Calcul formel et parallélisme : l'architecture du système PAC et son arithmétique rationnelle

Jean-Louis Roch

THESE

Présentée par Jean-Louis ROCH

pour obtenir le titre de **DOCTEUR**

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

(Spécialité: **Mathématiques Appliquées**)

Calcul Formel et Parallélisme

L'ARCHITECTURE DU SYSTEME PAC ET SON ARITHMETIQUE RATIONNELLE

Date de soutenance: 5 décembre 1989

Composition du jury:

Président:	Jean-Pierre VERJUS
Rapporteurs:	Daniel LAZARD
	Jean-Michel MULLER
Examinateurs:	Michel COSNARD
	Jean DELLA DORA
	Philippe FLAJOLET

Thèse préparée au sein du laboratoire TIM3, Unité Associée au CNRS n°397

Calcul Formel et Parallélisme

L'Architecture du Systeme

PAC

et son Arithmetique Rationnelle

A mes Parents,

A Françoise,

Et à toute notre famille !

Mon plus grand bonheur vient de la chaleur de notre famille. Au début de cette thèse, j'en avais une. Maintenant, nous en avons une encore plus grande.

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

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

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

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

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

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

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

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

E.N.S.H.G
ROWE Alain

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

E.F.P.
CHARUEL Robert

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

Laboratoires extérieurs

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

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.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD

Année 1988

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

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

'Don Quichotte sollicita un laboureur son voisin, homme de bien (...), mais qui avait fort peu de plomb en sa caboche. Il lui disait entre autres choses qu'il lui pourrait quelquefois arriver telle aventure, qu'il gagnerait, en moins d'un tour de main, quelque île, et qu'il l'en ferait gouverneur. Par ses promesses et autres semblables (...), Sancho Pança (ainsi s'appelait le laboureur) se mit pour écuyer de son voisin.'

Quelque temps plus tard...

'- Dieu veuille qu'il nous en tourne à bien, dit Sancho, et que le temps vienne bientôt de gagner cette île qui me coûte si cher, et que je meure ensuite.
- Je t'ai déjà dit, Sancho, que cela ne te donne aucun souci : car quand il manquerait une île, voilà le royaume du Danemark ou bien celui de Sobradise qui te viendront comme bague au doigt, et tu dois d'autant plus t'en réjouir qu'ils sont en terre ferme.'

"L'Ingénieux Hidalgo Don Quichotte de la Manche ", Cervantes

'S'il est vrai que 2 et 2 font de leur mieux pour faire 4, il n'en est pas moins vrai que 18 et 20 font tout ce qu'ils peuvent pour ne pas faire 39,95.'

"Pensées", Pierre Dac

'It makes me nervous to fly on airplanes since I know they are designed using floating point arithmetic.'

Anston Householder

REMERCIEMENTS

Je tiens à remercier Jean-Pierre Verjus, qui a bien voulu me faire l'honneur de présider le jury de cette thèse.

Je remercie tout particulièrement Jean-Michel Muller et Daniel Lazard d'avoir accepté d'être rapporteurs. Leurs conseils m'ont été d'une grande utilité.

Je remercie aussi Michel Cosnard, professeur à l'ENS de Lyon, et Philippe Flajolet, directeur de recherche à l'INRIA, qui ont accepté de participer à ce jury. L'importance de Michel Cosnard dans l'initiation du projet Machines Parallèles est fondamentale.

Je dois tout à Jean Della Dora, cet entremetteur de génie, responsable de la moralité de l'équipe, qui a su prendre en thèse deux étudiants faits l'un pour l'autre et qui ne se connaissaient pas encore... Mais aussi, j'apprécie toujours plus les discussions passionnées, les conseils essentiels, les idées intéressantes... et les gâteaux au chocolat (moins bons il est vrai que ceux de la maman de Françoise).

Si ce travail m'a beaucoup intéressé, c'est aussi parce que nous l'avons réalisé en équipe avec Françoise, Pascale Sénéchaud et Gilles Villard. Cette thèse est indissociable des leurs, et je tiens à les remercier tous pour leur sympathique collaboration, mais aussi pour les nombreuses erreurs qu'ils m'ont permis de corriger.

Travailler au sein d'une équipe est essentiel. Grâce à Bertrand Braschi, PAC est maintenant une vraie librairie. Sans les compétences de Bernard Tourancheau et Gilles Villard, la vectorisation des algorithmes n'aurait pu être menée à bien. L'étude des algorithmes d'arithmétique doit beaucoup à Jean-Michel Muller, qui m'a communiqué de nombreuses références, et dont les différentes corrections et suggestions ont été primordiales.

Jacque Ballasi et Denis Berthet ont développé, avec beaucoup de passion, la plupart des algorithmes polynomiaux, et la première partie de cette thèse leur doit beaucoup. Tout particulièrement Françoise, sans qui l'arithmétique polynomiale n'aurait pu être réalisée, mais aussi Line Perret et Nathalie Revol ont montré, avec beaucoup de courage, car il en fallait (!), que PAC était utilisable, tout en signalant les corrections à effectuer. J'ai particulièrement apprécié leur compagnie et je les remercie tous vivement de leur travail, essentiel dans l'élaboration et la réalisation du système.

Je n'oublie pas les conseils lors des traductions et des relectures de Brigitte Plateau et Denis Trystram, ni l'aide de Claire Dicrescenzo et Joëlle Prévost qui répondent toujours avec beaucoup de patience à mes questions, ni les discussions avec Dominique Duval, Yvon Siret et Evelyne Tournier qui m'ont appris beaucoup.

L'ambiance sympathique qui règne dans l'équipe doit un peu à chacun, et permet une collaboration entre tous : je remercie tous ceux que je n'ai pas encore cités, et dont j'ai tout particulièrement apprécié la compagnie : Guoting, Aziz, Abdel, Afonso, Mustapha, Yvan, Jean-Laurent, Adberezack, Karim, Jean-Yves et les Autres (comme dirait Truffaut). Sans oublier, bien sûr, la lointaine Christine Bernier, retranchée dans sa tour carrée, qui, outre les discussions sans fin entre numérique et formel, m'a fait découvrir les crêpes bretonnes (pas cet ersatz Grenoblois...) et le charme du vin d'Anjou familial.

Ce travail n'est certainement pas étranger à mes activités d'enseignement à l'Ensimag, qui m'apportent beaucoup. Je remercie tout spécialement Pierre Barras, François Robert et Patrick Witomski pour leur soutien en algorithmique numérique. J'apprends toujours beaucoup grâce au projet de compilation, et j'en remercie Marie-Laure Pottet, Jean-Ronan Vigauroux, Daniel Pilaud, Paul Jacquet, Florence Maraninchi et Roger Mohr.

Enfin, je remercie tous les membres du service de scolarité de l'INPG et du service de reprographie de l'IMAG pour la réalisation finale de cette thèse.

SOMMAIRE GÉNÉRAL

INTRODUCTION

Qu'est ce que le parallélisme ? Plus qu'un concept, c'est aujourd'hui une réalité : le nombre sans cesse grandissant d'architectures parallèles est là pour en témoigner.
Depuis longtemps les numériciens conçoivent et utilisent des algorithmes parallèles, alors que l'utilisation du parallélisme en calcul formel est relativement récente. Or, si dans les deux cas, la motivation est de permettre une puissance de calcul toujours plus grande, sans faire de clivage abusif, les différences profondes qui existent entre un programme purement numérique (besoin en calcul flottant) et un autre purement symbolique (besoin en calcul logique et en représentation d'objets) suggèrent différentes approches.

Dans tout système de calcul formel, il existe une dualité intrinsèque : d'une part un système comprend une partie informatique, qui permet la manipulation de symboles, l'utilisation de règles de simplification... D'autre part, il contient de nombreux outils mathématiques, dont une implantation naïve conduit souvent à des complexités trop importantes, qui en interdisent l'utilisation pour la résolution de problèmes pratiques.

Les systèmes classiques de calcul formel -Reduce, Macsyma, Maple, Scratchpad...- ne visent pas tant la puissance de calcul que l'aspect calculette mathématique sophistiquée.
Un système classique satisfait donc les besoins en manipulations symboliques de base (calcul de primitives, décompositions en éléments simples, géométrie algorithmique...). L'aspect prépondérant est celui d'une bibliothèque mathématique symbolique.
L'impossibilité sous Macsyma de calculer la forme d'Hermite d'une matrice carrée 6x6, à coefficients polynomiaux de degré trois, est significative.

La motivation principale de PAC (*Parallel Algebraic Computing*) est de montrer que, grâce au parallélisme, le Calcul Formel est un outil efficace pour le traitement de certains problèmes de l'ingénieur [DEL88]: formes normales de matrices denses à coefficients entiers ou polynômiaux (forme normale d'Hermite de matrices 160x160 à coefficients entiers [SIE89]),

systèmes linéaires de grandes tailles à coefficients entiers (systèmes denses 700x700 [VIL88]), bases de Gröbner (famille de 100 polynômes booléens [SEN90]), et dans le futur, certains problèmes d'équations différentielles.

Le système PAC repose alors sur deux problématiques :

1° Evaluer l'apport du parallélisme au calcul formel, en étudiant la parallélisation de certains algorithmes. Une réflexion sur l'organisation d'un système parallèle dédié au calcul formel est présentée dans la première partie de cette thèse. Elle est illustrée par l'exemple du produit de deux polynômes creux.
La parallélisation de ce problème en utilisant une découpe standard est immédiate : mais l'étude de complexité qui lui est associée doit être faite avec soin. Elle nécessite la modélisation précise des opérations effectuées. L'intérêt ici est double : la démarche peut se généraliser à d'autres problèmes, et elle donne pour le problème choisi la meilleure stratégie concernant le choix du polynôme à partager.
Par ailleurs, nous présentons une nouvelle méthode de parallélisation, basée sur un partage de type Karatsuba adapté à des polynômes creux.

Cette partie débouche sur la présentation générale du système parallèle de Calcul Formel PAC : une première version, déjà utilisée par les membres de l'équipe Algorithmique Parallèle et Calcul Formel de TIM3, est implantée sur un hypercube FPS-T40 (32 processeurs), et est interfacée avec le système de Calcul Formel Reduce [ROC89]. Elle permet d'effectuer différents calculs (séquentiels ou parallèles) avec des polynômes à coefficients rationnels et à plusieurs indéterminées. Une deuxième version, plus portable et plus générale, est en cours de développement: elle doit être portée sur une machine Telmat MégaNode (128 processeurs).

2° Construire un système de calcul formel adapté à quelques grands problèmes de l'ingénieur. Cet aspect suppose l'évaluation des algorithmes de base d'un système de calcul formel, essentiellement en ce qui concerne l'arithmétique qui, sauf pour des problèmes spécifiques, doit être nodale. Il serait en effet impossible de créer un système dont la vocation est de résoudre des problèmes de grande taille en ayant une arithmétique lente (ce qui n'était peut-être pas la motivation des systèmes classiques). C'est pourquoi l'étude de l'arithmétique de base est le centre de la deuxième partie de cette thèse.
Il s'agit donc d'une part d'évaluer les algorithmes existants et d'étudier plus précisément leur implantation, et d'autre part de dégager de nouveaux algorithmes, mieux adaptés aux entiers ayant de un à quelques centaines de chiffres décimaux, couramment manipulés en calcul formel.

Après l'étude du choix de la représentation et des conversions associées, les opérations de base sont étudiées.

Pour la multiplication, l'étude de l'algorithme de Karatsuba et de son seuil d'efficacité par rapport à l'algorithme standard est classique. Par contre, la généralisation de cet algorithme, adaptée à des opérandes de tailles quelconques (éventuellement différentes) est nouvelle.

Concernant la division, l'utilisation de l'itération de Newton a été introduite par Cook [COO66]. L'algorithme qui est élaboré ici est cependant un peu différent: notamment, il se distingue par le fait que tous les itérés successifs obtenus sont exacts, la correction étant assurée de manière déterministe par une division euclidienne sur des petits entiers. Il n'est alors plus besoin de faire des troncatures sur les itérés successifs. Ainsi, même si l'ordre de complexité de cet algorithme est le même que celui de l'algorithme de Cook, il est cependant plus rapide, et ce d'autant plus que les opérandes sont de tailles très différentes.

Le calcul du plus grand commun diviseur est un chapitre important. Après une rapide présentation des algorithmes standards, l'étude précise de l'algorithme de Lehmer conduit à de nombreuses optimisations. Elle permet notamment de donner le meilleur choix expérimental de la base de calcul, même si nous ne sommes pas parvenus à démontrer théoriquement la complexité de ce choix. Nous proposons un nouvel algorithme, appelé Lehmer généralisé, qui, expérimentalement, s'avère plus efficace même lorsque les opérandes sont de petite taille. Enfin, la présentation de l'algorithme de Schönhage est une première version française... La seule véritable référence est l'original en allemand de Schönhage; les versions de Mœnck ou d'Aho, Hopcroft et Ullman sont consacrées au calcul du pgcd de polynômes, et si elles peuvent être plus ou moins adaptées aux entiers, elles ne permettent pas d'obtenir la complexité donnée par Schönhage. Une version précise de cet algorithme, adaptée au calcul du pgcd (tests d'arrêt, calcul de matrices de Lehmer au lieu de considérer les développements en fractions continues) et qui se prête à l'implantation, est décrite précisément.

Finalement, la vectorisation des algorithmes est discutée, et évaluée expérimentalement sur une unité vectorielle flottante. Nous montrons que cette vectorisation ne s'adapte efficacement qu'aux algorithmes standards. Melenk [MEL88] a choisi la même représentation, et obtient de meilleurs résultats. Il est vrai que c'est sur un Cray, et qu'il n'a pas -à notre connaissance- comparé la vectorisation des algorithmes standards aux algorithmes rapides non vectorisés. L'unité vectorielle entière qui nous paraît la mieux adaptée au problème est décrite en conclusion.

Certes, il n'est pas question de concevoir un système parallèle global, recouvrant la connaissance entière des mathématiques : les bibliothèques numériques n'ont jamais eu cet objectif. Mais il faut créer l'équivalent de ces bibliothèques, adapté à la méthodologie du calcul formel.
Ainsi, les programmes numériques ont en général un flot de contrôle indépendant des données : la même séquence de calcul est effectuée un certain nombre de fois, et le coût de cette séquence est indépendant de la valeur des opérandes. Par exemple, un produit matrice-vecteur flottant prendra le même temps, quelles que soient les valeurs des coefficients. Par contre, en calcul formel, le coût des tâches est intrinsèquement lié au conditionnement des données.
Pourtant, il existe souvent des points pour lesquels les extrêmes se rejoignent. Ainsi, l'algorithme de Gauss numérique est un exemple de programme dépendant des données (Recherche du pivot), qui est tout à fait comparable à l'algorithme de Gauss formel dans un corps fini [VIL88].

Une réflexion plus profonde est celle de savoir s'il existe un clivage fondamental entre calcul numérique et calcul formel. Naturellement le noyau (numérique ou formel) est différent : ainsi le numéricien utilise à la base un sous-ensemble fini de l'ensemble des rationnels, alors qu'en calcul formel on prétend travailler avec *tous* les rationnels. Certes, cela entraîne nombre de différences profondes, mais qui, à mon sens, ne sont pas pour autant essentielles.
En effet, pour traiter un problème numérique ou formel, la démarche est la même, et peut se décomposer en deux étapes.
La première est celle du mathématicien appliqué, qui étudie un système d'équations (linéaire : A.x=b, non linéaire : f(x)=0, équations différentielles ou aux dérivées partielles). La connaissance est celle d'un mathématicien : noyaux, théorèmes d'existence et d'unicité, "bons espaces", etc... .

La deuxième étape est celle de l'algorithmicien appliqué, qui, de toute façon, associe à un problème donné dans un espace abstrait (espaces de Sobolev, de Gevrey...) un problème en dimension finie (linéaire : A.x=b, non linéaire : f(x)=0), et il n'y a pas de méthode pour échapper à cela.

Ce qui différencie alors un analyste numéricien et un spécialiste du calcul formel est très mince : les méthodes de base sont souvent issues d'une même idée (méthode de Gauss pour les systèmes linéaires, méthode de Newton pour les non-linéaires), mais elles sont appliquées sur des structures différentes et doivent donc être adaptées. Il est donc essentiel pour le mathématicien appliqué d'avoir à sa disposition les deux outils, numériques et formels, qui, issus d'une même démarche, sont adaptés à des types de problèmes différents. Ignorer l'un de ces outils pour le traitement d'un problème serait alors restrictif.

PREMIERE PARTIE

MODELE PARALLELE

POUR LE

CALCUL FORMEL

SOMMAIRE

INTRODUCTION

De nombreuses recherches théoriques ont contribué à l'étude d'algorithmes parallèles pour le calcul formel : la motivation est alors de montrer si un problème peut être traité en temps poly-logarithmique avec un nombre polynomial d'unités de calcul (classe $\mathcal{NC}$) [COO85]. Le modèle théorique classiquement utilisé est le modèle PRAM [BOR82] [KAL89a]: les unités de calcul peuvent manipuler les données stockées dans une mémoire partagée, et qui peut être accédée selon différents modes (CREW : lectures concurrentes et écritures exclusives est le mode le plus utilisé).

La parallélisation effective d'algorithmes, en vue de l'implantation sur une machine parallèle donnée, pose d'autres problèmes. D'une part, le nombre de processeurs, physiquement limité dans le cas pratique, dépendant de la taille du problème dans le cadre théorique, différencie fondamentalement les deux études. D'autre part, le modèle PRAM ne correspond pas aux architectures massivement parallèles -possédant un grand nombre de processeurs- actuellement disponibles, pour lesquelles la mémoire est distribuée : ainsi, le côut du partage d'information entre deux processeurs, non considéré dans le modèle théorique, doit être pris en compte lors de l'implantation sur une architecture.
Le choix d'un modèle spécifique pour la parallélisation d'algorithmes est donc essentiel : l'importance des temps de communication de données et de synchronisation des processeurs, souvent négligés dans les études théoriques, est une contrainte permanente dont dépend intrinsèquement l'efficacité d'un algorithme parallèle.

