{k}\log\left(\frac{k}{P}\right)\right)} \right\rfloor = 2$$

$$\Leftrightarrow \frac{\log 2 + \log k + \log\log\,(2k)}{\log k - \log P + \log(\log k - \log P)} \geq 2$$

$$\Leftrightarrow \log 2 + \log k + \log\log(2k) \geq 2\log k - 2\log P + 2\log(\log k - \log P)$$

$$\Leftrightarrow \log 2 + \log\log(2k) \geq \log k - 2\log P + 2\log(\log k - \log P)$$

En posant $x = \log k$ et $y = \log P$, l'équation devient

$$\log 2 + \log\log(2 + x) - x + 2y - 2\log(x - y) \geq 0.$$

En fixant la valeur de y (correspondant à 2, 4, 8, 16, 32... processeurs), on trouve la valeur de x pour laquelle l'équation change de signe, c'est-à-dire la valeur de k, et par conséquent de N au-delà de laquelle le plus grand élément ayant $\alpha_{max} \geq 2$ est assurément sur le processeur 0 (il peut y en avoir de plus grands que lui avec $\alpha_{max} = 1$ sur ce processeur).

Taille de N	k	x=log k	P=2 y=0,69	P=4 y=1,38	P=8 y=2,07	P=16 y=2,77	P=32 y=3,47	P=64 y=4,16	P=128 y=4,85	P=256 y=5,54
21	110	4,70	-3,52	-1,98	-0,4	1,3	+	+	+	+
24	160	5,07	-3,94	-2,41	-0,85	0,77	+	+	+	+
27	230	5,44	-	-	-	0,30	+	+	+	+
30	330	5,80	-	-	-	-0,14	1,49	+	+	+
33	500	6,21	-	-	-	-	0,97	+	+	+
36	725	6,59	-	-	-	-	0,50	+	+	+
39	1 050	6,96	-	-	-	-	0,06	+	+	+
42	1 200	7,09	-	-	-	-	-0,09	1,47	+	+
45	1 400	7,24	-	-	-	-	-	1,28	+	+
48	1 600	7,38	-	-	-	-	-	1,11	+	+
51	1 900	7,55	-	-	-	-	-	0,91	+	+
54	2 200	7,70	-	-	-	-	-	0,72	+	+
57	2 550	7,84	-	-	-	-	-	0,56	+	+
60	3 000	8,00	-	-	-	-	-	0,37	+	+
63	3 500	8,16	-	-	-	-	-	0,18	+	+
66	4 000	8,29	-	-	-	-	-	0,03	+	+
69	5 000	8,52	-	-	-	-	-	-0,23	+	+
72	6 000	8,70	-	-	-	-	-	-	1,08	+
75	7 000	8,85	-	-	-	-	-	-	0,91	+
78	8 300	9,02	-	-	-	-	-	-	0,71	+
81	10 000	9,21	-	-	-	-	-	-	0,49	+
84	12 000	9,39	-	-	-	-	-	-	0,28	+
87	15 000	9,62	-	-	-	-	-	-	0,02	+
90	18 500	9,83	-	-	-	-	-	-	-0,22	+
93	22 500	10,02	-	-	-	-	-	-	-	+
96	27 000	10,20	-	-	-	-	-	-	-	+
100	35 000	10,46	-	-	-	-	-	-	-	+

Les signes - et + signifient que la valeur de la fonction est respectivement négative et positive.

Figure 4.17. Valeurs de la fonction $\log(2)+\log(\log(2k))-\log(k)+2\log(P)-2\log(\log(k)-\log(P))$ pour $N \leq 10^{100}$ (i.e. $k \leq 10,46$) et $P \leq 256$.

Interprétation de ce tableau :

Avec un réseau comportant 2, 4 ou 8 processeurs, il est toujours possible en moyenne de mettre sur le processeur 0 tous les éléments de la base ayant $\alpha_{max} \geq 2$ pour $N \in \{10^{20}, \ldots, 10^{100}\}$.
Sur 16 processeurs, il faut que $N \geq 10^{30}$.
Sur 32 processeurs, $N \geq 10^{42}$.
Sur 64 processeurs, $N \geq 10^{69}$.
Sur 128 processeurs, $N \geq 10^{90}$.
Sur 256 processeurs et plus, il n'est pas possible que tous les éléments de la base ayant $\alpha_{max} \geq 2$ soient sur le processeur 0 pour $N < 10^{100}$.

Remarque : Ces valeurs sont moyennes et la distribution des nombres premiers et des résidus quadratiques influent sur elles.

Ainsi, on peut éviter de tester la valeur de α_{max} pour chaque envoi, dès que N et P respectent les conditions énoncées ci-dessus. De cette façon (tous les éléments de la base ayant $\alpha_{max} \geq 2$ sont regroupés sur le processeur 0), on peut alors assez facilement équilibrer les calculs des racines de $x^2 \equiv N \bmod p_i^{\alpha}$.

Tous les processeurs ont autant d'équations à résoudre que d'éléments de la base présents chez eux, sauf le processeur 0 (qui en a un peu plus). On utilise la technique de répartition des

éléments présentés ci-dessus (répartition des éléments, des feuilles vers la racine d'un arbre de diffusion de l'hypercube), pour que le processeur PE_0 garde un peu moins d'éléments et répartisse les autres éléments sur les autres processeurs.

Le stockage des solutions de ces équations est assuré par chaque processeur. Chaque équation a 2 solutions liées. On peut n'en conserver qu'une et déduire l'autre par une simple soustraction. En effet, soient x_1 et x_2 ces deux solutions. Alors $x_2 = p_i^{\alpha} - x_1$. Pour les éléments ayant $\alpha_{max} \geq 2$, il faut stocker 1 des deux racines (ou les 2 racines liées) par valeur de α, pour chaque élément. De plus, l'équation $x^2 \equiv N \bmod 2^{\alpha}$ admet 0, 1, 2 ou 4 solutions. C'est un cas particulier.

Dans ce paragraphe, la répartition des éléments et le calcul des racines sont deux opérations séparées. Nous allons voir qu'il est possible de les regrouper.

3.7.3. Calcul et répartition des k éléments en parallèle avec la résolution des équations

En fait, il existe une autre possibilité qui utilise encore mieux le parallélisme massif en ce qui concerne le calcul des k éléments de la base, puis des racines carrées de N modulo ces éléments.

Reprenons ces deux étapes. Pour la génération des k éléments, chaque processeur a son propre intervalle de recherche. Donc chaque processeur calcule tous les éléments de la base qui existent dans cet intervalle. A cet instant, les plus petites valeurs des éléments de la base ont été calculées, et les éléments manquants, s'il en manque, sont supérieurs à ceux qui viennent d'être générés.

Cette première étape est complètement parallèle : il n'y a pas de communications. Après ce calcul, soit il manque des éléments, soit il y en a assez, voire trop. Mais pour connaître cette information, il faut sommer sur un processeur les nombres d'éléments trouvés par chaque processeur. Ces communications sont effectuées selon le modèle présenté ci-avant (§2.1.5.) sur hypercube.

Un processeur connait alors le nombre total d'éléments générés. S'il y en a trop, on en supprime certains. S'il en manque, on en génère d'autres de la manière suivante : le processeur qui a centralisé la somme des nombres d'éléments calculés, le processeur 0 par exemple, désigne un de ses voisins, le processeur 1 par exemple, pour générer les éléments manquants.

Pendant ce temps, la répartition des éléments entre les autres processeurs commence, et dès qu'un processeur a le nombre voulu d'éléments, il commence immédiatement la recherche des racines carrées de N modulo les éléments qu'il possède.

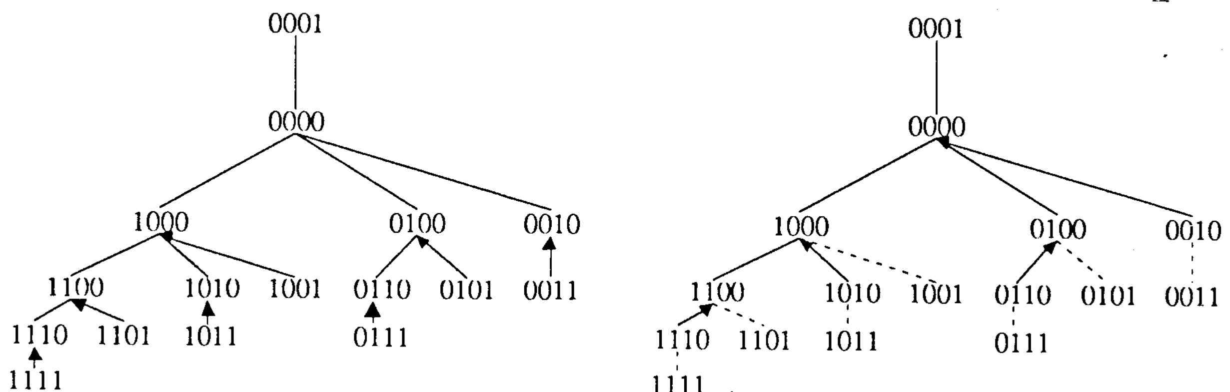


Figure 4.18 et 4.19. Première et deuxième étapes d'équilibrage sur un arbre de recouvrement d'un 4-cube.

Première étape : les feuilles s'octroient le nombre voulu d'éléments, pendant que le processeur 1 cherche les éléments manquants. Là encore, les échanges peuvent se faire vers les feuilles ou vers les pères.

Deuxième étape : les feuilles commencent à calculer les racines carrées de N modulo des éléments de la base, pendant que les feuilles de l'arbre de dimension inférieure s'octroient le nombre voulu d'éléments et que le processeur 1 cherche des éléments manquants.

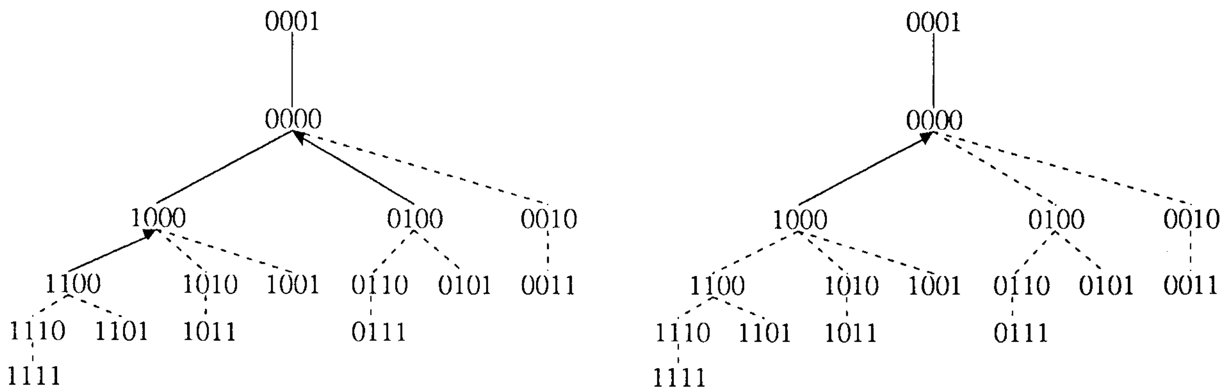


Figure 4.20 et 4.21. Troisième et quatrième étapes sur un arbre de recouvrement d'un 4-cube.

La suppression des feuilles se fait récursivement.

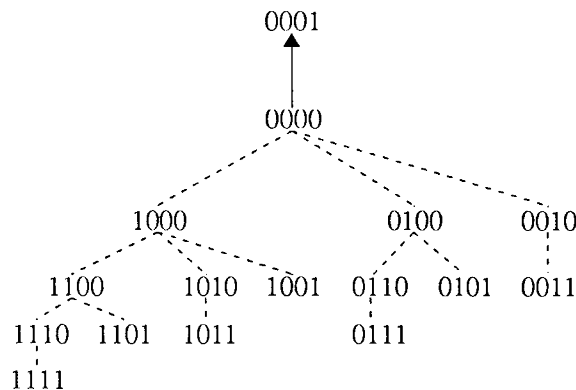


Figure 4.22. Dernière étape entre la racine et le processeur isolé

Dernière étape : répartition entre les deux derniers processeurs, dès que le processeur 1 a terminé sa recherche.

En fait, le nombre de communications est donc 1 + log P, car le processeur 1 nécessite une étape de communication qui ne peut avoir lieu que lorsque tous les autres noeuds de l'arbre se sont répartis les éléments. Afin d'optimiser le temps de calcul des racines carrées modulaires, on peut proposer de mettre quelques éléments supplémentaires sur les feuilles à l'étape 1, puis un peu moins à chaque étape suivante.

De cette manière, on assure une bonne répartition des tâches et les processeurs terminent en même temps le calcul des racines carrées de N. Avec cette solution, on utilise bien les processeurs en minimisant les communications : un seul regroupement d'informations, contre des regroupements et des diffusions successives dans la solution précédente.

Dorénavant, on appellera base ou base de facteurs l'ensemble des k éléments de la base et les racines carrées de N modulo chaque élément.

3.8. CONCLUSION

La base est complètement calculée, mais elle est répartie entre tous les processeurs, et de plus, de manière non équitable. Dans la suite du déroulement de l'algorithme, les processeurs vont avoir besoin soit de toute la base, soit chacun d'une partie de la base. Donc, il faudra soit faire connaître à chacun d'eux la totalité de la base, soit établir une répartition en fonction des besoins. Pour que chaque processeur ait connaissance de la base complète, il faut que chacun d'eux envoie à tous les autres sa propre partie de la base. Cette étape peut être réalisée de plusieurs manières :
- sur un anneau, on fait tourner les informations de tous les processeurs (P-1 rotations),
- chaque processeur, à son tour, envoie vers tous les autres (P diffusions),
- tous les processeurs envoient en même temps vers tous les processeurs, sur un hypercube, par multi-diffusion (voir [HoJ 89]), ce qui peut se faire par appel à la procédure all_to_all.

A ce stade, l'initialisation est terminée. Toutes les données nécessaires à la boucle de génération de la matrice sont prêtes.

4. LA BOUCLE DE GÉNÉRATION DE LA MATRICE DES FACTORISATIONS COMPLÈTES SUR LA BASE

Cette boucle est le cœur de l'algorithme. C'est aussi la partie qui nécessite le plus d'espace et le plus de temps. Informellement, c'est un ensemble de trois boucles imbriquées (la boucle *tant que* représente le test d'arrêt) :

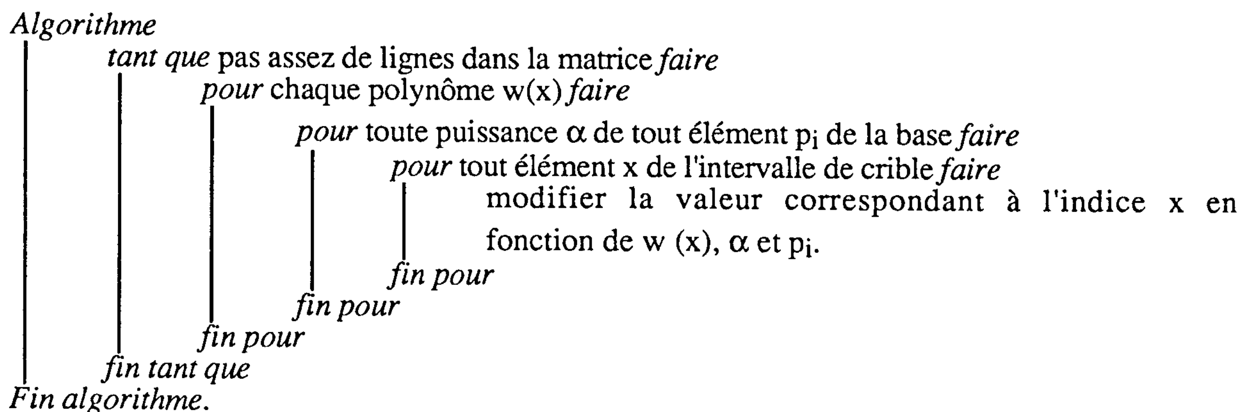


Figure 4.23. Algorithme de l'étape globale de construction de la matrice des factorisations

La matrice qu'on essaie de générer est la matrice des factorisations complètes de certains w_i sur la base de facteurs. Elle comporte k colonnes. Pour trouver assurément une combinaïson linéaire nulle entre des lignes, il faut au moins k lignes. Cependant, dès que le nombre L de lignes est supérieur à 0,96 k [Sil 87], on a une probabilité élevée de trouver au moins une combinaison linéaire. Mais cette combinaison linéaire peut conduire à des facteurs triviaux de N. Le nombre de lignes nécessaires est ainsi laissé à l'appréciation de l'utilisateur. C'est une variable supplémentaire de contrôle de l'algorithme.

4.1. PRÉSENTATION DE LA PARALLÉLISATION DE CETTE BOUCLE

La génération des lignes de la matrice se fait en plusieurs étapes pour chaque ligne :
- calcul des coefficients d'un nouveau polynôme,
- crible de [-M, M[avec ce polynôme et les éléments de la base,
- choix des w_i candidats à la factorisation complète sur la base,
- factorisation (ou tentative de factorisation) des w_i candidats et stockage dans la matrice des factorisations, des w_i complètement factorisés.

Dans l'algorithme informel présenté ci-dessus, chacune des trois boucles peut être parallélisée : les polynômes, la base de facteurs et l'intervalle de crible. On peut distribuer une des variables de ces trois boucles, tout en conservant sur chaque processeur les deux autres variables. Ceci nous conduit à deux solutions pour la parallélisation de cette étape :

- un calcul coopératif pour factoriser chaque w_i (distribution de la base de facteurs). En effet, distribuons la base de facteurs entre les processeurs. Alors, chaque processeur aura, à la fin d'une étape de crible avec un polynôme sur l'intervalle [-M, M[complet, une partie seulement de la factorisation des w_i. Pour obtenir un w_i totalement factorisé, il faut connaître les résultats des autres processeurs. C'est pourquoi on appelle cette technique de parallélisation, une coopération, car les processeurs exécutent une partie d'une tâche élémentaire. Cette tâche est la recherche d'une ligne de la matrice. Elle nécessite des communications sur une architecture distribuée.

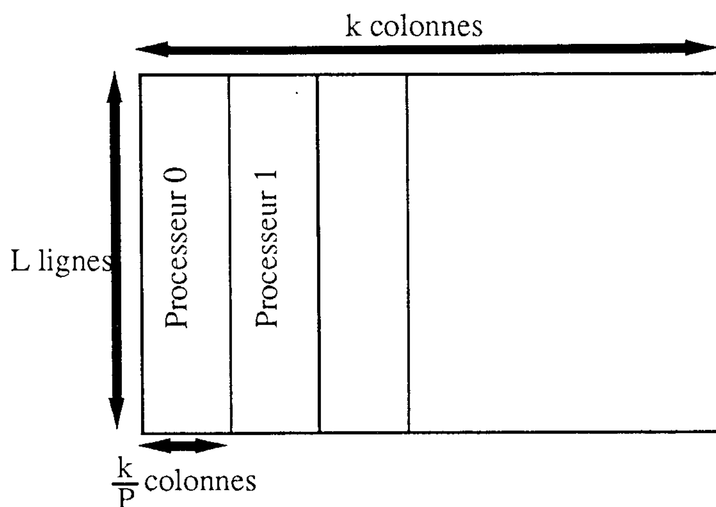


Figure 4.24. Répartition des colonnes entre les processeurs (calcul coopératif)

- un calcul indépendant de k/P w_i par processeur (distribution des polynômes ou de l'intervalle de crible). Si on distribue l'intervalle de crible, chaque processeur effectue donc un crible sur un petit intervalle. Mais rien d'autre n'est modifié : chaque processeur peut voir si ses w_i sont complètement factorisés. Les processeurs sont indépendants. De même, la distribution des polynômes implique que chaque processeur ait des polynômes différents. Mais les processeurs sont indépendants pour trouver les w_i factorisés.

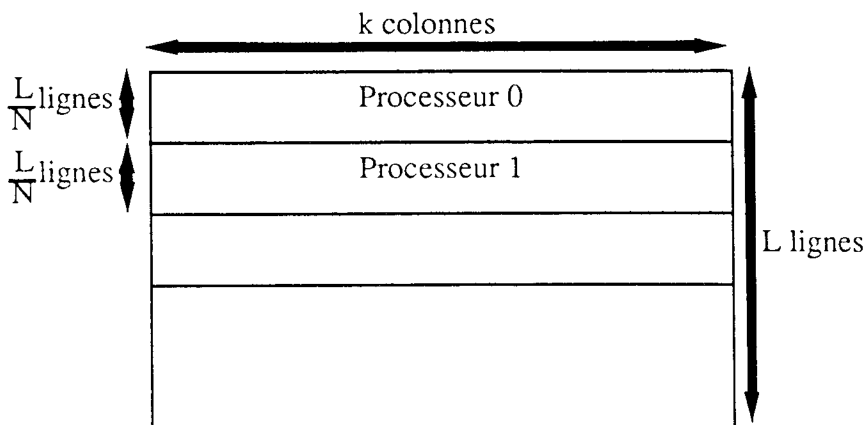


Figure 4.25. Répartition du calcul des lignes de la matrice des factorisations (calcul indépendant)

Nous allons étudier maintenant ces différentes possibilités de parallélisation pour chacune des quatre parties principales de cette boucle de construction de la matrice : la génération des polynômes, le crible de [-M, M[, le choix des candidats à la factorisation complète et leur factorisation, d'abord en imaginant une machine distribuée hypercube basée sur le FPS T40 où chaque nœud a une mémoire illimitée.

4.2. LA GÉNÉRATION DES POLYNÔMES w(x), PRÉSENTATION

L'algorithme du crible quadratique multipolynômial nécessite des polynômes du second degré, construits de telle manière que leurs valeurs sur l'intervalle de crible [-M, M[pourront, pour certaines d'entres elles, être factorisées complètement sur la base, ce qui fournira une ligne de la matrice finale.

Nous montrons que la génération d'un polynôme est en fait la génération du coefficent a de x^2. Ce coefficient dépend du nombre de polynômes nécessaires que nous évaluons statistiquement.

Puis nous présentons l'étude algorithmique de la génération des polynômes. Ils peuvent être tous générés avant la boucle de crible / factorisation, car la mémoire est infinie, ou un par un dans cette boucle. La première solution donne une génération dans la boucle d'initialisation. La deuxième solution, génération à l'intérieur de la boucle, peut être implantée de plusieurs manières. Nous montrons que si un processeur calcule les polynômes pour tous les autres, les envoie un par un à tous les processeurs, il y a une perte de temps due à la synchronisation de tous les processeurs. En revanche, si chaque processeur calcule, de façon redondante, les polynômes, il n'y a aucune communication avec les autres processeurs, et le temps perdu en redondance peut contrebalancer le temps perdu en synchronisation dans l'autre solution.

4.2.1. Théorie

La génération d'un polynôme est en fait la génération des coefficients du polynôme $w(x) = a^2x^2 + 2bx + c$, avec

- pour a :
  - a voisin de $\sqrt{\frac{\sqrt{2N}}{M}}$,
  - a de la forme 4j+3,
  - a premier,
  - N résidu quadratique de a ;
- pour b : $b = N^{((a(a-1)/2)+1)/2} \bmod a^2$ et $|b| < \frac{a^2}{2}$;
- pour c : $c = \frac{b^2 - N}{a^2}$.

b et c dépendent de a. Donc la génération d'un polynôme se résume à la génération de son coefficient a.

On remarque que a est de la taille de $\sqrt[4]{N}$ environ, ce qui, pour des valeurs de N de l'ordre de 10^{100}, donne des valeurs de a inférieures à 10^{25}. Donc b est de la taille de a^2 au maximum, donc inférieur à 10^{50}. Enfin c est négatif et de l'ordre de -10^{50}. Donc, a, b et c sont des entiers écrits en précision entière illimitée.

4.2.2. Le calcul du coefficient a

Le premier problème qui se pose tient au calcul de a. Calculer un résidu quadratique entre deux entiers écrits en précision entière illimitée est coûteux certes, mais moins que le test de primalité d'un tel entier.

En effet, tester la primalité exacte d'un nombre de 25 chiffres est très coûteux. Il est moins cher de tester la primalité probabiliste par le test de Rabin [Ros 84] à base de tests de pseudoprimalité forte sur plusieurs bases.

Mais en fait, il n'est pas nécessaire que a soit premier. En effet, le cas a non premier n'est pas gênant ni pour a lui-même, ni pour b. C'est pour c que cela risque de poser des problèmes. Car c est le résultat d'une division entière, dont le reste doit être nul.

Or si a est premier, on est sûr que ce reste est nul. Alors que si a n'est pas premier, il est possible que ce reste soit non nul. On peut donc choisir des coefficients a non premiers, à condition de vérifier que le reste de la division par a^2 lors du calcul de c est nul. Si oui, alors ce polynôme convient, sinon, il ne convient pas et il faut déterminer une nouvelle valeur de a. On peut, avec un coût réduit, éliminer une bonne proportion des a non premiers, en effectuant un test de pseudoprimalité forte sur la base 2 avec chaque a. Si a passe ce test, alors on calcule b et c, en vérifiant le reste de la division. Si a ne passe pas ce test, on choisit un autre a.

Cependant, que signifie "a voisin de $\sqrt{\frac{\sqrt{2N}}{M}} = Q$" ? Rappelons que cette condition permet de cribler avec des polynômes différents dont les valeurs sur [-M, M[restent à peu près constantes. Ceci signifie que les valeurs de a doivent être centrées autour de Q.

Donc les valeurs de a doivent s'étaler entre Q - ε et Q + ε. Cet ε dépend directement du nombre de polynômes à générer. Or on ne peut pas connaître précisément le nombre q de polynômes nécessaires pour construire la matrice, ni combien de valeurs de a doivent être essayées pour trouver q valeurs de a convenables. Mais on peut donner une valeur moyenne de q.

4.2.3. Le nombre de polynômes nécessaires

Le rapport entre le nombre de polynômes pour un entier de n+3 chiffres et celui pour un entier de n chiffres est environ 1,6 dans l'article de Caron et Silverman [Cas 88]. Par extrapolation, il faudrait environ 9 000 000 de polynômes pour un nombre de 100 chiffres avec un crible sur un intervalle de longueur 8 000 000. Pomerance, Smith et Tuler ont besoin de 5 polynômes sur un intervalle de longueur 138 000 000 pour obtenir une ligne de la matrice [PST 88]. Comme ils utilisent une base de 100 000 éléments, ils utilisent donc 500 000 polynômes.

Le nombre de cribles effectifs est de $7{,}3.10^{13}$ dans le premier cas, et de $6{,}9.10^{13}$ dans le deuxième cas (produit du nombre de polynômes et de la longueur de l'intervalle de crible), soit une différence de 6%.

Nb de décimales	Nb q de polynômes (*1000)	Taille de l'intervalle 2M (*1000)	2M.q
60	8	150	$1{,}200.10^{9}$
63	15	200	$3{,}000.10^{9}$
66	25	250	$6{,}250.10^{9}$
69	45	300	$1{,}350.10^{10}$
72	85	350	$2{,}975.10^{10}$
75	150	425	$6{,}375.10^{10}$
78	280	525	$1{,}470.10^{11}$
81	450	750	$3{,}375.10^{11}$
84	750	1000	$7{,}500.10^{11}$
87	1200	1300	$1{,}560.10^{12}$
90	*1920*	*1690*	$3{,}245.10^{12}$
93	*3072*	*2197*	$6{,}749.10^{12}$
96	*4915*	*2856*	$1{,}404.10^{13}$
99	*7864*	*3713*	$2{,}920.10^{13}$
100	*9000*	*4052*	$3{,}647.10^{13}$

Figure 4.26. Nombre de polynômes et taille de l'intervalle. En italique, les extrapolations

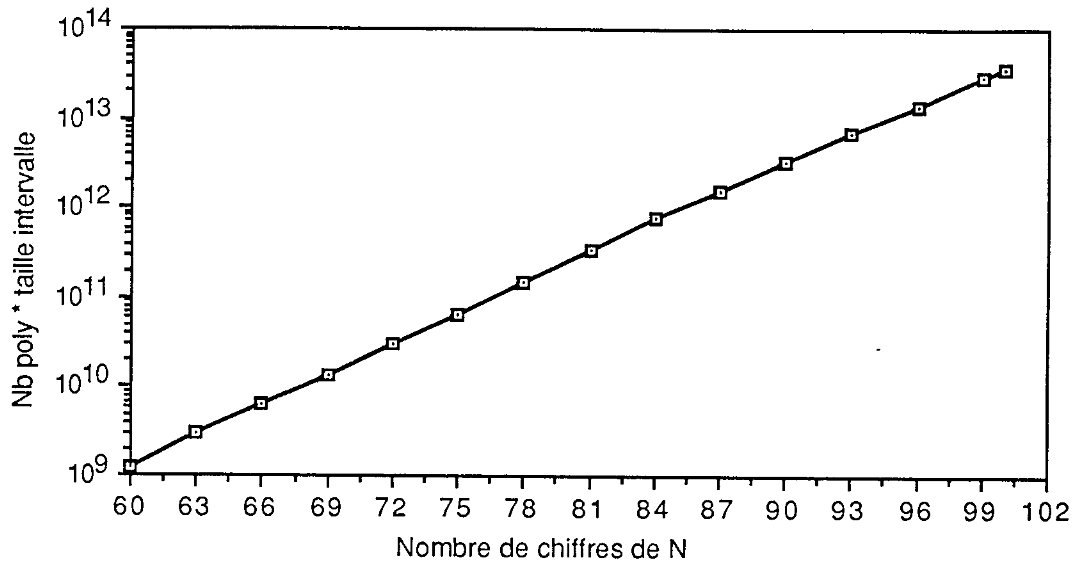


Figure 4.27. Courbe 2M.q en fonction de N

L'interpolation donnée par logiciel est $y = 259{,}45.10^{0{,}112x}$. On ramène cette équation en N, car $x = \log_{10} N$. Alors $2M.q = 259{,}45.10^{0{,}112\log_{10}N} = 259{,}45.N^{0{,}112}$

Cependant, il faut relativiser : Caron et Silverman utilisent, comme Pomerance et al., la large prime variation, ce qui leur permet de diminuer le nombre total de polynômes nécessaires. Cette courbe n'est donc qu'indicative, car nous n'utilisons pas la "large prime variation". Mais comme nous savons évaluer une longueur d'intervalle de crible en fonction de la taille de N, nous pouvons aussi en déduire une évaluation du nombre de polynômes. Par conséquent, il est possible de répartir les valeurs de a autour de Q, presque uniformément centrées.

Exemple :
Pour fixer les idées, et en reprenant les optima (k=100 000 et M=2 000 000 pour $N=10^{100}$) en termes d'espace mémoire, on en déduit le nombre q de polynômes par la relation $2Mq = 260.N^{0{,}112}$. On trouve q = 10 000 000 (soit 10 % de plus que la valeur préconisée par Caron et Silverman [CaS 88]).

Or pour obtenir 10 000 000 de polynômes, il faut générer 10 000 000 de coefficients a différents, premiers et tels que N soit un résidu quadratique de chacun d'eux. Il faut donc en tester en moyenne 20 000 000 log 20 000 000 = 336.10^6. Comme on prend a de la forme 4j+3, l'intervalle devient donc $\{Q-672.10^6, \ldots, Q+672.10^6\}$, dans lequel se trouveront les coefficients a. Rappelons que Q est de l'ordre de 10^{43} pour $N = 10^{100}$, ce qui implique un intervalle approximatif de $\{10^{43}-10^9, \ldots, 10^{43}+10^9\}$. Pour des valeurs de N plus petites, l'intervalle se réduit.

4.3. ALGORITHMIQUE DE LA GÉNÉRATION DES POLYNÔMES

Après ces considérations mathématiques, nous allons étudier la génération des polynômes d'un point de vue algorithmique dans un environnement distribué d'abord où la mémoire de chaque nœud est illimitée.

Dans un environnement où la mémoire est illimitée sur chaque processeur, on peut imaginer deux scénarios :

- les polynômes sont tous générés et stockés avant la boucle, lors de l'initialisation, puis utilisés chacun à leur tour, en fonction des besoins,
- les polynômes sont générés un par un à l'intérieur de la boucle.

4.3.1. Génération des polynômes lors de l'initialisation

En effet, avec une mémoire illimitée sur chaque nœud, on peut envisager de précalculer tous les polynômes qui seront nécessaires à la construction complète de la matrice. On rajoute donc une étape lors de l'initialisation, où on calcule et on stocke toutes les valeurs des coefficients a.

Comment ces coefficients sont-ils générés ?

- Soit un processeur s'occupe de tout, mais dans ce cas, un seul processeur travaille tandis que les autres restent inactifs. Ce n'est pas bon, d'autant plus qu'il faudra diffuser les résultats (ou une partie des résultats) vers les autres processeurs. Calculons la place que tient cet ensemble pour $N = 10^{100}$. Rappelons qu'il faut environ 10^7 polynômes. $a \approx 10^{25}$. $b \approx 10^{50}$. $c \approx -10^{50}$. Donc selon le codage de PaC [Roc 89], a occupe 16 octets, b et c chacun 28, soit un total de 72 octets par polynômes. D'où un total de 720 Mo environ. Diffuser 720 Mo dans un réseau est très coûteux. L'envoi entre deux voisins d'un message de 720 Mo sur l'hypercube FPS T40 prend déjà 18 minutes ! Bien sûr, pour la diffusion, on peut utiliser des techniques de découpage du message en paquets et d'envois pipelinés [HoJ 89], [BTV 90]. Mais le coût reste important.
- Soit les processeurs travaillent tous en même temps, et chacun génère tous les coefficients a (tous les polynômes) nécessaires à l'exécution du programme. On voit immédiatement une redondance totale.
- Soit les processeurs travaillent tous en même temps, mais sans redondance, c'est-à-dire que chacun génère des coefficients différents de ceux des autres processeurs. Il faut donc que les possibilités de génération soient disjointes entre deux processeurs quelconques. Mais la réunion des possibilités de tous les processeurs doit être égale à l'ensemble complet des possibilités. On doit donc réaliser une partition de l'ensemble des coefficients entre les processeurs. Caron et Silverman, dans leur implantation sur un réseau de stations de travail, découpent l'ensemble des coefficients, non pas en familles disjointes, mais en intervalles contigus [CaS 87]. Si bien que le risque existe qu'un processeur dépasse la limite supérieure de son intervalle (s'il n'a pas assez de lignes dans sa matrice) et utilise des polynômes qu'un autre processeur a déjà utilisés. Ceci ne peut pas survenir avec notre partitionnement de l'ensemble des polynômes. Chacun des P processeurs PE_i, i=0, ..., P-1, doit donc générer des cœfficients a de la forme $4Pj + 4i + 3$. Ces coefficients sont bien de la forme $4j'+3$ et de plus

$$\bigcup_{i=0}^{P-1} (4Pj + 4i + 3) = 4j'+3.$$

  De cette manière, on peut associer une famille à chaque processeur, et faire générer tous les coefficients a nécessaires sans aucune redondance. A la fin de cette étape, si on veut que chaque processeur ait connaissance de l'ensemble des polynômes, on communique selon un modèle de "gossiping" (multi-diffusion ou all-to-all). Ceci est très coûteux, mais le coût peut en être diminué en utilisant des arbres de recouvrement multiples et/ou des découpages des messages en paquets pour des envois pipelinés [HoJ 89], [BTV 90].

Cette technique est intéressante, car on peut l'implanter sans avoir besoin de communications.

4.3.2. Génération des polynômes un par un dans la boucle

Mais on peut aussi générer les polynômes au fur et à mesure des besoins, dans la boucle de crible-factorisation. Comment ces polynômes sont-ils alors générés ? Il y a de nombreuses manières. Nous allons en présenter quelques-unes avec leurs avantages et leurs inconvénients. De la même manière que lors de la génération complète des polynômes dans la phase d'initialisation, il y a ici 3 possibilités :

- un processeur dédié calcule les coefficients des polynômes et les envoie à chaque processeur sur demande expresse,
- un processeur dédié calcule les coefficients des polynômes et les envoie à tous les processeurs en même temps,

- chaque processeur génère pour lui-même tous les polynômes un par un selon ses besoins.

La deuxième solution est plus rapide que la première, car la procédure one_to_all est plus rapide que les communications point à point.

Nous pouvons difficilement la comparer avec la troisième solution. Cependant nous espérons, comme nous le montrons plus loin, que la redondance puisse entrainer une compensation (pas de communication, donc pas d'attente en synchronisation) et puisse contrebalancer le temps perdu en communication.