De quelle spécification parallèle a besoin le Calcul Formel ? Les résultats obtenus récemment en Intelligence Artificielle montrent que des langages tels que Multilisp [HAL88] ou Prolog [DUP88], si ils permettent la spécification d'un certain degré de concurrence, ne donnent pas toujours, après implantation, des résultats convaincants.
Comprendre quelles approches permettent de tirer le meilleur parti d'une machine parallèle est une étape essentielle vers la définition d'un langage permettant de spécifier le parallélisme dans un algorithme du Calcul Formel. Langages et applications sont liés : si la sémantique d'un langage permet de définir de nouvelles stratégies dans le développement d'applications, les spécifications essentielles à certains algorithmes -et d'autant plus lorsqu'ils sont parallèles- permettent de faire progresser les langages. Par exemple, l'importance de la dérivation formelle dans l'élaboration de Lisp est significative [MAC78].

De même, évaluer précisément l'efficacité des architectures nouvelles sur des problèmes choisis, ce qui est réalisé en Calcul Numérique avec les programmes-test ('benchmark') *Linkpac* [DON87] ou *Livermore Loops*

[MEU89], est une démarche fondamentale pour la définition d'une machine dédiée au Calcul Formel. Si le champ des applications est suffisamment grand pour être représentatif, les conclusions qui pourront en être tirées seront d'autant plus significatives.

PAC s'inscrit bien dans ce contexte : les machines massivement parallèles peuvent-elles ouvrir de nouveaux champs d'investigation au Calcul Formel ?

D'autres équipes travaillent sur l'utilisation du parallélisme en calcul formel.
S. Watt [WAT86] a étudié dans sa thèse la parallélisation de différents problèmes du calcul formel sur un réseau local. D'autres équipes, comme J. Fitch à Bath [FIT89] travaillent sur la répartition d'algorithmes sur des réseaux de stations de travail.
Différentes implantations ont été réalisées sur des machines à mémoire partagée, avec peu de processeurs : une implantation de Reduce sur Cray X-MP a été réalisée à Berlin [MEL88]; C. Ponder a étudiée l'implantation de différents algorithmes en Q-Lisp sur Alliant (4 processeurs) [PON88].
Dans le domaine des architectures massivement parallèles, D. Hilhorst [FEK89] a écrit un programme de manipulation de séries de Poisson sur NCube (16 processeurs).

Chapitre I

ADEQUATION DU PARALLELISME AU CALCUL FORMEL

I. Architectures Multi-Processeurs

Les architectures évoluent sans cesse : si cette constatation n'a rien d'original, elle nous conduit cependant à dresser un rapide bilan des architectures parallèles actuelles, de leurs particularités générales, préambule essentiel à la définition de l'architecture cible.
On peut essentiellement distinguer trois modèles parallèles parmi les architectures parallèles de type MIMD (Multiple Instruction Multiple Data)[FLY66].

I.1. Modèle à mémoire partagée

Le nombre de processeurs est faible, mais ils sont très puissants (éventuellement couplés à des unités vectorielles) et peuvent s'échanger des données via une mémoire partagée ou hiérarchisée. Par conséquent, les communications sont très rapides. Les machines les plus performantes s'inscrivent dans ce cadre (Cray II, Alliant... .). Le Cray le plus performant (Y-MP/832 8 processeurs) peut atteindre 2,5 GFlops : il résout un système linéaire dense 1000x1000 en à peine plus de 0,3 secondes...[TRY89].
Ces machines présentent donc l'avantage d'être en quasi-accord avec le modèle théorique PRAM au niveau des communications, mais aussi l'inconvénient majeur de disposer d'un nombre très limité de processeurs, du fait même du partage de la mémoire. Cette contrainte nuit au parallélisme, puisque le nombre de tâches parallèles traitées à un moment donné est faible, et influence par là même directement la granularité des tâches. Par ailleurs, ces architectures étant souvent très spécialisées pour atteindre de hautes performances, la parallélisation des algorithmes perd fortement en généricité.

I.2. Modèle massivement parallèle

Dans ce cadre, le nombre de processeurs, même s'il est borné, peut être très grand : 2^{14} processeurs pour l'utopique FPS T40000, 1024 processeurs pour le NCube ou le Telmat Méga-Node (réalisable même si non réalisé).

On considère un réseau de processeurs pouvant communiquer par envoi et réception de messages. Chaque nœud du réseau peut être constitué de différents éléments, qui lui confèrent une certaine autonomie en calcul : par exemple, un processeur muni d'une mémoire importante et de co-processeurs arithmétiques (unités scalaires, vectorielles ou dédiées à une application particulière).
La puissance de la machine est alors modulable, en ce sens que des puissances énormes (aussi bien en calcul qu'en espace mémoire) peuvent être obtenues en augmentant le nombre de nœuds. La puissance de crête - très théorique...- du plus puissant Méga-Node (1024 nœuds, chaque nœud équipé d'un transputer Inmos T800-30 et de 4 Mo de mémoire) est de 2 GFlops, avec un espace mémoire disponible de 4 Go : la vitesse du plus puissant des Cray n'est pas atteinte, mais la différence de prix est conséquente !
La connectique du réseau peut être statique (par exemple, topologie hypercube pour les machines Intel séries IPsc, FPS séries T, NCube) ou reconfigurable, c'est à dire qu'elle peut varier dynamiquement au cours de l'exécution d'un programme (projet Esprit Supernode).
La parallélisation gagne alors en généricité, puisqu'elle peut s'appuyer sur un nombre plus ou moins grand de processeurs, de toute façon extensible. En outre, chacun des processeurs peut autoriser l'exécution de différentes tâches en parallèle au niveau d'un même nœud, avec communications internes via la mémoire nodale : ainsi, les transputers d'Inmos intègrent un mécanisme de simulation du parallélisme, essentiel pour permettre d'étendre le comportement d'un algorithme à un nombre plus grand de nœuds.
L'inconvénient majeur de ce modèle réside alors dans les communications, qui ont un coût linéaire en fonction de la taille des données transmises lorsque les processeurs sont voisins, et qui sont fonctions de l'éloignement entre les deux processeurs communiquants. Pour être efficace, la parallélisation ne peut donc ignorer les temps de communication ou de synchronisation, souvent difficiles à modéliser, et ignorés dans les modèles théoriques. Pour un problème donné, il existe alors un optimum expérimental pour le nombre de processeurs à utiliser, en général bien inférieur au nombre de tâches indépendantes et parallèles du problème.
Un autre inconvénient de ce type d'architectures réside dans les temps de chargement d'un programme sur le réseau, qui peut être très grand lorsque le nombre de processeurs est important : environ deux minutes sur le FPS-T40, avec 32 processeurs. Il s'ensuit que le traitement de problèmes en parallèle ne devient expérimentalement intéressant que lorsque la résolution séquentielle est trop lourde en temps ou en occupation mémoire.

I.3. Modèle cellulaire

Le nombre de processeurs indépendants est alors très important, mais chaque nœud de l'architecture est constitué de processeurs élémentaires et

d'une mémoire de faible capacité. Une telle architecture permet un parallélisme de bas niveau, au niveau des instructions de base, les communications étant privilégiées par rapport au calcul : c'est un peu l'exemple de la Connection Machine, même si son fonctionnement est SIMD [FLY66] (comme la plupart des architectures cellulaires d'ailleurs). Un tel modèle est alors particulièrement adapté au traitement d'algorithmes découpés en tâches indépendantes simples, ne travaillant pas sur des objets à structure complexe, et ne faisant que des opérations élémentaires. Lorsque le problème est plus complexe au niveau des objets manipulés, le découpage du problème primitif en tâches élémentaires est souvent difficile. Certains langages modernes, qui sont aussi bien des langages de spécification que de programmation (Lustre [CHP87] ou Estelle [EST87] par exemple) permettent le passage d'un algorithme de haut niveau à un automate dont les différents états correspondent à des tâches élémentaires. Il est alors possible de vérifier formellement certaines propriétés de l'automate obtenu. Par ailleurs, ces langages intègrent des constructeurs parallèles : mais le problème du placement des tâches élémentaires concurrentes correspondantes sur une architecture cellulaire cible n'est cependant pas encore résolu.

I.4. Intérêt d'une approche parallèle des algorithmes du Calcul Formel

La -souvent grande- complexité intrinsèque des algorithmes sophistiqués du calcul formel a essentiellement deux origines :

- le nombre d'opérations élémentaires est important : certes, cette caractéristique n'est pas typique du calcul formel. Pourtant, des problèmes même de faible taille, si ils sont mal conditionnés, peuvent faire apparaître des termes intermédiaires importants, qui peuvent entraîner une augmentation considérable du nombre d'opérations machine élémentaires à effectuer.

- les objets manipulés ont une structure complexe, et demandent pour leur stockage une place mémoire importante, qui ne cesse de croître en général lors de l'exécution. La limite en stockage et en manipulation de structures de données qui souvent croissent exponentiellement est typique du calcul formel.

L'exemple du calcul de la forme d'Hermite d'une matrice à coefficients entiers est significatif de ce grossissement. Même si les coefficients de la forme finale sont bornés, la borne sur le coefficients intermédiaires est énorme : ainsi, l'espace mémoire nécessaire à l'exécution peut être considérable devant l'espace nécessaire au stockage du résultat [SIE89].

Remarque :

Il est important de noter que le temps passé en manipulation de symboles (i.e. génération de symboles -indéterminées-, substitution de symboles par des valeurs), du fait même du choix des structures de données, est négligeable devant le temps effectif nécessaire à l'évaluation d'une expression algébrique. Il y a en effet une différence structurelle entre un symbole et une valeur. La valeur est adressée directement et est temporelle, alors qu'un symbole indestructible peut être lié à une valeur par une opération s'effectuant en temps constant.

Les architectures massivement parallèles décritent précédemment peuvent apporter alors une solution à ces deux problèmes :

- Multiplier le nombre de processeurs calculant en parallèle multiplie -du moins en théorie- d'autant la puissance de calcul. Le problème étant de mettre en évidence l'indépendance de certaines tâches, de façon a pouvoir les exécuter parallèlement. Il existe cependant en pratique une limite à la puissance, due au coût des communications.

- Si la taille des mémoires disponibles est limitée par des contraintes technologiques (de l'orde de 4 Mo pour un micro-processeur simple, *et bon marché...*), le fait de disposer simultanément de plusieurs micro-processeurs permet encore de multiplier d'autant la taille mémoire utilisable. Le problème réside alors dans le partage d'un objet entre les différentes unités parallèles.

II. Différents niveaux de parallélisme d'un problème

Un des principes de base pour la réalisation d'un système de Calcul Formel est de permettre la construction de domaines imbriqués. Ainsi, il doit être possible, à partir du domaine des entiers $\mathbb{Z}$ supposé déjà construit, de construire l'ensemble des rationnels $\mathbb{Q}$, ou les polynômes à coefficients entiers ($\mathbb{Z}[X]$), et à partir de $\mathbb{Q}$, construire $\mathbb{Q}[X]$ et le corps des fractions $\mathbb{Q}(X)$.

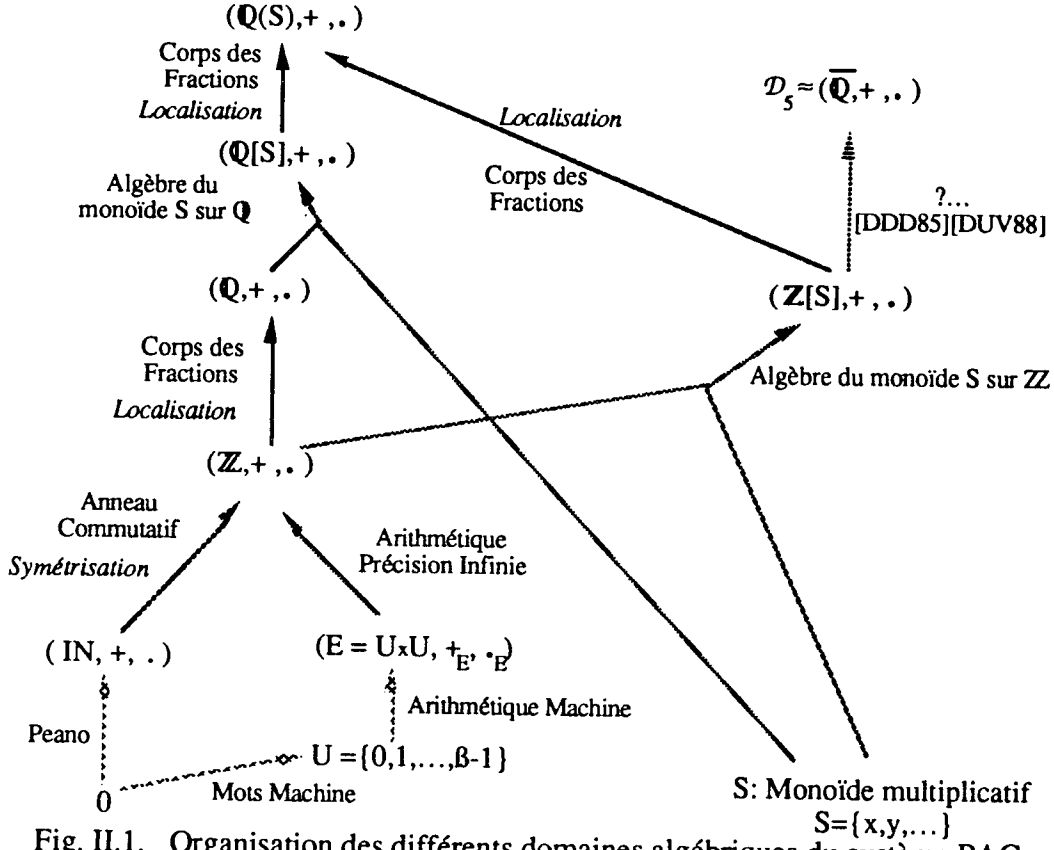


Fig. II.1. Organisation des différents domaines algébriques du système PAC

Ces extensions des domaines de base conduisent à la définition de domaines de plus en plus grands, et les objets de base de tels domaines ont une structure complexe, puisque construite à partir d'autres structures préalablement définies.
Chaque élément d'un domaine donné peut être manipulé et représenté. Ainsi, en théorie, lorsque l'on travaille dans $\mathbb{Q}$, on peut manipuler n'importe quel rationnel. En fait, manipuler un entier de trois millions de chiffres décimaux est impossible si l'on ne dispose que d'un Méga-Octet d'espace mémoire.

Deux niveaux d'approche parallèle peuvent être à priori considérés, correspondant à la complexité des différents domaines imbriqués.

II.1. Parallélisme de bas niveau

Le problème est décomposé en tâches élémentaires concurrentes, qui ne manipulent et qui ne s'échangent que des données très simples : caractères, entiers machines, etc... . Ce modèle, qui correspond à une description du problème sous forme d'automates élémentaires, est adapté à des algorithmes "simples", c'est à dire qui ne manipulent pas de structures de données complexes. Dans le cas d'algorithmes plus complexes, il ne permet pas d'aborder le parallélisme d'un problème à un haut niveau, ce qui est très restrictif lorsqu'il présente un parallélisme inhérent.

II.2. Parallélisme de haut niveau

Le problème est alors décomposé en tâches élémentaires qui travaillent sur des structures de données analogues à celles du problème primitif. Au niveau des algorithmes du calcul formel, ce problème permet aussi bien la spécification du ET-parallélisme (décomposition par division du problème initial : *Diviser Pour Régner*) qu'à celle du OU-parallélisme (traitement du même problème par deux méthodes différentes, ce qui est très intéressant lorsque la meilleure stratégie de résolution est indécidable : l'exemple du calcul des bases de Gröbner correspond bien à ce modèle).

II.3. Vers un Parallélisme à deux niveaux

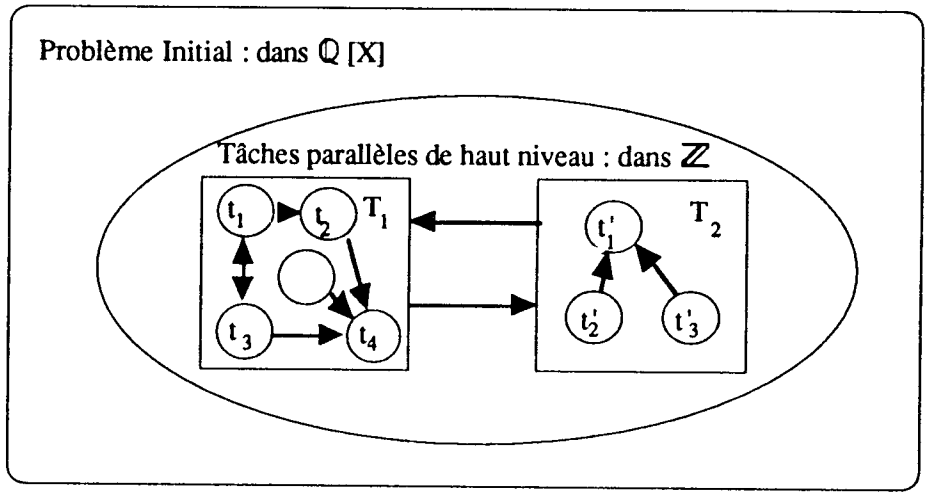
Considérons par exemple le calcul du pgcd de deux polynômes à coefficients dans l'ensemble des rationnels.
Une des méthodes pour traiter ce problème est de borner le degré du pgcd à calculer, puis de découper le problème modulairement en évaluant chacun des deux polynômes initiaux en diffférents points.
Cette méthode présente de toute évidence un ET-parallélisme inhérent. Chaque tâche correspond au calcul du pgcd de deux entiers, évaluations des polynômes en un point particulier. On interpole alors les différentes valeurs obtenues, pour trouver le polynôme résultat. La méthodologie est alors la suivante :

1° *Descente* : définition des tâches concurrentes
On définit une liste de tâches, chaque tâche correspondant à un calcul de pgcd de deux entiers.

2° *Calcul* : exécution parallèle des différentes tâches

3° *Remontée* : calcul du polynôme résultat (en faisant attention au choix éventuel de mauvais points d'interpolation)

Les tâches élémentaires travaillant sur des données complexes (pgcd d'entiers en précision infinie) correspondent à un haut niveau de parallélisme.
Chacune de ces tâches peut alors être découpée pour être traitée systoliquement à un bas niveau de parallélisme (Algorithme de Brent et Kung [BKU83]).



T : Tâches parallèles de haut niveau (dans ℤ) - t : Tâches parallèles de bas niveau (dans les entiers machines)

Fig. III.2.3 Différents niveaux de parallélisme autour du calcul du pgcd par interpolation

La complexité des données manipulées en calcul formel conduit donc à dégager sur un problème donné en premier lieu des tâches de haut niveau. Chacune de ce tâches peut alors être décomposée en sous-tâches de bas niveau. Les communications entre tâches de haut niveau correspondent à des données complexes, alors que les tâches de bas niveau s'échangent des données plus élémentaires.

III. Lien avec les architectures distribuées

Les tâches de haut niveau restent complexes : les communications nécessaires entre différentes tâches seront de coût négligeable devant le coût du calcul. Il apparaît donc naturel de répartir de telles tâches sur un réseau massivement parallèle, en affectant à chaque processeur une tâche.

Les sous-tâches concurrentes de bas niveau n'effectuent quant à elles que des calculs élémentaires : les communications entre de telles tâches doivent donc être très rapides, pour rester négligeables devant les temps de calcul.

Placées sur un même nœud d'une architecture à grosse granularité, elles peuvent communiquer efficacement via la mémoire associée au nœud.

Deux types d'architectures semblent donc convenir au Calcul Formel : les architectures massivement parallèles à mémoire distribuée (FPS T-Series, Intel Ipsc, Telmat...), ou les architectures partagées à mémoire hiérarchisée (Alliant).

Chapitre II

UN EXEMPLE DIDACTIQUE

I. Choix du produit de deux polynômes

Le produit de deux polynômes creux est un exemple typique de problème du Calcul Formel dépendant des données : il mêle manipulation de symboles -par l'intermédiaire des indéterminées- et calcul -par l'intermédiaire des produits et sommes des coefficients-.

Nous nous restreignons ici au calcul d'une représentation ordonnée du produit de deux polynômes creux ordonnés à une indéterminée X et à coefficients dans $\mathbb{Z}$. La méthode décrite, ainsi que l'analyse de complexité qui en est faite, sont cependant directement généralisables aux polynômes à plusieurs indéterminées. La résolution parallèle nécessite des méthodes utilisées classiquement en parallélisme : partitionnement du problème, répartition des données, remontée des résultats...

Le produit de deux polynômes est intrinsèquement parallèle, puisque distributif par rapport à la somme.

Soit $m = a.X^n$ un monôme de $\mathbb{Z}[X]$, avec $a \in \mathbb{Z}$.
Nous appelons *taille(a)* le nombre de chiffres nécessaires à la représentation de a (i.e. $\lfloor \text{Log}_B |a| + 1 \rfloor$ -cf Partie II Chap I-) . Par abus de langage, nous désignerons par *taille(*m*)* la taille du monôme m, assimilée à celle de son coefficient.
La taille d'un polynôme est alors définie comme la somme des tailles de chacun de ses monômes.

Soit P un polynôme de $\mathbb{Z}[X]$. Dans la suite, nous désignerons par :

- d(P) le degré de P
- $N_{mon}(P)$ le nombre de monômes non nuls et de degrés distincts de P
- B(P) la taille du plus grand coefficient en module de P

Un polynôme creux peut être décrit qualitativement comme ayant un nombre de monômes très inférieurs à son degré. Pour étudier le comportement d'un algorithme, nous définissons la *densité* $\rho(P)$ d'un polynôme P de degré supérieur à 1 par :

$$\rho(P) = \frac{N_{mon}(P)}{d(P)+1}$$

De manière évidente : $0 < \rho(P) \leq 1$

Autrement dit, plus la densité de P est proche de zéro, "plus P est creux". Au contraire, le polynôme est dense si sa densité est proche de un. Les algorithmes asymptotiquement optimaux pour calculer le produit de deux polynômes dépendent de la densité, autrement dit d'une appréciation qualitative. Par la suite, nous ne considérerons que des polynômes "relativement creux", c'est à dire tels que le nombre de monômes de leur produit soit égal au produit de leur nombre de monômes respectif :

$$\{P \text{ et } Q \text{ sont relativement creux}\} \Leftrightarrow \{ N_{mon}(P).N_{mon}(Q) = N_{mon}(P.Q) \}$$

Exemple : Soit $P = \sum_{i=0}^{n-1} x^{n.i}$ et $Q = \sum_{i=0}^{n-1} x^{i}$: P et Q ont tous deux n monômes.
P et Q sont relativement creux (aucun regroupement dans les produits de monômes), alors que Q et Q sont relativement denses (regroupement maximum des termes).

Pour étudier la complexité des algorithmes parallèles décrits, il est essentiel de définir le coût des opérations de base. Deux cas sont étudiés :

♦ les opérations arithmétiques sur les monômes sont effectuées en temps constant (ex. : $\mathbb{Z}/p\mathbb{Z}[X]$, ou des opérations sur des monômes dont le coefficient est représenté par un flottant et le degré borné ...)

♦ les opérations arithmétiques dépendent de la taille des monômes (cas général en Calcul Formel : ex. $\mathbb{Z}[X]$, $\mathbb{Q}[X]$...)

Modélisation des coûts des opérations de base :

Soit $m_1=a_1.X^{n_1}$ et $m_2=a_2.X^{n_2}$ deux monômes de $\mathbb{Z}[X]$, avec $(a_1, a_2) \in \mathbb{Z}^2$.

Pour modéliser les coûts arithmétiques, nous désignerons par :
$T_{mul}(m_1, m_2)$: le temps nécessaire à la multiplication de m_1 et m_2
$T_{add}(m_1, m_2)$: le temps nécessaire à la somme de m_1 et m_2
$T_{com}(m_1)$: le temps nécessaire pour communiquer m_1 d'un processeur quelconque du réseau à l'un de ses voisins (connexion physique du réseau entre les deux processeurs).

Axiome 1 Le temps $T_{com}(m_1)$, nécessaire pour communiquer un monôme d'un processeur à l'un de ses voisins, est supposé être une fonction affine croissante de la taille des données communiquées [SAA86]. D'où :

$$T_{com}(m_1) = \alpha + taille(m_1).\tau_{com} = \alpha + taille(a_1).\tau_{com} \qquad (\tau_{com} > 0, \alpha > 0)$$

Axiome 2 Le temps nécessaire à la multiplication de deux monômes est assimilé à celui de la multiplication de leurs coefficients, c'est à dire :

$$T_{mul}(m_1, m_2) = T_{mul}(a_1, a_2)$$

Axiome 3 Le temps nécessaire à l'addition de deux monômes est assimilé à celui de l'addition de leurs coefficients s'ils sont de même degré, et sinon il est nul, c'est à dire :

$$T_{add}(m_1, m_2) = \begin{cases} T_{add}(a_1, a_2) & \text{si } n_1 = n_2 \\ 0 & \text{sinon} \end{cases}$$

II. Analyse du problème et algorithmes parallèles

Nous désignerons par Algorithme de Jonhson [JOH74] [BAL89] l'algorithme qui consiste à calculer tous les produits de monômes, et à générer simultanément leur somme en regroupant par une technique de tri par tas ("heapsort") les monômes de même degré pour minimiser le nombre de comparaisons.

Le nombre d'opérations entre coefficients nécessaires au calcul du produit étant au moins supérieur au nombre de monômes du résultat, on en déduit que *cet algorithme est asymptotiquement optimal lorsque les polynômes sont relativement creux* : en effet, aucun regroupement de termes ne se produisant alors, le nombre de multiplications effectuées lors de l'algorithme est exactement le nombre de monômes du résultat.

Une approche parallèle directe amène à découper l'un des deux polynômes, et à effectuer des multiplications partielles sur chaque processeur. Les polynômes obtenus comme résultats intermédiaires sont ensuite additionnés deux à deux en parallèle. Cette méthode a été décrite par F. Siebert [RSS86] et a été observée parallèlement par C. Ponder [PON88].

Afin d'équilibrer les coûts des deux sous-problèmes obtenus après découpe, le polynôme découpé est séparé en deux parties ayant le même nombre de monômes.

Soient $P = \sum_{i=0}^{p} a_i.X^{\alpha_i}$ et $Q = \sum_{i=0}^{q} b_i.X^{\beta_i}$ deux polynômes.

Le schéma de calcul du produit en parallèle est décrit sur la figure de la page suivante.

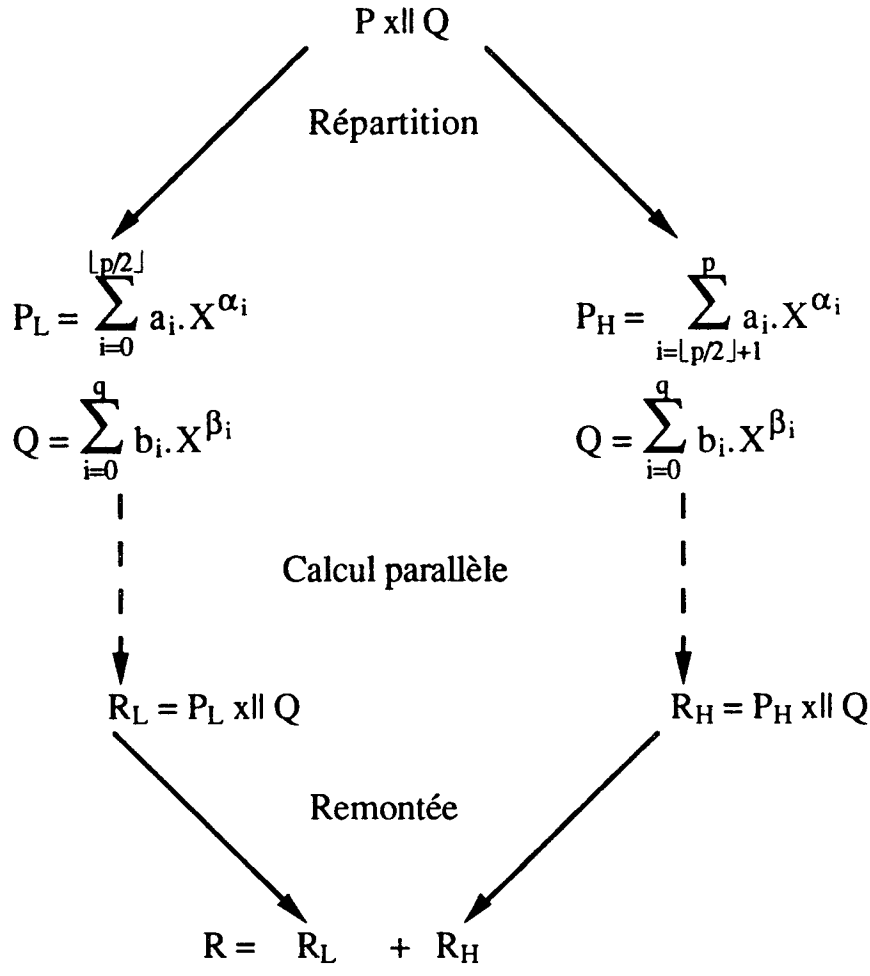


Fig. II. Produit parallèle de deux polynômes

III. Répartition d'un polynôme - Choix de la topologie

Le choix du polynôme à découper n'intervient pas pour le choix de la topologie : nous l'étudierons au paragraphe suivant. Nous supposerons donc ici qu'à chaque étape le polynôme découpé est P.

Le problème étant à chaque étape découpé en deux sous-problèmes de même complexité, après la $n^{ième}$ étape de découpage, 2^n sous-tâches ont été générées. Le nombre de processeurs nécessaires au calcul doit donc être une puissance de deux : soit 2^N le nombre de processeurs.
Le choix de la topologie est guidé par la parallélisation adoptée : à chaque étape k (k<N) tout processeur qui possède une tâche doit trouver un voisin inoccupé de façon à diviser son problème en deux.
La topologie hypercube est donc particulièrement adaptée à la répartition : à l'étape k, les processeurs du cube de dimension k-1 découpent leur tâche (P x|| Q) en deux sous-tâches (P_L x|| Q et P_H x|| Q), et envoient les polynômes P_H et Q à leur unique voisin du cube de dimension k.
En N étapes, le problème est donc réparti sur le N-cube (qui a 2^N sommets).

Les figures suivantes montrent la répartition des polynômes sur un hypercube de dimension quatre. Les numéros des processeurs sont écrits en base deux pour la fig. 1, et en base dix sur la fig. 2. Le nombre de flêches indique l'étape à laquelle s'effectue le transfert; le chiffre en gras qui les accompagne (fig.1) donne la proportion de monômes communiqués du polynôme découpé.

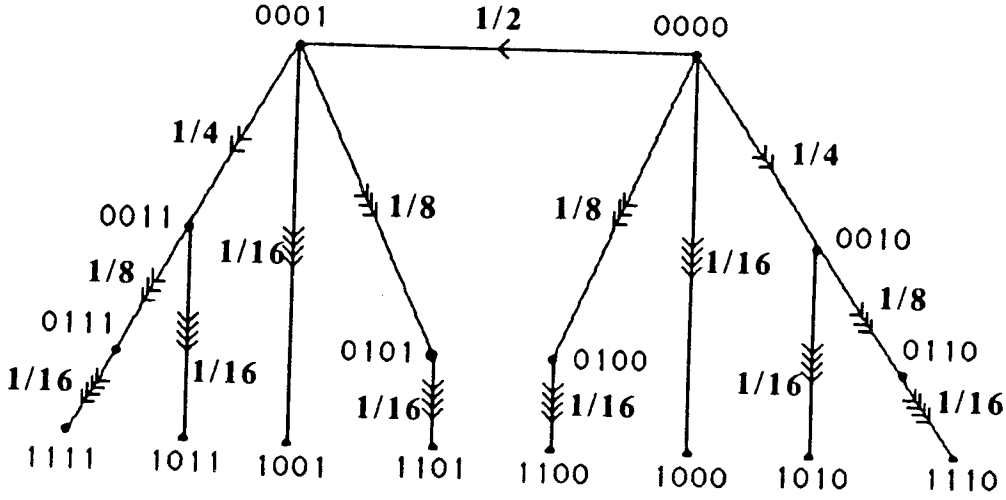


Fig. III.1. Graphe de répartition des polynômes sur le 4-cube

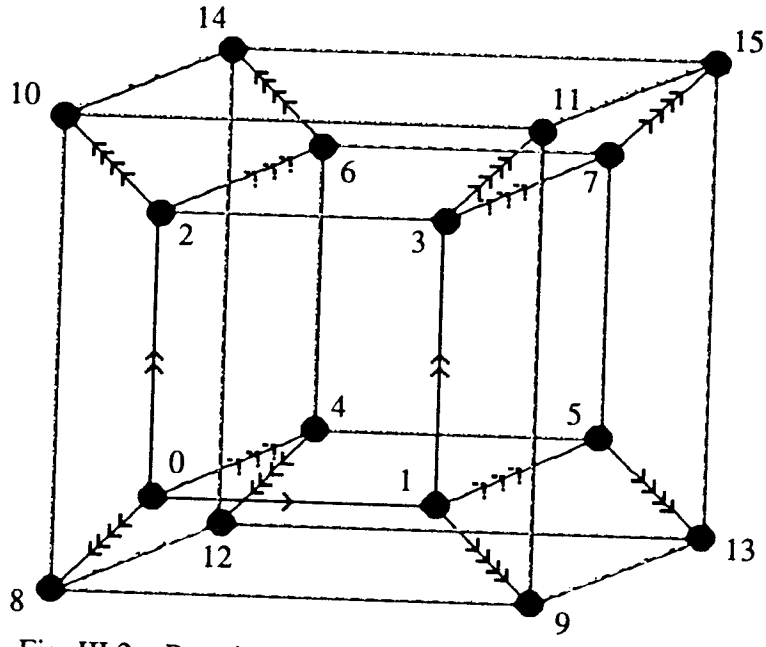


Fig. III.2. Représentation des répartitions sur le 4-cube

IV. Choix du polynôme à découper

Il s'agit de l'étude du meilleur choix à chaque étape du polynôme à découper.

Ce choix est lié à deux contraintes : minimiser le coût arithmétique du problème bien sûr, mais aussi minimiser le coût des communications, puisque plus le temps passé en communication est grand, moins important est le gain apporté par la parallélisation.

1. Contrainte arithmétique

Lemme 1 : il est préférable de découper le polynôme ayant le plus petit nombre de monômes

En effet, à nombres et *tailles* de monômes égaux, le nombre de comparaisons entre les degrés est moindre pour la multiplication de deux polynômes de tailles différentes que pour la multiplication de polynômes de même taille.

2. Contrainte de communication

Lemme 2 : il est préférable de découper le polynôme ayant la plus grande taille

Supposons que les monômes de P sont de même taille et qu'il en est de même pour les monômes de Q. Alors d'après l'axiome 1, la découpe de P entraîne un coût de communication proportionnel à (taille(Q) + (1/2).taille(P)). Le coût associé à la découpe de Q est lui proportionnel à (taille(P) + (1/2).taille(Q)), et donc est inférieur si taille(P) est supérieur à taille(Q).
On en déduit qu'il est moins coûteux de découper le plus grand polynôme.