4.3.2.a. Un processeur dédié calcule les polynômes et les envoie à un processeur à la fois

La première possibilité (un processeur dédié calcule les polynômes et les envoie un par un à chaque processeur sur sa demande expresse) correspond à un schéma Maître/Esclaves. Et l'étoile est la structure la mieux adaptée, avec le Maître au centre et les Esclaves autour. C'est d'ailleurs l'architecture choisie par Silverman [Sil 88] pour son réseau de Suns.

Plusieurs critères interviennent dans ce schéma Maître/Esclaves :

- Le temps de calcul d'un nouveau polynôme par le maître par rapport au temps de crible-factorisation par les esclaves ; ce qui donne le critère
  t(calcul nouveau poly) ≤ t(crible + facto) vs t(calcul nouveau poly) ≥ t(crible + facto).
- Le droit ou non du maître à effectuer plusieurs envois simultanés vers différents esclaves ; ce qui donne le critère
  envois simultanés vs envois non simultanés.
- Le fait que les communications du maître vers les esclaves soient ordonnées a priori par le programme ou que ces communications soient effectuées dans l'ordre dans lequel les esclaves demandent à communiquer avec le maître ; ce qui donne le critère
  communications ordonnées vs communications non ordonnées.
- Le fait qu'un esclave qui a besoin d'un nouveau polynôme puisse interrompre le maître dans son calcul et ainsi avoir immédiatement accès aux valeurs des coefficients d'un nouveau polynôme ou que l'esclave doive attendre que le maître ait terminé son calcul avant de pouvoir engager une communication ; ce qui donne le critère
  maître interruptible vs maître non interruptible.

Ces critères sont représentés par l'arbre suivant :

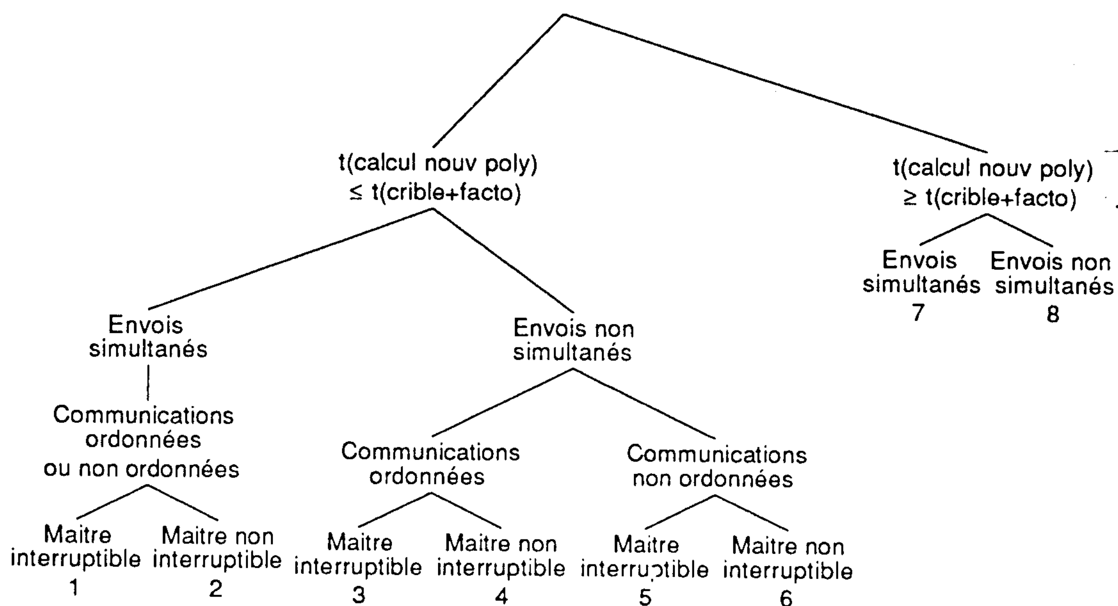


Figure 4.28. Arbre binaire des critères pour l'étape du crible entre un Maître et des Esclaves

Certains choix ont été supprimés (l'arbre n'est pas binaire complet) car, en ce qui concerne les envois simultanés (dans le sous-arbre gauche), il est évident que l'ordre des communications n'intervient pas ; en ce qui concerne le temps de calcul du maître par rapport aux esclaves (sous-arbre droit), que les communications soient ordonnées ou non importe peu puisque les esclaves doivent attendre que le maître ait terminé son calcul pour avoir accès au nouveau polynôme. De même, il ne sert à rien d'interrompre le maître, puisqu'il n'a pas de nouvelles informations à envoyer : il ne les a pas encore calculées. La seule variable qui peut entrer en jeu est celle des envois simultanés ou non du maître vers les esclaves.

En ce qui concerne les cas 1 à 6, où le temps de calcul d'un nouveau polynôme par le maître est inférieur au temps de crible-factorisation par les esclaves, le maître a du temps libre ou peut en avoir. Il est donc commode d'envisager que ce maître ait toujours un polynôme prêt pour les esclaves, et qu'il calcule et stocke en avance les coefficients des polynômes futurs. Eventuellement, il peut supprimer de sa mémoire les coefficients des polynômes dont plus aucun processeur n'aura besoin.

Cas 1 : Le maître met moins de temps pour calculer les coefficients du prochain polynôme que les esclaves pour cribler et factoriser avec le polynôme en cours. De plus, le maître peut communiquer avec plusieurs esclaves en même temps. Enfin, le maître peut interrompre ses calculs en cours pour effectuer une communication avec un ou plusieurs esclaves.

Les esclaves n'attendent jamais, puisque le maître a toujours terminé de calculer le prochain polynôme avant que les esclaves n'en aient besoin, et que ce maître est interruptible. Donc, dès qu'un esclave a terminé de cribler-factoriser, il a immédiatement accès au prochain polynôme. De plus, si plusieurs esclaves ont besoin des coefficients en même temps, le maître peut les leur envoyer en même temps.
Donc il n'y a aucune perte de temps dans ce cas de figure.

Cas 2 : Mêmes hypothèses que pour le cas 1, sauf en ce qui concerne l'interruption du maître qui n'est plus possible : le maître doit terminer ses calculs en cours avant de pouvoir initialiser des communications, c'est-à-dire que le calcul des coefficients d'un nouveau polynôme est une suite d'opérations non interruptible (atomique).

Un esclave qui a fini son étape de crible-factorisation demande un nouveau polynôme au maître. Si le maître n'est pas en train de calculer, il peut immédiatement accéder à la requête de l'esclave. S'il est en train de calculer, il se laisse le temps de terminer et s'occupe alors de toutes les communications demandées par les esclaves.
Mais, selon nos hypothèses, le maître n'est jamais libre : il calcule en continu de nouveaux polynômes. Donc les esclaves doivent toujours attendre un petit peu.

Cas 3 : Le maître met moins de temps pour calculer les coefficients du prochain polynôme que les esclaves pour cribler et factoriser avec le polynôme en cours. Mais le maître ne peut pas communiquer avec plusieurs esclaves en même temps. De plus, un ordre concernant les communications avec les esclaves est prédéterminé, ce qui implique que les esclaves parlent avec le maître toujours dans le même ordre, indépendamment de leurs besoins. Cependant, le maître peut interrompre ses calculs pour effectuer une communication avec un esclave.

Si plusieurs esclaves demandent un envoi de coefficients en même temps, il seront servis séquentiellement et dans l'ordre préétabli. De sorte que si le premier processeur prévu pour communiquer a du retard par rapport aux autres processeurs, il les bloque tous jusqu'à ce que sa communication ait été effectuée. D'où une perte de temps due à une synchronisation parfois nécessaire suivant la rapidité des processeurs. Au contraire, si le processeur prévu pour communiquer est prêt, il a même le droit d'interrompre le maître afin d'accéder immédiatement aux informations. Ainsi il libère le processeur suivant qui a le droit de communiquer dès cet instant avec le maître.

Cas 4 : Le maître met moins de temps pour calculer les coefficients du prochain polynôme que les esclaves pour cribler et factoriser avec le polynôme en cours. Mais le maître ne peut pas

communiquer avec plusieurs esclaves en même temps. De plus un ordre concernant les communications avec les esclaves est prédéterminé, ce qui implique que les esclaves parlent avec le maître toujours dans le même ordre. Et le maître ne peut pas interrompre ses calculs en cours pour effectuer une communication avec un esclave.

On se trouve dans un cas moins favorable que le cas précédent. Ce cas correspond au modèle de communication entre deux processeurs sur le FPS T40, le TNode et l'iPSC/2 indépendamment de la taille de leur mémoire. On voit ici qu'il y a une perte de temps assez importante.

Cas 5 : Le maître met moins de temps pour calculer les coefficients du prochain polynôme que les esclaves pour cribler et factoriser avec le polynôme en cours. Mais le maître ne peut pas communiquer avec plusieurs esclaves en même temps. Cependant, les communications entre les esclaves et le maître ne se font pas dans un ordre prédéterminé, mais dans l'ordre des requêtes des esclaves. Le maître peut interrompre ses calculs en cours pour répondre à la requête d'un esclave.

Par rapport au cas 3, ce cas est plus favorable, puisqu'un esclave qui a besoin d'informations peut accéder au maître sans devoir attendre son tour, sauf dans le cas où plusieurs esclaves se trouvent en même temps dans cette position, le maître ne pouvant en effet servir qu'un esclave à la fois.
On peut voir ce cas comme un buffer de réception sur les esclaves. L'esclave a toujours un polynôme d'avance dans son buffer. C'est le modèle de la Connection Machine (indépendamment de la taille de sa mémoire).

Cas 6 : Le maître met moins de temps pour calculer les coefficients du prochain polynôme que les esclaves pour cribler et factoriser avec le polynôme en cours. Mais le maître ne peut pas communiquer avec plusieurs esclaves en même temps. Cependant, les communications entre les esclaves et le maître ne se font pas dans un ordre prédéterminé, mais dans l'ordre de demande par les esclaves. Et le maître ne peut pas interrompre ses calculs en cours pour communiquer avec un esclave.

L'esclave qui veut des informations du maître doit attendre que le maître ait terminé ses calculs. Mais même si d'autres esclaves n'ont pas terminé, il aura accès à ces informations dans l'ordre où les esclaves les auront demandées, immédiatement ou presque.

Cas 7 : Le maître met plus de temps pour calculer les coefficients du prochain polynôme que les esclaves pour cribler et factoriser avec le polynôme en cours. Lorsque le maître a terminé ses calculs, il envoie à tous les esclaves en même temps.

C'est le modèle de communication utilisable sur le FPS T40 (à mémoire illimitée) sur un anneau (one-to-all). Les esclaves doivent attendre que le maître soit prêt à envoyer, mais à cet instant, la communication est instantanée vers tous les processeurs.

Cas 8 : Le maître met plus de temps pour calculer les coefficients du prochain polynôme que les esclaves pour cribler et factoriser avec le polynôme en cours. Lorsque le maître est prêt à émettre, il envoie les informations aux esclaves, mais à un esclave à la fois.

Ce modèle est celui de l'iPSC 2 et de la CM2 à cause de leur routeur, et du T-Node par sa reconfigurabilité, indépendamment de la taille de leur mémoire.
Il est clair que la perte de temps est énorme dans ce cas.

Evaluons les pertes de temps : la relation "x < y" signifie que le temps perdu dans le cas x est inférieur au temps perdu dans le cas y.

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

La meilleure option est 1, la pire est 8. Quant aux autres, on ne peut pas les ordonner facilement puisque cela dépend surtout de l'ordre préétabli par le programme pour les communications entre le maître et les esclaves.

4.3.2.b. Un processeur dédié calcule les polynômes et les envoie à tous les processeurs en même temps

La deuxième possibilité (un processeur dédié calcule les coefficients des polynômes et les envoie à tous les processeurs en même temps) correspond encore à un schéma Maître/Esclaves. Pendant que les esclaves criblent et factorisent, le maître calcule des polynômes. Si le temps de calcul du maître est supérieur à celui des esclaves, on est dans le cas 7, et c'est sur le maître que se fait la synchronisation, sinon c'est sur l'esclave le plus lent.

Ce schéma est réalisable sur le FPS T40 en utilisant une communication de type one-to-all sur un anneau. Mais les pertes de temps sont importantes.

Cette solution est meilleure car la procédure one_to_all de FPS est plus rapide que les communications point à point.

4.3.2.c. Redondance du calcul des polynômes sur chaque processeur

La troisième possibilité (chaque processeur génère pour lui-même tous les polynômes un par un selon ses besoins) permet aux processeurs de travailler indépendamment les uns des autres. Ils avancent à leur propre vitesse. Ils n'ont pas besoin de se synchroniser pour échanger des informations. Cependant, ils effectuent tous le même travail, d'où une redondance. Malgré tout, il n'est pas certain que cette technique ne permette pas un certain gain de temps. En effet, il faut comparer :

- la perte de temps due aux calculs redondants (troisième possibilité),
- la perte de temps due aux synchronisations et aux communications (deuxième possibilité) en supposant que le calcul d'un polynôme est plus rapide que le crible-factorisation.

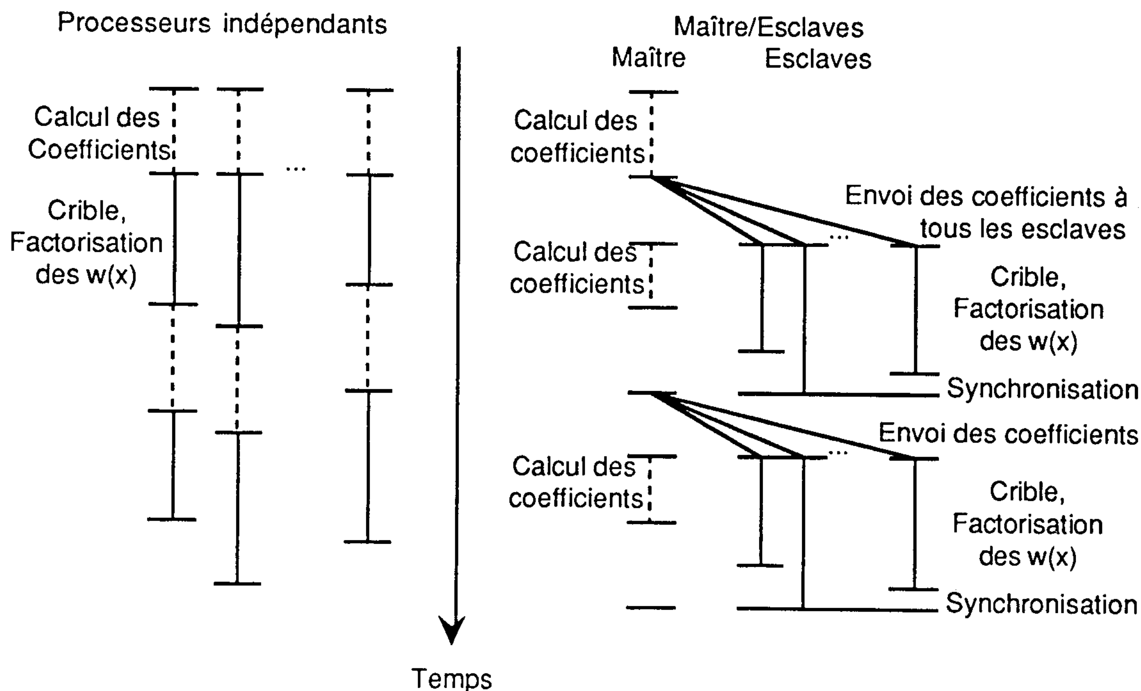


Figure 4.29. Comparaison de la génération des polynômes redondante puis par un maître.

Tout d'abord, on peut remarquer que la méthode Maître/Esclaves utilise P-1 processeurs pour le crible, alors que la méthode où les processeurs sont indépendants utilise les P processeurs disponibles.

Il faut maintenant établir, en moyenne, lequel de ces deux schémas est le moins coûteux en temps. Supposons que le temps de génération d'un polynôme soit plus court que le temps de crible-factorisation avec un polynôme.

Soit P le nombre de processeurs numérotés de 0 à P-1, L le nombre total de polynômes sur un processeur. Soit $t^{po}_{i,j}$ le temps de calcul des coefficients du i^e polynôme sur le processeur j. Soit $t^{cf}_{i,j}$ le temps de crible et factorisation avec le i^e polynôme sur le processeur j. Soit t^{co}_{i} le temps de communication des coefficients du i^e polynôme du maître vers les esclaves. Soient T_{ind} le temps d'exécution parallèle de l'algorithme des P processeurs indépendants et $T_{M/E}$ le temps total d'exécution parallèle de l'algorithme du maître 0 et des P-1 esclaves 1 à P-1.

$$T_{ind} = \max_{j=0...P-1} \left(\sum_{i=1}^{L} \left(t^{po}_{i,j} + t^{cf}_{i,j} \right) \right)$$

$$T_{M/E} = t^{po}_{0,1} + \sum_{i=1}^{L} \left(t^{co}_{i} + \max_{j=1...P-1} \left(t^{cf}_{i,j} \right) \right)$$

La comparaison de T_{ind} et $T_{M/E}$ est largement dépendante de la machine sur laquelle on exécute l'algorithme du crible quadratique, c'est-à-dire de la puissance de calcul et de la vitesse de communication.

Si on simplifie en posant $t^{po} \leq t^{co}$, alors statistiquement on assistera à une compensation dans le cas des processeurs indépendants, alors que pour les Maître/Esclaves, cette compensation ne pourra pas avoir lieu. Par suite de quoi, l'algorithme sur les processeurs indépendants s'exécutera plus rapidement. Si $t^{po} > t^{co}$, je ne peux rien dire (et c'est semble-t-il le cas sur la majeure partie des multiprocesseurs distribués).

4.3.3. Conclusion

Il est difficile de dire quelle méthode est la plus puissante, de la génération par un processeur, de l'envoi à tous, ou de la génération redondante sans communication. Les communications sont lentes sur le FPS T40, le processeur est puissant pour les calculs. On peut penser que la redondance équilibre le temps perdu en communication.

En revanche, si les Transputers étaient utilisés à leur puissance nominale en ce qui concerne les temps d'initialisation des communications (en OCCAM), il est certain que la redondance coûterait trop cher face à la rapidité de communication qu'offre OCCAM.

4.4. LE CRIBLE DE L'INTERVALLE [-M, M[

Le but du crible de l'intervalle [-M, M[est de découvrir les indices x de [-M, M[tels que w(x) se factorise complètement sur la base.

La première étape du crible consiste à calculer les points de départ pour chaque racine (2 en général) de chaque puissance α de chaque élément p_i de la base, en fonction du polynôme. Ce point de départ est l'indice x_0 de [-M, M[le plus petit possible (le plus près de -M) tel que $p_i^{\alpha} \mid w(x_0)$. Par la suite, le crible consiste à ajouter p_i^{α}, $2p_i^{\alpha}$, ... à x_0 pour trouver tous les w(x) divisibles par p_i^{α}. L'indice de [-M, M[correspondant à l'indice $x_0+jp_i^{\alpha}$ est alors augmenté de

log p_i. L'implantation de cette partie du crible doit être très efficace, car elle est répétée de nombreuses fois, pour chaque polynôme et pour chaque valeur de α et de p_i :

déterminer le point de départ x_0 le plus proche de -M
répéter
| ajouter log p_i à la valeur de tab_crible[x_0]
| $x_0 = x_0 + p_i^\alpha$
jusqu'à $x_0 \geq M$.

Cette boucle est indépendante de l'implantation parallèle choisie (sauf pour les bornes de variation de x_0). L'implantation naturelle consiste à définir un tableau tab_crible de réels, puisque le logarithme est un réel (64 bits sur une machine comme le FPS T40). Il faut donc 8 octets par élément de tableau. Or l'addition d'un réel est coûteuse. De plus la place nécessaire est très importante. Enfin la précision donnée sur 64 bits est inutile.

C'est pourquoi nous nous tournons vers les entiers. Les intérêts sont multiples : de 64 bits par élément, la place mémoire tombe à 32 bits ; la précision peut être maintenue avec une multiplication par 10, 100 ou 1000 des réels avant la troncature entière ; les opérations d'addition sont plus rapides.

Mais on peut encore diminuer la taille mémoire en codant chaque élément de tab_crible sur 8 bits. En effet, la valeur de w(x) est de l'ordre de $\sqrt{N}$. Avec $N \approx 10^{60}$, w(x) vaut environ 10^{30}. Or $10^{30} \approx 2^{100}$. Donc, le logarithme d'un w(x) ne dépassera pas la valeur 100. Et la somme des logarithmes ne dépassera pas cette valeur. Chaque élément de tab_crible peut être codé sur 8 bits. Mais la précision est faible. Cependant, il suffit d'en tenir compte dans la valeur moyenne V permettant le choix des w(x) candidats à la factorisation complète. Ainsi les additions se font entre entiers, les valeurs des logarithmes sont tronquées à leur partie entière et la place mémoire est mieux utilisée.

Nous voulons la solution la plus rapide.

Le crible dépend de la répartition de la base, de la répartition de l'intervalle de crible, et de la répartition des polynômes entre les processeurs. Nous montrons que la répartition de la base entraine des communications très importantes. Elle est moins bonne que la répartition de l'intervalle de crible qui consise à faire exécuter un crible normal mais sur un intervalle réduit. La meilleure est la répartition des polynômes.

4.4.1. Distribution des polynômes

Dans ce cas, les polynômes sont répartis entre les processeurs par famille (voir § 2.2.1) et chaque processeur possède l'intervalle de crible et la base de facteurs complets. Plusieurs stratégies d'acquisition de ces polynômes par chaque processeur sont envisageables.

* Chaque processeur a tous les polynômes dont il a besoin dans sa mémoire (ils ont été générés dans la phase d'initialisation). Il s'agit dans ce cas d'un crible quadratique complet avec un sous-ensemble de l'ensemble des polynômes. Or le but d'une étape du crible avec un polynôme est de découvrir les indices x tels que w(x) se factorise complètement sur la base. Il n'y a besoin d'aucune communication entre les processeurs durant toute cette phase, c'est-à-dire pour l'ensemble des polynômes stockés sur ce processeur. Le critère d'arrêt peut être de trouver un certain nombre de w(x) factorisés (lignes de la matrice) ou l'essai exhaustif de tous les polynômes prévus.

* Chaque processeur reçoit les polynômes d'un maître. Alors plusieurs hypothèses sont possibles (voir § 2.2.1) :

- Soit l'esclave peut interrompre le maître pendant ses calculs et donc accéder immédiatement au nouveau polynôme. Dans ce cas, il n'y a pas d'attente.

- Soit l'esclave doit attendre que le maître ait terminé ses calculs pour que la communication ait lieu. L'attente est faible.
- Soit l'esclave doit attendre que le maître ait terminé ses calculs et prendre son tour dans la file d'attente, car il doit attendre que d'autres esclaves aient été servis. Dans ce cas la perte de temps est plus importante, car d'autres processeurs plus lents que lui peuvent le retarder et le bloquer. C'est le cas si l'ordre des communications est préétabli par programme.
- Soit l'esclave doit se synchroniser sur l'esclave le plus lent afin que le maître diffuse des informations personnelles à tous les processeurs en même temps. C'est dans ce cas que l'attente est la plus longue.

* Chaque processeur calcule lui-même ses propres polynômes dès qu'il en a besoin. Evidemment, il n'y a pas de perte de temps en synchronisation et communications. Il n'y a pas non plus de redondance des calculs. De plus les processeurs travaillent tous : il n'existe pas de maître dédié à la génération des polynômes.

Comme on l'a vu au § 2.2.1, les temps de crible des processeurs ne sont pas tous les mêmes (ceci étant dû au fait que les processeurs n'ont pas les mêmes polynômes et que les coefficients a des w(x) sont différemment répartis dans de petits intervalles du fait de leur primalité et de la valeur de leur résidu quadratique). En fait cette stratégie est équivalente, en temps d'exécution, à celle qui ferait générer et stocker à chaque processeur tous les polynômes d'une famille dans l'étape d'initialisation, puis qui utiliserait les polynômes dans cette étape.

4.4.2. Distribution de l'intervalle de crible

Dans ce cas, tous les processeurs possèdent la base complète. Ils n'ont qu'un sous-intervalle de l'intervalle de crible. Ils travaillent donc tous avec les mêmes polynômes. Les stratégies dépendent du mode d'acquisition des polynômes.

* Chaque processeur a tous les polynômes en mémoire. Il a un intervalle de crible moins long, mais son cas revient au cas correspondant du paragraphe a précédent. Cependant, il a plus de polynômes. Or comme l'intervalle est distribué, il faut que tous les processeurs criblent, à un moment ou à un autre, avec le même polynôme. On le voit, les processeurs sont indépendants les uns des autres. Si on veut obtenir l'intervalle de crible complet, il suffit de remettre bout à bout les sous-intervalles de chaque processeur. C'est une vue logique qui n'implique aucun traitement, aucune communication. Avec cette stratégie, il n'y a pas de perte de temps. Chaque processeur reçoit les polynômes d'un maître. L'étude réalisée au paragraphe a précédent s'applique ici (même si l'intervalle de crible est plus court).

* Chaque processeur calcule lui-même ses propres polynômes dès qu'il en a besoin. Evidemment, il n'y a pas de perte de temps en synchronisation et en communication. En revanche, on effectue des calculs redondants puisque chaque esclave doit calculer le même polynôme que les autres à un instant donné. De même que précédemment, on peut envisager un phénomène de compensation entre les temps de crible des processeurs.

4.4.3. Distribution de la base de facteurs

Dans ce cas, tous les processeurs ont l'intervalle de crible complet.
Et ils travaillent donc tous avec les mêmes polynômes.

Que se passe-t-il sur un processeur lors du crible ? L'intervalle n'est pas complètement criblé, sur un processeur, puisqu'on n'a utilisé qu'un sous-ensemble de la base. Si tous les processeurs ont utilisé le même polynôme lors de ce crible, alors en additionnant indice par indice les valeurs du crible, on a l'intervalle complètement criblé.

Pour obtenir le crible avec toute la base, il faut que tous les éléments de la base rencontrent l'intervalle complet, et ceci avec chaque polynôme. On peut donc envoyer l'intervalle de crible de processeur en processeur, tous les processeurs ayant le même polynôme. Ainsi, le premier processeur crible avec les premiers éléments de la base, puis envoie l'intervalle ainsi criblé au

deuxième processeur qui ajoute sa contribution aux valeurs émises par le processeur précédent en criblant avec d'autres éléments de la base, et ainsi de suite jusqu'au dernier processeur. Après être passé sur tous les processeurs, l'intervalle est complètement criblé avec ce polynôme. C'est donc le dernier processeur qui crible qui s'occupe de la recherche des w(x) complètement factorisés. Ceci peut se réaliser par un parcours des processeurs sur un anneau. Le problème provient du fait qu'un seul processeur travaille alors que tous les autres restent inactifs. Si on garde l'idée du pipeline de base, tous les processeurs génèrent le même polynôme (d'où redondance) et tous les processeurs criblent avec leur sous-base.

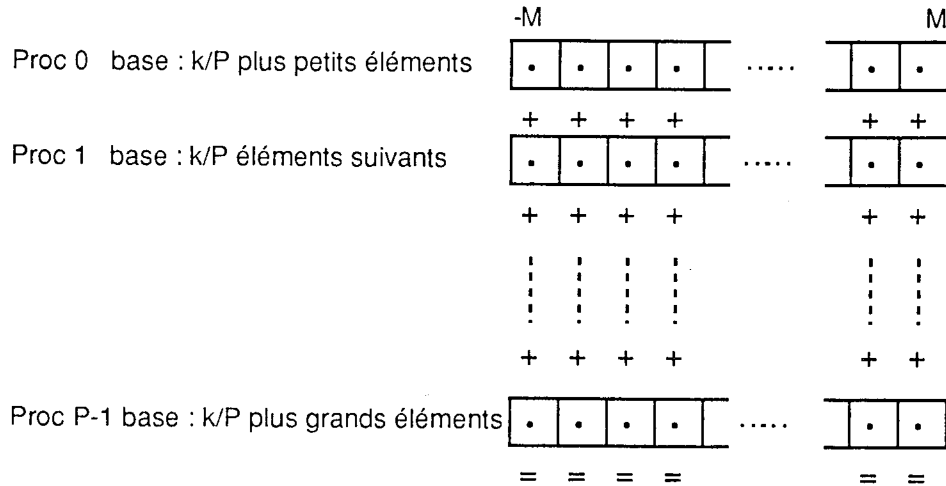


Figure 4.30. Sommation des contributions des processeurs (stratégie de la distribution de la base)

On peut atteindre un meilleur compromis avec un multipipeline. En effet, supposons que chaque processeur génère un polynôme propre (partitionnement de l'ensemble des polynômes en P familles distribuées chacune sur un processeur) dans un premier temps, puis qu'il crible avec ses propres éléments de la base et son polynôme dans un deuxième temps.

Imaginons ensuite qu'il envoie l'intervalle criblé au processeur suivant (sur un anneau par exemple), puis que chaque processeur crible avec ses propres éléments de la base et le polynôme reçu, l'intervalle reçu, et que chacun répète P-1 fois cette phase d'envoi et de crible, alors on aura à cet instant P intervalles criblés complètement avec n polynômes différents. Chaque processeur cherche alors sur l'intervalle qu'il possède actuellement les w(x) factorisés.

Après cette étape complète, chaque processeur recommence en générant un autre polynôme, criblant, envoyant au suivant et criblant P-1 fois, factorisant, et ainsi de suite. La factorisation des w(x) doit se faire sous forme de rotation sur l'anneau.

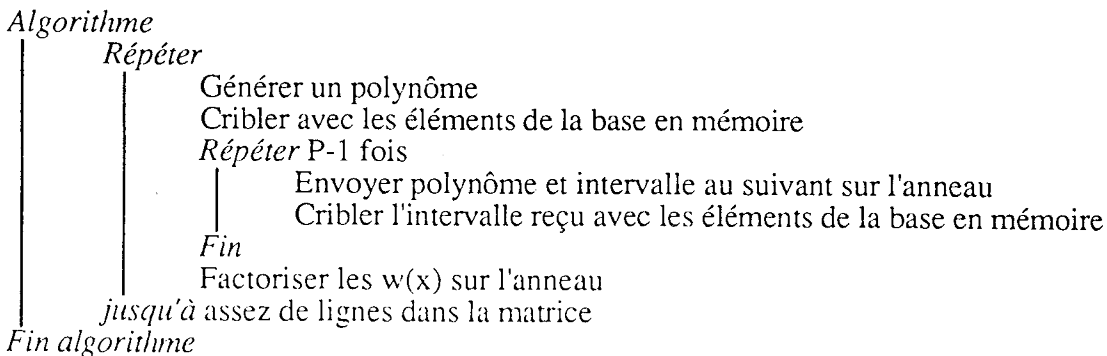

Algorithme
Répéter
Générer un polynôme
Cribler avec les éléments de la base en mémoire
Répéter P-1 fois
Envoyer polynôme et intervalle au suivant sur l'anneau
Cribler l'intervalle reçu avec les éléments de la base en mémoire
Fin
Factoriser les w(x) sur l'anneau
jusqu'à assez de lignes dans la matrice
Fin algorithme

Figure 4.31. Algorithme de l'étape de crible avec envoi de l'intervalle de crible

Bien sûr les opérations d'un seul pipeline ne sont pas de même taille. Mais comme tous les processeurs exécutent la même opération au même moment, il n'y a pas de problème d'inactivité ou de goulot d'étranglement.

On peut envoyer les éléments de la base de processeur en processeur plutôt que d'envoyer l'intervalle de crible. L'algorithme multipipeline précédent s'adapte à cette stratégie, sauf qu'il faut envoyer la partie de la base que possède ce processeur vers le suivant au lieu d'envoyer l'intervalle de crible.

En fait, l'envoi de la base peut se réduire à envoyer les indices de départ, (pour chaque p_i^α de la base) les p_i et les α. Ainsi le calcul de l'indice de départ pour ce processeur est facile : un simple modulo sur 32 bits (plus besoin de calculs en précision entière illimitée).

On peut aussi utiliser la réduction par addition. Chaque processeur génère un polynôme, le même pour tous, puis crible avec ses propres éléments de la base. Une étape de communication-réduction par addition est effectuée en log P (réseau avec P processeurs), où les intervalles criblés sont envoyés et additionnés indice à indice.

Evaluons maintenant le coût des communications pour les deux stratégies utilisant le multi-pipeline. Dans les deux cas, le nombre d'envois est le même, mais c'est la taille des messages qui varie. L'envoi de l'intervalle complet et du polynôme est beaucoup plus coûteux que l'envoi du polynôme et d'une partie de la base. En effet, l'intervalle de crible est beaucoup plus important que la P^e partie de la base.

4.4.4. Conclusion

Donc entre ces deux stratégies, il vaut mieux choisir celle qui transmet des parties de la base.

4.5. LE CHOIX DES w(x) CANDIDATS À LA FACTORISATION ET LEUR FACTORISATION

Comme pour toutes les autres étapes, celle-ci doit être rapide et efficace. Choisir les candidats ne peut se faire que séquentiellement, sauf si on dispose, comme sur le NEC SX2 ou le Cyber 205 d'opérations vectorielles de test. Ce n'est pas le cas du FPS. Quant à leur factorisation, les divisions successives coûtent trop cher, et nous utilisons une méthode de comparaison de congruences pour prouver qu'un élément de la base divise le candidat. Ainsi, on évite toutes les divisions inutiles, en multiprécision entière.

4.5.1. Choix des candidats

Il s'agit ici de découvrir parmi les $w(x)$, $x \in [-M, M[$, ceux qui se factorisent complètement sur la base. Pour ce faire, on définit le minimum de la valeur de $w(x)$ sur $[-M, M[$ et on prend un minorant du logarithme V de cette valeur.

$$w(-M) \approx w(M) \approx M\sqrt{2N}$$
$$w(0) \approx -M\sqrt{N/2}$$
$$|w(x)|_{moyen} \approx M\sqrt{N}$$

V est un minorant de $\log(M\sqrt{N})$.
Or les $w(x)$ qui se factorisent sur la base peuvent donc s'écrire :

$$w(x) = \prod_{i=1}^{k} p_i^{\alpha_i(x)}$$

et donc

$$\log|w(x)| = \sum_{i=2}^{k} (\alpha_i(x) \log p_i).$$

Rappelons que, à l'indice y dans l'intervalle de crible, nous avons sommé les logarithmes des éléments de la base qui divisent w(y). Donc, les w(y) qui se factorisent complètement sur la base sont ceux pour lesquels on trouve à l'indice y une valeur supérieure ou égale à V. Si on trouve une valeur inférieure, alors il manque dans la somme des log, un ou des log de facteurs supérieurs à p_k alors w(x) ne se factorise pas sur la base.

La première étape consiste donc à parcourir séquentiellement (pour chaque processeur) l'intervalle criblé, et déterminer les indices x pour lesquels la valeur de la somme des log des facteurs premiers de w(x) présents dans la base dépasse V. Il n'est pas possible de faire plus rapide sur le FPS T40.

4.5.2. Leur factorisation

Il y a deux possibilités :
- par divisions successives,
- par égalité de congruences.

4.5.2.a. Par divisions successives

La technique qui consiste à factoriser w(x) par divisions successives est très coûteuse. Rappelons qu'un entier y a, en moyenne, log log y diviseurs premiers [Rie 87 p. 164], ce qui donne 5 pour un entier de 100 décimales. Cependant, nous ne nous trouvons pas ici dans un cas général. En effet, tous les diviseurs premiers de w(x) sont inférieurs à p_k. Le nombre de facteurs premiers est donné dans le tableau suivant :

N	Nombre moyen de facteurs premiers de N
10^{100}	8,86
10^{80}	7,97
10^{60}	6,95
10^{40}	5,76

Figure 4.32. Nombre de diviseurs de w(x)

En fait, un nombre de 50 chiffres a au plus une trentaine de facteurs premiers différents. Il en a donc en moyenne un nombre inférieur.

Donc pour N de l'ordre de 10^{100}, k = 35 000, on peut s'attendre à ce que le nombre de diviseurs des w(x) qui se factorisent complètement sur la base soit de l'ordre de 10. En moyenne on effectuera donc un très grand nombre de divisions inutiles, puisque $k \gg h(w(x))$. De plus, comme w(x) est codé en multiprécision entière, ces divisions doivent être faites en multiprécision entière : elles sont très chères.