Le compromis arithmétique/communication sera donc particulièrement intéressant à étudier sur cet exemple. Nous distinguerons deux algorithmes : dans le premier, on découpe systèmatiquement le plus petit polynôme, dans le deuxième, le plus grand.

Pour simplifier les études de complexités, seuls sont considérés les coûts effectifs des communications et des opérations d'addition et de multiplication (supposées prépondérantes devant les comparaisons de degré lorsque les coefficients sont suffisamment importants)

V. Etude de complexité

Nous distinguons deux cas : dans le premier les opérations de base ont un coût indépendant des données. Dans le deuxième, leur coût est lié à la taille des données.

V.1. Les opérations de base sont indépendantes des données

Les coûts des opérations de manipulation de base de monômes : T_{com}, T_{mul} et T_{add} sont donc supposés constants.
Soient P et Q deux polynômes creux ayant respectivement p et q monômes. Dans la suite, nous supposerons $p \geq q \geq 2^N$. Le cas asymptotique $p \geq q \gg 2^N$ sera étudié en particulier.

V.1.1. Découpage du plus petit polynôme

Comme $q \leq p$, à chaque étape, c'est le polynôme q (et ses sous-parties) qui est découpé.

a/ Répartition

A l'étape k, les processeurs du cube de dimension k-1 découpent leur tâche (P x|| Q) en deux sous-tâches(P_L x|| Q et P_H x|| Q), et envoient les polynômes P_H et Q à leur unique voisin du cube de dimension k. Nous avons vu qu'en N étapes, le problème est alors réparti sur le N-cube (qui a 2^N sommets).

Soit $T_1(p,q)$ le coût de la répartition du problème sur tous les processeurs de l'hypercube.

$$T_1(p,q) = \sum_{i=1}^{N} \left(p + \frac{q}{2^i}\right).T_{com} = \left(N.p + q.\left(1 - \frac{1}{2^N}\right)\right).T_{com} \qquad (1.1)$$

b/ Calcul

Soit $T_2(p,q)$ le coût du calcul des produits sur chaque processeur. L'algorithme séquentiel consistant à calculer tous les produits de monômes, le coût du produit séquentiel, sur un processeur, d'un polynôme ayant j monômes par un autre de k monômes est : $j.k.T_{mul} + (j.k - 1)T_{add}$

On en déduit :

$$T_2(p,q) = \frac{p.q}{2^N}.T_{mul} + \left(\frac{p.q}{2^N} - 1\right).T_{add} \qquad (1.2)$$

c/ Remontée

Soit $T_3(p,q)$ le coût de la remontée du résultat vers le processeur racine. A l'étape k, les processeurs du cube de dimension N-k+1 envoient leur polynôme résultat vers leur unique voisin du cube de dimension N-k. Après réception, les processeurs du cube de dimension N-k somment leur résultat à celui qu'ils réceptionnent. En N étapes, la remontée est effectuée.
A chaque étape, les processeurs qui ont précédemment partagé leur tâche, additionnent deux polynômes déja ordonnés, pour obtenir un résultat ordonné. Dans le pire des cas, cette opération nécessite seulement $\frac{p.q}{2^i}$ additions de coefficients; lorsque les degrés des monômes diffèrent, la somme est réalisée par un simple parcours linéaire des deux polynômes, ayant chacun au plus $(p.q)/2^i$ monômes.

On a donc : $T_3(p,q) = \sum_{i=N}^{1} \frac{p.q}{2^i}.T_{com} + \frac{p.q}{2^i}.T_{add}$

D'où l'on tire :

$$T_3(p,q) = \left(1 - \frac{1}{2^N}\right) p.q.(T_{com}+T_{add}) \qquad (1.3)$$

d/ Etude de l'efficacité

On étudie ici l'accélération apportée par la parallélisation de l'algorithme. Pour cela, on considère l'efficacité e, définie comme étant le rapport du temps de l'algorithme séquentiel équivalent sur le temps de l'algorithme parallèle multiplié par le nombre de processeurs [CMR88] :

$$\boxed{e = \frac{T_{seq}}{2^N.T_{par}}}$$

Remarque :

L'algorithme séquentiel est souvent difficile à définir -et donc à évaluer- et il dépend du nombre de processeurs N. C'est pourquoi on considère en général comme algorithme séquentiel équivalent l'algorithme parallèle pour un seul processeur. Ici, quel que soit N, l'algorithme séquentiel est meilleur que l'équivalent séquentiel de l'algorithme parallèle : en effet, on regroupe directement tous les produits de monômes qui contribuent à un même degré, ce qui n'est pas vérifié par l'algorithme parallèle. Cependant, le coût arithmétique (additions et produits de coefficients) reste le même pour les deux algorithmes.

Ici les temps parallèles et séquentiels sont respectivement :

D'après (1.2) $T_{seq}(p,q) = p.q.T_{mul} + (p.q - 1).T_{add}$ (1.4)

Et $T_{par}(p,q) = T_1(p,q) + T_2(p,q) + T_3(p,q)$, d'où :

$$T_{par}(p,q) = \left[N.p + q.(p+1).\left(1-\frac{1}{2^N}\right)\right].T_{com} + \frac{p.q}{2^N}.T_{mul} + (p.q - 1).T_{add} \quad (1.5)$$

Nous nous intéressons à l'intérêt du parallélisme pour le traitement de gros problèmes. Il s'agit donc d'étudier le comportement asymptotique de l'efficacité, c'est à dire lorsque $p \geq q \gg 2^N$.

On a alors, en négligeant 1 devant 2^N :

$$e \approx \frac{p.q.(T_{mul} + T_{add})}{2^N.p.(N+q).T_{com} + p.q.T_{mul} + 2^N.p.q.T_{add}} \quad (1.6)$$

D'où l'on tire :

$$\boxed{e \xrightarrow[p \geq q \gg 2^N \gg 1]{} \frac{T_{mul} + T_{add}}{2^N.(T_{com} + T_{add}) + T_{mul}}} \quad (1.7)$$

e/ Conclusion

D'après (1.7), l'algorithme parallèle proposé est intéressant si les temps des opérations de base sont tels que :

$$T_{mul} \gg 2^N.(T_{com} + T_{add})$$

Pour des exemples suffisamment complexes, la parallélisation est donc d'autant plus efficace que les opérations arithmétiques sont complexes. En effet, dans ce cas, communications et additions sont négligeables devant la multiplication.

V.1.2. Découpage du plus grand polynôme

On suppose ici $p > q$, et à chaque étape de l'algorithme, chaque processeur occupé découpe le plus grand des deux polynômes qu'il possède.
De façon à simplifier les calculs, nous supposerons : $p = 2^{n_0}.q$

a/ Répartition

1° On suppose $n_0 > N$

A chaque étape, on découpe toujours le polynôme P. D'après (1.1), on a :

$$T_1'(p,q) = \sum_{i=1}^{N} \left(q + \frac{p}{2^i}\right).T_{com} = \left(N.q + p.\left(1 - \frac{1}{2^N}\right)\right).T_{com}$$

D'où :

$$T_1'(2^{n_0}.q, q) = q.\left(N + 2^{n_0}.\left(1 - \frac{1}{2^N}\right)\right).T_{com} \qquad (2.1)$$

2° On suppose $n_0 \leq N$

On découpe P jusquà l'étape n_0; puis, alternativement, à l'étape k de la répartition ($k \geq n_0$) on découpe : $\begin{cases} P & \text{si } (k-n_0) \text{ est impair} \\ Q & \text{si } (k-n_0) \text{ est pair} \end{cases}$

Le coût de la répartition est alors donné par :

$$T_1'(p,q) = \left(\sum_{i=1}^{n_0} \left(q + \frac{p}{2^i}\right) + \sum_{i=0}^{N-n_0/2} \left(\frac{q}{2^i} + \frac{p}{2^{n_0+i+1}} + \frac{q}{2^{i+1}} + \frac{p}{2^{n_0+i+1}}\right)\right).T_{com}$$

D'où :

$$T_1'(2^{n_0}.q, q) = q.\left(n_0 + 4 + 2^{n_0} - \frac{5}{2^{\frac{N-n_0}{2}}}\right).T_{com} \qquad (2.2)$$

b/ calcul et remontée

Après répartition, les polynômes sont équidistribués sur chacun des processeurs, et chaque processeur possède :

- si $n_0 > N$: $\left(p/2^N\right)$ monômes de P, et q monômes de Q
- si $n_0 \leq N$: $\left(p/2^{(N+n_0)/2}\right)$ monômes de P, et $\left(q/2^{(N-n_0)/2}\right)$ monômes de Q

Les produits étant ensuite effectués nodalement par l'algorithme de Johnson, le temps de calcul des produits de monômes est égal au produit de T_{mul} par le nombre de monômes de P et le nombre de monômes de Q. En tenant compte de la sommation des termes obtenus, on en déduit que dans tous les cas :

$$T_2'(p,q) = \frac{p.q}{2^N}.T_{mul} + \left(\frac{p.q}{2^N} - 1\right)T_{add} = T_2(p,q) \qquad (2.3)$$

Le nombre de monômes obtenus sur chaque processeur étant alors le même que lorsque l'on découpe le plus petit polynôme, on a :

$$T'_3(p,q) = T_3(p,q) = \left(1 - \frac{1}{2^N}\right).p.q.(T_{com}+T_{add}) \qquad (2.4)$$

c/ Comparaisons des algorithmes et étude de l'efficacité

Les étapes de calcul et de remontée étant de même coût, il suffit de comparer les coûts T_1 et T'_1 de la répartition. Nous nous plaçons toujours dans le cas : $p \gg q \gg 2^N$

On a alors, d'après (1.1), et en posant : $p = 2^{n_0}.q$:

$$T_1(2^{n_0}.q, q) = q.\left(N.2^{n_0} + 1 - \frac{1}{2^N}\right).T_{com}$$

et, d'après (2.1) :

$$T'_1(2^{n_0}.q, q) = q.\left(N + 2^{n_0}.\left(1 - \frac{1}{2^N}\right)\right).T_{com}$$

On en déduit donc qu'asymptotiquement (N grand) :

$$\boxed{T_1(p, q) \approx 2^{n_0}.T'_1(p, q)} \qquad (2.5)$$

Conclusion sur le choix du polynôme à découper :

La répartition du polynôme de plus grande taille est donc meilleure que celle du polynôme de plus petite taille. Mais, le coût de la remontée étant d'un ordre supérieur à celui de la répartition, l'approximation de l'efficacité (1.7) reste valable.

V.2. Les opérations de base dépendent des données

On suppose désormais que le coût des trois primitives de manipulation de monômes
($T_{com}(m_1, m_2)$, $T_{add}(m_1, m_2)$ et $T_{mul}(m_1, m_2)$) dépendent de la taille des monômes.
T_{add} et T_{com} sont supposées linéaires, et nous adopterons la convention d'écriture : $T_{add}(n) = T_{add}(n,n)$

Il est donc nécessaire de donner une borne sur la croissance des coefficients du résultat lors des opérations d'addition et de multiplication de polynômes. L'évaluation de cette borne est liée à deux contraintes :

(3.1) ♦ les polynômes opérandes sont supposés *relativement creux* (ou presque relativement creux...); ainsi, le nombre de monômes obtenus dans le produit est égal (ou proche) au produit des nombres de monômes des opérandes. De plus, les sous-polynômes obtenus sur chaque processeur seront eux aussi relativement creux (ou presque).

(3.2) ♦ les coefficients initiaux de chacun des polynômes opérandes sont tous du même ordre : autrement dit, le problème est supposé *équilibré* (ou conditionné). Nous reviendrons plus tard sur cette remarque.

Soient P et Q deux polynômes vérifiant les contraintes (3.1) et (3.2).
On pose : $p = N_{mon}(P)$ et $q = N_{mon}(Q)$. De manière évidente on a alors :

$$B(P.Q) \approx B(P)+B(Q) \tag{3.3}$$

$$N_{mon}(P.Q) \approx N_{mon}(P).N_{mon}(Q) \approx p.q \tag{3.4}$$

$$\begin{cases} Taille(P) \approx B(P).N_{mon}(P) \approx p.B(P) \\ Taille(Q) \approx B(Q).N_{mon}(Q) \approx q.B(Q) \\ Taille(P.Q) \approx p.q.(B(P)+B(Q)) \end{cases} \tag{3.5}$$

$T_1(P,Q)$, $T_2(P,Q)$ et $T_3(P,Q)$ désignent par la suite respectivement les coûts de répartition, calcul des produits partiels et remontée.

L'étude précédente a montré qu'il était plus avantageux de découper à chaque étape le polynôme de plus grande taille, même si le choix du polynôme à découper n'entraîne pas de modification de l'ordre de l'algorithme : les coûts des étapes de calcul et de remontée restent les mêmes. Nous supposerons donc, de façon à simplifier l'étude, que le polynôme découpé est toujours le même à chaque étape de la répartition : soit P ce polynôme. Ainsi, le cas du découpage du plus petit (respectivement du plus grand) des polynômes pourra être étudié en supposant $N_{mon}(P) \ll N_{mon}(Q)$ (respectivement $N_{mon}(P) \gg N_{mon}(Q)$).

a/ Répartition

D'après (3.2), P et Q sont assimilés respectivement à $N_{mon}(P)$ et $N_{mon}(Q)$ monômes de tailles B(P) et B(Q). En utilisant (1.1), on obtient :

$$T_1(P,Q) = \left(1 - \frac{1}{2^N}\right).p.T_{com}(B(P)) + N.q.T_{com}(B(Q))$$

La fonction T_{com} étant supposée linéaire, on en déduit donc :

$$T_1(P,Q) = \left(1 - \frac{1}{2^N}\right).T_{com}(Taille(P)) + N.T_{com}(Taille(Q)) \tag{3.6}$$

b/ Calcul

Après la répartition, chaque processeur possède $\left(P/_{2^N}\right)$ monômes de P, et q monômes de Q.
D'après (3.2), les produits nodaux étant effectués par l'algorithme de Johnson, le coût de chaque produit de monômes peut être borné par $T_{mul}(B(P),B(Q))$.
Le coût total des produits de monômes sur chaque nœud est donc :

$$\frac{p.q}{2^N}.T_{mul}(B(P),B(Q))$$

Les monômes obtenus sont de taille bornée par B(P)+B(Q). Leur somme pour former le produit partiel est donc obtenue avec un coût :

$$\left(\frac{p.q}{2^N} - 1\right)T_{add}(B(P)+B(Q), B(P)+B(Q))$$

L'addition étant linéaire, et en supposant que 2^N est négligeable devant p.q, le coût total des calculs des produits partiels sur chaque nœud est alors :

$$T_2(P,Q) = \frac{1}{2^N}.(p.q.T_{mul}(B(P),B(Q)) + T_{add}[p.q.(B(P)+B(Q))]) \quad (3.7)$$

c/ Remontée

En supposant la contrainte (3.1) vérifiée, les monômes obtenus sur chaque nœud sont de degrés différents. Le nombre de monômes du résultat est donc $\frac{p.q}{2^N}$, et la taille du polynôme obtenu sur chaque nœud est :

$$\frac{p.q}{2^N}.(B(P)+B(Q))$$

D'après (3.1), le nombre de monômes du résultat final est proche de p.q. A chaque étape, le nombre de monômes est donc doublé sur les nœuds récepteurs.
Par analogie avec (1.3), on en déduit :

$$T_3(P,Q) = \left(1 - \frac{1}{2^N}\right)p.q.[T_{com}(B(P)+B(Q)) + T_{add}(B(P)+B(Q))] \quad (3.8)$$

d/ Temps parallèle - Etude de l'efficacité

Le temps parallèle de l'algorithme est :

$$T_{par}(P,Q) = T_1(P,Q) + T_2(P,Q) + T_3(P,Q) \quad (3.9)$$

Le temps séquentiel est donné par : (analogie avec (1.4))

$$T_{seq}(P,Q) = p.q.T_{mul}[B(P),B(Q)] + (p.q - 1).T_{add}[B(P)+B(Q)] \quad (3.10)$$

Si p et q sont petits devant 2^N, l'efficacité est proche de zéro, et le parallélisme inefficace. On considère donc le traitement d'un gros produit, et on se place dans le cas asymptotique où p et q sont grands devant 2^N. On obtient alors, à partir de (3.6) (3.7) et (3.8), l'expresssion simplifiée de $T_{par}(P,Q)$:

$$T_{par}(P,Q) = p.q.\left(\frac{T_{mul}[B(P),B(Q)]}{2^N} + T_{add}[B(P)+B(Q)] + T_{com}[B(P)+B(Q)]\right)$$

L'efficacité est alors :

$$e \approx \frac{T_{mul}[B(P),B(Q)] + T_{add}[B(P)+B(Q)]}{T_{mul}[B(P),B(Q)] + 2^N.(T_{add}[B(P)+B(Q)] + T_{com}[B(P)+B(Q)])} \quad (3.11)$$

Si B(P) et B(Q) sont suffisamment grands devant 2^N, alors seuls les coûts des multiplications de coefficients prédominent, et l'efficacité est proche de un.

e/ Combien de processeurs, pour quel gain ?

Le compromis entre taille du problème et nombre de processeurs peut être étudié plus précisément dans le cas où les polynômes sont dans $\mathbf{Z}[X]$. En effet, on peut alors supposer (cf Partie II) :

$$T_{com}(n) = \tau.n + \alpha \quad (4.1)$$
$$T_{add}(n,p) = c_{add}.(n+p) \quad (4.2)$$
$$T_{mul}(n,p) = (n+p).g(n+p) \quad (4.3)$$

Dans la deuxième partie de cette thèse, nous présenterons et utiliserons les trois valeurs suivantes pour g, liées à trois algorithmes différents :

$g_1(n) = c_1.n$ (Algorithme standard)
$g_2(n) = c_2.n^{(Log_2 3)-1}$ (Karatsuba mixte)
$g_3(n) = c_3.Log(n).Log\ Log(n)$ (Schönhagge-Strassen)

Elles permettent de faire l'assertion que la fonction g est super-linéaire.

Il existe deux façons de considérer le parallélisme pour un problème donné :

1°/ Le problème nécessite un *certain temps* de traitement (qualitativement long, mais sans plus...) : l'étude précédente montre que dans le cadre du produit de polynômes, le parallélisme permet d'apporter une amélioration intéressante de la complexité. On peut ainsi espérer que

sur un CRAY avec quatre processeurs, le temps du problème pourra être divisé par quatre environ.

2°/ Le problème nécessite un *temps certain* de traitement (qualitativement très long...!): diviser son temps par un petit facteur n'est pas suffisant : une approche de type parallélisme massif est alors intéressante. Mais quels sont les limites du parallélisme sur un problème donné, et à partir de quel moment une approche parallèle ne permet plus une amélioration du temps de traitement ?

Il s'agit donc d'étudier l'évolution du temps de traitement, en fonction de la taille des données et du nombre de processeurs.

Soient P et Q deux polynômes, ayant respectivement p et q monômes, avec $p \geq q$.
Posons $B_p=B(P)$ et $B_q=B(Q)$.
L'expression (3.9) du temps parallèle sur 2^N processeurs pour effectuer leur produit donne :

$$\begin{aligned} T_{Par}^{(N)}(P,Q) = &\left(1 - \frac{1}{2^N}\right).p.T_{com}(B_p) + N.q.T_{com}(B_q) \\ &+ \frac{1}{2^N}.p.q.\left(T_{mul}(B_p,B_q) + T_{add}(B_p+B_q)\right) \\ &+ \left(1 - \frac{1}{2^N}\right).p.q.\left(T_{com}(B_p+B_q)+T_{add}(B_p+B_q)\right) \end{aligned} \qquad (4.4)$$

L'approche parallèle du problème n'apporte plus de gain effectif dès que le nombre de processeurs 2^{N_0} vérifie :

$$\begin{cases} T_{Par}^{(N_0-1)}(P,Q) > T_{Par}^{(N_0)}(P,Q) \\ T_{Par}^{(N_0)}(P,Q) \leq T_{Par}^{(N_0+1)}(P,Q) \end{cases} \qquad (4.5)$$

En supposant que N_0 est suffisament grand pour que 2^{N_0} soit négligeable devant 1, et en tenant compte des complexités (4.1) (4.2) et (4.3), on obtient la borne approchée pour N_0 :

$$\boxed{N_0 \geq \mathrm{Log}_2\left(1 + \frac{2.p.(B_p+B_q).g(B_p+B_q)}{\tau.B_q + \alpha}\right)} \qquad (4.6)$$

Conclusion

La fonction g étant super-linéaire, seul un grand nombre de processeurs permet d'exploiter pleinement le parallélisme du problème lorsque les données sont de tailles importantes (que ce soit par le nombre de monômes ou la taille des coefficients).

VI. Algorithme de Karatsuba

Lorsque les polynômes sont denses, l'algorithme de Johnson n'est plus optimal. Le meilleur algorithme séquentiel actuellement connu est basé sur un calcul de FFT [AHU74a]. Il présente cependant l'inconvénient majeur de n'être vraiment intéressant que pour de très grands polynômes denses. L'algorithme de Karatsuba devient lui intéressant relativement rapidement, et il permet en outre de s'adapter au cas où les polynômes ne sont pas tout à fait denses.

Le but de cette étude est double :

- Expliquer l'intérêt du choix de la topologie en fonction de l'algorithme

- Montrer comment la méthodologie d'étude de complexité présenté sur la parallélisation de l'algorithme de Johnson peut s'adapter à d'autres algorithmes

Dans un premier paragraphe, l'algorithme séquentiel est rapidement présenté. Puis nous présentons la parallélisation de cet algorithme, et le choix de topologie qui en découle. Enfin, l'étude de complexité est effectuée, en utilisant le même modèle de complexité et la même méthodologie que celles adoptées pour l'algorithme précédent.

L'algorithme de Karatsuba est étudié de manière précise dans la deuxième partie de cette thèse. Ici, nous ne rappelons que brièvement les résultats : l'intérêt visé n'est pas tant l'algorithme lui-même que sa parallélisation.

VI.1. Présentation de l'algorithme séquentiel

Cet algorithme est dû à A. Karatsuba [KAO62]. Fateman le désigne à juste titre sous l'appellation algorithme du partage (*"split algorithm"*) [MŒ72].

Soient P et Q deux polynômes de $\mathbb{Z}[X]$. De façon à simplifier l'exposé, nous supposons que les deux polynômes sont de même degré n. Nous partageons alors les deux polynômes de la manière suivante :

$$\begin{cases} P(x) = P_H(x).x^{n/2} + P_L(x) \\ Q(x) = Q_H(x).x^{n/2} + Q_L(x) \end{cases}$$

$$\text{avec } \begin{cases} \text{degré}(P_H) < n/2 \\ \text{degré}(Q_H) < n/2 \end{cases} \quad \text{et} \quad \begin{cases} \text{degré}(P_L) < n/2 \\ \text{degré}(Q_L) < n/2 \end{cases}$$

Le produit R de P et Q peut alors s'écrire :

$$R = P_H.Q_H.x^n + (P_H.Q_H + P_L.Q_L + (P_H - P_L)(Q_L - Q_H)).x^{n/2} + P_L.Q_L$$

Ainsi, le problème initial de taille p est équivalent à trois problèmes de taille p/2. Ces trois produits de taille p/2 sont calculés en leur appliquant récursivement la méthode. Pour reformer le polynôme R, il faut alors effectuer :

- deux additions de taille n/2 pour calculer $P_H - P_L$ et $Q_L - Q_H$
- quatre additions de taille n pour reformer R

Complexité

Posons T(p) le temps pour calculer le produit de deux polynômes de degré n et T_{add} (respectivement T_{mul}) le temps d'une addition (respectivement d'une multiplication) sur $\mathbb{Q}$. La complexité de l'algorithme est alors :

$$T(n) = 3.T(n/2) + 5.n.T_{add}$$

Posant $n = 2^p$, on obtient : $\begin{cases} T(2^p) = 3.T(2^{p-1}) + 5.T_{add}.2^p \\ T(2^0) = T_{mul} \end{cases}$

D'où l'on tire : $T(2^p) = 3^p.T_{mul} + 10.T_{add}.(3^p - 2^p)$

Et donc : $T(n) = (T_{mul} + 10.T_{add}).n^{\log_2 3} - 10.n.T_{add}$

Algorithme de Karatsuba mixte

Théoriquement, l'algorithme est d'une complexité inférieure à celle de l'agorithme de Johnson. En pratique, elle n'est intéressante que si les polynômes sont suffisamment denses. Par ailleurs le contrôle nécessité est important : en pratique, il est donc intéressant d'arrêter la récursivité lorsque n devient trop petit (inférieur à une valeur N_{kar} à déterminer), et de faire les produits des petits polynômes alors obtenus par l'algorithme de Johnson.

Ce résultat est étudié de manière précise dans la deuxième partie, pour le cas de la multiplication d'entier. Ici, il est assez difficile de trouver théoriquement la valeur optimale de N_{kar}. La courbe ci-dessous permet de déterminer expérimentalement que cette valeur pour l'algorithme implanté dans PAC [BAL89] est (cf fig. 1) :

$$\boxed{N_{kar}=7}$$

Il est à noter que R. Mœnck trouve la même valeur pour son implantation [MŒ72].

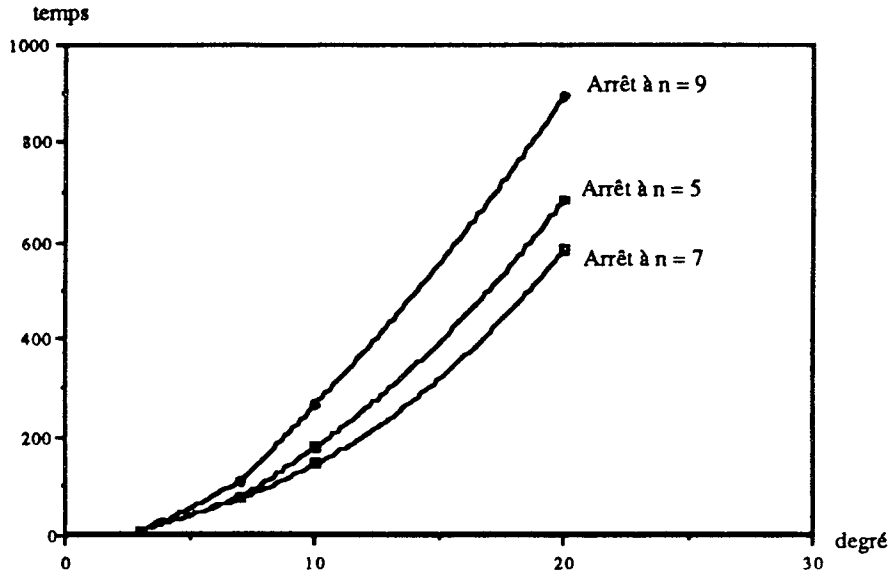


Fig. 1 Détermination expérimentale de N_{kar}
(Tests effectués sur des polynômes denses)

La figure ci-dessous montre le comportement des deux algorithmes (Johnson et Karatsuba) pour des polynômes denses. Les complexités expérimentales, calculées par meilleure approximation logarithmique, permettent de retrouver les complexités théoriques, à un facteur logarithmique près (dû au nombre de comparaisons, négligées dans l'étude).

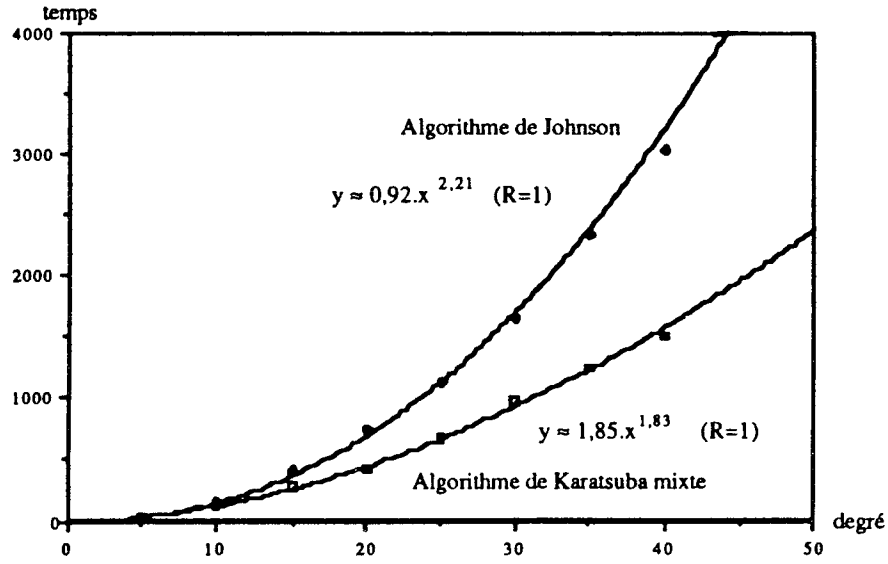


Fig. 2 Comparaison des algorithmes de Johnson et de Karatsuba
(Tests effectués sur des polynômes denses)

VI.2. Présentation de l'algorithme parallèle

L'algorithme de Karatsuba permet de découper un problème de taille n en trois sous problèmes de taille n/2. La parallélisation du problème est directe : à chaque étape, le processeur père génère trois sous-tâches; il en

répartit une sur chacun de ses deux voisins, exécute la sienne puis récupère les résultats de ses voisins pour les sommer avec le sien.
Il est important de constater que pour que l'algorithme de Karatsuba soit intéressant, il est essentiel que les polynômes à multiplier soient denses, et de même degré. Cette hypothèse est très restrictive en pratique, et réduit le champ d'applications. C'est pourquoi nous avons choisi d'adopter une stratégie de répartition basée sur un partage de type Karatsuba, et d'effectuer les calculs sur chaque nœud avec l'algorithme de Johnson, beaucoup mieux adapté au cas général des polynômes creux, et qui donne en pratique de bien meilleurs résultats.

Soient P et Q deux polynômes et soient P_L, P_H, Q_L, Q_H définis par :

$$\begin{cases} P(x) = P_H(x).x^{n/2} + P_L(x) \\ Q(x) = Q_H(x).x^{n/2} + Q_L(x) \end{cases}$$

$$\text{avec} \quad \begin{cases} \text{degré}(P_H) < n/2 \\ \text{degré}(Q_H) < n/2 \end{cases} \quad \text{et} \quad \begin{cases} \text{degré}(P_L) < n/2 \\ \text{degré}(Q_L) < n/2 \end{cases}$$

Le schéma du calcul du produit parallèle de P et Q est alors :

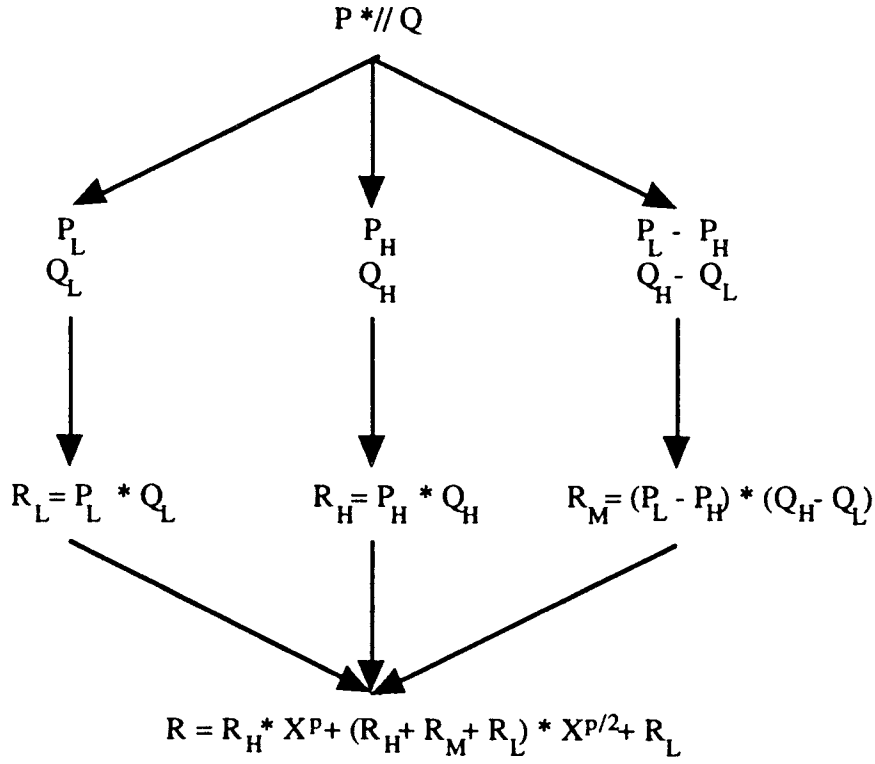


Fig. 3 Parallélisation de l'algorithme de Karatsuba

VI.3. Répartition - Choix de la topologie

A chaque étape, le problème est divisé en trois sous-tâches de même complexité. Il faut donc qu'à chaque étape, chaque processeur trouve deux voisins inoccupés et qu'il leur envoie à chacun une des sous-tâches.

La figure suivante montre la répartition des polynômes sur neuf processeurs en sachant que chaque processeur envoie P_H et Q_H à un de ses voisins, P_L et Q_L à un autre de ses voisins et qu'il calcule le produit $(P_H - P_L).(Q_L - Q_H)$. La hauteur de chaque nœud de l'arbre indique l'étape à laquelle s'effectue le transfert des polynômes.

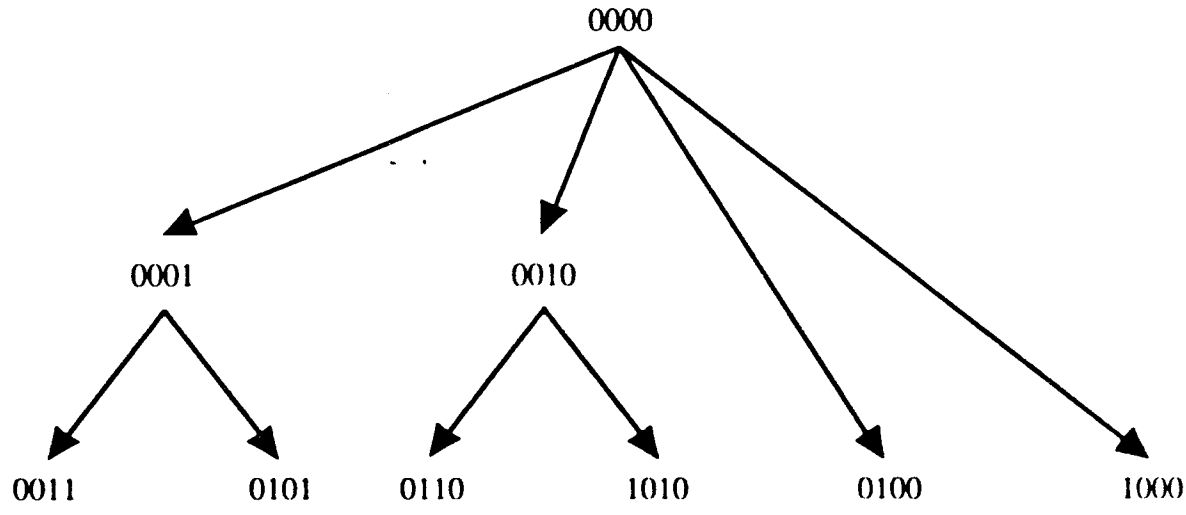


Fig. 4 Répartition de l'algorithme de Karatsuba sur neuf processeurs

La topologie, qui est toujours guidée par la parallélisation adoptée, est donc différente de celle choisie pour la méthode précédente. Cette parallélisation du produit de polynômes n'est pas du tout adaptée à la configuration hypercube puisque le nombre de voisins nécessaires pour chaque processeur est de 2.k pour réaliser k étapes (dans la méthode précédente, il fallait k voisins pour effectuer k étapes). La topologie idéale ici serait donc un réseau de 3^n processeurs, chaque processeur étant connecté à 2.n voisins.
Le tableau suivant donne le nombre de voisins nécessaires à chaque processeur pour un réseau à 3^n processeurs :

n	0	1	2	3	4	5
Nombre de processeurs	1	3	9	27	81	243
Nombre de voisins	0	2	4	6	8	10

Fig. 5 Topologie adaptée à la parallélisation de l'algorithme de Karatsuba

Remarque

La topologie hypercube du FPS-T20 sur lequel nous avons évalué l'algorithme permet donc d'aller jusqu'à n=2 avec ses 16 processeurs. La topologie exploitée correspond donc à 56% de la machine.