4.5.2.b. Par égalité de congruences

Une autre solution consiste à vérifier si le point de départ du crible sur l'intervalle pour ce polynôme sur ce processeur est congru à x, ceci pour chaque élément p_i de la base. Si oui, alors w(x) est divisible par p_i. On effectue alors la division en précision entière illimitée. On peut s'affranchir du test de congruence avec les puissances de cet élément, en divisant une fois de plus w(x) par cet élément de la base. Et on ne fait, pour chaque diviseur premier ayant $\alpha_{max} > 1$, qu'une division inutile.

Pour cette étape de factorisation des w(x), on a besoin de la base complète pour déterminer si un w(x) se factorise complètement sur cette base. En fait on a besoin des éléments de la base et soit des racines carrées (pour recalculer les points de départ du crible), soit de ces points de départ si on les a stockés.

La méthode la plus rapide pour cette factorisation est, bien entendu, que les processeurs aient connaissance de la base complète. De cette manière, on évite toute communication. Si malheureusement la base est répartie, on peut envisager de faire tourner les w(x) de processeur en processeur sur un anneau pour que chaque w(x) soit testé par rapport à tous les éléments de la base. Mais le problème vient du nombre variable de w(x) sur chaque processeur.

4.5.2.c. Factorisation en fonction de la variable de boucle qui est distribuée

Il faut donc introduire un algorithme de détection de la terminaison et de terminaison de ce calcul (tel celui présenté dans le chapitre 2). On peut aussi envisager de faire tourner les w(x) par paquets, c'est-à-dire que tous les candidats de chaque processeur sont testés ensemble (séquentiellement) sur le processeur qui les a découverts. Puis chaque processeur les fait parvenir à son successeur sur l'anneau, qui les testera tous séquentiellement par rapport à la partie de la base qu'il possède. Et ainsi de suite sur tous les processeurs. Dans ce cas, il n'y a plus besoin d'algorithme de détection de la terminaison, mais le temps de calcul est directement lié au nombre de w(x) candidats dans le processeur qui en a le plus. Ceci introduit des temps d'inactivité importants chez les processeurs qui ont moins de candidats à tester.

Une dernière solution consiste à faire tourner non plus les candidats, mais les parties de la base, afin que toutes les parties de la base rencontrent tous les processeurs. C'est le principe de multipipeline, chaque pipeline n'étant utilisé qu'avec un jeu de données.

Pour choisir entre ces 2 dernières solutions, il faut comparer la taille de l'ensemble des w(x) candidats et de la partie de la base que possède chaque processeur. L'envoi d'un w(x) nécessite la valeur de x et le coefficient a au moins (puisque b et c peuvent en être déduits facilement). C'est la taille minimum. Il est cependant utile de joindre à ces deux données la valeur non encore factorisée de w(x), c'est-à-dire, par exemple, les facteurs de w(x) déjà trouvés. Or il y a peu de candidats par polynôme [PST 88 p. 402]. La taille est faible, donc il est assez réaliste de transmettre les w(x) et les informations inhérentes plutôt que les parties de la base.

4.5.3. Conclusion

Le choix des candidats est séquentiel. On ne peut pas faire plus rapide, car il est nécessaire de tester pour chacun la valeur de la somme des logarithmes à cet indice.

Quant à laur factorisation, elle est rapide en utilisant l'égalité des congruences, mais seulement si la base est complète sur tous les processeurs. Dans les autres cas, il faut mettre en place des communications qui sont très coûteuses.

4.6. L'ÉLIMINATION DE GAUSS ET LA FACTORISATION DE N

La phase de crible précédente permet de calculer les factorisations de certains w(x) sur la base. Ces factorisations sont stockées dans une matrice $M = (M_{ij})$ où chaque ligne correspond à un w(x) complètement factorisé, chaque colonne correspond à un élément de la base de facteurs et où M_{ij} est la multiplicité de p_j dans la factorisation de $w_i(x)$ avec

$$w_i(x) = \prod_{j=1}^{k} p_j^{M_{ij}}.$$

Cette matrice M a donc k colonnes. Notre but est de trouver une ou des combinaisons linéaires de lignes telle(s) que le produit de ces lignes soit un carré parfait. Pour être sûr de trouver au moins une combinaison linéaire, il faut que la matrice ait au moins k lignes. Or, à partir de 0,96 k lignes environ pour k colonnes [Sil 87], la probabilité est grande de trouver au moins une combinaison, donc il n'est pas toujours nécessaire de chercher k lignes. De plus, M est creuse, puisque chaque ligne contient environ 10 éléments non nuls sur plus de 30 000 pour $N \approx 10^{100}$. Pour trouver les combinaisons linéaires, on transforme la matrice en remplaçant chacun de ses éléments par sa classe dans $\mathbb{Z}/2\mathbb{Z}$. Ensuite, on effectue une élimination de Gauss. Celle-ci nous dit s'il y a des combinaisons linéaires nulles, et en conservant la trace des transformations réalisées, permet de connaître les lignes qui sont combinaisons linéaires nulles

dans $\mathbb{Z}/2\mathbb{Z}$. L'élimination de Gauss dans $\mathbb{R}$ et dans $\mathbb{Z}/2\mathbb{Z}$ a été largement étudiée en séquentiel et en parallèle. Nous renvoyons le lecteur à [PaW 84], [Wie 86] et [CTV 87a], par exemple.

Lorsqu'on a trouvé une combinaison linéaire, on en déduit par quelques calculs (présentés dans l'introduction) et un pgcd, une factorisation de N. Avec une probabilité de 0,5 au moins, le pgcd n'est pas trivial (1 ou N). C'est pourquoi il est plus sûr, pour obtenir une factorisation non triviale de N, d'avoir plusieurs combinaisons linéaires. Si toutes les combinaisons linéaires conduisent à une factorisation triviale, il faut générer d'autres lignes de la matrice, trouver d'autres combinaisons linéaires jusqu'à l'obtention d'un pgcd différent de 1 ou de N.

5. CONCLUSION

Nous avons étudié la parallélisation théorique de l'algorithme du crible quadratique. Nous avons montré (indépendamment de la taille mémoire) que certaines possibilités convenaient meiux pour des machines que pour d'autres. Nous nous en inspirons au chapitre 5 pour l'implantation sur le FPS T40.

Sur des multiprocesseurs à mémoire illimitée, le crible quadratique multipolynomial s'implante de manière efficace, en choisissant de calculer et stocker tous les polynômes qui seront nécessaires à la factorisation du nombre N lors de l'étape d'initialisation. Puis, dans l'étape de crible-factorisation, il faut éviter les communications, en effectuant des calculs indépendants et non pas coopératifs pour la construction des lignes de la matrice.

En fait, on essaie, pour toutes les étapes, de se rapprocher le plus près possible d'une exécution sur une ferme de processeurs, c'est-à-dire sans aucune communication entre les processeurs. Ceci n'est pas toujours possible. En effet, dans l'étape d'initialisation, il faut connaître le nombre d'éléments générés. Pour ceci, il est nécessaire de communiquer des informations entre voisins. De même, si la matrice est répartie par blocs ou par groupes de lignes sur les processeurs, l'élimination de Gauss ne peut pas se faire sans communications. Mais l'étape la plus coûteuse, le crible lui-même (comprenant aussi la factorisation des w(x)) peut se faire sans une seule communication, pourvu qu'on choisisse de distribuer les polynômes ou l'intervalle de crible entre les processeurs. Si on répartit la base de facteurs, les communications sont importantes, ce qui occasionne des pertes de temps non négligeables.

On peut en déduire une méthodologie pour obtenir un algorithme efficace : il faut, dans notre cas, déterminer les variables qui sont nécessaires sur chaque processeur pour toute la durée de l'exécution, et celles qui ne le sont pas. Si on distribue les premières, on a besoin de communiquer pour les connaître à un moment ou à un autre. Dans notre cas, la distribution de la base implique de nombreuses et coûteuses communications. Si on distribue les autres, on risque de perdre du temps dans certaines phases de l'exécution pour recommencer des calculs déjà effectués, mais ce temps est négligeable par rapport aux communications. Dans notre cas, la distribution de l'intervalle ou des polynômes fournit une solution efficace, par rapport à la distribution de la base.

Après cette présentation des possibilités d'implantation de l'algorithme du crible quadratique sur des multiprocesseurs distribués à mémoire illimitée, nous resserrons les contraintes en diminuant la taille mémoire disponible à une valeur plus réaliste de 1 Mo par nœud. L'étude réalisée sous cette contrainte permettra alors de dériver une implantation sur une machine multiprocesseur à mémoire distribuée, le FPS T40, présentée au chapitre 5.

Chapitre 5

Le crible quadratique :

Parallélisation et implantation

sur l'hypercube FPS T40

1. INTRODUCTION

Dans le chapitre 4, nous avons étudié des parallélisations théoriques de chaque étape de l'algorithme du crible quadratique multipolynomial dans un environnement à mémoire distribuée de taille illimitée. Notre but, dans ce chapitre, est de dériver une implantation efficace pour cet algorithme sur le FPS T40, à partir des résultats et évaluations théoriques obtenus au chapitre précédent.

Nous nous fixons, comme taille des entiers à factoriser, des nombres de 40 à 60 chiffres. En dessous de 40 chiffres, il est assez facile et rapide de factoriser ces entiers. De plus, une large proportion des calculs peut être effectuée sur 32 bits. Or nous avons montré que, pour des entiers de 100 chiffres, les calculs sur les coefficients des polynômes nécessitent la multiprécision entière. Nous voulons nous placer dans le cadre général de la factorisation par le crible quadratique multipolynomial.

Cependant, nous sommes conscients que l'exécution de l'algorithme est très coûteuse en temps et en place mémoire. C'est pourquoi nous nous limitons à 60 chiffres environ. Nous espérons que les résultats obtenus puissent être crédibles, et à la base d'une extension de cet algorithme à des entiers plus grands, et ceci sans perte de généralité.

Nous ne revenons pas, dans ce chapitre, sur l'initialisation ni sur l'élimination de Gauss. En effet, l'initialisation peut être effectuée efficacement avec une mémoire de 1 Mo, car, même pour un entier de 100 chiffres, la base complète n'occupe pas plus de 500 Ko. Donc les restrictions sur la taille de la base n'influent pas sur l'efficacité. De la même manière, l'étape finale, l'élimination de Gauss, a été largement étudiée en parallèle. Nous renvoyons le lecteur à la bibliographie sur le sujet : [PaW 84], [Wie 86] et [CTV 87a] par exemple.

Ce chapitre est consacré à l'étude de l'implantation de la boucle de génération des lignes de la matrice (les w(x) factorisés sur la base) sur le FPS T40, puis à l'implantation de l'algorithme entier. Nous avons vu, au chapitre précédent, que deux solutions convenaient pour paralléliser cettte boucle, et qu'une troisième solution faisait intervenir trop de communications.

La première solution consiste à découper l'intervalle de crible et à le répartir entre les processeurs. Chaque processeur traite ce sous-intervalle avec la base complète et les mêmes polynômes. Nous avons montré que le calcul des polynômes posait des problèmes. Soit le

calcul est redondant de manière à éviter les communications, soit un processeur dédié génère tous les polynômes et les diffuse, ce qui implique des communications. De plus, le sous-intervalle est court. Or, cette étape est d'autant plus efficace que l'intervalle est long. Donc cette solution n'est pas très intéressante.

La deuxième solution consiste à répartir l'ensembles des polynômes. Chaque processeur génère ses propres polynômes dans une famille. Il n'y a aucune redondance de calcul. Il n'y a pas de communication. Cette solution est nettement meilleure. C'est celle que nous choisissons d'implanter sur le FPS T40.

Sur des entiers de 35 à 41 chiffres, nous étudions le taux d'utilisation des processeurs. En dépouillant les résultats, nous remarquons les problèmes qui se posent lors de la parallélisation, par exemple les communications, les synchronisations et la répartition des tâches entre les processeurs. Une analyse secondaire permet de dériver un deuxième algorithme beaucoup plus performant, tant du point de vue du temps d'exécution que de la répartition équilibrée des tâches. Nous essayons aussi de discuter des implantations possibles sur l'iPSC/2, le TNode (ou le MegaNode), la Connection Machine.

Quant à la troisième solution, elle consiste à distribuer la base de facteurs. Elle implique de nombreuses communications. Nous donnons une évaluation de ces coûts en essayant de répartir les sous-bases sur les processeurs ou de les lire sur disques. Nous n'envisageons pas une implantation de cette solution.

Notre but est de montrer que, grâce à une étude fine des étapes d'un algorithme, il est possible de tirer parti de la puissance de la machine-cible.

2. LA BOUCLE DE GÉNÉRATION DE LA MATRICE DES FACTORISATIONS COMPLÈTES

Rappelons informellement cette boucle :

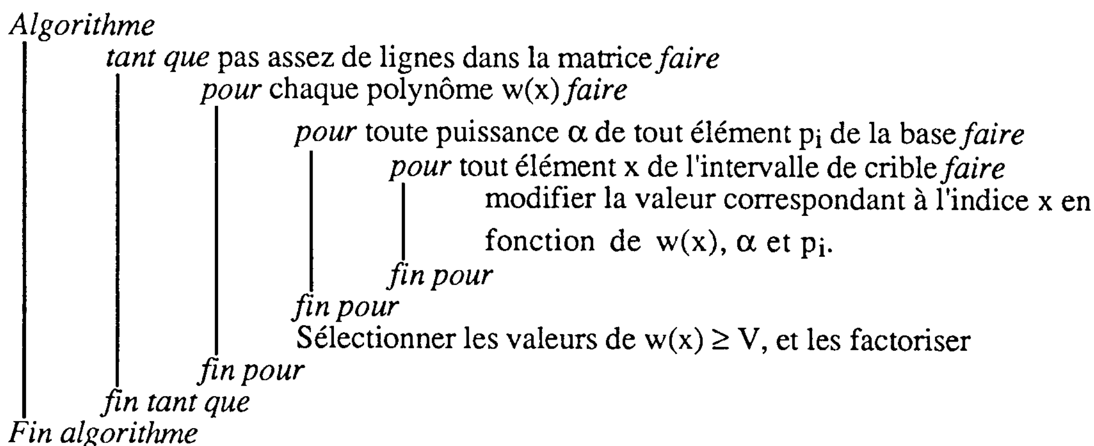


Figure 5.1. Boucles de génération de la matrice des w(x) factorisés

Elle consiste à générer les w(x) complètement factorisés (les lignes de la matrice).

Les méthodes mathématiques sous-jacentes ont été présentées ou référencées dans le chapitre précédent. Elles sont encore valables dans un environnement distribué où la mémoire est limitée à 1 M octets par nœud.

Nous reprenons donc ici les 3 grandes parties de l'étape de construction de la matrice des factorisations (génération des polynômes, crible de l'intervalle [-M, M[avec les polynômes et

factorisation des w(x) pour la construction de la matrice). Nous les étudions globalement car elles sont dépendantes les unes des autres.

Dans un environnement distribué où la mémoire est limitée sur chaque processeur, il est actuellement difficile de vouloir calculer tous les polynômes avant la boucle et de les stocker dans la mémoire des processeurs, même de manière répartie. En effet, rappelons que pour un nombre N de 100 chiffres, l'ensemble des polynômes représente un espace mémoire de 700 Mo environ. Il n'existe pas de machines ayant 700 Mo par nœud, et, à part la Connection Machine 2 (de 256 Mo à 1 Go pour la version à 64 K processeurs), aucun multiprocesseur distribué dont la mémoire totale atteigne ou dépasse 700 Mo.

En fait, il est très difficile d'évaluer les autres possibilités de génération des polynômes, adaptées à la mémoire illimitée (présentées au chapitre 4), indépendamment de la boucle complète, c'est-à-dire déconnectées de l'étape de crible et de factorisation des w(x). C'est pourquoi nous regroupons ces trois étapes et nous les étudions globalement.

Pour construire cette matrice, il est nécessaire de cribler un intervalle avec les éléments d'une base de facteurs afin de trouver des valeurs de polynômes qui se factorisent complètement sur cette base : les trois variables des trois étapes sont intimement liées.

La mémoire limitée est certes une contrainte, mais on peut la relâcher en définissant, dans certains cas, une mémoire virtuelle, accessible soit logiquement par l'algorithme, soit physiquement sur des disques externes par le programme. Cependant, sur le FPS T40, la capacité des disques est limitée, et le débit des liens qui les relient aux processeurs est faible par rapport au débit d'une mémoire centrale. C'est pourquoi les disques externes peuvent n'introduire, dans certains cas, que des gênes, des goulots d'étranglement, ou des pertes de temps.

Etudions les stratégies de distribution des variables (polynômes, intervalle, base) des boucles de l'algorithme de l'étape de crible-factorisation.

2.1. CONSTRUCTION DE LA MATRICE DES FACTORISATIONS PAR DISTRIBUTION DES POLYNÔMES

Les polynômes sont répartis par famille entre les P processeurs. Donc chaque processeur doit effectuer le crible sur l'intervalle complet avec la base complète. C'est la solution la plus rapide en mémoire illimitée.

Nous montrons que si la base et l'intervalle ne tiennent pas en mémoire, cette solution n'est pas aussi bonne que nous l'avions prédit, puisqu'elle nécessite des échanges de données, plus rapides entre voisins que par des lectures sur disque.

Sur aucune des trois machines FPS T40, iPSC/2, TNode, la mémoire, même totale, n'est assez importante pour permettre de stocker une famille complète de polynômes par processeur. Par exemple sur le FPS T40, chaque famille représente près de 22 Mo pour $N=10^{100}$. Et la mémoire n'offre que 1 Mo (dans lequel il faut loger le code et les données). Et même avec un disque externe de 85 Mo par cube de 8 processeurs, l'espace n'est pas suffisant. L'option de calcul de tous les polynômes et de stockage lors de l'initialisation est impossible à réaliser sur les machines. Sur le TNode, la mémoire est de même taille, donc le problème persiste. Sur l'iPSC/2, il y a 4 Mo de mémoire par nœud. Mais elle ne suffit pas à contenir une famille de polynômes.

Quant à la Connection Machine 2, elle peut disposer d'un espace total de 1 Go. Donc l'ensemble des polynômes tient dans la mémoire totale et en laisse libre 30 % environ. Or chaque nœud ne possède que 256 Ko au total. Il reste donc moins de 80 Ko utilisable sur chaque nœud, ce qui ne permet en aucun cas de contenir la base et l'intervalle complets. Chacun d'eux nécessite respectivement au moins 300 Ko pour la base et environ 8 Mo pour l'intervalle. Donc, les polynômes ne sont pas stockés a priori par les processeurs. Si chaque

processeur reçoit tous les polynômes d'un maître qui les calcule, la mémoire disponible sur chaque nœud peut ne pas être suffisante, sur aucune des 4 machines, pour contenir à la fois la base complète et l'intervalle de crible, suivant la valeur de N. Mais dans ce cas, on peut envisager de découper en morceaux l'intervalle, et même la base, si cela s'avère nécessaire.

2.1.1. Découpage de l'intervalle de crible et de la base en sous-ensembles

Il est aisé de couper l'intervalle en sous-intervalles, car en fait, on réalise des cribles complets avec la base sur de petits intervalles. Dès qu'on a criblé un sous-intervalle, on recherche les w(x) candidats à la factorisation complète et on les factorise. Puis on passe au sous-intervalle suivant. Il est alors judicieux de conserver les points de départ dans le sous-intervalle précédent afin d'éviter des calculs déjà effectués. Et ainsi de suite jusqu'à avoir criblé l'intervalle complet. Mais ceci n'est possible que dans le cas où la base tient entièrement sur chaque processeur. Si ce n'est pas le cas, il faut couper la base en sous-bases, et stocker les sous-bases sur disque, ou éventuellement sur des voisins, et respectivement lire sur disque ou communiquer avec ces voisins lorsqu'il faut travailler avec une autre sous-base.

Dans ce cas (l'intervalle est lui aussi coupé en sous-intervalles) pour cribler un intervalle complet, il faut cribler séquentiellement les sous-intervalles. Et pour chaque sous-intervalle, il faut cribler avec toutes les sous-bases, ce qui nécessite un nombre important d'échanges avec le disque ou avec les voisins. Ce qui coûte très cher en temps. Cette stratégie sera étudiée plus loin sur le FPS T40, et son étude s'adapte aussi aux autres machines.
Cette technique de découpage peut d'ailleurs s'appliquer à la première solution proposée ci-dessus pour la Connection Machine. Mais elle ne semble pas optimale : pourquoi engorger la mémoire locale avec des polynômes (dont un seul est utile à chaque boucle) alors que la base et l'intervalle sont nécessaires à chaque instant ?

Si chaque processeur calcule lui-même ses polynômes selon ses besoins, il n'y a pas de perte de temps en communication des polynômes. Mais il peut y avoir des pertes de temps en lecture sur disque ou en communication avec des voisins, s'il est nécessaire de découper l'intervalle en sous-intervalles et éventuellement la base en sous-bases. Car rappelons que chaque processeur possède toute la base et tout l'intervalle de crible.

Evaluons le coût des communications entre processeurs pour s'échanger des données et le coût des lectures sur disque pour acquérir ces mêmes données (que des voisins possèdent peut-être), sur le FPS T40.

2.1.2. Evaluation des coûts d'accès aux sous-ensembles de la base et de l'intervalle

2.1.2.a. Communications entre voisins

Les communications ont cet avantage d'être peu coûteuses en temps et cet inconvénient de nécessiter que les deux protagonistes soient prêts à communiquer (synchronisation obligatoire). De plus, il doit exister en mémoire sur chaque processeur une zone libre pour recevoir les données.

Une communication entre voisins coûte $0{,}00086 + 0{,}144.10^{-5}.n$ secondes pour transférer un message de n octets entre deux voisins sur l'hypercube.

2.1.2.b. Lectures sur disques

Les lectures sur disque ont cet avantage de n'avoir pas besoin de synchroniser deux processeurs pour avoir lieu et ces inconvénients d'être coûteuses et d'être exclusives si deux processeurs d'un même cube de 8 processeurs veulent accéder au disque simultanément. Voici la courbe donnant les temps de lecture et d'écriture en fonction du nombre de processeurs et de la taille de l'information :

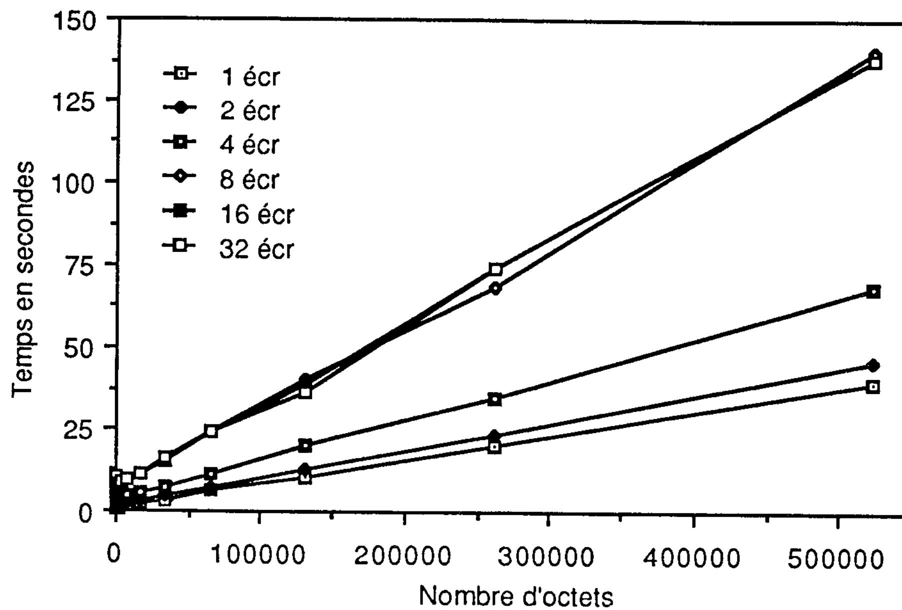


Figure 5.2. Temps d'écriture sur disque en fonction de la taille du message et du nombre de processeurs

Quant aux temps de lecture sur disque, nous les avons mesurés en fonction de la taille du message à lire et du nombre de processeurs qui accèdent en même temps à un disque (8 processeurs au plus) ou à plusieurs disques (16 et 32 processeurs) puisqu'il y a un disque par cube de 8 processeurs. Nous en avons profité pour mesurer les temps d'écriture dans les mêmes conditions. Nous donnons les temps de lecture et d'écriture dans le tableau suivant :

Nb octets	1 écr	1 lec	2 écr	2 lec	4 écr	4 lec	8 écr	8 lec	16 écr	16 lec	32 écr	32 lec
4	1,15	0,35	2,32	0,38	4,69	0,57	9,42	0,91	10,67	0,94	10,21	0,96
8	1,18	0,35	2,02	0,36	3,98	0,52	8,01	0,93	8,00	0,93	8,22	0,99
16	1,12	0,32	1,98	0,38	3,97	0,52	7,94	0,92	7,82	0,89	7,76	0,95
32	1,12	0,35	2,03	0,36	3,83	0,55	7,69	0,92	7,69	0,92	7,68	0,97
64	1,11	0,32	1,98	0,40	3,81	0,54	7,62	0,91	7,67	0,97	7,82	0,99
128	1,12	0,36	2,02	0,40	3,85	0,54	7,75	0,91	7,75	0,96	7,86	1,01
256	1,18	0,35	2,11	0,40	3,84	0,56	7,71	0,91	7,66	0,90	7,82	0,97
512	1,12	0,35	2,11	0,40	3,83	0,55	7,66	0,94	7,65	0,98	7,93	0,95
1 024	1,11	0,32	2,12	0,40	3,89	0,57	7,69	0,92	7,66	0,94	7,81	0,87
2 048	1,17	0,41	2,12	0,48	4,02	0,70	7,89	1,19	7,93	1,27	8,10	1,30
4 096	1,33	0,54	2,34	0,65	4,33	0,84	8,45	1,66	8,45	1,71	8,48	1,81
8 192	1,63	0,83	2,58	1,00	4,77	1,33	9,40	2,63	9,60	2,74	9,65	2,75
16 384	2,22	1,41	3,36	1,65	5,78	2,42	11,55	4,53	11,49	4,69	11,56	4,75
32 768	3,55	2,72	4,71	2,93	7,62	4,15	15,47	8,44	15,42	8,69	15,83	8,63
65 536	6,15	5,34	7,23	5,58	11,57	8,02	24,03	16,49	23,98	16,68	24,16	16,92
131 072	10,64	9,85	12,61	10,82	19,77	16,00	40,03	32,17	38,41	30,93	36,56	29,34
262 144	20,29	19,26	23,56	21,89	35,08	30,99	68,71	61,08	73,83	64,80	73,90	64,52
524 288	39,77	38,87	45,82	43,64	68,15	63,63	140,00	133,01	138,10	128,92	138,05	129,01

Figure 5.3. Temps de lecture et d'écriture en secondes sur les disques du T40 en fonction du nombre de processeurs et du nombre d'octets lus ou écrits.

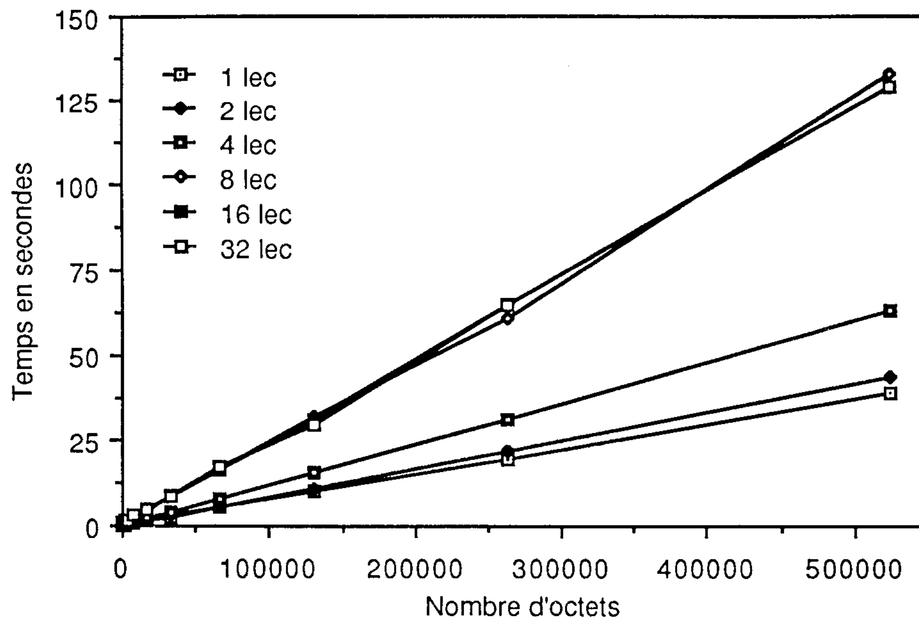


Figure 5.4. Temps de lecture sur disque en fonction de la taille du message et du nombre de processeurs

On remarque que les temps sont sensiblement les mêmes lorsque plus de 8 processeurs accèdent simultanément aux disques. C'est normal puisqu'il y a un disque par groupe de 8 processeurs. Lorsqu'on dépasse 8 processeurs en accès simultanés, on augmente le nombre de disques, sans augmenter le nombre de processeurs par disque.

Les temps d'initialisation pour la lecture et l'écriture sont importants. En effet voici les régressions linéaires donnant les temps de lecture et d'écriture, en secondes, en fonction du nombre x d'octets du message.

Nb de Proc en écr	Temps d'écriture en s	Temps de lecture en s	Nb de Proc en lec
1 proc en écriture	$1{,}0960 + 7{,}365.10^{-5}$ x	$0{,}3003 + 7{,}333.10^{-5}$ x	1 proc en lecture
2 proc en écriture	$1{,}9883 + 8{,}319.10^{-5}$ x	$0{,}3167 + 8{,}242.10^{-5}$ x	2 proc en lecture
4 proc en écriture	$3{,}8312 + 12{,}19.10^{-5}$ x	$0{,}4172 + 11{,}97.10^{-5}$ x	4 proc en lecture
8 proc en écriture	$7{,}5713 + 24{,}87.10^{-5}$ x	$0{,}5262 + 24{,}80.10^{-5}$ x	8 proc en lecture
16 proc en écriture	$7{,}7306 + 24{,}86.10^{-5}$ x	$0{,}7349 + 24{,}38.10^{-5}$ x	16 proc en lecture
32 proc en écriture	$7{,}7477 + 24{,}79.10^{-5}$ x	$0{,}6987 + 24{,}33.10^{-5}$ x	32 proc en lecture

Figure 5.5. Temps de lecture et d'écriture sur disque (régressions linéaires)

On remarque deux choses :
- l'opposition lecture / écriture,
- l'opposition moins de 8 / plus de 8 processeurs en accès simultanés.

En ce qui concerne l'opposition lecture / écriture, le temps d'initialisation est vraiment différent, mais son augmentation en lecture est beaucoup moins importante qu'en écriture lorsque le nombre de processeurs croît.

En ce qui concerne l'opposition moins de 8 / plus de 8 processeurs en accès simultanés, c'est sur l'écriture que les différences sont les plus importantes. Le temps d'initialisation double lorsqu'on double le nombre de processeurs en accès simultanés sur le même disque. Bien évidemment, au-delà de 8 processeurs, on augmente le nombre de disques sans augmenter le nombre de processeurs par disque, donc les temps sont constants.

Il est plus coûteux de lire sur disque que de communiquer entre deux voisins. En effet le temps d'initialisation d'une communication est moins élevé que celui de lecture sur disque : 0,000 86 s contre 0,3 à 0,7 s. De même, le débit pour les communications entre voisins est plus élevé que le débit pour les lectures sur disque : 1,44 µs/octet contre 73 à 243 µs/octet. Cependant, le disque reste un support de grande taille pour l'archivage.

Après cette évaluation des temps de lecture sur disque, revenons au cas où chaque processeur génère ses polynômes (dans une famille qui lui est propre). S'il veut cribler l'intervalle entier avec la base de facteurs complète, il peut le simuler par criblage séquentiel de petits sous-intervalles, dans l'hypothèse où la base tient en mémoire. Sinon il est obligé de couper la base en sous-bases.

Pour $N = 10^{100}$ sur le FPS T40, la base occupe 500 Ko (300 Ko au minimum avec des calculs redondants à éviter) et l'intervalle de crible occupe 8 Mo. Or on a 750 Ko par processeur pour ces données (200 à 250 Ko sont réservés pour le code et le système), à répartir entre un sous-intervalle de taille I octets et un sous-ensemble de E octets de la base de facteurs. On a donc $I + E = 750\ 000$. Pour cribler un sous-intervalle, il faut faire $\lceil 500\ 000/E \rceil$ accès aux informations (sur un voisin ou sur disque). Et il y a $\lceil 8\ 000\ 000/I \rceil$ sous-intervalles à cribler. Au total, il y a donc $\lceil 500\ 000/E \rceil . \lceil 8\ 000\ 000/I \rceil$ accès obligatoires. Chaque lecture représente E octets. Donc au total, on accède à $T = E.\lceil 500\ 000/E \rceil . \lceil 8\ 000\ 000/I \rceil$ octets.

On veut minimiser T sous la contrainte I+E = 750 000.

$$\begin{cases} \min\left(E.\left\lceil \frac{500\ 000}{E} \right\rceil . \left\lceil \frac{8\ 000\ 000}{I} \right\rceil\right) \\ I + E = 750\ 000 \end{cases}$$

T est minimum pour I maximum. Mais ceci n'est vrai que si les lectures sur disque et les communications entre voisins sont simultanées, ce qui n'est pas le cas avec le FPS T40. Donc il faut faire beaucoup de petites lectures car on minimisera la taille totale lue, et par conséquent le temps pour ce transfert. En effet, plus E est petit, plus I est grand, et moins il faudra faire de calculs redondants. Avec E = 1 Ko et I = 749 Ko, il faut faire 11 000 lectures de 1 Ko, soit 11 Mo au total.

2.1.3. Conclusion

Les lectures sur disques sont plus coûteuses que les communications entre voisins. Nous voulons minimiser le temps d'exécution. C'est pourquoi il faut choisir d'échanger des données entre voisins plutôt que lire sur disques.

Cependant, nous savons que jusqu'à $N = 10^{100}$, la base tient sur chaque nœud (500 Ko). Donc, ce n'est qu'avec des entiers de plus de 100 chiffres que ce problème se pose.

2.2. CONSTRUCTION DE LA MATRICE DES FACTORISATIONS PAR DISTRIBUTION DE L'INTERVALLE DE CRIBLE

Distribuer l'intervalle de crible revient, pour l'exécution de l'algorithme, à réaliser un crible complet sur un intervalle plus petit. Il n'y a donc pas de problème, tant que la base tient sur chaque processeur. C'est ce que nous montrons dans ce paragraphe.

Evaluons la place nécessaire sur un processeur pour contenir la base complète et le sous-intervalle, sur le FPS T40 pour $N = 10^{100}$. La base occupe 500 Ko. L'intervalle complet nécessite 8 Mo, donc chaque sous-intervalle occupe 250 Ko sur chacun des 32 processeurs. Au total, il faut 750 Ko. Cet espace est disponible.

Bien sûr, pour N légèrement plus grand, on est de nouveau limité en place, et il faut alors couper l'intervalle sur chaque processeur en sous-intervalles et éventuellement la base en sous-bases. C'est en fait le problème rencontré si on choisit la taille de la base préconisée par Pomerance et al. [PST 88] ainsi que leur taille de crible. Dans ce cas, la base occupe 1 Mo et chaque sous-intervalle occupe 416 Ko sur chacun des 32 processeurs.

Rappelons que tous les processeurs travaillent avec les mêmes polynômes.

- Que chaque processeur ait tous les polynômes en mémoire à un même instant est impossible, même en utilisant les disques externes.
- Que chaque processeur reçoive ses polynômes d'un maître implique des communications trop importantes (et des synchronisations très coûteuses en temps).
- Il reste la solution où chaque processeur génère ses polynômes selon ses besoins. Il y a redondance, car tous les processeurs vont générer à un moment ou à un autre les mêmes polynômes. Cependant il n'y a pas de perte de temps en communication.