L'algorithme n'a pas été implanté sur le T40 (32 processeurs) : il aurait permis d'utiliser 27 des 32 processeurs, donc une exploitation à 84% de la topologie physique.
L'implantation est par suite plus complexe que celle de l'algorithme de Johnson.

Calculs nodaux

Une fois toute les tâches réparties, chaque processeur effectue son calcul en utilisant l'algorithme de Johnson : en effet, lorsque les polynômes sont creux, il est intéressant de n'effectuer qu'un certain nombre d'étapes de l'algorithme de Karatsuba.
Soient p_{min} et q_{min} (resp. p_{max} et q_{max}) les degrés des monômes de degré minimal (resp. maximal) de P et Q. Alors, le degré d de séparation entre P_H et P_L d'une part, Q_L et Q_H d'autre part, est défini par :

$$d = \frac{\text{Max}\,(p_{min}, q_{min}) + \text{Min}\,(p_{max}, q_{max})}{2}$$

La justification de cette découpe équilibrée est donné dans la deuxième partie, pour le cas entier.

VI.4. Complexité

Pour calculer la complexité de l'algorithme de Karatsuba parallèle, nous utilisons les mêmes notations que précédemment (§I), et le même modèle de complexité (§V). Nous supposerons d'abord que le coût des opérations de base ne dépend pas de la taille des données puis qu'il en dépend.
Nous calculerons les différentes complexités pour deux polynômes P et Q ayant respectivement p et q monômes. Nous nous placerons pour l'étude de l'efficacité asymptotique dans le cas : $p \geq q \gg 3^N$.

VI.4.1. Les opérations de base ne dépendent pas des données

Comme pour l'algorithme de Johnson en parallèle, trois étapes sont à distinguer : la répartition, le calcul, et la remontée.

VI.4.1.1. Répartition

A l'étape k, les processeurs de dimension k-1 découpent leur tâche P.Q en trois sous-tâches $P_L.Q_L$, $P_H.Q_H$, $(P_H-P_L).(Q_L-Q_H)$ et envoient les polynômes (P_L, Q_L) à un de leurs voisins, (P_H, Q_H) à un autre de leurs voisins et calculent $(P_H-P_L).(Q_L-Q_H)$.

En N étapes, le problème est réparti sur les 3^N processeurs.

Soit $T_1(p,q)$ le coût de la répartition des polynômes P et Q sur tous les processeurs utilisés. Par analogie à (2.1), on a :

$$T_1(p,q) = \sum_{i=1}^{N} 2.\left(\frac{p+q}{2^i}\right).T_{com} + \left(\frac{p+q}{2^i}\right).T_{add} = \left(1 - \frac{1}{2^N}\right).(p+q).(2.T_{com}+T_{add})$$

VI.4.1.2. Calcul

Soit $T_2(p,q)$ le coût du calcul des produits sur chaque processeur. Par analogie à (1.2), le coût du produit nodal d'un polynôme de taille p par un polynôme de taille q étant $p.q.T_{mul} + (p.q - 1).T_{add}$, on en déduit :

$$T_2(p,q) = \left(\frac{p.q}{4^N}\right).T_{mul} + \left(\frac{p.q}{4^N} - 1\right).T_{add}$$

VI.4.1.3. Remontée

Soit $T_3(p,q)$ le coût de la remontée du résultat vers le processeur racine. A l'étape k, les processeurs de dimension N-k+1 envoient leur résultat vers leur voisin de dimension N-k. Ces processeurs de dimension N-k+1 vont par couple, et les deux processeurs d'un couple envoient leurs résultats vers le même processeur père (*une histoire de famille, quoi !*). A chaque étape, les processeurs qui reçoivent doivent effectuer quatre additions de taille $p.q/4^i$, avant d'envoyer à leur père le résultat ainsi obtenu. On obtient donc :

$$\begin{aligned} T_3(p,q) &= \sum_{i=1}^{N} 2.\left(\frac{p}{2^i}.\frac{q}{2^i}\right).T_{com} + 2.4.\left(\frac{p.q}{4^i}\right).T_{add} \\ &= \frac{p.q}{3}.\left(1 - \frac{1}{4^N}\right).(T_{com} + 4.T_{add}) \end{aligned}$$

VI.4.1.4. Temps parallèle

Soit $T_{kar}(p,q)$ le coût total de la multiplication de P par Q en utilisant une découpe de type Karatsuba. On a :

$$T_{kar}(p,q) = T_1(p,q) + T_2(p,q) + T_3(p,q)$$

D'où l'on tire :

$$T_{kar}(p,q) = T_{com}.\left[2.(p+q).\left(1 - \frac{1}{2^N}\right) + \frac{p.q}{3}.\left(1 - \frac{1}{4^N}\right)\right] + T_{mul}.\frac{p.q}{4^N}$$

$$+T_{add}.\left[(p+q).\left(1 - \frac{1}{2^N}\right) + 4.\frac{p.q}{3}.\left(1 - \frac{1}{4^N}\right) + \frac{p.q}{4^N} - 1\right]$$

Dans le cas asymptotique $p \geq q \gg 3^N \gg 1$, on obtient donc :

$$\boxed{T_{kar}(p,q) = p.q.\left(\frac{1}{3}.T_{com} + \frac{4}{3}.T_{add} + \frac{1}{4^N}.T_{mul}\right)}$$

L'efficacité est ici difficile à définir, puisque la parallélisation n'a plus rien de commun avec l'algorithme séquentiel utilisé, qui serait ici un algorithme de Karatsuba avec un arbre d'appels de profondeur N, puis une multiplication par l'algorithme de Johnson.
Les limites du modèle théorique apparaissent : il est difficile de prendre en compte le comportement d'algorithmes dont la stratégie de diffusion est différente de la stratégie de calcul.

Il apparaît cependant que l'algorithme parallélisé par Karatsuba est plus efficace, dans le cas de polynômes suffisament denses (au sens d'équilibré), que l'algorithme classique de multiplication en parallèle : cela est justifié par l'ordre théorique de complexité.

VI.4.2. Les opérations de base dépendent des données

Nous utilisons les mêmes hypothèses et notations que lors de l'étude de la multiplication classique en parallèle.

VI.4.2.1. Répartition

Chaque processeur fait toujours les mêmes opérations on en déduit donc :

$$T_1(p,q) = \left(1 - \frac{1}{2^N}\right).\{\ T_{com}(Taille(P)) + T_{com}(Taille(Q))\ ) + 2.T_{add}(Taille(P)) + 2.T_{add}(Taille(Q))\ \}$$

VI.4.2.2. Calcul - Remontée - Temps parallèle

Les autres calculs se font de la même manière et nous obtenons :

$$T_2(p,q) = \frac{p.q}{4^N}.\ \{\ T_{mul}(\ B(P), B(Q)\ ) + T_{add}(\ B(P) + B(Q)\ )\ \}$$

et

$$T_3(p,q) = \frac{p.q}{3}.\left(1 - \frac{1}{4^N}\right).\ \{\ T_{com}(\ B(P) + B(Q)\ ) + 4.T_{add}(\ B(P) + B(Q)\ )\ \}$$

En simplifiant les termes du premier ordre et en négligeant 1 devant 4^N, nous obtenons pour le temps parallèle :

$$T_{kar}(p,q) = \frac{p.q}{3}.\left[\frac{3.T_{mul}(B(P), B(Q))}{4^N} + T_{com}(B(P) + B(Q)) + 4.T_{add}(B(P) + B(Q)) \right]$$

Le temps trouvé est alors -relativement au nombre de processeurs utilisés- inférieur à celui de l'algorithme classique de multiplication en parallèle.

VII. Résultats expérimentaux

VII.1. Algorithme de Johnson parallèle

Expérimentalement plus le nombre de processeurs est important, plus le temps de calcul est faible (*heureusement...*). La courbe ci-dessous nous donne les temps de calcul du produit de deux polynômes en fonction de la taille de ces deux polynômes pour 2,4,8,16,32 processeurs. La taille limite des problèmes à partir de laquelle le parallélisme devient intéressant se situe aux alentours de 200 monômes : ce n'est guère étonnant, le problème étant très déséquilibré (coefficients énormes pour les termes de degrés moyens, très petits pour ceux de degrés extrêmes).

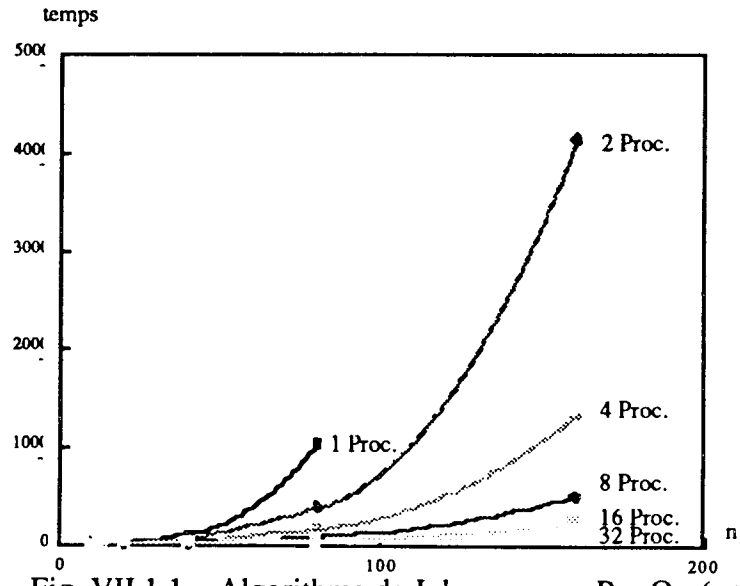


Fig. VII.1.1. Algorithme de Johnson pour P = Q= $(x+y)^{5.n}$

Les courbes précédentes permettent d'obtenir les efficacités suivantes :

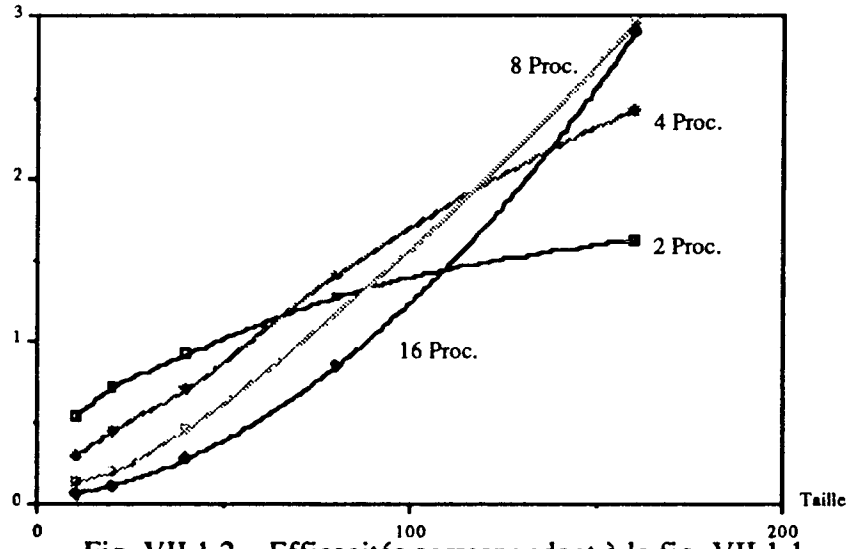


Fig. VII.1.2. Efficacités correspondant à la fig. VII.1.1.

Comme l'étude théorique l'a montré, à nombre de processeurs constant, l'efficacité croît jusqu'à une asymtote limite (cf. pour 2 ou 4 processeurs). A une taille donné de problèmes, correspond un optimum pour le nombre de processeurs : dans l'exemple ci-dessus, pour une taille de 150, cet optimum est de 8 processeurs (et non de 16).

VII.2. Comparaison avec l'algorithme Karatsuba parallèle

Les expériences ont été réalisées en comparant la distribution stantard sur 16 processeurs (division en 16 tâches réparties en 4 étapes) à la distribution de Karatsuba sur 9 processeurs (division en 9 tâches réparties en 3 étapes). L'exemple considéré est le même que précédemment.

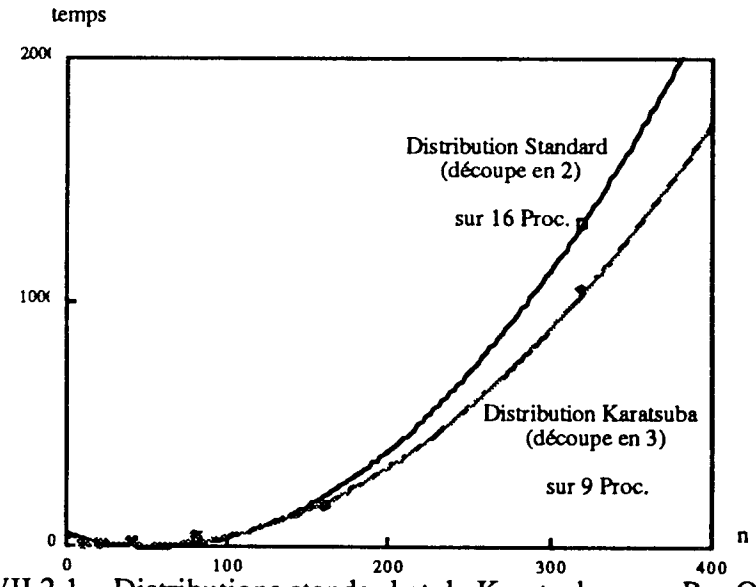


Fig. VII.2.1. Distributions standard et de Karatsuba pour $P = Q = (x+y)^{5.n}$

Expérimentalement, pour des problèmes denses de taille suffisament grande, la distribution de Karatsuba est plus intéressante.

Ce résultat est confirmé par la courbe des efficacités comparées des deux algorithmes pour les tests précédents.

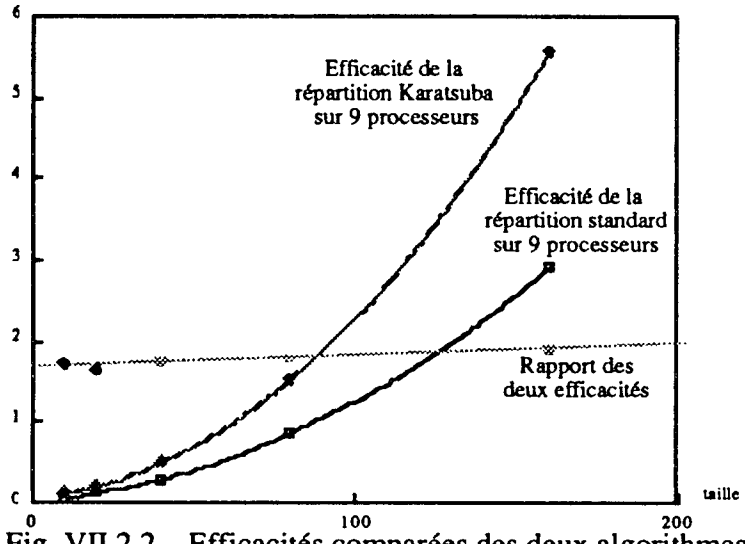


Fig. VII.2.2. Efficacités comparées des deux algorithmes

Il apparaît clairement que sur l'exemple considéré, la répartition Karatsuba est mieux adaptée que la répartition standard.

VII.3. Conclusion

Les résultats expérimentaux montrent que l'efficacité peut devenir supérieure à un. Bien que surprenante au premier abord, cette remarque s'explique par le fait que les calculs effectués ne sont pas les mêmes dans les algorithmes séquentiels et parallèles. L'efficacité expérimentale ne correspond donc pas tout à fait à l'efficacité théorique. Ce phénomène est aggravé par une caractéristique essentielle du Calcul Formel : les données n'étant pas conditionnées, les temps des opérations de base ne sont pas constants.

Cette constatation peut être illustrée par l'exemple suivant :
Supposons que l'on veuille calculer le produit de quatre entiers r = a.b.c.d
Un calcul séquentiel direct évalue : r = ((a.b).c).d
Un calcul distribué sur deux processeurs évalue le résultat sous la forme :

$$r_1 = a.b,\ r_2 = c.d$$
$$r = r_1.r_2$$

Les deux algorithmes ne sont pas véritablement équivalents : mais il est souvent peu naturel de programmer l'algorithme séquentiel comme l'algorithme parallèle.
Soit M(x,y) le coût du calcul du produit de deux entiers x et y.

D'après la partie II, on a : M(x,y) est une fonction sous-linéaire, croissante.