On remarque que dans cette stratégie, la base ne tient pas sur un processeur. Donc, il ne sert à rien de découper l'intervalle de crible de chaque processeur. On doit découper la base. Or le code occupe 200 à 250 Ko. Il reste donc 750 Ko pour les données. Comme l'intervalle occupe 416 Ko, il reste 334 Ko pour la base. On découpe donc en trois parties égales de 333 Ko chacune et on fait 3 accès (un pour chacune des 3 sous-bases) pour chaque sous-intervalle de crible.

On a donc pour chaque polynôme trois lectures sur disque, et aucune écriture puisqu'on garde le sous-intervalle en mémoire, et que lorsqu'on change de sous-intervalle, on remplace le précédent sans devoir le stocker : en effet on a trouvé les w(x) factorisés, par conséquent on n'a plus besoin de l'intervalle. Pour chaque polynôme, les 3 accès aux sous-bases représentent 1 Mo d'informations à transférer.

Comparons avec la solution du paragraphe 2.1. Celle-ci réalise le crible avec un polynôme propre à chaque processeur sur un intervalle complet par processeur. Et en fin de compte, elle obtient P intervalles complets criblés, P = 32 pour le FPS T40.

Combien cela coûte-t-il en termes d'accès ?
Au mieux, un processeur accède à 11 Mo. Donc les 32 processeurs accèdent à 352 Mo.
La solution actuelle permet d'obtenir un intervalle complet criblé avec un polynôme (car l'intervalle est distribué) et coûte 32 Mo avec 32 processeurs. Pour obtenir P intervalles criblés, il faut donc accéder à 1024 Mo.

La solution du §2.1 coûte moins cher !

2.3. CONSTRUCTION DE LA MATRICE DES FACTORISATIONS PAR DISTRIBUTION DE LA BASE DE FACTEURS

L'intervalle de crible ne tient pas sur un processeur, mais on peut le découper logiquement (en criblant séquentiellement des sous-intervalles) sans avoir recours à des communications concernant l'intervalle ou les sous-intervalles.
De même pour la génération des polynômes, chaque processeur génère pour lui-même ses propres polynômes, même si cela crée de la redondance.

Le problème maintenant est de savoir, dans cette stratégie, où aller réellement chercher les sous-bases : sur disque ou chez les voisins ? Communiquer avec des voisins est peu coûteux mais nécessite une synchronisation qui, elle, peut être très coûteuse. Lire sur disque est très coûteux et exclusif dans un cube de 8 processeurs mais ne nécessite aucune synchronisation.

On peut distinguer deux cas : soit chaque processeur a sa propre famille de polynômes et crible l'intervalle complet, soit tous les processeurs ont les mêmes polynômes et criblent des sous-intervalles. Etudions les échanges nécessaires (avec le disque ou avec un autre ou plusieurs autres processeurs) pour $N = 10^{100}$, la base occupant 500 Ko et l'intervalle 8 Mo.

Nous montrons que le crible des sous-intervalles est moins coûteux.

2.3.1. Le processeur s'occupe de l'intervalle de crible complet.

Evaluons la taille des informations nécessaires, et dont ce processeur ne dispose pas dans sa mémoire.

2.3.1.a. Taille des informations à acquérir

Premièrement il est clair qu'il n'aura en mémoire qu'un sous-intervalle de l'intervalle complet. Soit I la longueur en octets de ce sous-intervalle. Il va cribler complètement ce sous-intervalle avec la base complète. Comme la mémoire utile pour les données est 750 Ko, il devra ne cribler qu'avec (750 000-I) octets de la base. Il devra donc effectuer $\lceil 500\,000/(750\,000-I) \rceil$ lectures sur disque ou communications avec un ou des voisins pour chaque sous-intervalle. Il va effectuer ces accès sur $\lceil 8\,000\,000/I \rceil$ sous-intervalles donc un total de $\lceil 500\,000/(750\,000-I) \rceil . \lceil 8\,000\,000/I \rceil$. Chaque accès se fait sur (750 000 - I) octets.

Supposons que les 32 processeurs effectuent cette lecture simultanément sur disque. Cela coûte $7{,}7477 + 2{,}479.10^{-5}\,(750\,000 - I)$ secondes par lecture, et au total :

$$\left\lceil \frac{500\,000}{750\,000 - I} \right\rceil . \left\lceil \frac{8\,000\,000}{I} \right\rceil . \left(7{,}7477 + 2{,}479.10^{-5}.(750\,000 - I)\right).$$

Cette fonction de I est minimum pour I = 375 000, et elle vaut alors 1790 secondes, donc environ une demi-heure de lecture sur disque par polynôme.

En ce qui concerne les communications entre processeurs, il faut répartir les informations dans l'hypercube de manière à n'avoir pas de routage à faire pour accéder aux informations : il faut donc que les informations se trouvent sur un voisin, et ceci pour chaque processeur. Il faut minimiser le nombre de startups et de synchronisations, donc le nombre de communications. Le nombre de communications (et de synchronisations) est 44 pour un polynôme, valeur minimum atteinte pour I = 375 000.

Comment répartir les 3 sous-ensembles de la base sur les sommets d'un hypercube de manière qu'un sommet ait toujours parmi ses voisins les deux autres sous-ensembles ?

Ce n'est pas possible sur un hypercube de dimension 2 :

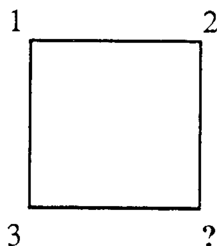


Figure 5.6. Répartition des trois sous-bases sur un hypercube de dimension 2

En revanche, sur un hypercube de dimension 3, il est possible de trouver un agencement des sous-ensembles qui satisfasse la condition ci-dessus :

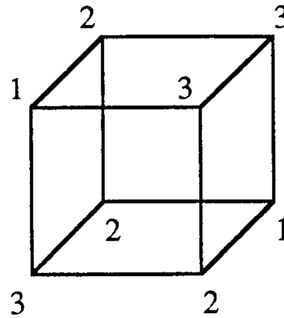


Figure 5.7. Répartition des trois sous-bases sur un hypercube de dimension 3

Il y a même plusieurs possibilités pour agencer ces trois sous-bases en respectant la condition de voisinage. La figure 5.7 donne une de ces configurations.

Ensuite, par récurrence, il est très simple de trouver un agencement : pour construire un hypercube de dimension 4, on juxtapose deux hypercubes de dimension 3 avec l'agencement préconisé ci-dessus. Puis on lie deux à deux les sommets correspondants à ces deux hypercubes. On ne modifie pas les liens existants. Donc on ne modifie pas les voisinages. On rajoute un voisin par processeur. Par la suite, ce processeur aura peut être un choix plus grand pour le processeur avec lequel il voudra échanger des données :

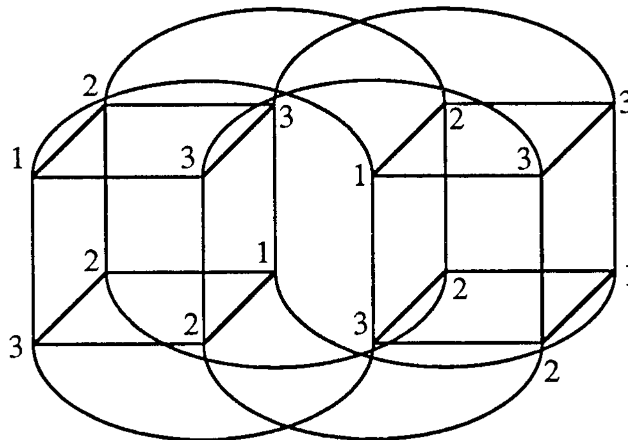


Figure 5.8. Répartition des trois sous-bases sur un hypercube de dimension 4

En fait, il faut ajouter que lorsqu'un processeur a terminé avec un sous-ensemble 1 par exemple, il le remplace par un sous-ensemble 2, puis par un sous-ensemble 3, et par la suite, il doit pouvoir récupérer le sous-ensemble 1 précédent. Donc, il faut une certaine logique dans les rotations des sous-ensembles, de manière qu'à chaque instant les processeurs aient toujours les deux sous-ensembles complémentaires du leur, à portée directe. On peut mettre au point cette stratégie sur un cube de dimension 3, si elle existe, et la porter par récurrence sur les hypercubes de dimensions supérieures qu'on aura pris soin de séparer en cubes de dimension 3.

2.3.1.b. Echange des informations par communication entre voisins sur un 3-cube

Voyons donc s'il existe une possibilité d'avoir toujours cette condition vraie sur un cube de dimension 3. On oublie pour l'instant comment les processeurs échangent leurs blocs de données, s'ils le font de manière synchronisée (huit par huit) par une rotation sur un anneau, ou deux par deux ou quatre par quatre.

Etudions tous les choix possibles de coloration d'un hypercube de dimension 3 avec trois couleurs, tels que cette coloration respecte la relation : un sommet de couleur donnée doit avoir parmi ses voisins immédiats au moins un sommet de chacune des deux autres couleurs.

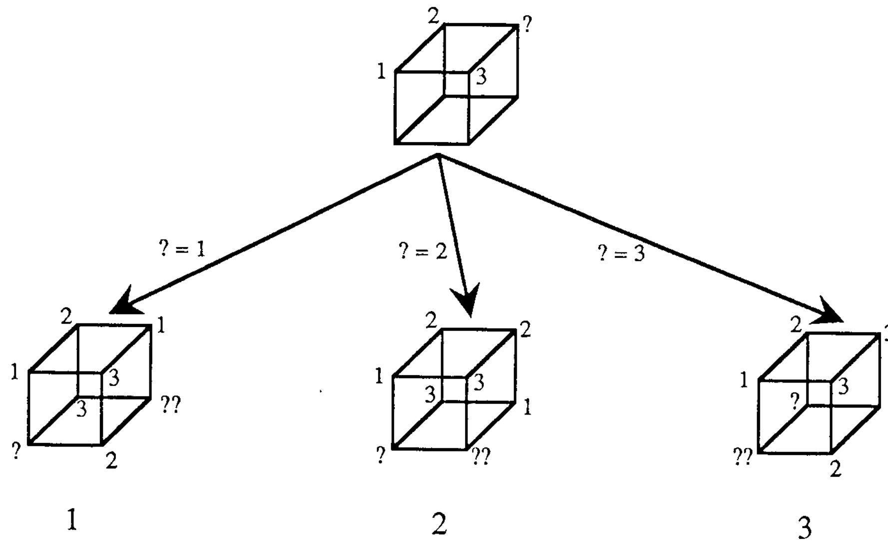


Figure 5.9. Premier niveau d'un arbre de répartition des trois sous-bases

De cet arbre découlent trois sous-arbres qui sont donnés ci-dessous :

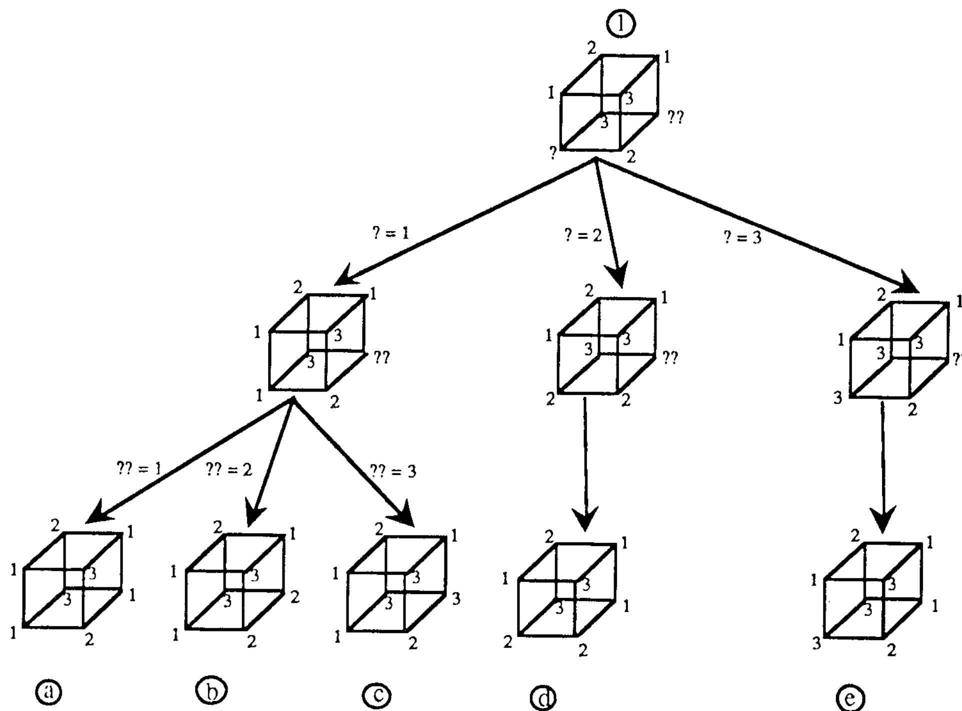


Figure 5.10. Sous-arbre gauche de l'arbre de la figure 5.9.

Chacun de ces cubes respecte la coloration demandée. Mais si on échange deux à deux la coloration des sommets, sans synchroniser les processeurs tous ensemble, on arrive fatalement à une coloration qui ne respecte pas la contrainte de voisinage.

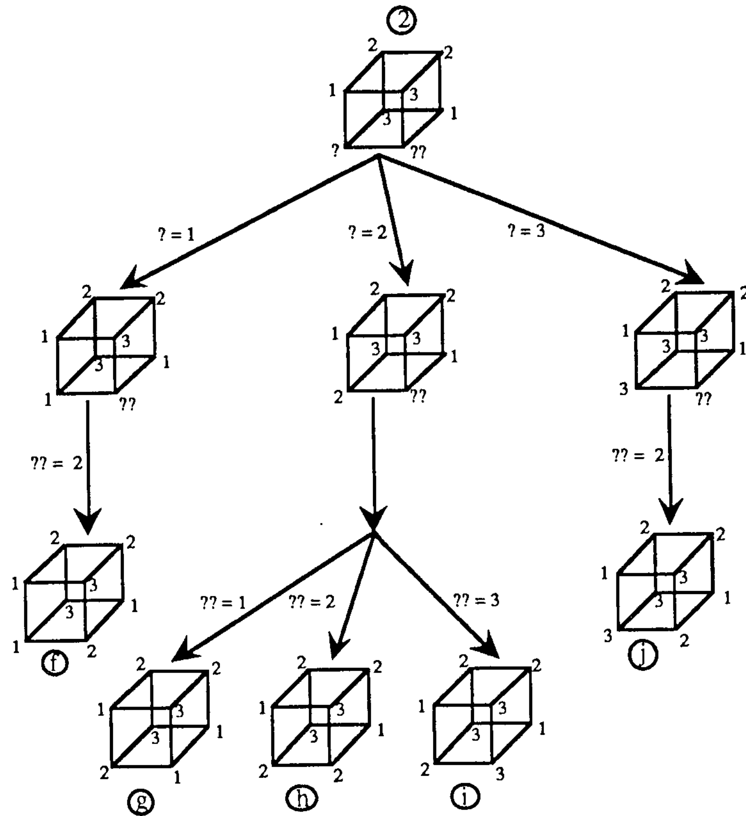


Figure 5.11. Sous-arbre central de l'arbre de la figure 5.9.

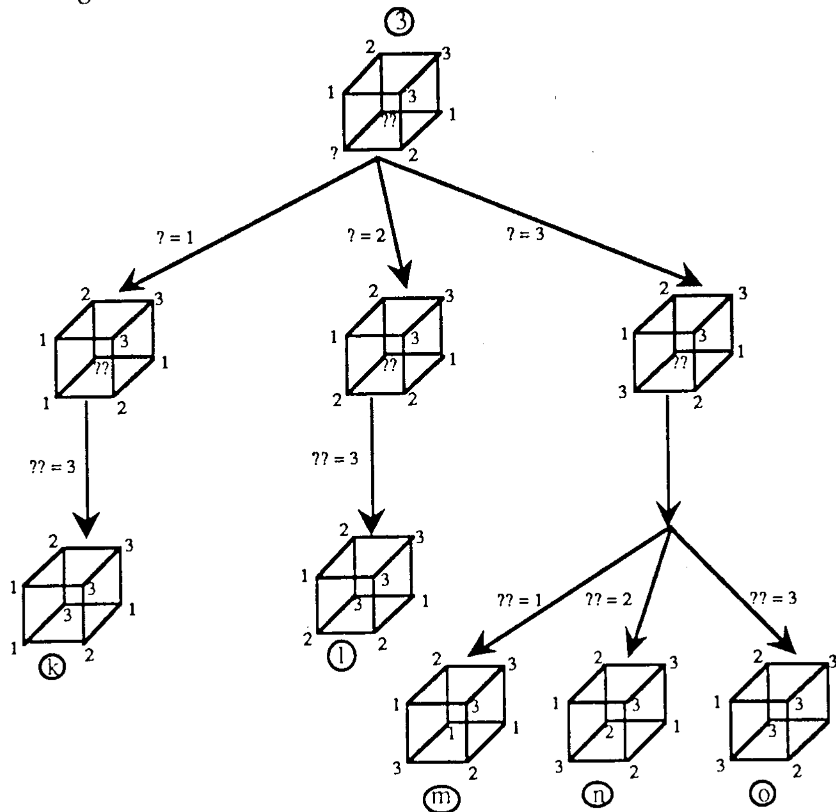


Figure 5.12. Sous-arbre droit de l'arbre de la figure 5.9.

On a exactement le même problème avec ces deux sous-arbres.

On envisage donc une synchronisation d'un plus grand nombre de processeurs, par exemple 4 processeurs. On synchronise donc 4 processeurs, et on effectue des échanges entre ces processeurs par rotation des informations.

Il s'agit d'échanges entre processeurs, c'est-à-dire qu'on remplace le contenu de la mémoire d'un processeur par celle de l'autre processeur et vice versa. Il ne faut pas perdre de données. Et c'est là que se pose un autre problème. Si on a quelques octets à échanger on n'a pas de souci à se faire pour la place mémoire. Alors que dans notre cas, on va échanger plusieurs centaines de Ko, 500 par exemple. Comment se passe cette communication ? Il faut savoir qu'un échange nécessite deux zones mémoires : une pour l'envoi et une pour la réception. Clairement, on n'a pas 500 Ko disponibles. Mais on peut y arriver, car sur tous les hypercubes ci-dessus, on remarque que le carré du haut et le carré du bas, quel que soit l'hypercube, possèdent toujours les 3 couleurs.

2.3.1.c. Echange des informations sur un 2-cube

On choisit donc de couper les cubes ou hypercubes en de tels carrés. Le principe de la communication sera le suivant (il est donné sur la figure 5.16) :

- Deux processeurs ont la même couleur sur ce carré.
- On décide d'établir une communication en réseau linéaire. On définit comme processeur de queue, l'un des deux processeurs ayant les deux mêmes couleurs. Ce processeur recevra la couleur du processeur précédent, et ceci est valable pour le troisième et le deuxième processeurs. Le premier et le deuxième processeurs auront donc la même couleur.
- On modifie le réseau, de manière que les deux processeurs qui ont la même coloration soient chacun à une extrémité du réseau.
- On recommence au plus trois fois pour que les processeurs aient été coloriés par les trois couleurs.

Il n'y a plus besoin de cette prérépartition qui colorie le graphe de manière précise et contraignante. Il faut maintenant initialiser tous les carrés de la même manière, par exemple en fonction du reste modulo 3 du numéro du processeur, ce reste attribuant automatiquement une couleur, et par conséquent un jeu de données. Ceci implique que le même jeu de données soit dupliqué sur chaque carré.

Etudions cet algorithme sur un exemple (d'ailleurs, il y a une symétrie entre les diverses possibilités de couleur, donc un seul exemple suffit).

Soit un carré dont les sommets ont les couleurs suivantes (la couleur 1 apparaît deux fois).

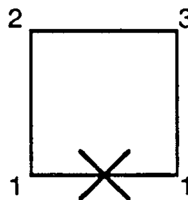


Figure 5.13. Répartition initiale des sous-bases sur un carré

On choisit de construire un réseau linéaire dont les deux extrémités ont la même couleur.

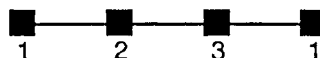


Figure 5.14. Ouverture de ce carré en réseau linéaire

Entre deux étapes de traitement, on effectue un décalage à droite avec recopie de l'information du processeur le plus à gauche.

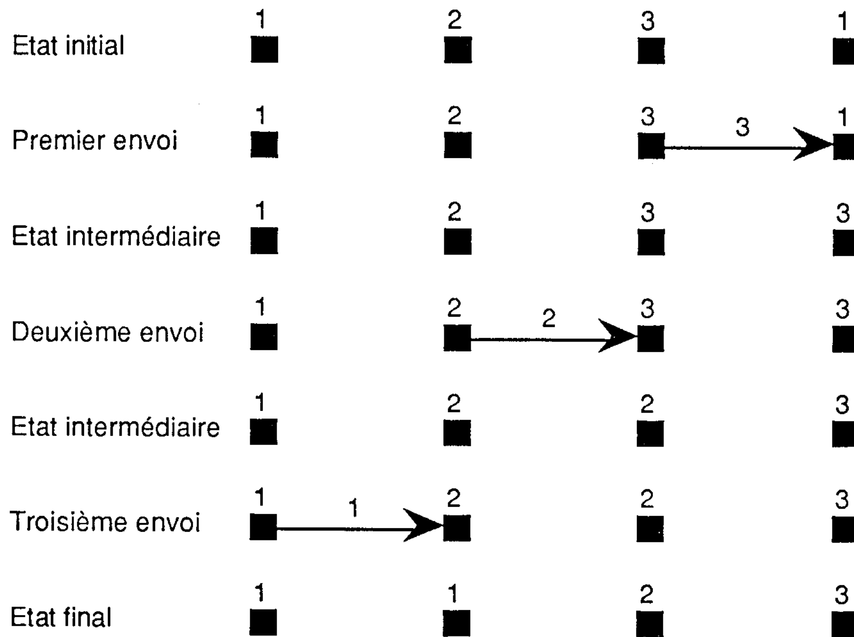


Figure 5.15. Une étape de décalage vers la droite

On utilise ce décalage de l'information dans l'algorithme suivant : après un décalage à droite, on effectue un traitement correspondant aux informations transmises. Puis on reconstruit le carré et on redécoupe entre les deux processeurs qui ont les mêmes informations, pour créer un nouveau réseau linéaire sur lequel on appliquera les mêmes opérations que précédemment. Il faut obligatoirement trois étapes de décalages (rotations) pour réaliser le all-to-all sur ce réseau.

Il n'est pas possible de le réaliser en deux étapes seulement à cause de la répartition des données.

Sur le schéma suivant, on donne en gras les couleurs actuelles de chaque processeur, et en clair les couleurs que ce processeur a déjà traitées.

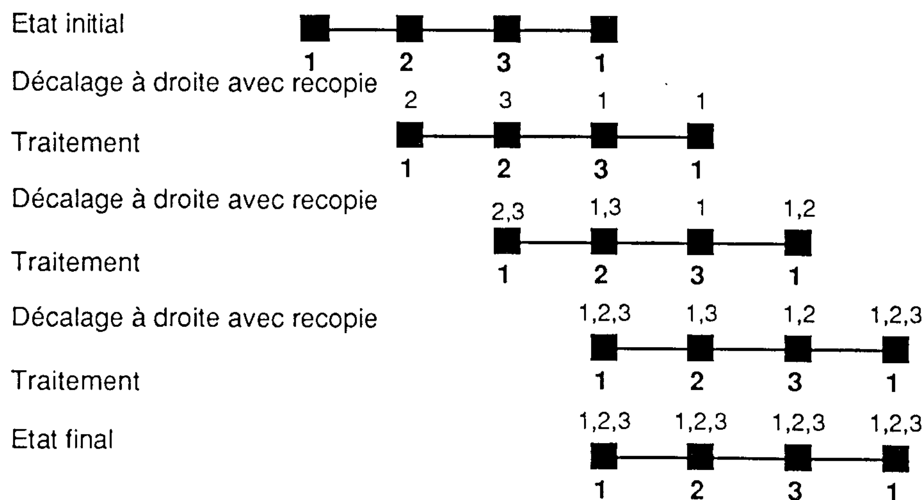


Figure 5.16. Trois étapes de décalage pour que les processeurs aient toutes les informations

Après trois décalages, tous les processeurs ont eu les données. Il est cependant dommage de constater que des processeurs qui reçoivent une donnée qu'ils ont déjà utilisée pour un traitement ne peuvent rien faire d'autre qu'attendre.

En fait, il y a une autre méthode qui permet de ne synchroniser que deux processeurs. Les deux processeurs ont des couleurs différentes. En supposant qu'ils puissent se les échanger facilement, on imagine une première étape de traitement avec leurs couleurs initiales, puis une phase d'échange, puis un nouveau traitement, puis une phase de lecture sur disque de la couleur qui leur manque, et un traitement de cette couleur.

Cependant, il est difficile d'échanger des gros paquets de données entre deux voisins. On pourrait alors utiliser le principe de l'algorithme précédent avec trois éléments de communication, le premier étant le disque et les deux autres étant deux processeurs. Il n'y a besoin de synchroniser que deux processeurs.

Le principe en est le suivant. Les deux processeurs ont initialement des couleurs différentes. Il travaillent avec ces couleurs. Ensuite, le processeur du milieu envoie au dernier processeur sa couleur. Le processeur du milieu reçoit du disque une autre couleur. Les deux processeurs traitent (voir figure 5.17). Il a fallu deux étapes seulement mais deux lectures sur disque, ce qui coûte cher.

Cependant, avec cette solution, il faut peu de place sur disque, uniquement 3 fichiers correspondant aux trois couleurs (attention il faut les mettre sur chaque disque). De plus, les processeurs ne restent plus inactifs à cause de la réception d'une information qu'ils ont déjà traitée, alors que d'autres processeurs travaillent.

Essayons d'évaluer le temps de chacune de ces deux solutions. Les processeurs se transmettent ou lisent sur disque des paquets de 300 Ko environ.

Dans la première solution, on peut étudier le temps d'un seul carré.
C'est le temps de 4 traitements t_{traite}, plus le temps de 9 (= 3*3) communications t_{comm} de 300 Ko entre deux processeurs voisins. Ceci vaut :

$$4\ t_{traite} + 9\ t_{comm} = 4\ t_{traite} + 9\ (\beta + 300\ 000\ \alpha)$$

avec $\beta = 860.10^{-6}$ s et $\alpha = 1{,}44.10^{-6}$ s.

Soit :

$$4\ t_{traite} + 9(860.10^{-6} + 300000*1{,}44.10^{-6}) = 4\ t_{traite} + 9*0{,}43286 = 4\ t_{traite} + 3{,}89574 \text{ s.}$$

Le temps total est donc : $4\ t_{traite} + 3{,}89574$ s.

2.3.1.d. Echange des informations entre deux processeurs et un disque

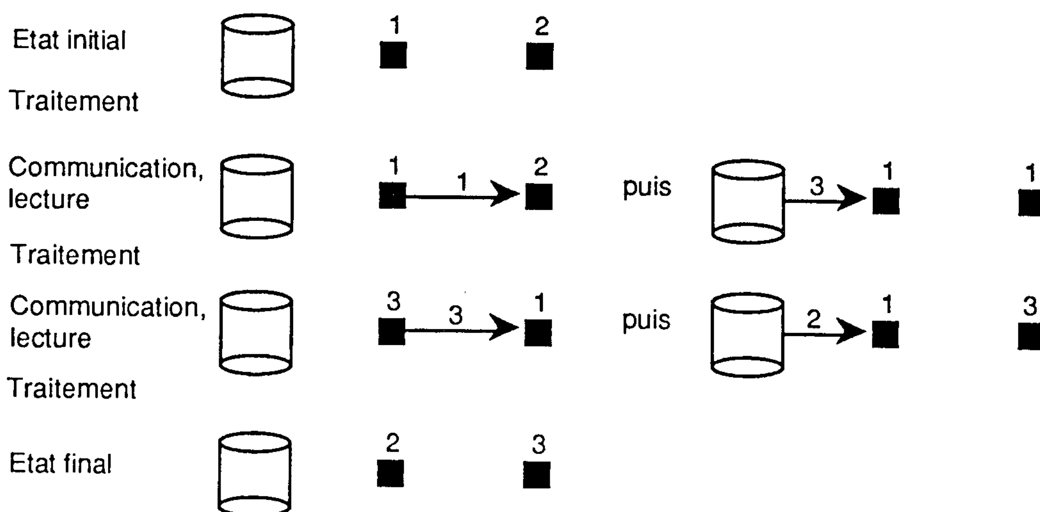


Figure 5.17. Echange des trois sous-bases entre deux processeurs et un disque

Les processeurs risquent d'effectuer des accès en lecture simultanés. On est obligé de prendre en compte la structure globale de la machine et plus particulièrement le nombre de processeurs. Sur un même disque, 4 processeurs au plus vont accéder en même temps à l'information. Le principe est le suivant pour deux processeurs :

Le temps total est :

$3\ t_{traite} + 2\ t_{comm} + 2\ t_{disque}$
$= 3\ t_{traite} + 2\ (\beta_{proc} + 300\,000\ \alpha_{proc}) + 2\ (\beta_{disque} + 300\,000\ \alpha_{disque})$
$= 3\ t_{traite} + 2\ (860.10^{-6} + 300\,000*1{,}44.10^{-6}) + 2\ (0{,}4172 + 300\,000*1{,}20.10^{-4})$
$= 3\ t_{traite} + 2*0{,}43286 + 2*36{,}4172$
$= 3\ t_{traite} + 0{,}96572 + 72{,}8344$
$= 3\ t_{traite} + 73{,}80012$ s.

Le temps total est donc : $3\ t_{traite} + 73{,}80012$ s.

2.3.1.e. Evaluation des solutions

La différence entre les deux temps (§c et §d ci-dessus) est $t_{traite} - 69{,}90438$ s.

Si $t_{traite} > 69{,}90438$ alors la deuxième solution est la meilleure. Sinon, la première solution est la meilleure.

Dans une méthode optimale, le minimum que l'on pourrait avoir est 3 traitements (1 par couleur) et 2 envois (1 par couleur non initialement présente sur un processeur).

Ceci coûte donc au minimum :

$3\ t_{traite} + 2\ t_{comm}$
$= 3\ t_{traite} + 2\ (\beta_{proc} + 300\,000\ \alpha_{proc})$
$= 3\ t_{traite} + 2\ (860.10^{-6} + 300\,000*1{,}44.10^{-6})$
$= 3\ t_{traite} + 2*0{,}43286$
$= 3\ t_{traite} + 0{,}86562$ s

Par rapport à la première solution ($4\ t_{traite} + 3{,}89574$ s) on perd ($1\ t_{traite} + 3$ s).

Par rapport à la deuxième solution ($3\ t_{traite} + 73{,}80012$ s) on perd 73 s.

Un traitement consiste à cribler le sous-intervalle que l'on a en mémoire avec le polynôme défini et la sous-base disponibles en mémoire. (Rappelons que c'est la sous-base qui changera entre deux traitements sur le même sous-intervalle avec le même polynôme).

Combien coûte un traitement ?
Pour un traitement, il faut définir les points de départ du crible à partir des racines carrées de N modulo les puissances des éléments de la base. Ceci se fait en deux opérations mod_Z de PaC [Roc 89] sur deux entiers en précision infinie, puis avec deux ou trois opérations sur des entiers 32 bits, qui sont donc négligeables.

Puis pour chaque racine carrée de N sur chaque puissance de chaque élément de la base (en fait on peut simplifier et omettre la boucle sur les puissances, puisque, très rapidement lorsque les éléments de la base croissent en valeur, la puissance maximale autorisée est 1), on crible le sous-intervalle.

Il y a deux racines carrées par élément de la base. Le sous-intervalle a une longueur d'environ 400 Ko. Le crible par une racine carrée consiste à additionner la partie entière du log de l'élément p_i de la base à la valeur correspondant à un indice et à incrémenter l'indice de parcours de l'intervalle de p_i^α. L'addition de p_i^α se fait $\frac{400\,000}{p_i}$ fois environ, puisqu'on prend $\alpha = 1$.

Donc, pour chacun des trois sous-ensembles E_1, E_2 et E_3 de la base où les éléments vont respectivement

de 2 à $\frac{2k \log(2k)}{3}$, de $\frac{2k \log(2k)}{3} + 1$ à $2.\frac{2k \log(2k)}{3}$ et de $2.\frac{2k \log(2k)}{3} + 1$ à $2k \log(2k)$,

on fait k/3 fois ces deux additions, une fois pour chacun des éléments des sous-ensembles.

Combien d'additions effectue-t-on sur le total des trois intervalles ? Pour un intervalle, on supposera que chaque intervalle fournit le même nombre d'éléments à la base

$$2 \sum_{i=1}^{k} \frac{400\,000}{2i \log(2i)}$$

environ. Dans notre cas, si on pose k = 30 000, le nombre d'additions est

$$2 \sum_{i=1}^{30\,000} \frac{400\,000}{2i \log(2i)} = 800\,000.1{,}8595905 \approx 1\,500\,000.$$

On effectue 1 500 000 additions 32 bits et 30 000 divisions modulaires en précision illimitée. Ce qui nous donne 50 000 additions 32 bits et 10 000 divisions modulaires avec un entier sur 21 décimales et le même nombre avec un entier sur 42 décimales en précision infinie pour chacun des trois sous-ensembles E_1, E_2 et E_3.

Calculons le temps de la division modulaire du système de calcul en précision infinie de PaC [Roc 89] et le temps des opérations 32 bits sur l'hypercube (sans utiliser le VPU). Rappelons que les fonctions de PaC dépendent de la taille des opérandes. Les divisions modulaires que j'effectue se font sur des opérandes de 21 décimales environ par un nombre écrit sur 32 bits et de 42 décimales par un nombre écrit sur 32 bits.

Opération	Temps
addition entre 2 entiers 32 bits	$1{,}95.10^{-6}$ s
soustraction entre 2 entiers 32 bits	$1{,}95.10^{-6}$ s
multiplication entre 2 entiers 32 bits	$3{,}68.10^{-6}$ s
division entière entre 2 entiers 32 bits	$4{,}97.10^{-6}$ s
division modulaire entre un entier de 21 décimales écrit en précision infinie et un entier 32 bits	$8{,}64.10^{-4}$ s
division modulaire entre un entier de 42 décimales écrit en précision infinie et un entier 32 bits	$8{,}85.10^{-4}$ s

Figure 5.18. Temps des opérations de base

500 000 additions 32 bits coûtent $500\,000.1{,}95.10^{-6} = 0{,}975$ s.
10 000 divisions modulaires (21 décimales) coûtent $10\,000.8{,}64.10^{-4} = 8{,}64$ s.
10 000 divisions modulaires (42 décimales) coûtent $10\,000.8{,}85.10^{-4} = 8{,}85$ s.

Ceci donne un total de 18,5 s pour un sous-ensemble de la base, et près de 55 secondes pour toute la base, sur un sous-intervalle de 400 Ko. Rappelons que dans ce cas, chaque processeur s'occupe de la 32ème partie de l'intervalle de crible, et qu'il n'y a pas de redécoupage de cet intervalle sur les processeurs.

2.3.1.f. Choix et conclusion

Or nous avons vu que nous perdions 73 s ou t_{traite} + 3 s = 22 s. La solution la moins coûteuse est, d'après les chiffres, celle qui utilise des communications entre deux processeurs et un disque. Ceci contredit ce que nous avions pensé auparavant. Mais c'était sans compter avec les synchronisations. Pour deux processeurs et un disque, il y a moins de temps de synchronisation que pour quatre processeurs.

En fait, chaque processeur peut travailler avec une base de 300 à 330 Ko et sur un sous-intervalle de 400 Ko. Et chaque processeur a un polynôme propre. Donc pour cribler l'intervalle de 8 Mo il faut à chaque processeur 20 cribles de sous-intervalles de 400 Ko chacun. Mais rappelons qu'il n'y a pas besoin de synchroniser les processeurs pour cette méthode car chaque processeur génère des polynômes dans une famille qui lui est propre. Le coût du crible de l'intervalle complet avec un polynôme (réalisé sur un seul processeur) coûte donc environ 20 minutes.

Mais en 20 minutes, on crible avec autant de polynômes que de processeurs. Donc en fait en 20 minutes, on crible l'intervalle complet avec 32 polynômes, ce qui ramène le temps de crible à 40 s par polynôme.

2.3.2. Le processeur s'occupe d'un sous-intervalle de l'intervalle de crible complet

Dans ce cas, les processeurs ont tous le même polynôme qu'ils calculent de manière redondante. Rappelons que cette technique peut permettre, sur un grand nombre de calculs de polynômes, une compensation et un équilibrage des temps de calcul.

Il faut calculer le temps de crible d'un sous-intervalle en oubliant le temps de calcul du polynôme, pour l'instant.

Comment les processeurs vont-ils échanger les informations dont ils ont besoin, à savoir les autres parties de la base qu'ils ne peuvent pas avoir en mémoire ?

Ils peuvent les lire sur disque ou les chercher sur les processeurs voisins. On en revient à la technique du cas 1. Mais il y a une grosse différence entre la théorie et la réalisation. En effet, les processeurs ne peuvent pas conserver en mémoire l'ensemble de la base de facteurs. Et ceci va entraîner des communications ou des lectures sur disque, et donc des synchronisations. Bien évidemment, on va perdre l'acquis théorique du non-besoin de synchronisations et de communications, ce qui permettait une compensation des différences de temps de crible entre les processeurs, et, à terme, une bonne efficacité, malgré la redondance due au calcul des polynômes.

On retombe dans le cas précédent : un processeur, pour cribler un sous-intervalle, est obligé de découper en trois parties la base des éléments. Et il ne peut en conserver qu'une partie en mémoire. Les deux autres parties sont soit sur disque, soit sur des processeurs voisins directement accessibles.

Pour cribler un intervalle complet, il faut que les P sous-intervalles, un par processeur, aient été criblés avec la base complète. On peut reprendre la technique du cas 1 : soit on groupe les processeurs par 2, soit on les groupe par 4. Si on les groupe par 2, des lectures sur disque sont nécessaires. Si on les groupe par 4, il y a des temps d'inactivité sur certains processeurs. Les temps calculés pour cribler des intervalles de 400 Ko avec des sous-ensembles de la base de 300 à 330 Ko sont les mêmes que dans le cas 1.

Or nous espérions un algorithme sans communication, ce qui n'est pas le cas. En effet, les processeurs échangent des informations. Sur 4 processeurs comment se déroule cet algorithme ? Nous présentons son chronogramme sur la figure 5.19.

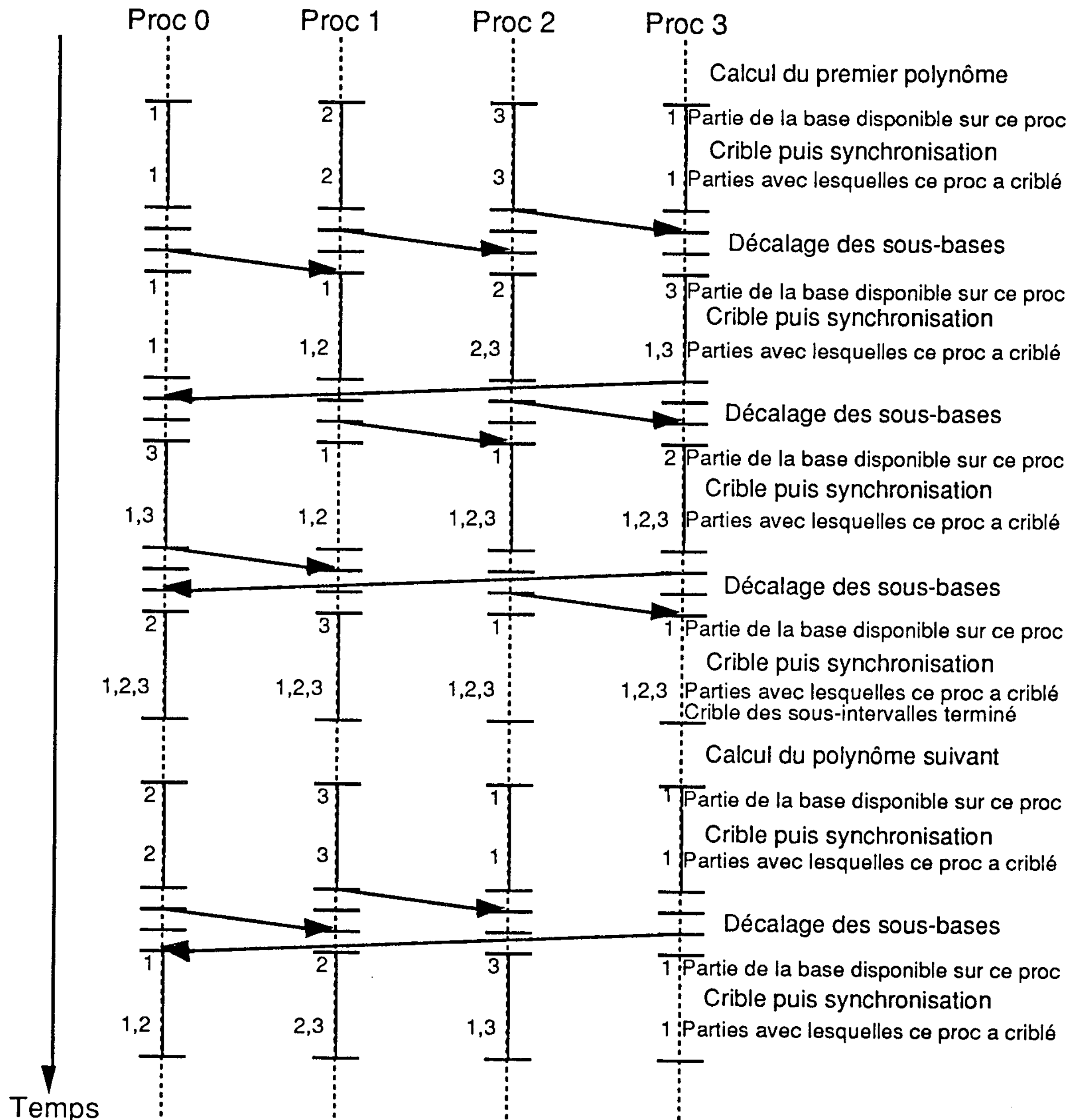


Figure 5.19. Chronogramme des échanges de sous-bases pour le crible

On voit que beaucoup de temps est perdu par les processeurs qui ne criblent pas, parce qu'ils ont déjà criblé avec le sous-ensemble de la base qu'ils reçoivent actuellement.

Remarque : chaque processeur est inactif exactement une fois dans cet algorithme. On peut donc mettre à profit ce temps d'inactivité pour calculer le prochain polynôme dont aura immédiatement besoin chaque processeur. Suivant le temps nécessaire au calcul des coefficients du nouveau polynôme par rapport au temps de crible d'un processeur sur un sous-intervalle de l'intervalle de crible avec un sous-ensemble des éléments de la base, on saura si on peut garder cette technique de calcul redondant des polynômes.

Sinon, on peut envisager de répartir le calcul du nouveau polynôme sur les processeurs dans l'ordre dans lequel ils deviennent inactifs. Et ils se passent les informations pour ne pas rechercher deux fois ou plus si un coefficient a convient ou non.

2.3.3. Evaluation

Il s'agit de comparer maintenant les deux solutions : découpage ou non de l'intervalle de crible. Dans ces deux cas, nous coupons la base en trois parties. De même, nous coupons l'intervalle de crible en sous-intervalles de 400 Ko environ. Mais c'est la gestion des polynômes qui n'est pas la même, et le travail à moyen terme d'un processeur qui diffère.

Dans le premier cas, chaque processeur calcule ses polynômes dans une famille qui lui est propre. Son but est de cribler tout l'intervalle de crible en faisant des cribles complets de sous-intervalles. Ces parties font 400 Ko. Il faut donc environ 32 sous-cribles pour réaliser le crible sur un intervalle complet.

Dans le deuxième cas, tous les processeurs ont le même polynôme (on ne sait pas encore si un seul processeur le calcule pour tout le monde et l'envoie ensuite à tous les processeurs, ou si chaque processeur calcule ce polynôme pour lui seul). Le but de chaque processeur est de cribler un sous-intervalle de 400 Ko complètement, car l'intervalle complet est réparti entre tous les processeurs. Pendant qu'un processeur crible sur 400 Ko (en trois fois, puisque la base est découpée en trois) les autres processeurs criblent sur les autres sous-intervalles de crible. Quand tous les processeurs ont terminé cette étape, le crible est réalisé sur l'intervalle complet.

Dans les deux cas, de toute manière, il faut chercher les w(x) factorisables et les factoriser immédiatement.

En utilisant les temps morts des processeurs qui ne travaillent pas parce qu'ils ont en mémoire un sous-ensemble de la base avec lequel ils ont déjà criblé, on peut faire en sorte que chaque processeur calcule le polynôme suivant sans perdre de temps. Il faudrait que le temps de calcul d'un polynôme soit inférieur au temps du crible d'un sous-intervalle par un sous-ensemble de la base.

2.3.3.a. Temps de calcul pour la génération d'un polynôme

Calculons le temps de recherche d'un polynôme. En fait, le temps nécessaire est surtout celui de recherche du coefficient a :

- a voisin de $Q = \sqrt{\sqrt{2N} / M}$ n'est pas une condition coûteuse. On calcule une fois cette valeur. On initialise a à une valeur proche de $Q - \varepsilon$, où ε est une valeur dépendant de N, et dépendant du nombre de a à essayer pour obtenir tous les coefficients a utilisables. Par exemple, pour N de l'ordre de 10^{100}, ε est de l'ordre de 10^9.

- a de la forme 4j+3. Au début, on cherche un a de cette forme. Et par la suite, on ajoute 4 ou un multiple de 4 à la valeur précédente de a pour avoir le nouveau a candidat.

- a premier n'est pas nécessaire. Donc on peut effectuer un test de Miller par rapport à la base 2, et vérifier que le coefficient c est bien entier. Ce test est logarithmique.

- N résidu quadratique de a. Ce test est aussi logarithmique.

En moyenne, pour trouver un coefficient a qui vérifie les conditions de primalité et de résidu quadratique, il faut essayer (pour un entier de 100 chiffres à factoriser) 34 coefficients a. Pour chaque coefficient, il faut calculer tout d'abord (N/a) puis la valeur du test de Miller sur la base 2, mais seulement dans le cas où (N/a) = 1. Le calcul d'un nouveau a est le suivant. Tout d'abord rappelons que a est codé en précision infinie puisqu'il vaut environ 10^{21} si N est de l'ordre de 10^{100}.

- On commence par ajouter 4 (ou un multiple de 4) à la précédente valeur de a.

- On calcule ensuite (N/a). N et a sont tous les deux codés en précision illimitée. La complexité de ce calcul correspond à celle de l'algorithme d'Euclide, c'est-à-dire en au plus le nombre de bits de l'entier le plus petit. Ce pire cas est obtenu pour deux termes consécutifs de la suite de Fibonacci (montré par Lamé en 1844). Il y aura donc au plus 70 (bits) fois le même groupe d'opérations qui contient une division par 2, une multiplication (ou deux). Ceci coûte donc environ $70.(7 + 6{,}19).10^{-4}$ soit moins de 0,1 s.

- Puis, dans le cas où (N/a) = 1, on teste la pseudo-primalité de a, par la méthode du test de Miller (test probabiliste).

- On cherche la valeur s de l'exposant de 2 dans la factorisation de a-1.

- On pose $a-1 = 2^s t$. Soit 2 la base avec laquelle on va effectuer un test de Miller. Ce test comporte s+1 élévations à la puissance d'un entier en précision 32 bits (la base 2) par un entier en précision infinie. Dans la première élévation on calcule 2^t mod a et on regarde si cette valeur est 1. Dans la deuxième, on calcule une par une les valeurs $2^{2^j t}$ mod a pour tout j tel que $0 \leq j \leq s-1$ et on regarde si elles sont égales à -1 (ou a-1).

Opération	Temps
Modulo d'un entier 10^{21} par un entier 32 bits	$8{,}68.10^{-4}$ s
Modulo d'un entier 10^{42} par un entier 32 bits	$8{,}90.10^{-4}$ s
Modulo entre deux entiers 10^{21}	$13{,}38.10^{-4}$ s
Division entière d'un entier 10^{21} par un entier 32 bits	$7{,}00.10^{-4}$ s
Multiplication de deux entiers 10^{21}	$6{,}19.10^{-4}$ s

Figure 5.20. Temps d'opérations modulaires sur des entiers en précision infinie

Pour une élévation à la puissance (+ modulo) entre trois opérandes en précision infinie, il y a autant de groupes d'opérations à faire que de bits de l'exposant. Chaque groupe d'opérations contient au plus 1 division entière entre un entier infini et un entier 32 bits, 2 multiplications infinies entre des entiers de l'ordre de grandeur de a et 2 modulos infinis entre des entiers de l'ordre de grandeur de a. Or a est un entier de 70 bits environ. Donc, chaque exponentiation demande 70 divisions entières infinies, 140 multiplications infinies et 140 modulos infinis.

D'après ce tableau, le temps d'une exponentiation modulaire est donc d'environ 0,323 s : $70*(7.10^{-4} + 2*6{,}19.10^{-4} + 2*13{,}38.10^{-4}) = 0{,}323$ s. Or dans le test de Miller, on fait au plus s+1 exponentiations avec $s \leq 70$ (l'égalité est atteinte pour $a = 2^{70} + 1 = 1{,}18.10^{21}$). Mais en moyenne on effectuera moins de 3 exponentiations, ce qui coûte de l'ordre de 1 s.

Il est donc clair que pour tester si un coefficient a convient, il faut environ 1 s dans les cas moyens (et 20 s dans les cas extrêmes), dans l'hypothèse où a convient. Si on doit éliminer 30 coefficients pour en trouver un qui convienne, il faut environ 30 s pour obtenir un polynôme. Ce temps est compatible avec les temps d'inactivité des processeurs : on peut donc calculer (de façon redondante) les polynômes sur les processeurs, et éviter des communications.

2.3.3.b. *Modification du travail des processeurs lors du crible par sous-intervalle*

Les processeurs vont donc pouvoir travailler pendant la période où ils restaient inactifs précédemment. En effet, lorsqu'ils recevaient une partie de la base avec laquelle ils avaient déjà travaillé, ils restaient inactifs. Dorénavant, ils cherchent le prochain polynôme. Ils sont mieux utilisés. Le schéma devient alors :

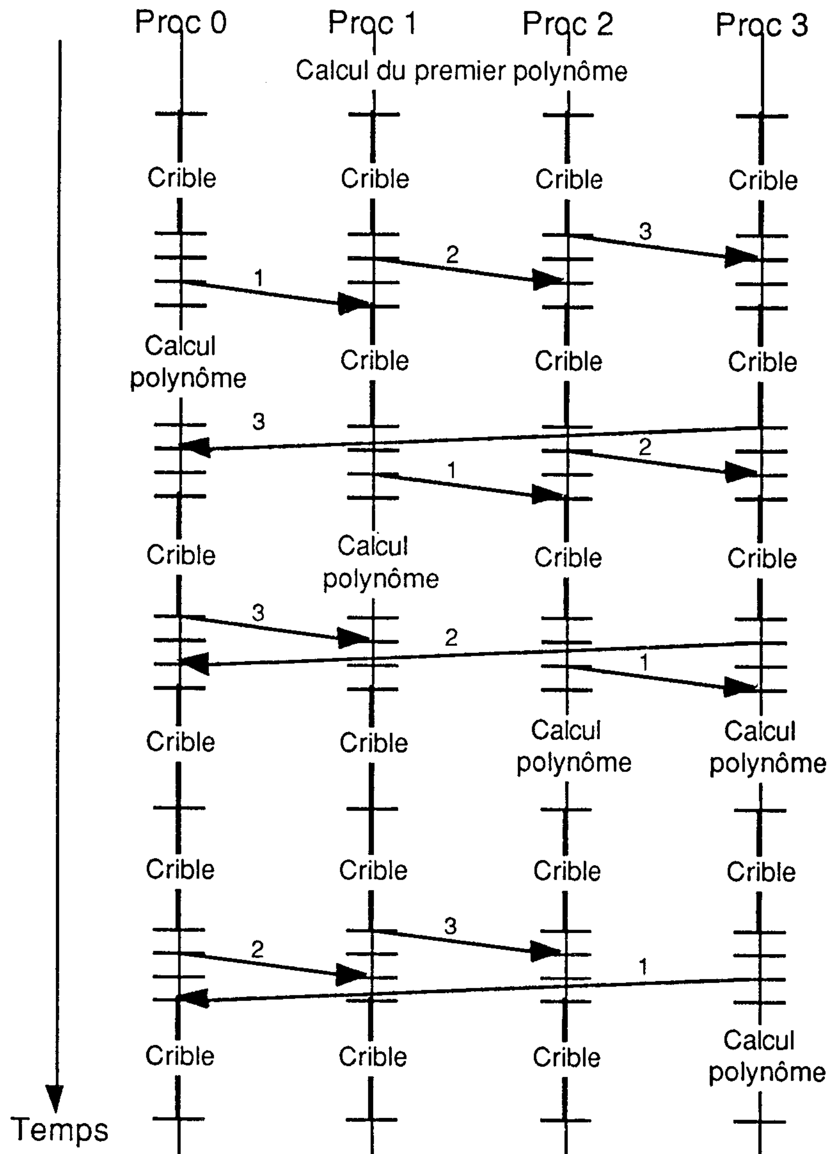


Figure 5.21. Chronogramme des échanges des sous-bases pour le crible, avec calcul des polynômes pendant les temps d'inactivité des processeurs

On gagne le temps de calcul d'un nouveau polynôme.

3. CONCLUSION DE LA PRÉ-ÉTUDE D'IMPLANTATION SUR L'HYPERCUBE FPS T40

En résumé, on peut dire que l'algorithme du crible quadratique peut être parallélisé assez efficacement, mais qu'il est possible d'envisager de nombreuses variantes selon la distribution choisie et les caractéristiques de la machine-cible.
On peut distribuer une des trois variables de boucle : les polynômes, la base de facteurs, ou l'intervalle de crible.

3.1. DISTRIBUTION DES POLYNÔMES

Chaque processeur génère ses propres polynômes, mais possède en mémoire toute la base et tout l'intervalle. Il n'y a pas besoin de communications, puisque les processeurs peuvent cribler

l'intervalle complet avec toute la base de facteurs, puis chercher les w(x) candidats à une factorisation complète et enfin les factoriser sans avoir recours à des informations qu'ils ne possèdent pas. Lorsqu'un w(x) se factorise complètement, les paramètres (a, b, c, x et la factorisation de w(x)) sont stockés.
Cependant, avec une mémoire de taille limitée, il n'est pas toujours possible de garder à la fois la base entière et l'intervalle complet en mémoire, surtout lorsque N devient grand. Mais, si l'espace mémoire est suffisant pour contenir la base de facteurs et un sous-intervalle de crible, il est possible de cribler complètement sans aucune communication.

L'exécution n'est pas aussi rapide qu'avec une mémoire illimitée, parce que pour chaque sous-intervalle, il est nécessaire de calculer un nouvel indice de départ pour chaque racine carrée de chaque puissance de tous les éléments de la base. Malgré cela, les processeurs n'ont pas à se synchroniser, sauf au moment où ils terminent la construction de la matrice, pour l'élimination de Gauss.

Si la base ne tient pas en mémoire, d'importantes communications sont à redouter.

3.2. DISTRIBUTION DE LA BASE DE FACTEURS

Tous les processeurs ont les mêmes polynômes et l'intervalle complet. Ils criblent avec un sous-ensemble de la base de facteurs. Lorsque le crible avec un polynôme est achevé, certains w(x) peuvent se factoriser sur la base, mais leur factorisation est distribuée entre les processeurs, à l'instar de la base. Pour obtenir le résultat final du crible, des étapes de communication sont nécessaires pour sommer indice à indice les résultats de chaque processeur.

Ainsi, après chaque phase de crible, il faut synchroniser les processeurs, communiquer et réduire les résultats. Ceci induit une perte de temps, surtout si les processeurs ne travaillent pas à la même vitesse.

Les autres stratégies de rotation des sous-bases par exemple que nous avons présentées au paragraphe 2.3. nécessitent les mêmes synchronisations, même si les tailles des messages transmis, et, par suite, les temps de communication, sont différents.

3.3. DISTRIBUTION DE L'INTERVALLE À CRIBLER

Tous les processeurs ont les mêmes polynômes, ainsi que la base de facteurs complète. Cette stratégie revient à réaliser un crible complet, mais sur un intervalle plus petit. Comme les processeurs vont tous avoir besoin des mêmes polynômes à un certain instant, ce serait une perte de temps importante si chaque processeur génèrait ses polynômes pour son propre compte. Ainsi, dans l'initialisation, tous les processeurs pourraient générer et stocker tous les polynômes nécessaires, et ceci d'une manière parallèle efficace. Ceci est possible avec une mémoire illimitée. Mais avec une mémoire limitée, il est impossible de stocker, a priori, tous les polynômes qui seront utilisés pendant la phase de crible.

C'est pourquoi ces polynômes doivent être générés en cours de boucle, soit par chaque processeur (redondance de calculs, mais compensation possible) soit par un processeur central dédié (perte de temps lors des synchronisations des processeurs, tous ensemble pour une diffusion ou par paire avec le processeur central).

3.4. ÉVALUATION

Sur un multiprocesseur à mémoire limitée, il est difficile de départager a priori les deux stratégies de distribution des polynômes ou de l'intervalle. Il est certain que la distribution de l'intervalle implique une redondance de calculs afin d'éviter des communications, ce que la distribution des polynômes ne nécessite pas. Il est donc plus facile d'implanter efficacement la première stratégie (distribution des polynômes) que la troisième (distribution de l'intervalle). Le problème crucial provient de la taille de la mémoire : tant qu'elle est suffisante pour contenir la base entière, même s'il faut couper l'intervalle en sous-intervalles et cribler chaque sous-

intervalle séquentiellement, la perte de temps est minime. Mais lorsque la base doit être découpée en sous-bases, des accès disque ou des communications supplémentaires (et des synchronisations) viennent augmenter très fortement le temps d'exécution.

4. LES AMÉLIORATIONS POSSIBLES POUR L'ALGORITHME DU CRIBLE QUADRATIQUE MULTIPOLYNOMIAL

Le crible quadratique est puissant. Mais on peut lui ajouter des raffinements algorithmiques qui l'accélèrent énormément.

Pomerance et al., dans [PST 88], proposent plusieurs variations :

4.1. LE MULTIPLICATEUR

L'utilisation d'un multiplicateur h. L'idée consiste à remplacer N par hN où h est un petit entier, non carré fixé, afin de biaiser la base vers les petits nombres premiers. Cette variation permet parfois d'accélérer l'exécution par un facteur 2 ou 3. Mais ceci pose un problème : les valeurs de $|w(x)|$ sont plus grandes d'un facteur $\sqrt{h}$.

Il est possible de trouver un compromis en définissant la fonction

$$f(h, N) = \frac{1}{2} \log h + \sum_{p \leq B} E_p^{(h)}$$

où la somme est faite sur les nombres premiers inférieurs à B.

$$E_p^{(h)} = 0 \qquad \text{si } t^2 \equiv h\,N \bmod p \text{ n'a pas de solution,}$$

$$E_p^{(h)} = \begin{cases} \dfrac{2 \log p}{p-1}, \text{ si p impair et p ne divise pas h,} \\ \dfrac{\log p}{p}, \text{ si p impair et p divise h,} \\ \dfrac{\log 2}{2}, \text{ si p = 2 et hN} \equiv 2 \text{ ou } 3 \bmod 4, \\ \log 2, \text{ si p = 2 et hN} \equiv 5 \bmod 8, \\ 2 \log 2, \text{ si p = 2 et hN} \equiv 1 \bmod 8, \end{cases}$$

$$\text{si } t^2 \equiv h\,N \bmod p \text{ a des solutions [Pom 85].}$$

Avant de construire la base des k éléments, il faut chercher la valeur de h qui maximise f(h,N). En pratique, h < 100 et on remplace B par une plus petite valeur : 1000 par exemple.

4.2. LA VARIATION DES PETITS NOMBRES PREMIERS

Si q est une puissance d'un nombre premier de la base de facteurs, alors le temps pour cribler avec une racine correspondant à q est proportionnel à M/q. C'est ainsi que nous passons beaucoup plus de temps à cribler avec des petits q, qu'avec des grands q. De plus la contribution de ces q à la somme des log est faible.
En effet

$$\sum_{p \leq 30} \log p \leq 23 \text{ et } \sum_{p^\alpha \leq 30,\ \alpha \in \mathbb{N}} \log p \leq 38.$$

Cette valeur est faible devant celle de log $|w(x)|$. On peut donc "oublier" de cribler avec ces petites valeurs. Avantage : gain possible de 20 % sur le temps total. Inconvénient : on risque d'essayer des valeurs de w(x) qui ne se factoriseront pas, mais on ne perd aucun w(x) complètement factorisé.

4.3. LA VARIATION DU GRAND NOMBRE PREMIER

Si on trouve des indices x tels que

$$\log |w(x)| - \sum_{\substack{m^{\alpha} \mid w(x) \\ m \in \text{base}}} \log m < 2 \log B,$$

alors soit w(x) se factorise complètement sur la base, soit w(x) se factorise sur la base à un facteur p près tels que $B < p < B^2$. Ainsi, en gardant en mémoire les w(x) de ce type, triés selon les valeurs de p, on peut, par le paradoxe des anniversaires, trouver deux valeurs de w(x), w_i et w_j, qui aient le même p. Dans ce cas w_i/w_j se factorise complètement sur la base. Il s'agit de trouver des collisions.

Une première possibilité est de trier les p au fur et à mesure de leur arrivée, en temps logarithmique pour chaque p.

Une deuxième possibilité est de stocker les éléments et de ne les trier qu'en fin de boucle. Le tri sur hypercube peut être très performant [FeG 89].

Une autre possibilité consiste à utiliser la technique de hachage. Et dans ce cas, on est satisfait lorsqu'on trouve une collision. De cette manière, il faut un nombre constant d'opérations proche de 1 en moyenne pour insérer un p dans la liste.

Avantage : gain d'un facteur 2 ou 3 sur le temps d'exécution.
Inconvénient : cette variation nécessite un grand espace mémoire.

Dans une implantation, on restreint la condition $p < B^2$ à $p < D$ où D tient sur 32 bits.

4.4. LA VARIATION DES DEUX GRANDS NOMBRES PREMIERS

C'est le même principe théorique que pour la variation avec un seul grand nombre premier, mais en posant : $B^2 < q < B^3$

Dans ce cas q est soit un nombre premier, soit le produit de deux nombres premiers p et p' tels que $B < p < B^2$ et $B < p' < B^2$. Donc il faut savoir factoriser ces nombres q=pp', ou tester leur primalité (si q est premier), et avoir la place de les stocker afin de trouver des doubles collisions ou des collisions à 3.

Avantage : gain d'un facteur 3 sur le temps d'exécution.
Inconvénient : grande perte de place et nécessité de factoriser des nombres intermédiaires.

4.5. CONCLUSION

Toutes ces améliorations conjuguées peuvent permettre un gain de temps allant d'un facteur 20 à un facteur 30. Cependant, elles nécessitent une grande place mémoire, du moins pour la variation des grands nombres premiers. Nous ne les implantons pas, car elles ne modifient pas l'étude algorithmique parallèle. En revanche, pour une implantation efficace, il serait nécessaire de les inclure. C'est ce que nous allons faire pour l'implantation sur de nouvelles machines.

5. IMPLANTATION DU CRIBLE QUADRATIQUE MULTIPOLYNOMINAL SUR L'HYPERCUBE FPS T40

Notre implantation ne contient aucune des variations présentées ci-dessus.
L'élimination de Gauss a été implantée sur un nœud afin de tester la validité de l'ensemble de l'algorithme, mais n'a pas été implantée en version distribuée, lorsque la matrice occupe plus d'1 Mo. Nous nous contentons ici de construire cette matrice.

Notre but est de valider l'étude théorique faite pour le FPS T40. Nous vérifions tout d'abord que l'algorithme construit une matrice qui permet d'extraire, d'une manière probabiliste, une factorisation de N. Nous mesurons les temps d'exécution des différentes phases. L'accélération est superlinéaire. Ceci provient de l'augmentation de la taille mémoire lorsqu'on augmente le nombre de processeurs. Nous notons des différences de temps d'exécution importantes entre les processeurs, pour la factorisation d'entiers ayant entre 35 et 41 chiffres. Nous analysons les résultats pour découvrir que ces différences sont dues à une mauvaise répartition des coefficients des polynômes. Nous modifions l'affectation des polynômes sur les processeurs. L'algorithme s'exécute alors à la même vitesse sur tous les processeurs. Nous le vérifions sur des entiers de 35 à 60 chiffres.

Notre but est atteint. Nous savons factoriser des entiers de 60 chiffres ayant deux grands facteurs premiers, en utilisant la puissance d'un multiprocesseur distribué. Tous les processeurs travaillent le même temps, il n'y a pas de période d'inactivité. De plus, les étapes de l'algorithme, prises séparément, sont elles aussi bien réparties sur les processeurs : elles mettent toutes le même temps sur chaque processeur. Enfin, les communications sont réduites, puisque la phase la plus coûteuse en temps, la boucle de génération de la matrice, se déroule sur une ferme de processeurs.

5.1. FACTORISATION DE NOMBRES AYANT ENTRE 35 ET 41 CHIFFRES

Le schéma suivant donne les temps d'exécution pour factoriser sur 32 processeurs des nombres ayant entre 35 et 41 chiffres.

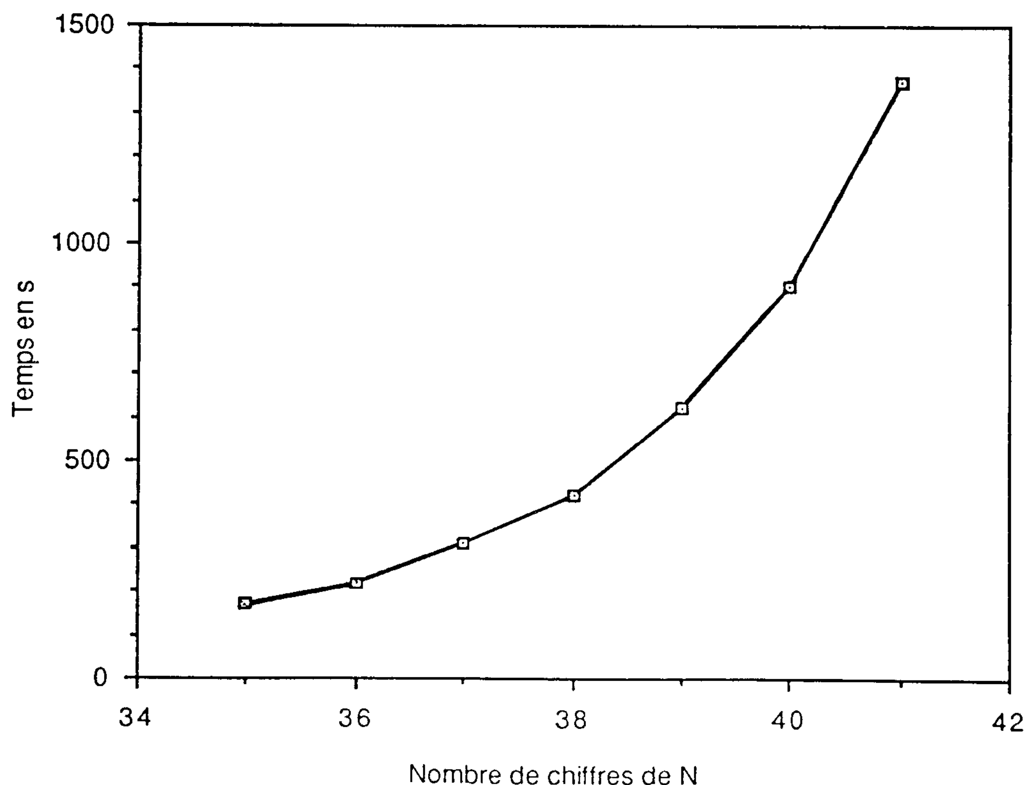


Figure 5.22. Temps de crible pour des entiers ayant entre 35 et 41 chiffres

Le temps de factorisation croît rapidement en fonction du nombre N à factoriser.

Notre but est de minimiser le temps total, de répartir la charge de travail de manière équilibrée sur les processeurs, et d'utiliser au maximum ces processeurs. La stratégie qui consiste à distribuer les polynômes par famille sur les processeurs évite toute communication, tant que l'espace mémoire local à chaque processeur peut contenir la base complète des facteurs.

Pour $N \in \{10^{34}, ..., 10^{40}\}$, la base occupe de 10 à 20 Ko et l'intervalle de 15 à 25 Ko (le programme occupe moins de 250 Ko et chaque nœud a une mémoire de 1 Mo).

N est le produit de deux nombres premiers ayant à peu près la même taille.

Pour N de l'ordre de 10^{40}, étudions l'accélération en fonction du nombre de processeurs. Nous exécutons le même programme avec les mêmes données sur des nombres différents de processeurs. L'accélération que nous calculons est la suivante pour P processeurs :
(temps sur un processeur) / (temps sur p processeurs).

Nombre de processeurs	Temps en s	Accélération
1	166 000	1
2	47 800	3,4
4	14 820	11,2
8	5 790	28,6
16	2 750	60,4
32	1 260	131,8

Figure 5.23. Accélérations pour le crible d'un entier de 41 chiffres

L'accélération est plus grande que l'accélération maximale attendue, égale au nombre de processeurs. Elle est superlinéaire.

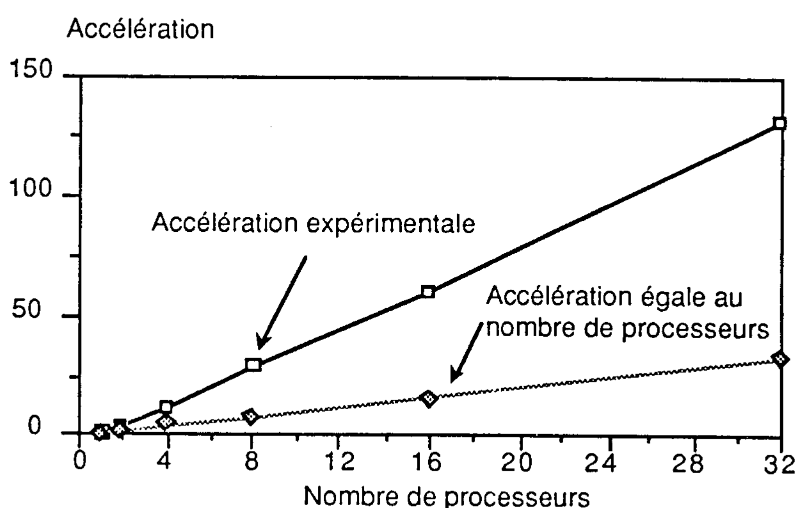


Figure 5.24. Accélérations superlinéaires

On s'attendait en multipliant le nombre de processeur par x, à diviser le temps de calcul par x, ce qui correspond à l'accélération maximale linéaire. Or l'accélération expérimentale est supérieure à cette valeur. Ceci est dû au fait qu'en augmentant le nombre de processeurs, on augmente aussi la mémoire totale. Or comme le programme gère dynamiquement la mémoire qu'il utilise, l'augmentation de la mémoire totale lui évite un grand nombre de libérations et d'allocations. Donc le temps total diminue, puisque le nombre d'opérations n'est pas modifié.

Comment se répartit le temps d'exécution en fonction des phases de l'algorithme, et de l'identité du processeur ? Pour ce faire, nous prenons le cas d'un entier N de 35 chiffres et celui d'un entier N de 41 chiffres. Pour $N \approx 10^{34}$, k vaut 650, donc la matrice doit avoir environ 645 lignes. Pour $N \approx 10^{40}$, k vaut 1150.

Les processeurs construisent la matrice. Mais comment vont-ils se répartir le travail, c'est-à-dire quel sera leur critère d'arrêt ? Autrement dit, dans l'algorithme, comment est implanté le test "assez de lignes dans la matrice" ? On sait qu'il faut k lignes. Donc chacun des P processeurs calcule k/P lignes, c'est-à-dire qu'il doit trouver au moins k/P w(x) complètement factorisés sur la base. En effet, on ne l'arrête pas dès qu'il en a trouvé k/P, on laisse chaque processeur tester les w(x) candidats du polynôme en cours, car on sait que plus la matrice contient de lignes, plus la probabilité est grande de trouver une factorisation non triviale de N. Sur le tableau suivant sont regroupés les temps, en secondes, de chaque étape sur chaque processeur, pour $N \approx 10^{34}$.