L'efficacité du calcul est alors : $e = \frac{M(a,b) + M(a.b,c) + M(a.b.c,d)}{2.(M(a,b) + M(c,d) + M(a.b,c.d))}$

La fonction M étant croissante, selon les valeurs de a, b, c et d, l'efficacité peut varier, et devenir supérieure à un.
Ainsi, lorsque d est grand et a, b et c petits, l'efficacité tend vers zéro. Par contre, si a est grand et b, c et d petits, elle tend vers un.

Par ailleurs, le temps passé en gestion mémoire permet d'expliquer les valeurs "anormales" de l'efficacité. En effet, plus les exemples sont gros et plus morcelée est la mémoire (même si il n'y a que peu de libérations). La mémoire nodale sur la machine utilisée pour les exemples n'étant que de un Mo, le temps passé en allocation devient important rapidement. Il est donc essentiel de rapporter les efficacités à la taille des tâches traitées. Les modèles d'efficacité de Gustavson [DON86] [TOU89] permettent de prendre en compte en partie ces problèmes.
Cette justification est apparue clairement lors du tracé de ces courbes : pour chaque point, le temps peut varier d'un facteur très important si la même expression est évaluée deux fois de suite, sans réinitialisation du système.

Chapitre III

MODELE PARALLELE DE PAC

Le chapitre précédent, avec l'exemple du produit de polynômes, montre que le parallélisme est d'autant plus efficace que les problèmes traités sont importants. Mais il prouve également que, pour tirer pleinement parti du parallélisme sur un problème donné, il est nécessaire de disposer d'un suffisamment grand nombre de processeurs. La résolution parallèle avec peu de processeurs limite nécessairement la taille des problèmes traités.

I. Modèle nodal

La taille d'un problème du Calcul Formel n'est souvent pas caractérisée par un seul paramètre, mais dépend de plusieurs facteurs.
Ainsi, le coût du produit de deux polynômes dépend tout autant du nombre de monômes que de la taille des coefficients. Si P et Q ont peu de monômes, et que les coefficients de chacun des monômes ont plusieurs millions de chiffres, leur produit ne doit pas être évalué parallèlement de la même façon que dans le cas où P et Q sont des polynômes à coefficients dans $\mathbf{Z}/p.\mathbf{Z}$ (p petit), avec plusieurs millions de monômes.

Une approche parallèle doit donc être liée à la formulation même du problème à traiter, avec ses données. Avant de passer à une parallélisation fine des tâches de moindre coût, améliorer le temps de résolution d'un problème passe nécessairement par une étude de la parallélisation des tâches les plus coûteuses.
En Calcul Formel, des tâches aussi anodines qu'un produit d'entiers peuvent s'avèrer très coûteuses, aussi bien en temps qu'en espace mémoire. Il est donc essentiel, pour aborder le parallélisme d'un problème à son niveau le plus haut, que chacune des unités distribuées aient une capacité de calcul et de mémoire suffisamment grande pour leur donner une autonomie dans le traitement des tâches.

Il est donc naturel de s'orienter vers un parallélisme à grosse granularité, dans laquelle chaque nœud est équipé de différents éléments :

- unités arithmétiques rapides : il est donc possible d'effectuer des calculs sur chaque nœud. Ces calculs peuvent être cablés (unités vectorielles, opérateurs arithmétiques on-line [GUY89], polynômieur [MUL88]) ou simulés par un ensemble de primitives arithmétiques efficaces, disponibles nodalement.

♦ espace mémoire important : chaque nœud peut stocker de manière autonome des objets de taille importante. Il possède donc son propre système de gestion mémoire (allocation, libération et ramasse-miettes). Ainsi, les calculs nodaux pourront être effectués sur des données de taille importante, et par suite, des problèmes de grandes tailles -ceux pour lesquels le parallélisme est un outil primordial- pourront être traités.

Sous l'hypothèse que le temps de communication est une fonction linéaire de la taille des données, les opérations sous-linéaires et certaines opérations linéaires (comme l'addition d'entiers - lorsque le coût de communication de base est supérieur à celui d'une addition élémentaire-) pourront ainsi être effectuées plus rapidement nodalement qu'en parallèle sur plusieurs nœuds. Plus généralement, le sur-coût apporté par les communications entraîne que pour n'importe quel problème (même sur-linéaire), il existe un seuil au dessous duquel le parallélisme ne peut rien apporter : il est donc essentiel de définir pour chaque problème quel est le seuil de "rentabilité parallèle" qui lui est associé.

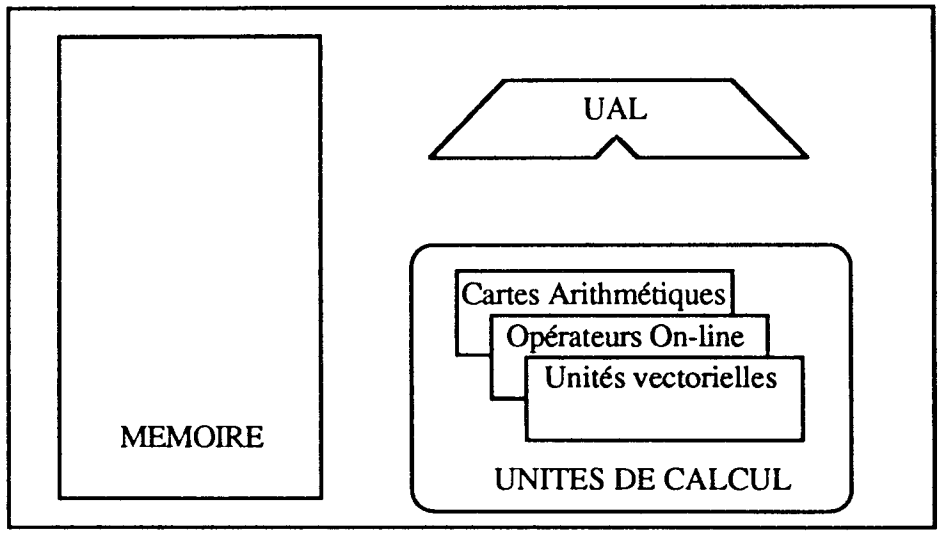


Fig. 1 Modèle Nodal

II. Modèle Parallèle
PAC : un co-processeur algébrique parallèle

Disposer d'un grand nombre de processeurs, chacun doté d'une autonomie suffisante, conduit vers le modèle des architectures massivement parallèles, à grosse granularité, seul modèle existant répondant aux contraintes énoncées ci-dessus.
Il est cependant fondamental de constater que le modèle choisi n'a d'intérêt que si le problème à traiter est suffisamment conséquent pour en interdire une résolution séquentielle.

Cette vision est quelque peu en opposition avec celle des systèmes de Calcul Formel traditionnels : il ne s'agit pas ici de mettre à la disposition de l'utilisateur une machine sur-puissante, munie d'un système interactif, alors que la plupart des problèmes généralement traités ne peuvent tirer aucun parti de cette puissance, bien au contraire.

C'est pourquoi PAC se présente comme un co-processeur algébrique parallèle, destiné à la résolution de problèmes qu'un système séquentiel ne peut traiter.

PAC se présente comme une librairie installée sur un périphérique parallèle. Il est constitué :

- des primitives nodales : elles permettent de doter chaque nœud d'une arithmétique suffisamment puissante pour lui permettre la manipulation d'expressions algébriques même complexes. En outre, elles permettent la manipulation de symboles.

- des algorithmes parallèles : adaptés à des problèmes fondamentaux spécifiques, il permettent la résolution de problèmes de grande taille.

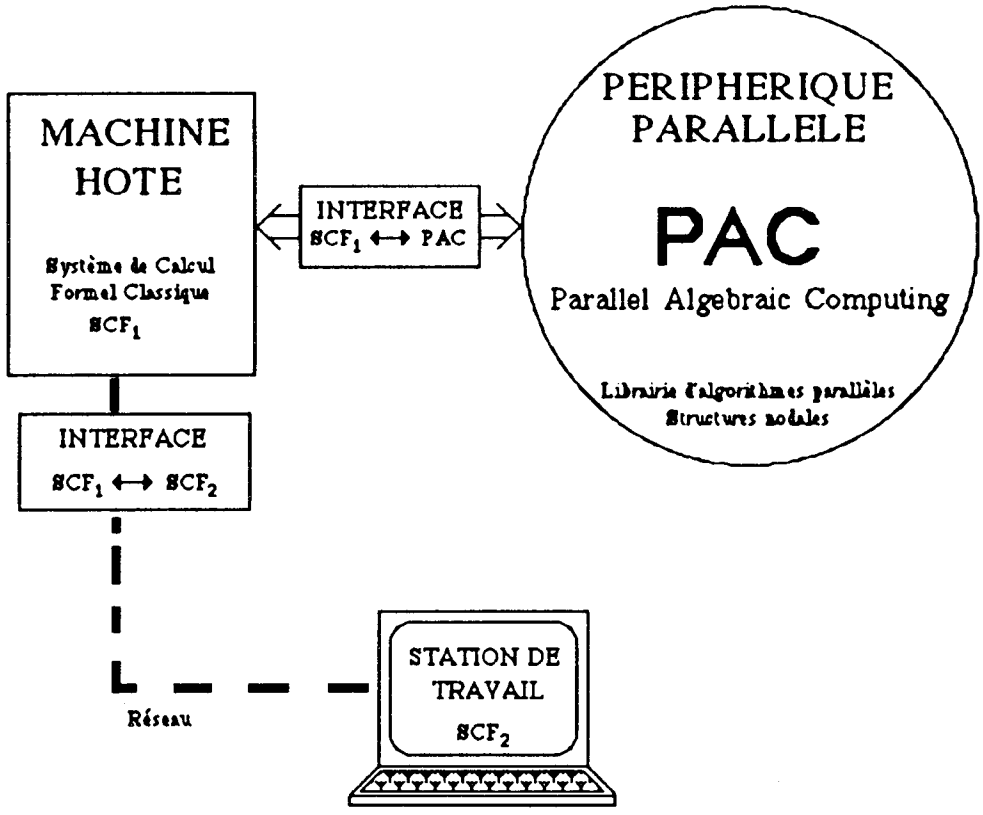


Fig. 2 PAC : un co-processeur parallèle pour le Calcul Formel

Il existe deux façons d'utiliser PAC :

♦ comme un outil de base pour le développement d'applications parallèles : il est possible d'utiliser aussi bien les algorithmes nodaux ou parallèles pour développer d'autres applications destinées à être directement exécutées sur le périphérique parallèle. Certaines applications de la librairie sont données à titre de modèle pour développer d'autres algorithmes parallèles.

♦ comme un co-processeur de traitement : connecté à un système de Calcul Formel classique, il peut être utilisé de manière transparente pour la résolution d'un problème spécifique donné.

III. Etat actuel de PAC (du moins le 15 Novembre 1989)

Une première version de PAC a été développée sur l'hypercube FPS-T20, qui est devenu depuis T40. Un rapport technique [ROC89] décrit l'ensembles des primitives nodales, et donne des schémas directeurs pour le développement d'algorithmes. De manière générale, le système comprend :

♦ Sur chaque nœud, différents modules qui permettent d'effectuer les opérations suivantes :

- utilisation de symboles
- gestion d'objets : allocation, libération, envoi et réception d'objets
- arithmétique rationnelle
- arithmétique polynomiale dans $\mathbb{Q}[X_1,...X_n]$

♦ Sur le périphérique parallèle, différents programmes qui effectuent :

- Produit de polynômes [BAL89]
- Résolution de systèmes linéaires [VIL88]
- Calcul de formes normales [SIE89]
- Calcul de bases de Gröbner booléennes [SEN89]
- Algorithme de Buchberger dans $\mathbb{Q}[X_1,...X_n]$ [PER89]

♦ Un interpréteur : installé sur le périphérique, il facilite le développement d'applications

♦ Un interface, qui permet un appel depuis Reduce de manière transparente. Cet interface existe dans une version préliminaire : son plus grand intérêt est... de montrer que c'est possible et que *ça marche...*

CONCLUSION ET PERSPECTIVES

Les différents exemples étudiés ([BAL89] [PER89] [ROC89] [SEN90] [SIE90] [VIL89]) montrent que le parallélisme inhérent à de nombreux problèmes du Calcul Formel peut être exploité sur une machine massivement parallèle, et ce d'autant plus efficacement que le problème est de taille conséquente. L'étude de la parallélisation d'un problème donné doit alors tenir compte des contraintes dues au modèle, et notamment des communications de données, pour décrire de manière fidèle le comportement d'un algorithme. L'expérimentation sur une machine parallèle réelle s'avère alors essentielle pour valider ou non l'implantation.

Les choix des topologies d'exécution, et des stratégies de répartition sont fondamentaux. L'exemple de l'adéquation topologie/algorithme, mise en évidence pour le produit de polynômes - algorithmes de Johnson (Chap.III-§III) et de Karatsuba (Chap.III-§VI) -, est significatif. Concernant le même problème, le meilleur choix de répartition, à savoir le polynôme à découper (Chap.III-§V), est caractéristique, et prouve l'importance d'une étude précise de l'algorithme avant l'implantation.

Dans son état actuel, PAC est implanté sur le FPS-T40 : il permet de travailler nodalement dans $\mathbb{Q}$ ou $\mathbb{Q}[S]$ ($S=\{x,y,\ldots\}$), et de communiquer les structures associées. Au niveau de cette arithmétique, seul le produit de polynômes (Chap. III) est implanté en parallèle : le parallélisme d'un problème peut donc être décrit de manière statique, en associant une tâche précise à chaque processeur. En conséquence, seuls des algorithmes bien synchronisés (donc bien conditionnés en fonction des données) permettent de dégager des efficacités intéressantes.
Cette contrainte est relativement limitative : le non-conditionnement intrinsèque des problèmes du Calcul Formel (non numériques) conduit à envisager une répartition dynamique des tâches, seule capable d'assurer un équilibre entre les activités des différents processeurs du réseau. Le portage de PAC sur la machine Méga-Node de Telmat s'inscrit dans ce cadre : profiter des possibilités d'expression d'un pseudo-parallélisme sur chaque nœud pour intégrer des primitives de répartition dynamique des tâches sur le réseau.

DEUXIEME PARTIE

ARITHMETIQUE

PRECISION INFINIE

SOMMAIRE

INTRODUCTION

La partie précédente a permis de dégager un modèle parallèle dans lequel chaque unité nodale possède une certaine autonomie en calcul : c'est pourquoi chaque processeur est muni d'une arithmétique rationnelle et polynômiale. La volonté de s'intéresser à des problèmes de taille conséquente nécessite que ce noyau nodal soit particulièrement performant.

Disposer d'une arithmétique exacte est fondamental, puisqu'elle permet, outre l'application d'algorithmes mathématiques, la résolution de problèmes pour lesquels les méthodes numériques - approchées - s'avèrent non satisfaisantes. Ainsi, résoudre un grand système linéaire mal conditionné est difficile sans outils permettant de contrôler les erreurs numériques [MUL89][FRA89]. Plus généralement, certains problèmes ne peuvent être résolus qu'avec des calculs exacts : le calcul des formes normales (Frobenius) de matrices, dépendant éventuellement de la structure algébrique de la matrice est un problème qui ne peut être traité que par des algorithmes formels [OZE87]. Le problème, essentiel en Automatique, du calcul de la forme d'Hermite d'une matrice en est un autre exemple [SIE89].

Le problème est ici double. D'une part, il s'agit de dégager les algorithmes de calcul en précision infinie les plus rapides sur l'architecture parallèle modèle définie précédemment, et donc d'étudier précisément le problème des communications de données. D'autre part, l'architecture de base des processeurs arithmétiques évolue sans cesse : le faible temps de cycle des processeurs RISC, la puissance de calcul des unités vectorielles permettent de résoudre de plus en plus rapidement les problèmes : encore faut-il que les algorithmes soient adaptés.

Après une étude sur la structure de données la mieux adaptée au modèle parallèle, la représentation interne des entiers est définie. Puis, les méthodes de calcul classiques sont décrites, ainsi que la manière de les combiner pour obtenir les algorithmes les plus efficaces pour des rationnels quelconques. Enfin, l'implantation de cette arithmétique sur un processeur vectoriel est discutée.
La plupart des études théoriques conduisent à des algorithmes permettant d'effectuer des opérations sur des entiers ayant le même nombre de chiffres. Il est donc essentiel d'étudier les meilleurs algorithmes pour le cas -le plus courant en pratique- où les entiers sont de tailles différentes : l'étude de la multiplication et de la division d'entiers présentée dans cette partie s'inscrit dans ce cadre.
L'importance du calcul du pgcd en arithmétique rationnelle, du fait même de sa forte complexité théorique et pratique, implique une étude particulière de ce problème, qui conduit à la présentation d'un algorithme spécifique.

Chapitre I

PRINCIPES DE BASE

I. Primitives de Calcul Machine

Nous rappelons ici différentes caractéristiques de l'arithmétique machine, de façon à dégager des structures générales dont on peut disposer sur la plupart des processeurs [MUL89].

I.1. Arithmétique entière

Dans la plupart des unités arithmétiques modernes (par exemple Motorola 68000 [JAU84], ou Inmos T414 [TRA85] et T800), les opérations arithmétiques de base sont implantées de façon à permettre une récupération des dépassements de capacité lors d'opérations avec des entiers non-signés - *overflow* -.

Soit M le nombre de bits d'un entier standard (en général M est la taille du mot machine). Pour représenter un entier "machine", on utilise alors comme représentation standard la notation "non signée".