Proc	Temps total (s)	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Nombre polyn nécess	Nombre lignes trouvées
0	88,87	4,01	17,73	56,65	10,21	6	23
1	102,35	4,90	20,33	65,67	11,47	7	23
2	122,41	4,90	28,31	76,52	12,44	8	22
3	72,90	4,90	13,07	46,22	8,54	5	23
4	100,90	4,90	18,42	65,29	12,08	7	23
5	120,35	4,90	25,19	76,85	13,16	8	22
6	87,39	4,90	16,06	56,31	9,92	6	23
7	104,79	4,90	21,89	66,42	11,36	7	21
8	89,55	4,90	17,67	56,63	10,17	6	21
9	72,68	4,90	12,89	45,99	8,73	5	21
10	133,93	4,90	26,47	86,97	15,33	9	21
11	107,83	4,90	24,73	66,30	11,69	7	22
12	114,39	4,90	30,60	66,61	12,07	7	21
13	110,23	4,90	26,14	67,23	11,35	7	23
14	91,76	4,90	19,88	57,34	9,45	6	21
15	74,47	4,90	14,25	46,28	8,87	5	22
16	148,74	4,90	32,21	94,92	16,43	10	24
17	97,44	4,90	24,78	57,90	9,67	6	24
18	106,98	4,90	24,10	65,40	12,43	7	29
19	98,25	4,90	26,52	56,09	10,54	6	21
20	104,89	4,90	22,42	65,17	12,18	7	21
21	70,72	4,90	11,17	45,92	8,55	5	21
22	91,92	4,90	20,71	55,91	10,21	6	23
23	86,99	4,90	17,03	54,56	10,30	6	22
24	102,34	4,90	19,52	66,13	11,58	7	22
25	122,76	4,90	30,10	75,03	12,49	8	21
26	94,89	4,90	21,88	57,40	10,53	6	25
27	92,44	4,90	19,72	56,79	10,84	6	25
28	119,01	4,90	25,68	75,45	12,74	8	22
29	77,53	4,90	15,98	47,66	8,82	5	22
30	106,73	4,90	23,96	65,87	11,80	7	25
31	107,14	4,90	25,73	65,11	11,20	7	21

Figure 5.25. Temps d'exécution des phases de l'algorithme pour un entier de 35 chiffres
*N = (100 000 000 000 000 013 * 100 000 000 000 000 003)*

Les figures suivantes visualisent les résultats contenus dans le tableau ci-dessus.

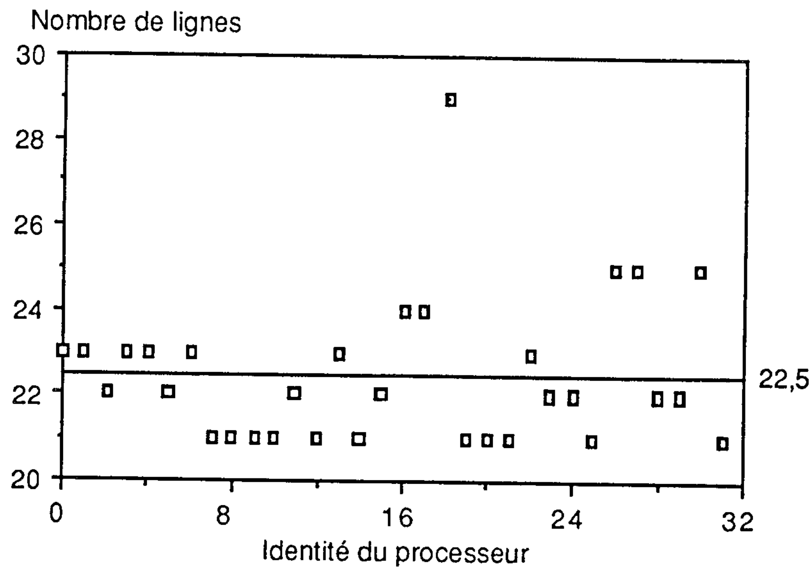


Figure 5.26. Nombre de lignes trouvées par processeur pour un entier de 35 chiffres

Les processeurs trouvent au moins 2 à 3 w(x) factorisés par polynôme.

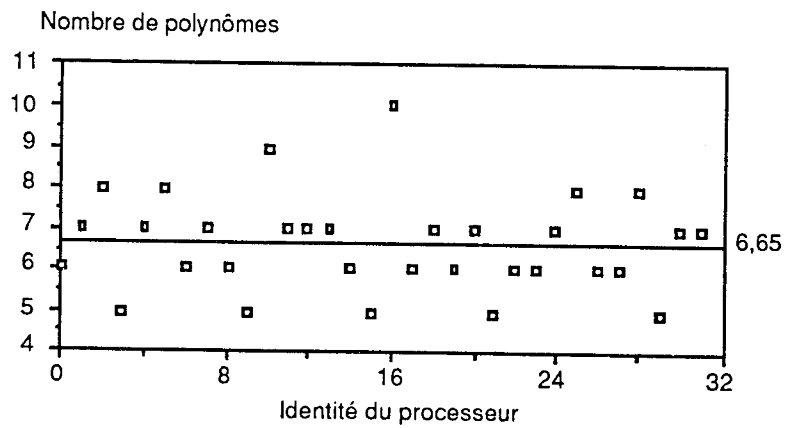


Figure 5.27. Nombre de polynômes nécessaires à la factorisation d'un entier de 35 chiffres

Un processeur génère 10 polynômes, d'autres 5, pour obtenir le nombre de lignes requis.

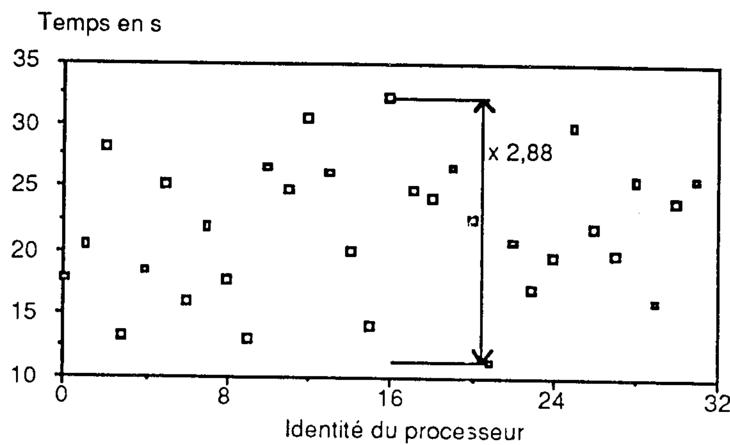


Figure 5.28. Temps de calcul des polynômes pour un entier de 35 chiffres

Le temps de calcul des polynômes suit les variations du nombre de polynômes.

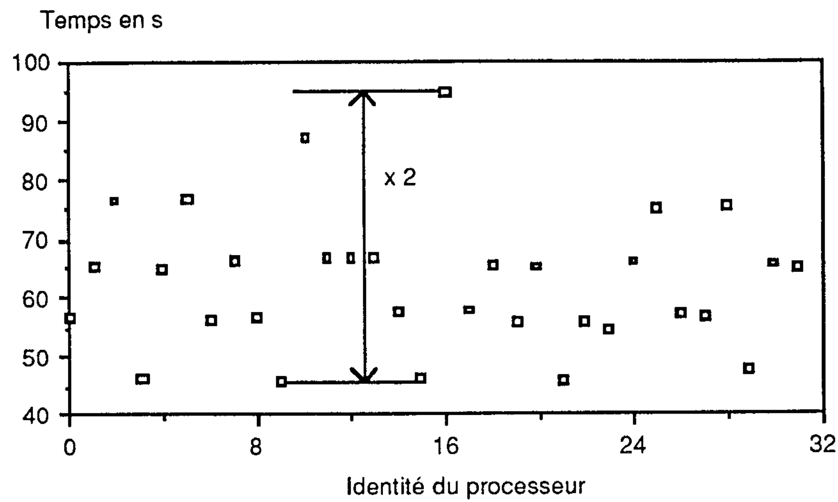


Figure 5.29. Temps de crible pour un entier de 35 chiffres

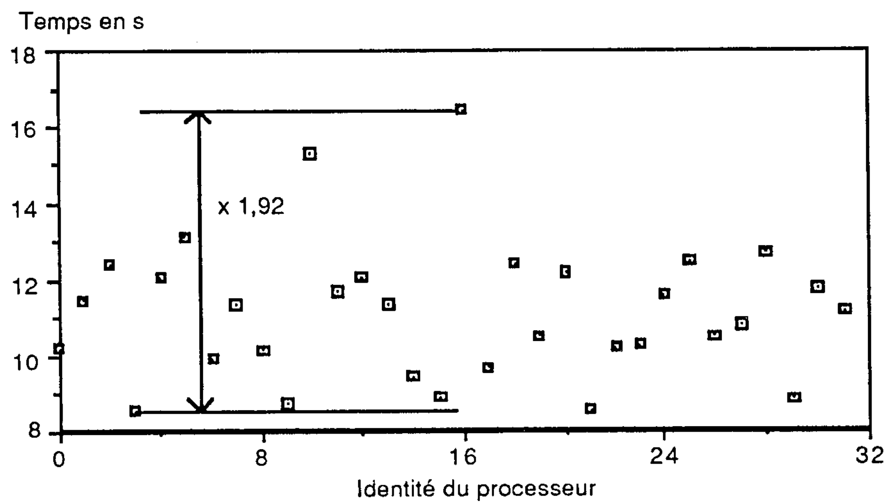


Figure 5.30. Temps de factorisation des w(x) pour un entier de 35 chiffres

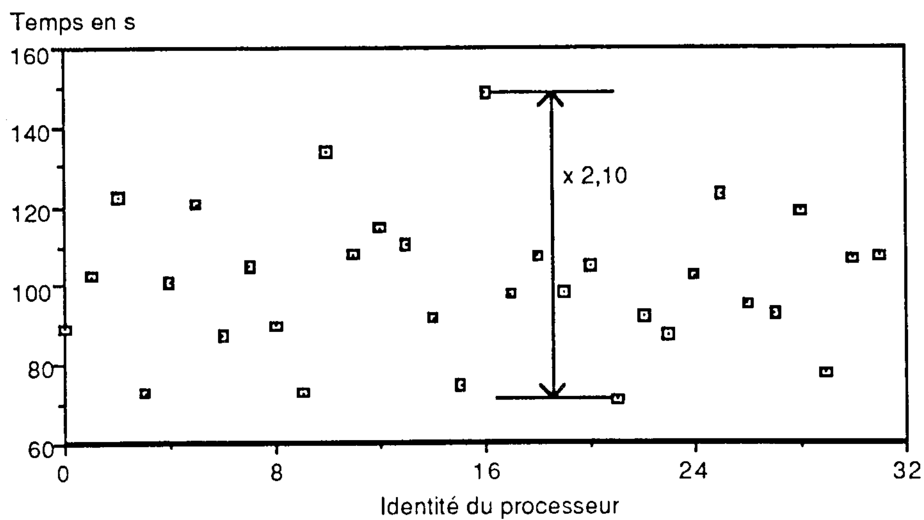


Figure 5.31. Temps total d'exécution pour un entier de 35 chiffres

Le processeur le plus rapide met deux fois moins de temps que le processeur le plus lent. Il reste donc inactif 50 % du temps total de crible. En moyenne les processeurs restent inactifs près du tiers du temps total, car la charge de travail (workload) vaut 67,7%.

Et par conséquent on retrouve le phénomène sur le temps total. Il est à noter que le processeur le plus lent est celui qui a dû générer le plus grand nombre de polynômes pour trouver le nombre de lignes requis. C'est lui aussi qui a eu besoin du temps le plus long pour générer tous ses polynômes. Quant au processeur le plus rapide, c'est lui qui a eu besoin du plus petit nombre de polynômes, et qui a mis le temps minimum pour générer tous ses polynômes.

On remarque donc que le nombre de polynômes générés influe directement sur le temps d'exécution total.

Voyons si cette tendance se vérifie avec $N \approx 10^{40} = (10^{20} + 39).(10^{20} - 11)$

Voici les temps d'exécution en s pour les étapes de la construction de la matrice pour $N \approx 10^{40}$ avec k = 1150.

Proc	Temps total	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Nombre polyn nécess	Nombre lignes trouvées
0	1433,4	8,12	213,0	1130,2	80,4	35	37
1	1298,8	8,96	210,5	1002,2	75,7	32	37
2	1349,6	8,96	217,5	1048,3	73,3	33	37
3	1392,8	8,96	226,6	1074,2	81,4	35	40
4	926,6	8,96	128,3	732,6	55,6	24	37
5	1392,6	8,96	182,7	1119,0	80,4	35	37
6	1007,1	8,96	181,1	757,4	58,6	24	37
7	1343,3	8,96	190,9	1065,9	76,0	34	37
8	1257,4	8,96	194,0	985,3	67,7	31	38
9	865,0	8,96	104,2	699,2	51,6	23	37
10	1662,6	8,96	277,8	1287,3	86,7	39	38
11	1271,8	8,96	185,3	1003,7	72,4	32	37
12	1411,8	8,96	203,5	1120,2	77,5	34	37
13	1272,0	8,96	191,9	1000,1	69,6	32	37
14	1379,0	8,96	199,6	1094,3	74,3	34	39
15	897,6	8,96	136,8	699,5	51,3	23	37
16	1497,5	8,96	244,9	1164,4	77,7	35	37
17	1491,9	8,96	260,6	1142,0	78,8	35	38
18	1114,9	8,96	198,8	846,7	59,2	26	37
19	1696,4	8,96	259,0	1327,0	99,6	41	38
20	933,3	8,96	130,6	738,7	53,9	24	38
21	1132,8	8,96	172,9	886,5	63,1	28	38
22	1517,9	8,96	250,8	1173,9	82,5	37	39
23	1062,4	8,96	155,1	837,8	59,2	27	38
24	1251,2	8,96	191,3	980,2	69,2	31	37
25	1168,8	8,96	167,0	926,2	165,3	30	37
26	985,9	8,96	153,9	762,2	59,6	25	37
27	1276,3	8,96	195,0	998,5	72,4	31	39
28	1205,0	8,96	165,3	963,6	65,7	30	37
29	1418,4	8,96	226,6	1100,5	80,7	35	38
30	1282,5	8,96	221,3	983,7	67,1	31	39
31	1460,4	8,96	220,9	1150,8	78,2	35	38

Figure 5.32. Temps d'exécution des phases de l'algorithme pour un entier de 41 chiffres

Les figures suivantes visualisent les résultats contenus dans le tableau ci-dessus.

On remarque toutefois que le nombre de polynômes générés varie de façon importante entre les processeurs.

Il en est de même en ce qui concerne le temps d'exécution de l'algorithme.

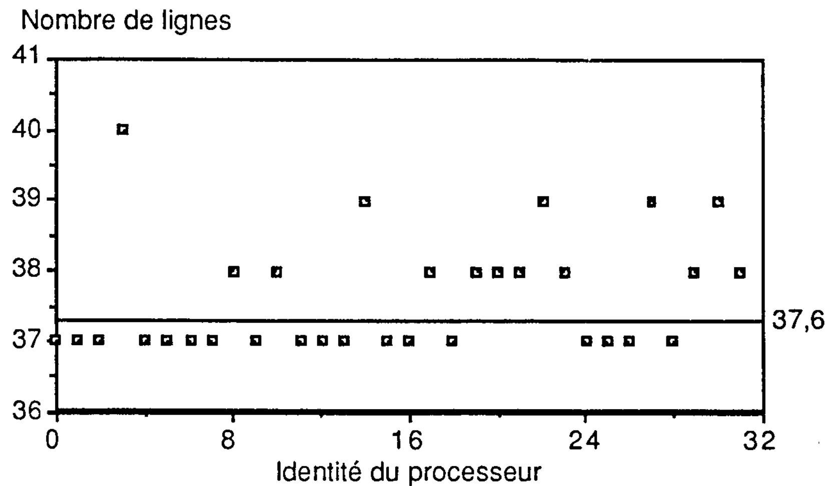


Figure 5.33. Nombre de lignes trouvées par processeur pour un entier de 41 chiffres

Les processeurs trouvent en moyenne un w(x) complètement factorisé par polynôme.

L'écart entre le nombre de polynômes sur le processeur qui en génère le plus et le nombre de polynômes sur celui qui en génère le moins est un facteur 1,8. Cependant il reste élevé, comme la différence de temps entre les processeurs pour la génération des polynômes (fig. 5.34).

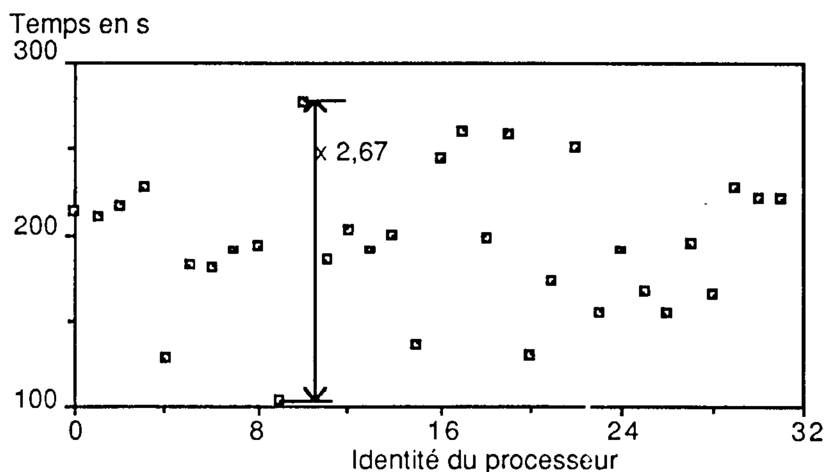


Figure 5.34. Temps de calcul des polynômes pour un entier de 41 chiffres

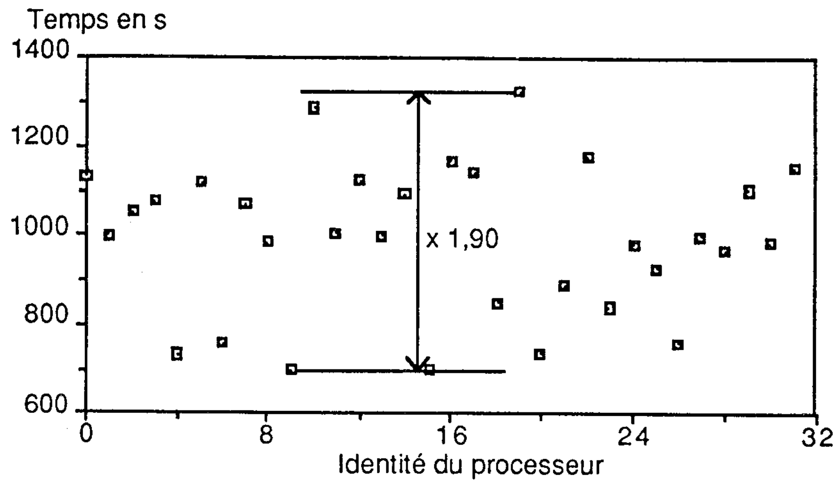


Figure 5.35. Temps de crible pour un entier de 41 chiffres

Le processeur le plus rapide crible en 2 fois moins de temps que le processeur le plus lent.

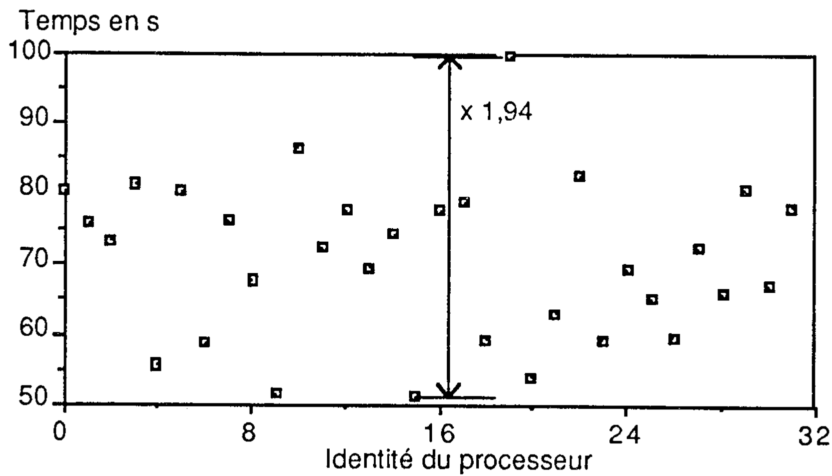


Figure 5.36. Temps de factorisation des w(x) pour un entier de 41 chiffres

Même tendance pour la factorisation des w(x) candidats.

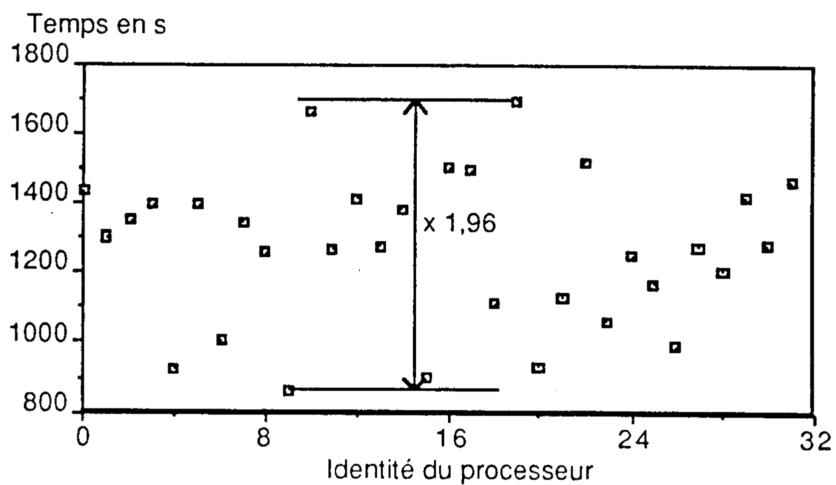


Figure 5.37. Temps total d'exécution pour un entier de 41 chiffres

On retrouve ce rapport 2 sur le temps total d'exécution. On retrouve encore la relation entre nombre de polynômes générés et temps de crible d'une part, et factorisation et temps total d'autre part, pour le processeur le plus lent ainsi que pour le processeur le plus rapide. Pour cet exemple, la charge de travail (workload) est 75%.

5.2. ANALYSE DES RÉSULTATS ET MODIFICATION DE L'ALGORITHME

Il est clair que la charge de travail n'est pas équilibrée entre les processeurs. Donc chacun des P processeurs ne doit pas générer k/P lignes de la matrice. Il faut donc une autre implantation pour le test "assez de lignes dans la matrice". Le but est d'équilibrer la charge entre les processeurs dans la boucle génération des polynômes/crible/factorisation des w(x).

Le déséquilibre provient de deux choses :

- Le temps de calcul du nouveau polynôme : en effet le temps nécessaire pour le calcul d'un nouveau polynôme est relativement long. Et s'il ne convient pas, il faut en essayer un autre, ceci jusqu'à ce qu'on en trouve un qui convienne. Donc il faudrait que tous les processeurs travaillent avec le même nombre de polynômes. De cette manière, les temps de calculs seraient vraisemblablement les mêmes, et la différence de temps entre les processeurs serait en partie gommée.
- Le nombre de w(x) factorisés (et par conséquent le nombre de lignes) n'est pas le même suivant les polynômes. Car il se trouve des polynômes qui ne génèrent aucun w(x) factorisé alors que d'autres polynômes en génèrent un, voire plusieurs.

Il ne faut donc pas déterminer le travail de chaque processeur en fonction du nombre de w(x) complètement factorisés à trouver, mais en fonction du nombre de polynômes qu'on estime nécessaires afin d'obtenir les k (environ) lignes voulues.

Nombre de polynômes théoriquement nécessaires (d'après Caron et Silverman) :

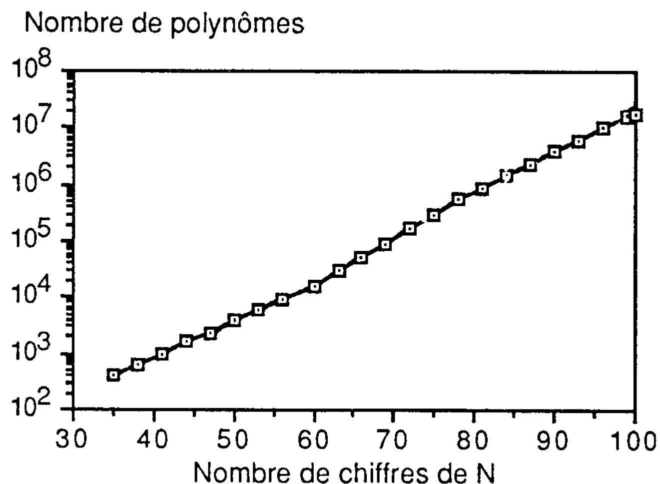


Figure 5.38. Nombre théorique de polynômes nécessaires à la factorisation d'un entier ayant entre 35 et 100 chiffres

Nous savons ainsi calculer le nombre de polynômes nécessaires pour factoriser un entier N. Nous pouvons modifier l'algorithme pour que la répartition des charges soit meilleure.

5.3. RÉSULTATS OBTENUS AVEC L'ALGORITHME INTÉGRANT LA NOUVELLE RÉPARTITION DES TÂCHES

En fait, dans le nouvel algorithme, c'est le test d'arrêt qui est modifié. Il se faisait sur le nombre de w(x) factorisés (il en fallait k/P par processeur). Maintenant, on affecte à chaque processeur

un certain nombre de polynômes à traiter. Lorsque le processeur a terminé avec ces polynômes, il s'arrête quel que soit le nombre de w(x) factorisés qu'il a trouvés, car on a évalué le nombre de polynômes de manière que le nombre de w(x) factorisés soit proche du nombre souhaité.

Avec le même N ≈ 10^{34}, on obtient les résultats suivants. On met en valeur le nombre de polynômes qu'il faut tester pour trouver 6 polynômes utilisables par processeur (au total, 190).

Proc	Nombre lignes trouvées	Nombre polyn testés	Nombre polyn OK	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Temps total
0	23	73	6	4,01	16,4	55,8	10,1	86,6
1	19	77	6	4,90	16,9	54,5	9,9	86,3
2	19	90	6	4,90	17,5	56,2	9,6	88,4
3	25	52	6	4,90	14,7	55,1	10,1	85,0
4	19	41	6	4,90	13,9	54,4	10,2	83,6
5	14	97	6	4,90	18,1	56,4	9,8	89,5
6	23	58	6	4,90	15,1	55,4	9,9	85,4
7	18	75	6	4,90	17,4	55,8	9,2	87,5
8	21	73	6	4,90	16,3	55,8	10,1	87,3
9	23	73	6	4,90	16,4	55,0	10,0	86,5
10	17	68	6	4,90	16,5	55,7	10,9	88,2
11	18	97	6	4,90	19,0	55,8	9,9	89,7
12	19	164	6	4,90	24,2	55,9	10,1	95,3
13	20	116	6	4,90	20,5	56,4	9,8	91,7
14	21	84	6	4,90	18,7	56,4	9,4	89,6
15	24	73	6	4,90	16,5	54,5	10,5	86,6
16	13	48	6	4,90	14,6	55,1	9,7	84,5
17	24	129	6	4,90	22,4	56,5	9,6	93,6
18	19	53	6	4,90	14,8	55,1	10,2	85,1
19	21	153	6	4,90	24,7	55,2	10,5	95,4
20	18	78	6	4,90	16,8	55,0	10,6	87,5
21	24	38	6	4,90	14,2	54,0	9,9	83,2
22	23	99	6	4,90	19,1	54,5	10,1	88,8
23	22	64	6	4,90	15,8	53,9	10,2	85,1
24	16	56	6	4,90	14,7	55,8	9,8	85,4
25	16	107	6	4,90	19,6	55,8	9,7	90,2
26	25	118	6	4,90	20,5	55,8	10,4	91,8
27	25	77	6	4,90	18,2	55,8	10,8	89,9
28	15	77	6	4,90	16,6	55,2	9,6	86,5
29	26	60	6	4,90	16,5	55,8	10,3	87,6
30	18	82	6	4,90	18,2	55,2	9,6	88,1
31	20	110	6	4,90	21,0	55,2	9,5	90,8

Figure 5.39. Temps des étapes de la factorisation de l'entier de 35 chiffres avec équilibrage des tâches

Le temps d'exécution est plus faible qu'avec l'algorithme précédent. Certains processeurs vont plus vite, alors que d'autres travaillent plus lentement. Cependant, il n'y a plus ce facteur 2 entre le temps du processeur le plus lent et celui du processeur le plus rapide, comme le montre la figure suivante.

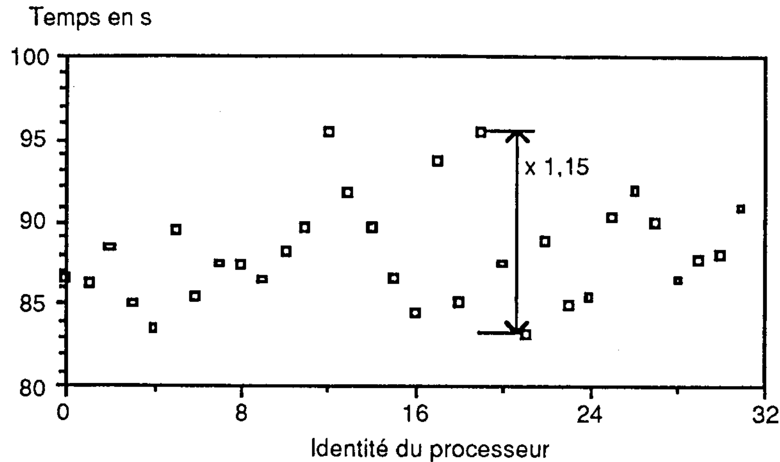


Figure 5.40. Temps d'exécution pour un entier de 35 chiffres avec équilibrage des tâches

La différence entre le processeur le plus lent et le processeur le plus rapide est faible : 15 %. Elle était de 100 % avec l'algorithme précédent. Le taux d'utilisation des processeurs est 92,4 %.

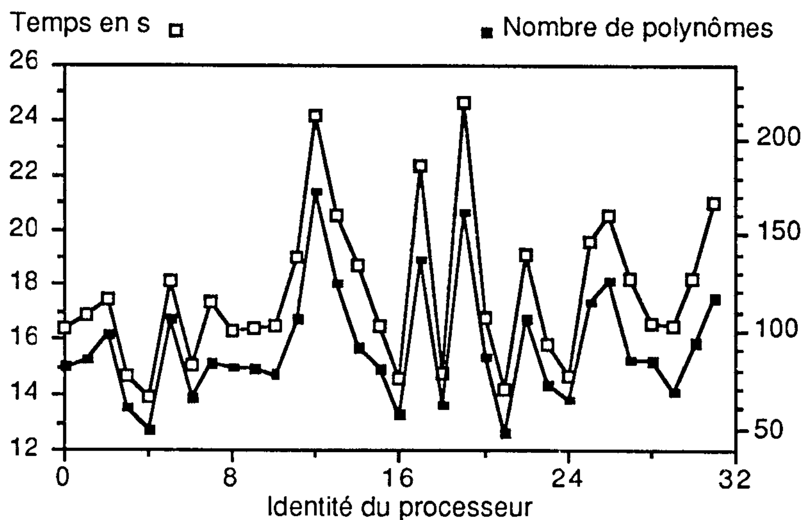


Figure 5.41. Temps de génération des polynômes et nombre de polynômes testés pour la factorisation d'un entier de 35 chiffres

Ces 2 courbes suivent exactement les mêmes variations : le temps est proportionnel au nombre de polynômes testés. Ceci signifie que même si on teste des coefficients qui ne conviennent pas, on ne perturbe pas le temps global : il y a compensation, et ce temps est faible.

De plus, la valeur minimale de ces deux courbes vaut la moitié de la valeur maximale de chacune d'elles. Donc, on voit que pendant l'étape de génération des polynômes, le processeur qui génère le plus petit nombre de polynômes reste inactif la moitié du temps de génération du processeur qui met le plus long temps.

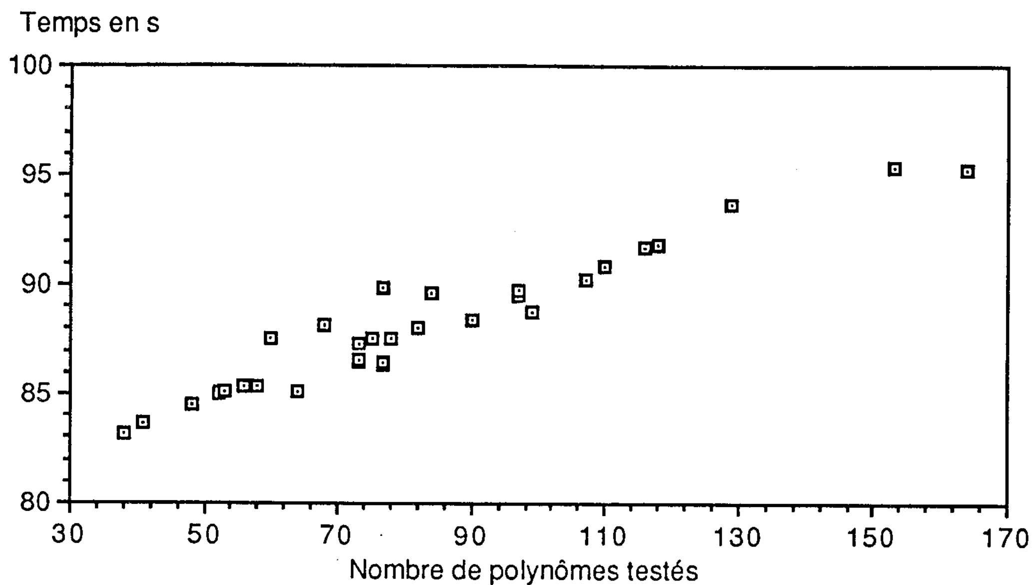


Figure 5.42. Temps total d'exécution de l'algorithme en fonction du nombre de polynômes testés

Le temps de calcul total sur chaque processeur n'est pas le même car certains processeurs ont plus de polynômes à tester pour trouver les bons coefficients que d'autres processeurs. Cependant, les temps sont proches : la charge est presque équirépartie.

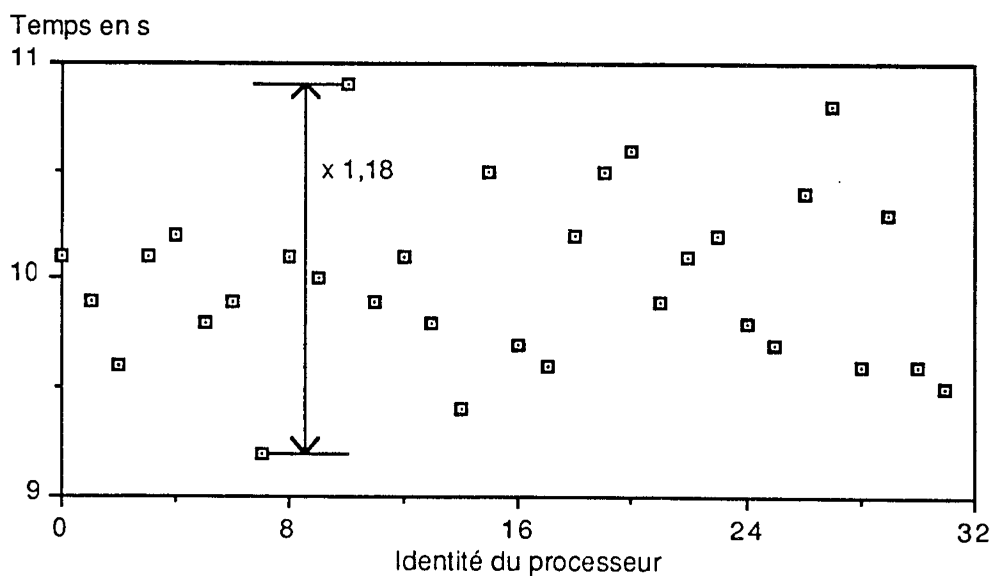


Figure 5.43. Temps de factorisation des w(x) pour un entier de 35 chiffres avec équilibrage

Le temps de factorisation des w(x) est lui aussi assez bien équilibré entre les processeurs.

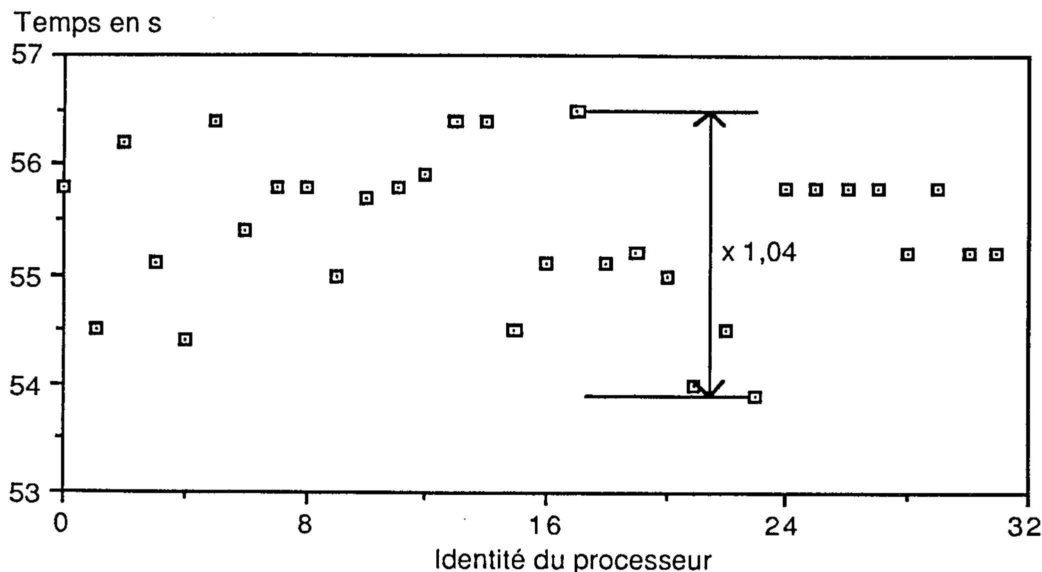


Figure 5.44. Temps de crible pour un entier de 35 chiffres avec équilibrage

Pour factoriser ce nombre de 35 chiffres, il a fallu essayer 2 660 polynômes pour en trouver 192 qui soient corrects. Ils ont permis de trouver 648 lignes d'une matrice qui comporte 669 colonnes (Nombre de lignes = 97 % du nombre de colonnes).

La théorie donne le nombre de polynômes à tester en fonction du nombre de polynômes voulus : un polynôme est équivalent à un coefficient a. Si on a besoin de L polynômes, alors en moyenne, il faudra tester 2L log (2L) coefficients a. Dans notre cas, pour avoir 192 polynômes il faudrait tester théoriquement 2 285 coefficients (il y a donc une erreur de près de 20 %, car on en a essayé 2660). On peut donc estimer a priori le nombre de polynômes à tester au total, et le répartir sur les processeurs.

Pour l'entier de 41 chiffres, on obtient les résultats suivants :

Proc	Nombre lignes trouvées	Nombre polyn testés	Nombre polyn OK	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Temps total
0	33	697	38	7,29	149	557	88	802
1	35	848	38	9,14	168	549	89	815
2	46	809	38	9,14	161	553	86	809
3	31	765	38	9,14	160	548	87	805
4	42	514	38	9,14	134	550	87	780
5	34	549	38	9,14	137	558	88	793
6	46	973	38	9,14	185	552	89	836
7	37	571	38	9,14	140	551	87	787
8	41	737	38	9,14	157	552	84	802
9	51	543	38	9,14	137	549	92	787
10	35	746	38	9,14	160	558	85	812
11	33	684	38	9,14	153	555	87	805
12	37	675	38	9,14	152	558	86	805
13	37	653	38	9,14	149	550	83	791
14	39	623	38	9,14	149	554	85	798
15	48	850	38	9,14	171	553	87	820
16	33	818	38	9,14	168	559	85	821
17	36	848	38	9,14	170	554	86	820
18	50	966	38	9,14	182	557	85	833

19	28	628	38	9,14	144	554	91	798
20	57	747	38	9,14	157	555	87	808
21	40	753	38	9,14	160	558	97	814
22	34	733	38	9,14	160	551	86	807
23	41	641	38	9,14	150	551	85	796
24	40	817	38	9,14	164	556	89	818
25	35	760	38	9,14	157	552	83	801
26	40	779	38	9,14	164	548	89	810
27	49	688	38	9,14	157	558	90	814
28	44	607	38	9,14	146	555	87	796
29	33	729	38	9,14	159	551	87	807
30	41	802	38	9,14	172	555	85	820
31	36	692	38	9,14	159	561	85	814

Figure 5.45. Temps des étapes de la factorisation d'un entier de 41 chiffres avec équilibrage des tâches

Workload : 96,5%. On voit que la charge de travail est bien répartie entre les processeurs. De plus, les étapes prennent le même temps sur les processeurs : la charge est bien équilibrée, au niveau des étapes de l'algorithme.

Pour un nombre de 45 chiffres $(10^{22} + 9).(10^{22} + 57)$, on obtient les résultats suivants.

Proc	Nombre lignes trouvées	Nombre polyn testés	Nombre calcul OK	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Temps total
0	57	1492	57	10,95	295	1190	180	1677
1	62	1455	57	11,87	287	1190	177	1667
2	50	1171	57	11,87	262	1187	175	1637
3	60	1576	57	11,87	309	1195	176	1693
4	57	1298	57	11,87	278	1188	175	1654
5	55	1096	57	11,87	254	1193	181	1641
6	61	1069	57	11,87	262	1187	175	1638
7	64	1184	57	11,87	268	1194	178	1652
8	46	1189	57	11,87	265	1188	173	1646
9	70	1619	57	11,87	312	1191	183	1700
10	68	975	57	11,87	245	1188	185	1632
11	65	1027	57	11,87	247	1191	174	1625
12	71	1097	57	11,87	261	1194	181	1649
13	58	1262	57	11,87	274	1200	177	1664
14	79	1459	57	11,87	301	1188	181	1683
15	63	1222	57	11,87	275	1195	182	1666
16	59	1368	57	11,87	282	1191	178	1664
17	57	1510	57	11,87	304	1197	175	1689
18	59	1078	57	11,87	255	1187	177	1633
19	62	1043	57	11,87	250	1183	175	1621
20	58	1227	57	11,87	267	1195	176	1652
21	70	1358	57	11,87	288	1197	175	1673
22	64	1262	57	11,87	278	1190	178	1659
23	61	1246	57	11,87	283	1192	175	1663
24	64	1358	57	11,87	299	1188	183	1680
25	61	1013	57	11,87	246	1186	177	1622

26	66	1297	57	11,87	286	1193	179	1671
27	42	1196	57	11,87	271	1188	173	1645
28	66	1179	57	11,87	267	1194	185	1659
29	69	990	57	11,87	247	1184	176	1620
30	74	1240	57	11,87	276	1191	179	1660
31	63	1094	57	11,87	263	1188	179	1643

Figure 5.46. Temps des étapes de la factorisation d'un entier de 45 chiffres avec équilibrage des tâches

Workload : 97,4%

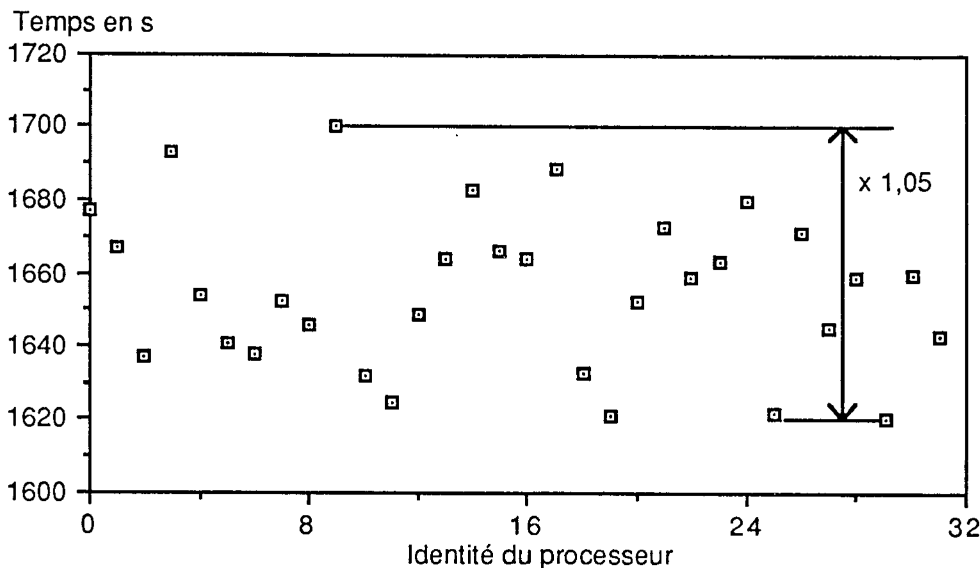


Figure 5.47. Temps d'exécution de l'algorithme pour un entier de 45 chiffres avec équilibrage

On obtient 5 % de différence entre le processeur le plus rapide et le processeur le plus lent : la différence décroît avec l'augmentation de la taille de N. Une perte minime de 5 % environ représente 80 s de perte sur 1 680 s au total, soit 1 min 20 s sur 28 min.

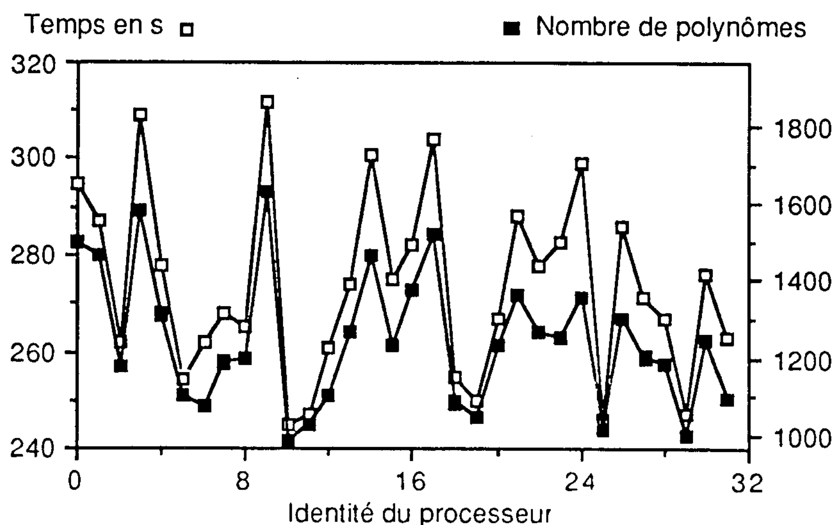


Figure 5.48. Temps de calcul des polynômes et nombre de polynômes testés pour un entier de 45 chiffres avec équilibrage

Sur la figure 5.48, on voit que les courbes suivent les mêmes variations. La courbe du nombre de polynômes (carrés noirs) est décalée vers le bas, de manière à ne pas recouvrir la courbe du temps, car ces deux courbes se superposent presque parfaitement.

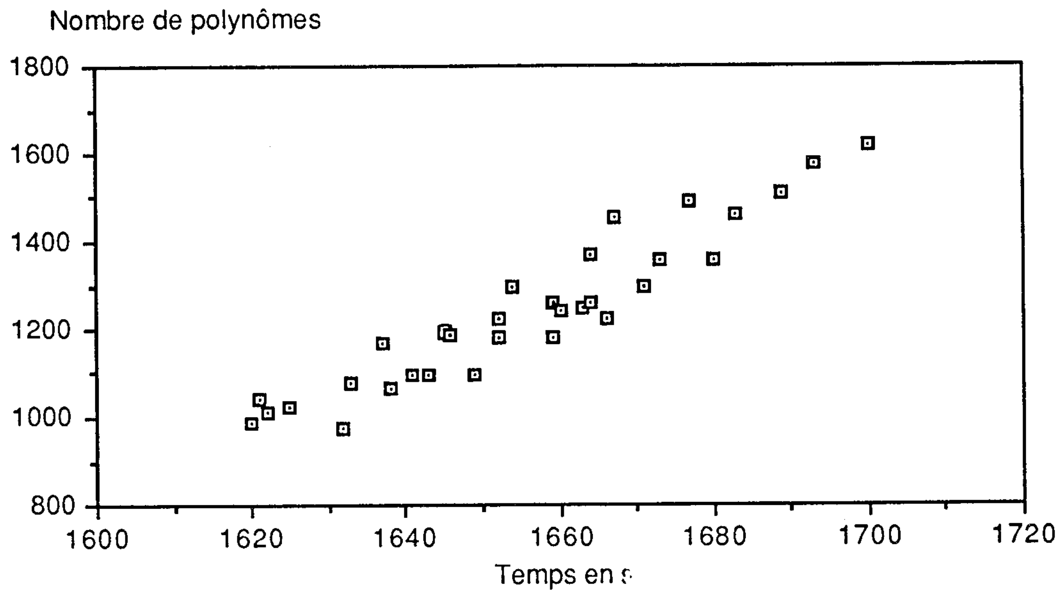


Figure 5.49. Temps total et nombre de polynômes testés pour un entier de 45 chiffres avec équilibrage

Il y a une différence peu importante entre le nombre de polynômes du processeur qui en génère le plus et le nombre de polynômes du processeur qui en génère le moins. C'est ce qui permet d'avoir une bonne répartition des charges.

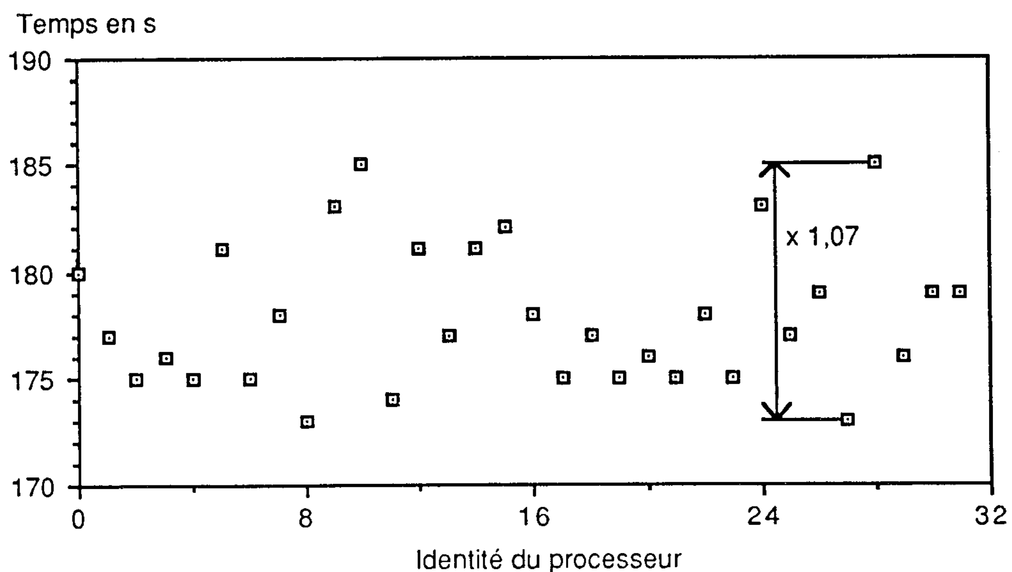


Figure 5.50. Temps de factorisation des w(x) pour un entier de 45 chiffres avec équilibrage

Là encore la différence de temps entre les processeurs de temps extrêmaux est faible : 7 %.

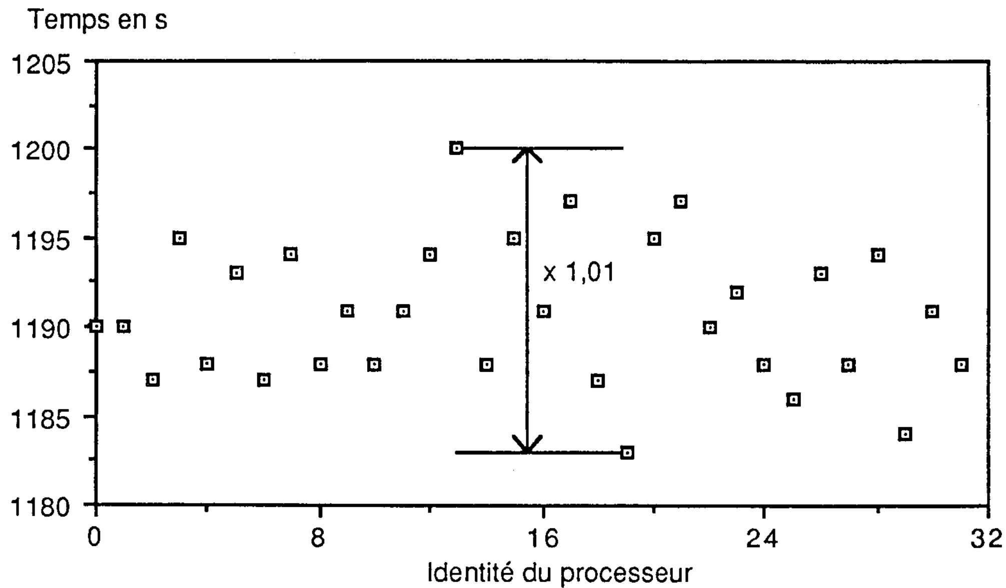


Figure 5.51. Temps de crible pour un entier de 45 chiffres avec équilibrage

Les écarts sont minimes, même dans les étapes prises séparément.

Poursuivons les exécutions, en augmentant la taille des entiers à factoriser. Pour un nombre de 51 chiffres $(10^{25} + 13).(10^{25} + 223)$, on obtient les résultats suivants.

Proc	Nombre lignes trouvées	Nombre polyn testés	Nombre polyn OK	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Temps total
0	58	3276	144	15,25	1114	4649	796	6577
1	63	3433	144	16,07	1139	4624	779	6561
2	62	3386	144	16,07	1154	4634	782	6590
3	76	3838	144	16,07	1206	4635	786	6646
4	76	3980	144	16,07	1271	4657	795	6742
5	71	3150	144	16,07	1113	4646	781	6559
6	60	3373	144	16,07	1161	4648	800	6627
7	75	3350	144	16,07	1149	4664	795	6626
8	64	3695	144	16,07	1196	4627	776	6619
9	59	3463	144	16,07	1157	4652	776	6604
10	54	3151	144	16,07	1114	4642	776	6552
11	71	3948	144	16,07	1270	4642	790	6721
12	56	3543	144	16,07	1190	4638	778	6626
13	58	3673	144	16,07	1203	4639	784	6646
14	65	3335	144	16,07	1174	4629	778	6600
15	70	3535	144	16,07	1196	4650	785	6651
16	57	3449	144	16,07	1171	4647	786	6623
17	57	4175	144	16,07	1285	4630	784	6718
18	70	3655	144	16,07	1207	4646	779	6652
19	54	3479	144	16,07	1169	4640	783	6611
20	64	3359	144	16,07	1172	4644	793	6628
21	63	3343	144	16,07	1146	4659	796	6620
22	60	3841	144	16,07	1257	4654	791	6722
23	62	3176	144	16,07	1130	4628	783	6560
24	59	3459	144	16,07	1175	4635	775	6604

25	54	3830	144	16,07	1229	4650	782	6680
26	76	3978	144	16,07	1276	4638	794	6727
27	53	3139	144	16,07	1132	4654	778	6583
28	55	3592	144	16,07	1230	4657	782	6688
29	52	3474	144	16,07	1197	4654	776	6646
30	78	3324	144	16,07	1170	4629	791	6609
31	70	3459	144	16,07	1196	4643	785	6628

Figure 5.52. Temps des étapes de la factorisation d'un entier de 51 chiffres avec équilibrage des tâches

Workload : 98,6%

Il n'y a que 3 % de différence entre le processeur le plus rapide et le processeur le plus lent.

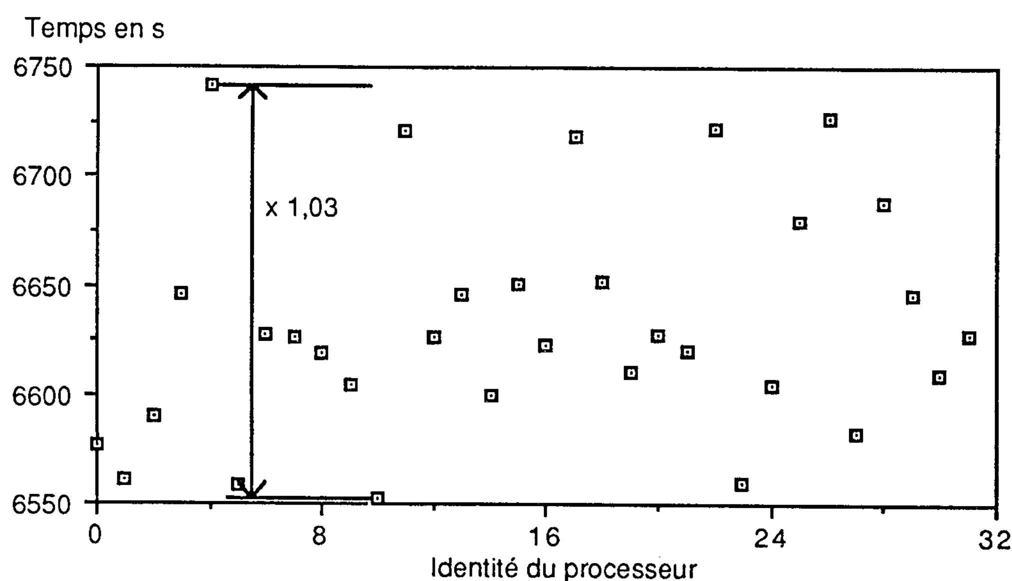


Figure 5.53. Temps d'exécution de l'algorithme pour un entier de 51 chiffres avec équilibrage

Pour un nombre de 54 chiffres $(10^{26} + 67).(10^{27} + 103)$, les temps sont les suivants :

Proc	Nombre lignes trouvées	Nombre polyn testés	Nombre polyn OK	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Temps total
0	77	8550	344	19,76	2975	13079	2189	18271
1	91	8582	344	20,62	2940	13060	2200	18228
2	73	9059	344	20,62	3074	13078	2199	18379
3	66	9495	344	20,62	3150	13045	2189	18413
4	85	8413	344	20,62	2996	13051	2206	18282
5	71	8784	344	20,62	3026	13023	2200	18278
6	75	8016	344	20,62	2912	13029	2183	18152
7	69	9844	244	20,62	3245	13055	2191	18520
8	80	9156	344	20,62	3108	13024	2188	18348
9	94	9159	344	20,62	3096	13079	2214	18417
10	77	8399	344	20,62	3002	13044	2200	18274
11	76	8367	344	20,62	3001	13074	2216	18319

12	81	8868	344	20,62	3097	13038	2200	18364
13	89	9799	344	20,62	3223	13071	2198	18521
14	71	9398	344	20,62	3209	13056	2203	18497
15	83	9436	344	20,62	3196	13038	3186	18449
16	80	9071	344	20,62	3104	13038	2200	18370
17	67	8862	344	20,62	3083	13062	2199	18373
18	71	9547	344	20,62	3195	13044	2203	18471
19	80	8413	344	20,62	2995	12999	2191	18204
20	82	9228	344	20,62	3182	13067	2202	18479
21	64	8557	344	20,62	3031	13023	2191	18273
22	77	9047	344	20,62	3172	13051	2193	18445
23	84	9192	344	20,62	3166	13046	2196	18436
24	85	9117	344	20,62	3147	13046	2199	18421
25	82	8389	344	20,62	2989	13025	2180	18223
26	60	9207	344	20,62	3196	13075	2183	18484
27	74	8742	344	20,62	3086	13064	2206	18385
28	77	9761	344	20,62	3299	13020	2200	18548
29	82	8612	344	20,62	3057	13012	2186	18284
30	74	8958	344	20,62	3188	13058	2193	18467
31	69	8615	344	20,62	3092	13049	2179	18349

Figure 5.54. Temps des étapes de la factorisation d'un entier de 54 chiffres avec équilibrage des tâches

Workload : 99,1%

Dans ces résultats, on voit qu'il n'y a que 2 % d'écart entre le processeur le plus lent et le processeur le plus rapide pour générer 344 polynômes chacun, cribler l'intervalle et factoriser les w(x) candidats. Ici, on obtient 2 466 lignes pour la matrice.

Or on en n'avait besoin que de 2 200. Avec environ 9 450 polynômes, on avait une probabilité élevée d'obtenir assez de lignes pour qu'il y ait des combinaisons linéaires. De même pour le temps, on aurait obtenu ces lignes suffisantes en 16 350 s environ, soit 4 h 30 environ.

Le nombre de polynômes n'est pas cohérent avec celui de la table donnée à partir des expériences de Caron et Silverman, car ils utilisent des améliorations algorithmiques que nous n'avons pas implantées.

Pour un nombre de 60 chiffres enfin $(10^{29} + 319).(10^{30} + 57)$, le plus grand que nous ayons testé, les résultats sont les suivants :

Proc	Nombre lignes trouvées	Nombre polyn testés	Nombre polyn OK	Temps calcul base	Temps calcul polyn	Temps calcul crible	Temps calcul facto	Temps total
0	92	47561	1594	31,6	17318	79113	16258	112757
1	85	49092	1594	32,4	17623	79059	16263	113013
2	94	47095	1594	32,4	17437	79043	16229	112777
3	108	49181	1594	32,4	17773	79126	16286	113252
4	104	47876	1594	32,4	17676	79079	16249	113073
5	93	47848	1594	32,4	17604	79128	16241	113042
6	83	46697	1594	32,4	17458	79106	16255	112888
7	95	47192	1594	32,4	17503	79135	16235	112941
8	84	46299	1594	32,4	17314	79162	16274	112818

9	108	46690	1594	32,4	17341	79056	16281	112747
10	98	47046	1594	32,4	17599	79096	16227	112992
11	87	48978	1594	32,4	17901	79057	16286	113314
12	102	49477	1594	32,4	18260	78939	16227	113494
13	91	46256	1594	32,4	17323	79028	16249	112668
14	100	45609	1594	32,4	17450	79050	16254	112823
15	83	50215	1594	32,4	18302	79004	16239	113613
16	103	48783	1594	32,4	17872	79088	16264	113293
17	113	45240	1594	32,4	17068	79106	16253	112495
18	92	47008	1594	32,4	17530	79016	16253	112868
19	93	46691	1594	32,4	17456	79068	16292	112284
20	111	49111	1594	32,4	18126	79047	16309	113550
21	93	48773	1594	32,4	17866	79082	16248	113265
22	103	46089	1594	32,4	17532	79217	16265	113083
23	111	46617	1594	32,4	17560	79098	16267	112993
24	93	47998	1594	32,4	17879	79021	16218	113186
25	75	45726	1594	32,4	17340	79089	16265	112762
26	96	48244	1594	32,4	18081	79075	16228	113452
27	86	47186	1594	32,4	17631	79074	16268	113042
28	98	47894	1594	32,4	18100	79149	16233	113551
29	94	46321	1594	32,4	17472	79069	16270	112879
30	91	47756	1594	32,4	18126	79039	16273	113507
31	84	46853	1594	32,4	17768	79125	16243	113205

Figure 5.55. Temps des étapes de la factorisation d'un entier de 60 chiffres avec équilibrage des tâches

Workload : 99,5%

On a k = 3 000. Et ici, on obtient 3 043 lignes dans le temps de 113 600 s, soit environ 31,5 h .

La différence de temps entre le processeur le plus rapide et le processeur le plus lent est de moins de 1,0 %. Ceci signifie que le processeur qui termine le premier reste moins de 19 minutes inactif, avant que le dernier ne termine à son tour. 19 mn sur 31,5 h (soit 1890 mn) représentent environ 1 %.

De plus, la place totale allouée et libérée tout au long de l'exécution de l'algorithme, uniquement pour les calculs en précision entière illimitée, va de près de 290 Mo (290 297 363 octets exactement) sur le processeur le plus rapide (17) à près de 300 Mo (300 176 758 octets exactement) sur le processeur le plus lent (15). Ce qui nous donne un total de près de 9,6 Go de mémoire nécessaire au total. Comme le code tient 153 Ko, la pile 4 Ko environ, la base 85 Ko, et l'intervalle de crible 450 Ko, ce qui utilise 700 Ko environ, il reste 300 Ko pour la mémoire allouable. Donc chaque processeur a, en moyenne, réutilisé 1 000 fois chaque emplacement mémoire.

Ces 9,6 Go ont permis de tester 1 459 402 polynômes différents pour obtenir 51 008 polynômes convenables pour générer des w(x) factorisables, soit une moyenne de 6 578 octets par polynôme testé, 188 205 octets par polynôme utilisable et 28,6 valeurs de a testées pour trouver une valeur adéquate. Rappelons que pour un nombre de 100 chiffres, nous avions évalué à 34 le nombre de tests nécessaires en moyenne pour obtenir un coefficient a utilisable. Cette différence est due au fait que les valeurs de a ici sont plus petites, ce qui permet de trouver plus de nombres premiers.

6. CONCLUSION

Les temps d'exécution de l'algorithme sont importants, et ils ne nous permettent pas d'envisager des records, quand on sait que sur un Cray XMP, il n'a fallu que 9,5 heures pour factoriser un nombre de 71 chiffres [Sil 87].

Il est vrai qu'en implantant les variations algorithmiques, on pourrait théoriquement diviser le temps par un facteur variant entre 10 et 30. Cependant, n'oublions pas que ces variations nécessitent une place mémoire bien plus importante (par nœud), place dont les multiprocesseurs actuels ne disposent pas !

Quoi qu'il en soit, voici les temps d'exécution du crible quadratique que nous avons obtenus sur le FPS T40.

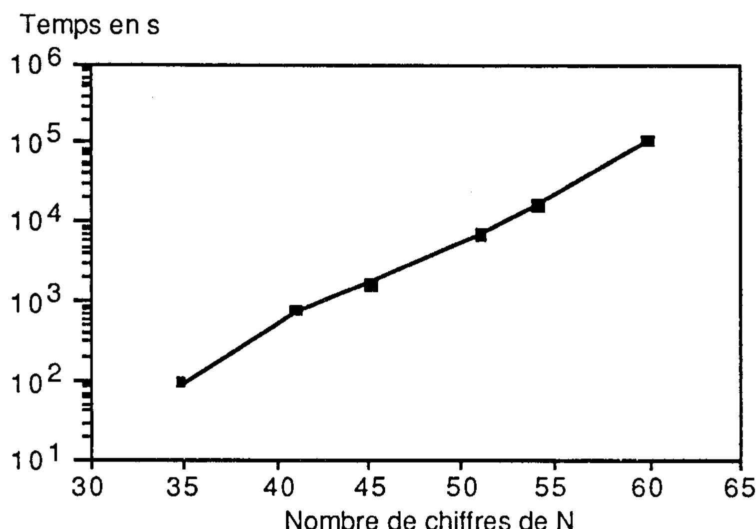


Figure 5.56. Courbe du temps moyen de factorisation d'un entier ayant entre 35 et 60 chiffres

Nombre de chiffres	Temps
35	90 s
41	720 s
45	1 650 s
51	6 650 s
54	16 350 s
60	113 500 s

Figure 5.57. Temps moyen de factorisation d'un entier ayant entre 35 et 60 chiffres

Mais d'un point de vue algorithmique, nous avons montré que la parallélisation habituelle (distribution du travail dirigée par les résultats : même nombre de lignes de la matrice trouvées par chaque processeur), qui a été implantée sur des réseaux de stations de travail, ne s'adapte pas bien à une machine multiprocesseur à mémoire distribuée. En effet nous avons obtenu des différences de temps d'exécution sur différents processeurs allant du simple au double avec cette stratégie de répartition des tâches qui consiste à faire calculer à chacun des P processeurs la P^e partie de la matrice à construire. Cette répartition était basée sur les résultats de l'algorithme. Nous l'avons modifiée en la faisant porter sur les données de l'algorithme. Et même si, pour des petites tailles de N, l'équilibrage des charges sur les processeurs n'est pas parfait, il tend à s'améliorer lorsque la taille de N croît. Et, surtout, il est bien meilleur qu'avec la première stratégie.

Nous avons donc atteint notre but de concevoir un algorithme adapté aux machines multiprocesseurs à mémoire distribuée.

En effet, le taux d'utilisation des processeurs devient proche du maximum :

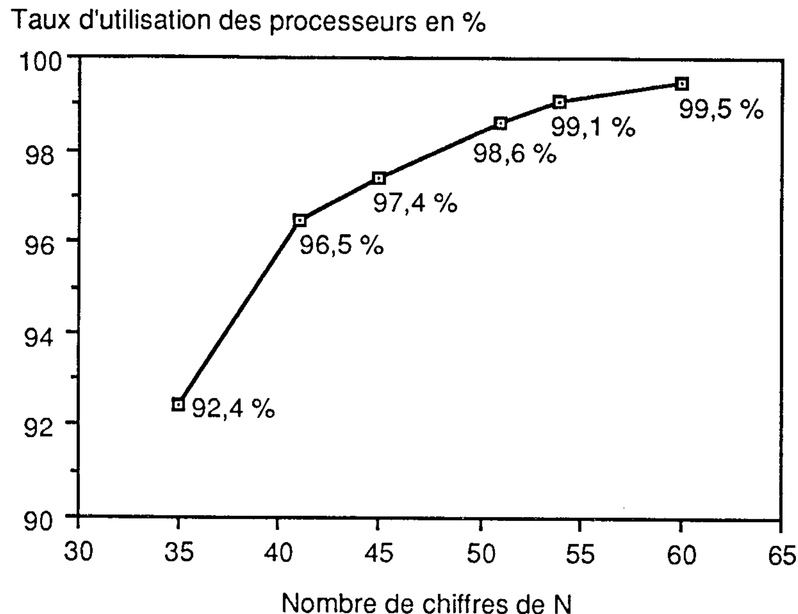


Figure 5.58. Taux d'utilisation des processeurs en % (workload)

De plus, chaque étape de l'algorithme est bien équilibrée sur les différents processeurs.

Quant à l'étude théorique, si elle fait apparaître de nombreuses possibilités, bien peu sont réellement implantables. Mais elle permet de comparer certaines stratégies de répartition ou d'accès aux données, sur des machines hypothétiques qui, pourquoi pas, existeront peut-être un jour ou l'autre.

Le crible quadratique est bien adapté au traitement parallèle, puisqu'il est possible de découper l'algorithme en tâches indépendantes, et les données en une partition de sous-ensembles disjoints, tels que l'algorithme soit exécutable (en ce qui concerne la boucle de construction de la matrice, phase la plus coûteuse en temps) sur une ferme de processeurs, c'est-à-dire sans aucune communication entre les processeurs. Pour obtenir des performances élevées, il faudrait des processeurs puissants capables de "peigner" la mémoire, comme sur un Cray. C'est-à-dire qu'un processeur est capable, en un cycle, d'effectuer une opération vectorielle sur une zone mémoire, mais en des points régulièrement espacés de cette mémoire. Ceci est très pratique pour le crible, puisqu'on ajoute log p tous les p indices. De plus, ce processeur doit avoir une mémoire de grande taille, de manière à pouvoir contenir la base complète et un grand intervalle, puis pour les variations avec le (ou les) grands(s) nombre(s) premier(s), toutes les informations relatives aux w(x) qui entrent dans cette catégorie.

Il n'est pas nécessaire de construire un processeur capable d'effectuer des opérations en multiprécision entière, car les opérations de ce type se rencontrent surtout dans la phase d'initialisation et dans le calcul du pgcd pour la découverte des facteurs de N. On en rencontre aussi quelques-unes dans la phase de crible pour calculer les points de départ. D'autres proviennent du calcul des polynômes. La subtibilité de tester la pseudo-primalité de a par rapport à une seule base (la base 2 par exemple) et de vérifier ensuite que la division pour le calcul de c rend bien un reste nul, est très utile et permet de gagner beaucoup de temps en calcul et en gestion de la mémoire dynamique.

Quant à l'implantation sur d'autres machines, à mon avis, elle ne pose pas de problème sur le TNode ou l'iPSC/2 puisque les tailles mémoire sont comparables à celle du FPS T40, et les modes de communication sont assez proches. Pour ce qui est de la Connection Machine, dont la mémoire de chaque nœud est assez réduite, il faut envisager d'une part des communications entre voisins (NEWS) pour se transmettre les parties de la base qui a été distribuée selon un modèle analogue à celui présenté au paragraphe 2.3.2.c. De plus, il doit être possible d'utiliser le Data Vault, et même de faire effectuer sur le frontal certains des calculs. Cependant, on ne peut plus compter sur la compensation entre les processeurs, puisque les instructions sont envoyées par le frontal à tous les nœuds en même temps (SIMD), ce qui ne permet pas une désynchronisation des processeurs.

L'avènement de processeurs plus puissants (Intel i860, Motorola 68040, Transputer H1, ...) peut aider les multiprocesseurs à mémoire distribuée dans leur combat face aux multiprocesseurs à mémoire partagée. En effet, les futurs iPSC/3 ou machines à base de Transputers H1 pourront flirter avec les puissances des Cray. Et le crible quadratique pourrait alors connaître de nouveaux records.

Conclusion

L'étude que nous venons de présenter porte principalement sur deux thèmes :

- l'élaboration d'outils et de méthodologies pour la parallélisation d'algorithmes,
- l'étude de la parallélisation d'algorithmes issus de la théorie des nombres,

sur des machines multiprocesseurs à mémoire distribuée, et plus particulièrement le FPS T40.

En ce qui concerne les méthodologies de parallélisation, nous avons trouvé une répartition théorique asymptotiquement optimale des entiers pour le crible d'Eratosthène sur un multiprocesseur à mémoire partagée (chapitre 1, paragraphe 3.4). Malheureusement, cette méthode ne donne pas de bons résultats lorsqu'on l'exécute sur le FPS T20 : les temps d'exécution sont très longs, à cause d'une gestion très lourde.

Nous avons dérivé des algorithmes de génération des nombres premiers par la méthode des divisions successives sur un anneau de processeurs (chapitre 1, paragraphe 5). Ces algorithmes sont de plus en plus performants, car ils intègrent successivement des améliorations algorithmiques, fruits de l'analyse des résultats donnés par l'exécution des algorithmes précédents. C'est ainsi que nous réduisons le temps de communication en groupant les messages par paquets et que nous diminuons le temps de calcul en instaurant une redondance limitée des informations. Tout ceci contribue à l'amélioration de l'implantation de l'algorithme, car le temps total d'exécution décroît.

Nous avons élaboré des algorithmes de détection de la terminaison distribuée (chapitre 2) pour l'application de génération des nombres premiers par divisions successives sur un anneau. Les algorithmes sont très efficaces : ils sont symétriques (tout processeur peut détecter la terminaison, indépendamment des autres) et un processeur ne génère pas de message de contrôle tant qu'il est actif. Ils utilisent le principe de comptage des processeurs devenant passifs, sur chaque processeur pour le premier algorithme, puis sur des jetons valués circulant sur l'anneau, en ce qui concerne le deuxième algorithme. De plus, nous construisons un processus de terminaison qui remet les processeurs et les voies de communication dans un état cohérent, et ceci, très rapidement, après la détection la terminaison distribuée globale.

Ces études nous permettent de construire un outil logiciel d'aide à la parallélisation et à l'implantation d'applications Maître / Esclaves sur un réseau linéaire ou une grille de processeurs (chapitre 3). D'un point de vue facilité d'emploi, l'objectif est atteint : l'outil que nous avons développé gère les communications et l'état des processeurs de façon transparente pour le programmeur. Ce dernier doit donner les instructions du Maître, des Esclaves, les conditions de changement d'état, la structure des données à communiquer du Maître vers les Esclaves et la répartition initiale des données entre les processeurs. Quant aux performances, elles se dégradent (30 %) lorsqu'on transfère une même application sur le réseau linéaire sur la grille. Ceci tient à la gestion plus complexe des voies logiques de communication, et à la connectivité qui entraîne des attentes en synchronisation.

Ces travaux ne représentent qu'une petite partie de ce qu'il est nécessaire de construire. En effet, la restriction à certaines topologies doit être levée. Nous proposons des idées pour développer des méthodologies d'implantation d'algorithmes Maître / Esclaves sur l'arbre et l'hypercube, gérées par le même type de méta-algorithme. Quant aux types d'applications, il

faut étudier ces mêmes méta-algorithmes pour des fermes de processeurs et des applications de type "diviser pour régner", qui peuvent elles aussi se découper en tâches précises. Mais il est d'abord nécessaire de regrouper les applications selon leur type, c'est-à-dire effectuer un classement des algorithmes.

En ce qui concerne la parallélisation d'algorithmes de la théorie des nombres, le crible d'Eratosthène a été étudié dans un environnement à mémoire partagée, et une répartition optimale des données a été démontrée (chapitre 1). Mais, dans un environnement à mémoire distribuée, cet algorithme ne permet pas de générer efficacement des nombres premiers. C'est le crible d'Eratosthène non modifié qui est le plus efficace. Cela tient au fait que les opérations de base sont simples et rapides : il n'y a pas de divisions, tout est basé sur l'addition.

Nous avons aussi analysé le crible quadratique multipolynomial de manière à extraire le maximum d'informations sur l'algorithme séquentiel, son découpage en tâches, ses besoins en données, ... (chapitre 4). Cette étude fine permet d'élaborer un algorithme parallèle qui tire parti des machines-cibles. En effet, jusqu'à présent, des implantations parallèles triviales avaient été réalisées, où seule la boucle de crible/factorisation avait été parallélisée. Nous étudions chaque étape et nous l'accélérons étape par étape (chapitre 5). C'est ainsi que notre implantation permet d'atteindre des accélérations superlinéaires, car l'analyse nous a conduits à un algorithme dont la partie la plus coûteuse en nombre d'opérations (et en temps d'exécution) se déroule sur une ferme de processeurs. Il n'y a pas de communications entre les processeurs, donc pas de perte de temps en synchronisation. Les autres parties de l'algorithme ont été adaptées de manière à générer le plus petit nombre de communications.

Nous avons analysé ensuite les résultats de l'implantation sur le FPS T40, et nous en avons déduit une nouvelle stratégie de répartition des données, qui permet d'obtenir une charge de travail très proche de 1. Cette stratégie nous a permis de factoriser des nombres de 60 chiffres.

Cette stratégie de répartition des données, distribution des polynômes par familles, permet d'éviter des calculs redondants, car les processeurs ont des données toujours différentes. Nous évitons un écueil rencontré par Caron et Silverman dans leur implantation [CaS 88].

L'implantation sur le FPS T40 de l'algorithme de base du MPQS donne des résultats encourageants. Des améliorations mathématiques et algorithmiques sont possibles. Et dans les futures implantations sur d'autres machines, nous ne manquerons pas de les inclure, afin de repousser encore les limites de l'exécution du crible quadratique sur des machines multiprocesseurs.

Nous avons apporté quelques réponses à quelques problèmes, mais chacune d'elles a soulevé de nouvelles questions. L'algorithmique parallèle est en pleine croissance. Les machines commerciales commencent à apparaître sur le marché. Il faut répondre aux utilisateurs qui ont besoin de méthodologies et d'outils afin d'exploiter la puissance de ces machines. Gelernter [Gel 86] se demande si nous pourrons répondre à ces besoins. Mais nous sommes résolument optimistes.

Bibliographie

[AIR 88] **M. Adam, P. Ingels, M. Raynal,**
"Algorithmes distribués synchrones et systèmes répartis asynchrones : concepts, mises en œuvre et expérimentations",
Rapport de Recherche, IRISA, Université de Rennes, juin 1988.

[AnJ 87] **F. André, A. Joubert,**
"SIGLE : un outil d'aide à la mise en oeuvre d'algorithmes sur les architectures réparties",
Actes du 2ème coloque C^3, Angoulême, 20-22 mai 1987, A. Arnold, éd, pp. 1-8.

[Apt 84] **K.R. Apt,**
"Correctness proofs of distributed algorithms",
Rapport de Recherche 84-51, LITP, Paris 7, septembre 1984.

[Apt 86] **K.R. Apt,**
"Correctness proofs of distributed algorithms",
ACM Trans. Program. Lang. Syst., Vol. 8, n° 3, pp. 388-405.

[ARG 86] **R.K. Arora, S.P. Rana, M.N. Gupta,**
"Distributed termination detection algorithm for distributed computations",
IPL, Vol. 22, 1986, pp. 311-314.

[BaH 77] **C. Bays, R.H. Hudson,**
"The segmented sieve of Eratosthenes and primes in arithmetic progressions to 10^{12}",
Bit, 17, 1977, pp. 121-127.

[BaW 86] **M.R. Barbacci, J.M. Wing,**
"Durra: a task-level description language",
Reference Manual V. 0.1, 15 novembre 1986, Carnegie Mellon University.

[Ben 86] **S.A. Bengelloun,**
"An incremental primal sieve",
Acta Informatica, 23, 1986, pp. 119-125.

[Bla 88] **P. Blanc,**
"Distributed termination detection when messages arrive out of sequence",
IFIP WG 10.3 Working Conference on Parallel Processing, 25-27 avril 1988, Pise, Italie.

[BLS 83] **J. Brillhart, D.H. Lehmer, J.L. Selfridge, B. Tuckerman, S.S. Wagstaff,**
"Factorizations of $b^n \pm 1$, b=2,3,5,6,7,10,11,12 up to high powers",
Contemp. Math., Vol. 22, Amer. Math. Soc., Providence, R.I., 1983.

[BMS 88] **J. Brillhart, P.L. Montgomery, R.D. Silverman,**
"Tables of Fibonacci and Lucas factorizations",
Math. of Comp., Vol. 50, n° 181, janvier 1988, pp. 251-260.

[Bok 87] **S. H. Bokhari,**
"Multiprocessing the sieve of Eratosthenes",
IEEE Computer, Vol. 20, n° 4, Avril 1987, pp. 50-58.

[BTV 90] **J.Y. Blanc, D. Trystam, G. Villard,**
"Desynchronized communication schemes for distributed memory architectures",
DMCC5, Columbia, 8-12 avril 1990, IEEE Computer Society Press, Los Alamitos, à paraître.

[Bue 87] **D.A. Buell,**
"Factoring : algorithms, computations and computers",
J. of Supercomputing, 1, 1987, pp. 191-216.

[CaS 88] **T.R. Caron, R.D. Silverman,**
"Parallel implementation of the quadratic sieve",
The Journal of Supercomputing, 1, 1988, pp. 273-290.

[CCD 88] **D.V. Chudnovsky, G.V. Chudnovsky, M.M. Denneau, S.G. Younis,**
"A design of general purpose number-theoretic computer",
ICS, Vol. II, 1988, pp. 498-499.

[CCG 88] **E.J. Cameron, D.M. Cohen, B. Gopinath, W.M. Keese, L. Ness, P. Uppaluru, J.R. Vollaro,**
"The IC* model of parallel computation and programming environment",
IEEE Trans. on Software Engineering, Vol. 14, n° 3, mars 1988, pp. 317-326.

[ChL 85] **K.M. Chandy, L. Lamport,**
"Distributed snapshots: determining global states of distributed systems",
ACM Trans. Comp. Syst., Vol. 3, n° 1, février 1985, pp. 63-75.

[ChM 85] **K.M. Chandy, J. Misra,**
"A paradigm for detecting quiescent properties in distributed computations",
In Apt K.R. (ed), Logics and models of concurrent systems, Springer, Berlin, 1985, pp. 325-341.

[ChM 88] **K.M. Chandy, J. Misra,**
"Parallel program design. A foundation",
Addison-Wesley Publishing Company, 1988, 516 pages.

[CoP 89] **M. Cosnard, J.L. Philippe,**
Discovering new parallel algorithms: the sieve of Eratosthenes revisited",
CAP, juillet 1988, Grenoble, Academic Press 1989, pp. 1-18.

[CoP 90] **M. Cosnard, J.L. Philippe,**
"The quadratic sieve factoring alogorithm on distributed memory multiprocessors",
DMCC5, Columbia, 8-12 avril 1990, IEEE Computer Society Press, Los Alamitos, à paraître.

[COS 86] **D. Coppersmith, A.M. Odlyzko, R. Schroeppel,**
"Discrete logarithms in GF(p)",
Algorithmica, 1, 1986, pp. 1-15.

[CRB 85] **D.M. Chiarulli, W.G. Rudd, D.A. Buell,**
"DRAFT - a dynamically reconfigurable processor for integer arithmetic",
Proceedings of the 7th international symposium on computer arithmetic,
Urbano, IL., 1985, pp. 307-317.

[CRT 88] **M. Cosnard, Y. Robert, B. Tourancheau, G. Villard,**
"Speedup and data allocation strategies on distributed memory architectures",
Rapport Technique 40, TIM3, IMAG, Grenoble, septembre 1988.

[CTV 87a] **M. Cosnard, B. Tourancheau, G. Villard,**
"Gaussian elimination on message passing architectures",
Proceedings of ICS 1987, Athènes, 1987, Springer Verlag.

[CTV 87b] **M. Cosnard, B. Tourancheau, G. Villard,**
"Présentation de l'hypercube T20 de FPS",
Bigre+Globule, 56, 1987, pp. 12-17.

[DaH 84] **J.A. Davis, D.B. Holdridge,**
"Most wanted factorizations using the quadratic sieve",
Sandia Report SAND 84-1658, 1984.

[DaH 88] **J.A. Davis, D.B. Holdridge,**
"Factorization of large integers on a massively parallel computer",
"Eurocrypt '88 Abstracts, a workshop on the theory and application of cryptographic techniques", IACR, 1988.

[Dew 88] **A. Dewdney,**
"Les orpailleurs de nombres",
Pour la Science, n° 131, septembre 1988, pp. 88-94.

[DFG 83] **E.W. Dijkstra, W.H.J. Feijen, A.J.M. van Gasteren,**
"Derivation of a termination detection algorithm for distributed computation",
IPL, Vol 16, 1983, pp. 217-219.

[DHS 85] **J.A. Davis, D.B. Holdridge, G.J. Simmons,**
"Status report on factoring",
In Advances in Cryptology, Lecture Notes in Computer Science, Vol. 209, 1985, pp. 183-215.

[Dij 72] **E. W. Dijkstra,**
"Notes on structured programming",
In "Structured programming", E.W. Dijkstra et C.A.R. Hoare eds, Academic Press, New York, 1972, pp. 1-82.

[DiS 80] **E.W. Dijkstra, C.S. Scholten,**
"Termination detection for diffusing computations",
IPL, Vol. 11, n° 1, août 1980, pp. 1-4.

[Eri 88] **O. Eriksen,**
"A termination detection protocol and its formal verification",
Journal of Parallel and Distributed Computing, Vol. 5, 1988, pp. 82-91.

[FeG 89] **A.G. Ferreira, M. Gastaldo,**
"Expérimentations de deux algorithmes de tri sur l'hypercube T20 de FPS",
Rapport de Recherche 89-11, LIP-IMAG, ENS Lyon, septembre 1989, 20 pages.

[FeR 87] **D. Ferment, B. Rozoy,**
"Solutions for the disitributed termination problem",
in Allbrecht A., Jung H., Mehlhorn K. (eds), "Parallel algorithms and architectures", Springer, LNCS, Vol. 269, 1987.

[Fra 80] **N. Francez,**
"Distributed termination",
ACM Trans. on Progr. Lang. and Syst., Vol. 2, n° 1, janvier 1980, pp. 42-55.

[Fra 89] **P. Fraigniaud,**
"Performance analysis of broadcasting in hypercubes",
in Hypercube & Distributed Computers, F. André et J.P. Verjus (eds), Elsevier Science Publishers, B.V. (North-Holland), 1989, pp. 311-327.

[GaS 87] **K. Gates, D. Socha,**
"Programming NCUBEs with a graphical parallel programming environment versus an extended sequential language",
Hypercube Multiprocessors, 1987, M.T. Heath, (ed), Philadelphie, pp. 17-27.

[Gel 86] **D. Gelernter,**
"Domesticating parallelism",
IEEE Computer, août 1986, pp. 12-16.

[Ger 83] **J. Gerver,**
"Factoring large numbers with a quadratic sieve",
Math. Comp., Vol. 41, 1983, pp. 287-294.

[GiS 88] **A. Giacalone, S.A. Smolka,**
"Integrated environments for formally well-founded design and simulation of concurrent systems",
IEEE Trans. on Software Engineering, Vol. 14, n° 6, juin 1988, pp. 787-802.

[Gor 85] **J. Gordon,**
"Strong primes are easy to find",
LNCS, Vol. 209, Springer Verlag, 1985, pp. 216-233.

[GrM 78] **D. Gries, J. Misra,**
"A linear sieve algorithm for finding prime numbers",
Comm of the ACM, décembre 1978, Vol. 21, n° 12, pp. 999-1003.

[Gus 88] **J.L. Gustafson,**
"Reevaluating Amdahl's law",
Comm. of the ACM, Vol. 31, n° 5, 1988, pp. 532-533.

[Haa 87] **A. Haas,**
"The multiple prime random number generator",
ACM Transactions on Mathematical Software, Vol. 13, n° 4, décembre 1987, pp. 368-381.

[HaS 88] **S. Haldar, D.K. Subramanian,**
"Ring based termination detection algorithm for dsitributed computations",
IPL, Vol. 29, 1988, pp. 149-153.

[HaW 79] **G.H. Hardy, E.M. Wright,**
"An introduction to the theory of numbers",
5ème édition, Oxford University Press, 1979, pp. 73-79.

[HaZ 87] **C. Hazari, H. Zedan,**
"A distributed algorithm for distributed termination",
IPL, Vol. 24, 1987, pp. 293-297.

[HeM 88] **E.C.R. Hehner, A.J. Malton,**
"Termination conventions and comparative semantics",
Acta Informatica, Vol. 25, 1988, pp. 1-14.

[Her 86] **A. Herscovici,**
"Introduction aux grands ordinateurs scientifiques",
Eyrolles, Paris, 1986, 169 pages.

[HeR 88] **J.M. Hélary, M. Raynal,**
"Construction méthodique d'un algorithme réparti de détection de la terminaison",
Rapport de Recherche 440, décembre 1988, IRISA, Univ. Rennes.

[HiK 80] **T. Hikita, S. Kawai,**
"Parallel sieve methods for generating prime numbers",
Information Processing 80, S.H. Lavington (ed.), North Holland, IFIP, 1980, pp. 257-262.

[HJP 87] **J.M. Hélary, C. Jard, N. Plouzeau, M. Raynal,**
"Detection of stable properties in distributed applications",
Proc. 6th ACM-IEEE Symposium on Principles of Distributed Computing, Vancouver, août 1987.

[Hoa 72] **C.A.R. Hoare,**
"Notes on data structuring",
In "Structured programming", E.W. Dijkstra et C.A.R. Hoare (eds), Academic Press, New York, 1972, pp. 83-174.

[Hoa 78] **C.A.R. Hoare,**
"Communicating Sequential Processes",
Comm. ACM, Vol 21, n° 8, août 1978, pp. 666-677.

[Hoa 88] **C.A.R. Hoare,**
"Occam 2 Reference Manual",
Prentice Hall International Series in Computer Science, Cambridge, 1988, 133 pages.

[HoJ 89] **C.T. Ho, S.L. Johnsson,**
"Distributed routing algorithms for broadcasting and personalized communication in hypercubes",
IEEE Trans. on Computers, Vol. 38, n° 9, 1989.

[Hua 88] **S.T. Huang,**
"A fully distributed termination detection scheme",
IPL, Vol. 29, 1988, pp. 13-18.

[Hua 89] **S.T. Huang,**
"Termination detection by using distributed snapshots",
IPL, Vol. 32, 1989, pp. 113-119.

[Hwa 87] **K. Hwang,**
"Advanced parallel processing with supercomputer architectures",
Proceedings of the IEEE, Vol. 75, n° 10, octobre 1987, pp. 1348-1379.

[HwB 84] **K. Hwang, F.A. Briggs,**
"Computer architecture and parallel processing",
Mac Graw Hill Book Company, 1984, 846 pages.

[JaJ 88] **C. Jard, J.M. Jézéquel,**
"Un compilateur Estelle multiprocesseurs pour l'expérimentation d'algorithmes sur systèmes distribués",
Rapport Technique, IRISA, Université de Rennes, novembre 1988.

[JMG 88] **C. Jard, J.F. Monin, R. Groz,**
"Development of Véda: a prototyping tool for distributed algorithms",
IEEE Trans. on Software Engineering, Vol. 14, n° 3, mars 1988, pp. 339-352.

[KnT 76] **D.E. Knuth, L. Trabb Pardo,**
"Analysis of a simple factorization algorithm",
Theoret. Comput. Sci., 3, 1976, pp. 321-348.

[Knu 81] **D.E. Knuth,**
"The art of computer programming",
Volume 2, 2ème édition, Addison Wesley, 1981, p. 626.

[KoT 88] **R. Koo, S. Toueg,**
"Effects of message loss on the termination of distributed protocols",
IPL, Vol. 27, 1988, pp. 181-188.

[Kra 26] **M. Kraïtchik,**
"Théorie des nombres - Tome II",
Gauthier-Vilars, Paris, 1926.

[KrL 88] **B. Kruatrachue, T. Lewis,**
"Grain-size determination for parallel processing",
IEEE Software, janvier 1988, pp. 23-32.

[Lam 78] **L. Lamport,**
"Time, clocks and the ordering of events in a distributed system",
Comm. ACM, Vol. 21, n° 7, 1978, pp. 558-565.

[Lec 85] **P. Le Certen,**
"LC^3, un langage de programmation parallèle",
Actes du 1er coloque C^3, Angoulême, 16-18 septembre 1985, A. Arnold (ed).

[Lee 89] **M.A. Leeder,**
"The Queen's tree Machine : an experimental parallel computer",
Master thesis, Queen's University, Kingston, Ontario, juin 1989, pp. 95-100.

[LeM 89] **A.K. Lenstra, M.S. Manasse,**
"Factoring by electronic mail",
Proceedings Eurocrypt '89.

[Len 87] **H.W. Lenstra,**
"Factoring integers with elliptic curves",
Annals of Mathematics, 126, 1987, pp. 649-673.

[LeP 86] **S.T. Levi, B.D. Plateau,**
"A distributed algorithm for deadlock and termination detection of distributed computations",
Rapport de Recherche 1750, Univ. Maryland, décembre 1986.

[LLM 90] **A.K. Lenstra, H.W. Lenstra, M.S. Manasse, J.M. Pollard,**
"The number field sieve",
Communication personnelle de F. Morain.

[LMO 85] **J.C. Lagarias, V.S. Miller, A.M. Odlyzko,**
"Computing $\pi(x)$: the Meissel-Lehmer method",
Mathematics of Computation, Vol. 44, n° 170, avril 1985, pp. 537-560.

[Mai 77] **H. G. Mairson,**
"Some new upper bounds on the generation of prime numbers",
Comm. of the ACM, septembre 1977, Vol. 20, n° 9, pp. 664-669.

[Mat 85] **N. Matelan,**
"The Flex/32 multicomputer",
Proc. 12th Int'l Symp. Computer Architecture, 17-19 juin 1985, Los Alamitos, Computer Society Press, 1985, pp. 209-213.

[Mat 87] **F. Mattern,**
"Algorithms for distributed termination detection",
Distributed Computing, Vol. 2, 1987, pp. 161-175.

[Mat 89] **F Mattern,**
"An efficient distributed termination test",
IPL, Vol. 31, 1989, pp. 203-208.

[Mis 83] **J. Misra,**
"Detecting termination of distributed computation using markers",
Proc. of the 2nd annual ACM Symposium on Principles of Distributed Computing, Montreal, août 1983, pp. 290-294.

[MiV 88] **D. Millot, J. Vautherin,**
"Dynamic creation of processes on a Transputer network",
Rapport de Recherche 433, juillet 1988, LRI, Université de Paris Sud.

[MoB 75] **M.A. Morrison, J. Brillhart,**
"A method of factoring and the factorization of F_7",
Math. Comput., 29, 1975, pp. 183-205.

[Mor 87] **F. Morain,**
"Implémentation du crible quadratique",
Rapport, 7 avril 1987, Limoges.

[Mor 88] **F. Morain,**
"La factorisation de F_{11} est achevée",
Pour la Science, n° 132, octobre 1988, p. 17.

[Mor 90] **F. Morain,**
Communication personnelle, 1990.

[Nic 84] **J.L. Nicolas,**
"Sur la distribution des nombres entiers ayant une quantité fixée de facteurs premiers",
Acta Arithmetica, XLIV, 1984, pp. 191-200.

[Pat 82] **D.A. Patterson,**
"A performance evaluation of the Intel 80286"
Computer Architecture News, (ACM SIGArch Newsletter), Vol. 10, n° 5, 1982, pp. 16-18.

[PaW 84] **D. Parkinson, M. Wunderlich,**
"A compact algorithm for gaussian elimination over GF(2) implemented on highly parallel computers",
Parallel Computing, 1984, pp. 65-73.

[Pép 1877] **P. Pépin,**
"Sur la formule $2^{2^n} + 1$",
CRAS, Paris, v. 85, 1877, pp. 329-331.

[Phi 90] **J.L. Philippe,**
"Superlinear speedups for parallel factorization of large integers",
International Conference on Parallel Computing: Achievements, Problems and Prospects, 3-7 juin 1990, Capri, Italie, à paraître.

[Pom 82] **C. Pomerance,**
"Analysis and comparison of some integer factoring algorithms",
In Computational Methods in Number Theory, Lenstra et Tijdeman (eds), Math. Centrum Tract 154, 1982, pp. 89-139.

[Pom 85] **C. Pomerance,**
"The quadratic sieve factoring algorithm",
In Advances in Cryptology, Lectures Notes in Computer Science, 209, 1985, pp. 169-182.

[PoM 89] **R. Pozo, A.E. MacDonald,**
"Performance characteristics of scientific computation on the Connection Machine 2",
Rapport Technique, University of Colorado at Boulder, juin 1989.

[PoW 83] **C. Pomerance, S.S. Wagstaff,**
"Implementation of the continued fraction integer factoring algorithm",
Congressus Numerantium, 37, 1983, pp. 99-118.

[PRG 88] **J.M. Purtilo, D.A. Reed, D.C. Grunwald,**
"Environments for prototyping parallel algorithms",
Journal of Parallel and Distributed Computing, Vol. 5, 1988, pp. 421-437.

[Pri 79] **P. Pritchard,**
"On the prime example of programming",
Proceedings of the Symposium on Language Design and Programming Methodology, Sidney, 10-11 septembre 1979, pp. 85-94.

[Pri 81] **P. Pritchard,**
"A sublinear additive sieve for finding prime numbers",
Comm. of the ACM, janvier 1981, Vol. 24, n° 1, pp. 18-23.

[Pri 82] **P. Pritchard,**
"Explaining the wheel sieve",
Acta Informatica, 17, 1982, pp. 477-485.

[Pri 83] **P. Pritchard,**
"Fast compact prime number sieves (among others)",
Journal of Algorithms, 4, 1983, pp. 332-344.

[Pri 87] **P. Pritchard,**
"Linear prime-number sieves : a family tree",
Science of Computer Programming, North Holland, 9, 1987, pp. 17-35.

[PST 88] **C. Pomerance, J.W. Smith, R. Tuler,**
"A pipeline architecture for factoring large integers with the quadratic sieve algorithm",
SIAM J. Comput., Vol. 17, n° 2, avril 1988, pp. 387-403.

[Qui 88] **M.J. Quinn,**
"Designing efficient algorithms for parallel computers",
Mc Graw Hill International Editions, Singapour, 2e édition, 1988, 288 pages.

[Ran 83] **S.P. Rana,**
"A distributed solution of the distributed termination problem",
IPL, Vol. 17, 1983, pp. 43-46.

[Ray 85] **M. Raynal,**
"Algorithmes distribués et protocoles",
Eyrolles, Paris, 1985, pp. 71-94.

[Ray 89] **M. Raynal,**
"Prime numbers as a tool to design distributed algorithms",
IRISA, Rennes, P.I. 458, Fév. 1989, 11 pages.

[Rie 87] **H. Riesel,**
"Prime numbers and computer methods for factorization",
Progress in Mathematics, Vol. 57, J. Coates and S. Helgason eds, Birkäuser, 2ème édition, 1987.

[Rie 88] **H.J.J. te Riele,**
"Optimization of the MPQS factoring algorithm on the Cyber 205 and the NEC SX 2",
Supercomputer, juillet 1988, pp. 42-50.

[RLW 88] **H.J.J. te Riele, W.M. Lioen, D.T. Winter,**
"Factoring with the quadratic sieve on large vector computers",
Rapport de Recherche NM-R8805, Centrum voor Wiskunde in Informatica, juin 1988, Amsterdam.

[Ros 84] **K.H. Rosen,**
"Elementary number theory and its applications",
Addison-Wesley, 1984, p. 158.

[RSA 78] **R. Rivest, A. Shamir, L. Adleman,**
"A method for obtaining digital signatures and public-key cryptosystems",
Comm. ACM, 21, 2, février 1978, pp. 120-126.

[Saa 85] **Y. Saad,**
"Communication complexity of the Gaussian elimination algorithm on multiprocessors",
Rapport de Recherche 348, Comp. Sc. Dep., Yale University, 1985.

[Sal 87] **J. Salmon,**
"CUBIX: programming hypercubes without programming hosts",
Hypercube Multiprocessors, 1987, M.T. Heath (ed), Philadelphie, pp. 3-9.

[SBB 87] **L.R. Scott, J.M. Boyle, B. Bagheri,**
"Distributed data structures for scientific computations",
Hypercube Multiprocessors, 1987, M.T. Heath (ed), Philadelphie, pp. 55-66.

[Sch 86] **M.R. Schroeder,**
"Number theory in science and communication",
Springer series in information sciences, Springer Verlag, Second Edition, 1986, 374 pages, pp. 116-117.

[ScL 84] **C.P. Schnorr, H.W. Lenstra,**
"A Monte-Carlo factoring algorithm with linear storage",
Mathematics of Computation, Vol. 43, n° 167, juillet 1984, pp. 289-311.

[Sha 71] **D. Shanks,**
"Class number, a theory of factorization and genera.",
Proc. Sympos. Pure Math., Vol. 20, Amer. Math. Soc., Providence, R.I., 1971, pp. 415-440.

[Sha 72] **D. Shanks,**
"Five number-theoretic algorithms",
Proc. Second Manitoba Conference on Numerical Math., 1972, pp. 51-70.

[Sil 87] **R.D. Silverman,**
"The multiple polynomial quadratic sieve",
Mathematics of Computation, Vol. 48, n° 177, janvier 1987, pp. 329-339.

[Sil 88] **R.D. Silverman,**
"Factoring large integers in parallel",
ICS, Vol. 2, 1988, pp. 488-497.

[SmW 83] **J.W. Smith, S.S. Wagstaff,**
"An extended precision operand computer",
Proceedings of the 21st Southeast Region ACM Conference, 1983, pp. 209-216.

[SRV 88] **K. Schwan, R. Ramnath, S. Vasudevan,**
"A language and system for the construction and tuning of parallel programs",
IEEE Trans. on Software Engineering, Vol. 14, n° 4, avril 1988, pp. 455-471.

[TeM 89] **G. Tel, F. Mattern,**
"Comments on "Ring based termination detection algorithm for distributed computations"",
IPL, Vol. 31, 1989, pp. 127-128.

[TeL 87] **G. Tel, J. van Leuwen,**
"Comments on "A distributed algorithm for distributed termination"",
IPL, Vol. 25, 1987, p. 349.

[Top 84] **R.W. Topor,**
"Termination detection for distributed computations",
IPL, Vol. 18, 1984, pp. 33-36.

[TYN 83] **Y. Takahashi, Y. Yamane, K. Nishiyama, F. Yoshitani, K. Inoue**
"Efficiency of parallel computation on the binary-tree machine CORAL '83",
Journal of Information Processing, Vol. 8, n° 4, 1985, pp. 288-299.

[Ver 87] **J.P. Verjus,**
"On the proof of a distributed algorithm",
IPL, Vol. 25, 1987, pp. 145-147.

[Vil 88] **G. Villard,**
"Calcul formel et parallélisme. Résolution de systèmes linéaires",
Thèse de doctorat INPG, Grenoble, 1988.

[Vin 88] **P. Vincent,**
"Nouvelles architectures d'ordinateurs",
Editests, Groupe de la Cité, Paris, 1988, 336 pages.

[Wie 86] **D.H. Wiedeman,**
"Solving sparse linear equations over finite fields",
IEEE Trans. Inform. Theory, IT-32, 1986, pp. 54-62.

[Wir 73] **N. Wirth,**
"Systematic programming : an introduction",
Prentice Hall, Englewood Cliffs, New Jersey, 1973.

[Wun 84] **M.C. Wunderlich,**
"Factoring numbers on the Massively Parallel Processor",
In Advances in Cryptology, David Chaum (ed), Plenum Press, New York, 1984, pp. 87-102.

[Wun 85] **M.C. Wunderlich,**
"Implementing the continued fraction factoring algorithm on parallel machines",
Math. of Comput., 44, 1985, pp. 251-260.

[WuW 87] **M.C. Wunderlich, H.C. Williams,**
"A parallel version of the continued fraction integer factoring algorithm",
J. of Supercomputing, 1, 1987, pp. 217-230.

[YoB 88] **J. Young, D.A. Buell,**
"The twentieth Fermat number is composite",
Math. of Computation, Vol. 50, n° 181, Janv. 1988, pp. 261-263.

département des études doctorales

A U T O R I S A T I O N de S O U T E N A N C E

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

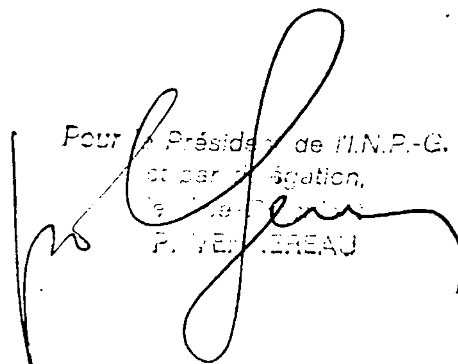
- **Monsieur Jean-Louis NICOLAS**
- **Monsieur Yves ROBERT**

Monsieur PHILIPPE Jean-Laurent

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité

"Informatique"

Fait à Grenoble, le **01 Juin 1990**



Abstract

This thesis consists in two parts: developments related to prime numbers generation, and implementation of the quadratic sieve.

In the first part, we analyze data allocation strategies on the processors for the sieve of Eratosthenes in a shared-memory environment, in order to improve the workload balance. Then, we propose implementations on the FPS T40 distributed-memory hypercube. But the sieve of Eratosthenes is a Master/Slaves algorithm. It is not adapted to a distributed architecture. Hence we study a new algorithm for generating primes through sequential divisions on a ring. This algorithm requires a technique for detecting its distributed termination. This is done by counting the processors having terminated their computation. Finally, from this study we derive methodologies for implementing Master/Slaves applications on a linear array and a grid of processors.

The second part is devoted to the multiple polynomial quadratic sieve, an algorithm for factoring large integers, which is used for cryptography. Our goal is to extract the parallelism from each step of this algorithm in a distributed environment. We want a better use of the available power of massively parallel computers. This study leads to an efficient implementation on the FPS T40 hypercube.

Key-words : distributed programmation, massively parallel machines, cryptography, integer factorization, hypercube.

Résumé

Cette thèse est composée de deux parties : les développements liés à la génération des nombres premiers et l'implantation du crible quadratique.

Dans la première partie, nous analysons les stratégies d'allocation des données aux processeurs pour le crible d'Eratosthène dans un environnement à mémoire partagée en vue d'améliorer l'équilibrage de la charge de travail. Puis, nous proposons des implantations sur l'hypercube FPS T40 à mémoire distribuée. Comme le caractère centralisé du crible d'Eratosthène (de type Maître/Esclaves) s'accommode mal des exigences de l'architecture distribuée, nous étudions un algorithme de génération des nombres premiers par divisions successives sur un anneau. Cet algorithme nécessite la mise en œuvre d'une technique de détection de la terminaison distribuée, par un dénombrement des processeurs ayant terminé l'exécution de leur programme. Enfin, l'aspect Maître/Esclaves du crible d'Eratosthène permet l'étude de méthodologies d'implantation de ce type d'algorithmes sur un réseau linéaire et une grille de processeurs.

La deuxième partie est consacrée au crible quadratique multipolynomial, algorithme de factorisation des grands entiers, utilisé en cryptographie. Notre but est d'extraire le maximum de parallélisme de chacune des étapes de cet algorithme dans un environnement distribué, afin d'utiliser au mieux la puissance des calculateurs massivement parallèles. Cette étude conduit à une implantation efficace sur l'hypercube FPS T40.

Mots-clés : programmation distribuée, machines massivement parallèles, cryptographie, factorisation d'entiers, hypercube.