Le mot $(b_{M-1}\ b_{M-2} \ldots b_1\ b_0)$ correspond alors à l'entier positif : $\sum_{i=0}^{M-1} b_i \cdot 2^i$

L'ensemble U des entiers non signés représentés est donc :

$$U = \{ 0, 1, \ldots, ß-1 \} \qquad \text{avec } ß = 2^M$$

Les opérations d'addition et de multiplication sont alors définies dans U comme retournant le résultat de l'opération dans **N** modulo ß. Le quotient (a/b) et le reste (a%b) dans U (a,b ∈ U) correspondent au quotient et au reste de la division euclidienne de a par b dans **N** : les opérandes appartenant à U, le reste et le quotient lors de la division euclidienne appartiennent aussi à U.

Remarque :

Pour permettre le calcul avec des entiers négatifs, on utilise une autre représentation : la notation en *complément à deux.*

Le mot $(b_{M-1}\ b_{M-2} \ldots b_1\ b_0)$ correspond alors en notation "complément à 2" à l'entier signé : $-b_{M-1}.2^{M-1} + \sum_{i=0}^{M-2} b_i.2^i$

L'ensemble des entiers signés représentés est donc :

$$S = \{ -2^{M-1}, \ldots, 0, \ldots, 2^{M-1} - 1 \}$$

Sur S, les opérations d'addition, soustraction, multiplication, division et reste sont définies. Elles ne correspondent aux opérations sur $\mathbf{Z}$ que lorsque le résultat de l'opération dans $\mathbf{Z}$ appartient à S.

I.2. Arithmétique étendue

Pour que les opérations arithmétiques dans U correspondent aux opérations arithmétiques dans $\mathbf{N}$, les opérations arithmétiques sur U sont étendues, et donnent alors un résultat sur 2.M bits, dans l'ensemble U_2 :

$$U_2 = \{ 0, 1, \ldots, ß^2 - 1 \}$$

Un résultat est alors stocké sur deux mots. Cette arithmétique étendue est essentielle : elle permet d'effectuer des opérations sur les entiers machine en récupérant éventuellement les retenues ou les dépassements occasionnés par le calcul, et cela pour un coût analogue au coût des opérations arithmétiques machines.

Il est clair que les instructions qui permettent de nommer les opérations en arithmétique étendue dépendent du processeur. Néanmoins, elles réalisent toujours les mêmes types de calcul, et nous pouvons donc considérer de manière générale que l'on dispose des opérations suivantes :

LONGADD (op_gche, op_dte, retenue_in) → retenue_out, résultat

avec
- retenue_in ∈ {0,1}
- résultat = (op_gche + op_dte + retenue_in) mod ß
- retenue_out = (op_gche + op_dte + retenue_in) div ß

LONGDIFF (op_gche, op_dte, retenue_in) → retenue_out, résultat

avec
- retenue_in ∈ {0,1}
- résultat = (op_gche - op_dte - retenue_in) mod ß
- retenue_out = $\mathbf{1}$(op_gche ≥ (op_dte + retenue_in))

LONGPROD (op_gche, op_dte, retenue_in) → résultat_hi, résultat_lo

avec
- résultat_lo = (op_gche x op_dte + retenue_in) mod ß
- résultat_hi = (op_gche x op_dte + retenue_in) div ß

LONGDIV (dividende_hi, dividende_lo, diviseur) → quotient, reste

avec
- quotient = (dividende_hi x ß + dividende_lo) div ß
- reste = (dividende_hi x ß + dividende_lo) mod ß
- On suppose que 0 ≤ quotient < ß

SHIFTRIGHT (hi.in, hi.lo, shift) → hi_out, lo.out

$$\text{avec} \quad \begin{cases} lo\_out = ((hi.in \times ß + lo.in) \text{ div } 2^{shift}) \text{ mod } ß \\ hi\_out = ((hi.in \times ß + lo.in) \text{ div } 2^{shift}) \text{ div } ß \end{cases}$$

SHIFTLEFT (hi.in, hi.lo, shift) → hi_out, lo.out

$$\text{avec} \quad \begin{cases} lo\_out = ((hi.in \times ß + lo.in) \times 2^{shift}) \text{ mod } ß \\ hi\_out = ((hi.in \times ß + lo.in) \times 2^{shift}) \text{ div } ß \end{cases}$$

Le tableau ci-dessous donne l'équivalence entre ces fonctions et les mnémoniques assembleur pour le transputer T414 d'Inmos [TRA85] et le processeur 68000 de Motorola [JAU84]. Le nombre de cycles est difficile à définir pour le 68000 (CISC), car il dépent de l'expression dans laquelle figure l'opération.

Fonction	Mnémonique correspondante		Nombre de cycles
	T414	68000	T414
Accès param.			8
addition	add	add	1
LONGADD	lsum	addx	3
soustraction	diff	sub	1
LONGDIFF	ldiff	subx	3
multiplication	prod	muls	39
LONGPROD	lprod	mulu[1]	34
reste	rem	divs	38
division	div	divs	40
LONGDIV	ldiv	divu	36
shift droite (n)			3+n
SHIFTRIGHT(n)	lshr	lsr	n<32 4+n n>31 n-25
shift gauche (n)			3+n
SHIFTLEFT (n)	lshl	lsl	n<32 4+n n>31 n-24

Fig I.1 Opérations en Arithmétique non signée étendue

Les opérations en arithmétique étendue serviront de brique de base à la construction du calcul en précision infinie, puiqu'elles permettent des calculs exacts sur l'ensemble U des entiers non signés, et cela avec la même rapidité que les opérations signées.

Remarque :

Pour le noyau arithmétique du système SAC2 [COL80], l'ensemble U est défini à partir de S - ensemble des entiers-machine signés -, en prenant une restriction de l'ensemble S telle que le résultat d'une opération à opérandes dans U ait toujours un résultat dans S.

Si ß est la base machine ($S = \{x \in \mathbf{Z} / 0 \leq |x| < \beta\}$), la base δ choisie pour la décomposition d'un entier quelconque ($U = \{x \in \mathbf{Z} / 0 \leq |x| < \delta\}$) doit donc vérifier : $\forall(a,b,c) \in U^3 \ (a.b)+c \in S$

D'où l'on déduit : $$\delta \leq \frac{\sqrt{4.ß-3}+1}{2}$$

Notations
Nous désignerons désormais par :

- τ_{add} : coût de LONGADD ou LONGDIFF (supposés égaux)
- τ_{mul} : coût de LONGPROD
- τ_{div} : coût de LONGDIV
- τ_{shift} : coût de SHIFTLEFT ou SHIFTRIGHT (supposés égaux)

II. Objets de base du Calcul Formel

La structure liste :

Nous avons vu que l'un des principes de base du Calcul Formel est d'autoriser la construction de domaines imbriqués. La structure de liste est classiquement utilisée pour représenter récursivement des objets sous un format arborescent adapté.

C'est ainsi que la couche élémentaire du système SAC 2 -écrit en Fortran- (le noyau *List Processing System*) est constitué d'un gestionnaire de liste. Dans les systèmes plus modernes, construits autour de Lisp (Reduce, Macsyma [DST87], Scratchpad II [SSV87]) ou de C (Maple), la manipulation de listes est particulièrement aisée.

La structure de liste constituée uniquement de paires, dont les éléments peuvent être des pointeurs sur d'autres paires, présente en outre l'avantage d'être relativement facile à gérer en mémoire : l'allocation de blocs de taille constante peut être réalisée facilement à l'aide d'un buddy-system [AHU83][ASS85].
Par contre, le stockage d'un objet en liste nécessite une place mémoire relativement importante: outre les données, il est nécessaire de stocker les différents pointeurs. L'objet pouvant être stocké de manière éparse en mémoire, l'accès à certains champs peut s'avérer coûteux.
Pour limiter la présence de pointeurs vers l'élément suivant, lorsque cela est possible, certaines techniques ont été élaborées. La technique cdr-coding [SIR88] permet par exemple d'éviter de stocker dans la deuxième partie d'une cellule l'adresse de la cellule consécutive lorsque celle-ci est physiquement adjascente - contiguë -.

La structure tableau

Un autre mode de structure possible, plus proche de la mémoire physique, est le stockage sous format de tableau : le gain de place est effectif, mais la manipulation d'objets est souvent lourde (cf. Altran). Outre les problèmes liés a la gestion mémoire de blocs de taille quelconque, l'un des gros problèmes provient de la difficulté à estimer la taille d'un objet lors de son allocation. Ainsi, quelle taille faut-il pour stocker le produit de deux polynômes, connaissant les deux polynômes opérandes, mais sans faire explicitement leur multiplication ?

Un compromis

La dichotomie liste/tableau est trop brutale : le choix d'une structure est en fait lié de très près au problème traité, de façon à en diminuer la complexité. R.T. Moenck [MŒ70] montre que pour les opérations classiques en arithmétique polynômiale, un stockage en tableau judicieux permet d'obtenir de très bons résultats aussi bien au niveau de l'occupation mémoire qu'au niveau du coût d'accès aux champs. Le problème inhérent reste l'allocation dynamique de la place mémoire pour le stockage d'un polynôme.

Il est donc fondamental de définir, dans le cadre du modèle parallèle choisi, la structure la mieux adaptée à la représentation des objets. L'idée de base est de chaîner dans des listes des éléments de base stockés en tableaux. Cette structure peut être vue comme un intermédiaire entre la liste plate et le tableau contigu.

III Contraintes en liaison avec le modèle

Le modèle défini dans la première partie est massivement parallèle. Sur chaque nœud, on devra pouvoir manipuler des objets plus ou moins complexes selon la granularité du problème. Il s'agit ici de définir la structure nodale générale la mieux adaptée à l'architecture modèle.

Contraintes liées aux communications inter-nodales

Le temps pour communiquer n mots mémoire contigus entre deux nœuds voisins est : $T_{com} = \alpha + n.\tau$ avec $\tau \gg \alpha$ [SAA86]

Il est donc préférable de communiquer une donnée stockée sous forme de tableau, que sous forme de liste à cellules plus ou moins importantes.

Contraintes liées à la communication par rendez-vous

Lorsque deux nœuds veulent communiquer un objet par rendez-vous, il est nécessaire qu'ils soient synchronisés : le récepteur (resp. l'émetteur) doit

attendre l'émetteur (resp. le récepteur) jusqu'à ce qu'il soit prêt à lui envoyer (resp. réceptionner) l'objet.

Si l'objet est réparti en mémoire (format liste ou suite de tableaux ...), la synchronisation doit avoir lieu pour chacun des éléments de l'objet. L'accès aux différents éléments nécessitant du calcul (calcul d'adresse pour l'émetteur, allocation de mémoire pour le récepteur), le temps total de l'émission est perturbé de manière importante par ces synchronisations. Si l'objet est stocké de manière contiguë, une synchronisation entre les processeurs d'envoi et de réception est suffisante -la première qui précède la communication-.

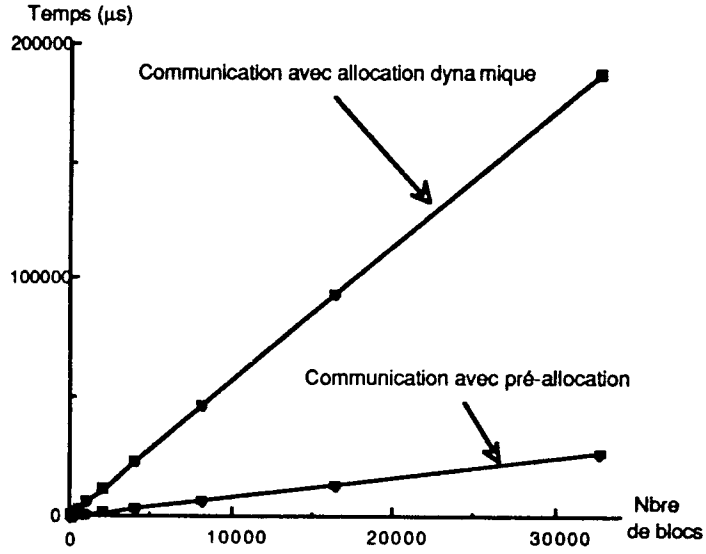


Fig II.3.1 Communication de 32768 octets découpés en blocs de taille égale
Mesures effectuées sur l'hypercube FPS T20

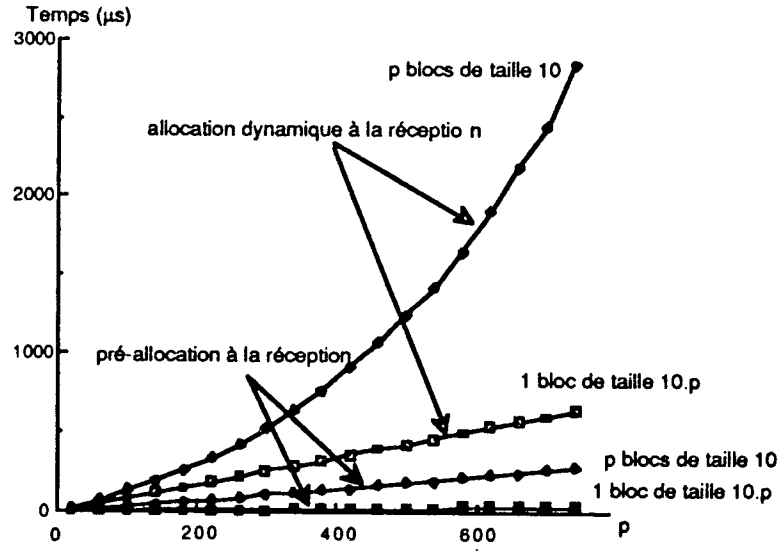


Fig II.3.2 Distribution de données sous format liste ou tableau
Mesures effectuées sur l'hypercube FPS T20

Il est donc clair que le stockage contigu est le mieux adapté aux communications inter-nodales par rendez-vous :

- minimisation du temps de communication pure
- les allocations dynamiques lors de la réception retardent le processeur émetteur : il est donc essentiel d'allouer la zone à recevoir avant la communication (ce qui est incompatible avec le concept de listes), ou d'utiliser des zones tampons pour la réception des objets.

Contraintes liées à l'allocation mémoire pour chaque nœud

Le volume de mémoire manipulée en Calcul Formel est souvent énorme : il est donc essentiel de gérer dynamiquement la mémoire sur chaque nœud. Pour cela, on peut distinguer deux genres d'objets :

- les objets *temporaires* : un tel objet est utilisé comme valeur intermédiaire lors d'un calcul "élémentaire". Sa valeur a donc une portée locale au calcul, et la place qu'il occupe sera libérée une fois le calcul terminé.

- les objets *permanents* : leur valeur est essentielle en dehors de l'environnement dans lequel elle a été calculée.

Une valeur est donc considérée comme temporaire si entre l'instant où un emplacement mémoire a été alloué pour son stockage et celui où cette place est libérée, il n'y a pas (ou "peu") eu d'autres allocations et libérations de mémoire.

Le temps d'allocation est lié au caractère temporaire/permanent d'un objet. L'allocation et la libération d'un objet temporaire a effet un faible coût, et surtout ne perturbe pas la mémoire. Par contre, la libération d'objets permanents morcelle la mémoire.

Minimisation du temps d'allocation :

Il est facile d'allouer ou de libérer des blocs de taille fixe - par exemple des doublets -, et ce en temps constant [AHU83]. L'allocation peut se faire en allouant le dernier bloc libéré par exemple -LIFO-, ou le premier -FIFO-. Un ramasse-miettes peut-être activé lorsqu'il n'y a plus d'espace libre, de façon à libérer les blocs alloués qui ne sont plus utilisés.

L'avantage d'une telle gestion mémoire est sa souplesse d'utilisation. Mais elle présente des inconvénients majeurs :

1° les objets d'une taille quelconque sont inévitablement morcelés en mémoire, découpés en blocs de taille fixe. L'accès direct aux différents champs de l'objet -par incrément à partir d'une adresse de début- est donc

impossible : on ne peut accéder en général à un champ que par adressage indirect.

Ex. 1 : Supposons que l'on désire stocker la décomposition en base ß d'un entier naturel N. Si l'on dispose de blocs de taille 2 pour le stockage, il est clair que pour accéder au $k^{ème}$ coefficient de la décomposition, il faudra effectuer k+1 adressages indirects.

2° Les blocs étant de taille fixe, le stockage d'un objet complexe - de grande taille- ne peut se faire qu'en utilisant plusieurs blocs élémentaires chaînés entre eux par l'intermédiaire de pointeurs. Le stockage de tels pointeurs de reprise peut s'avérer coûteux.

Ex.2 : Pour reprendre l'exemple précédent (ex.1), bien que l'entier N n'ait que $\lfloor Log_B N \rfloor + 1$ chiffres en base ß, le stockage de sa décomposition en base ß coûte $2(\lfloor Log_B N \rfloor + 1)$.

3° La détermination de la taille fixe pour les blocs élémentaires est difficile, et souvent liée au problème traité. Supposons que cette taille soit T.

Si l'algorithme manipule des objets de taille très inférieure à T, de la place mémoire sera allouée et non utilisée, donc gaspillée.

Par contre, si l'algorithme manipule des objets de taille très supérieure à T, de la place mémoire sera utilisée pour le stockage des pointeurs permettant d'accèder aux différents blocs de taille T de l'objet.

Il est donc clair que pour minimiser la place mémoire occupée par un objet, il faut stocker de manière contiguë les données relatives à un même champ.

IV. Structure interne de base

Tenant compte des contraintes de communication et de gestion mémoire sur l'architecture modèle, il est nécessaire de pouvoir disposer de tableaux de taille variable.

Définition :

un bloc est un tableau de taille variable, stocké de manière contiguë en mémoire, tel qu'à tout instant de l'évaluation il soit possible de connaître ou de modifier la taille du bloc ainsi que son type.

Pour l'implantation, un bloc de taille n est un tableau de (n+1) mots :

- le libellé (premier mot) contient les informations de taille (n) et de type (une partie du libellé reste utilisable pour d'autres informations)

- la valeur du bloc stockée sur n mots

- le nombre d'occurence du mot, c'est à dire le nombre d'objets qui pointent sur le bloc

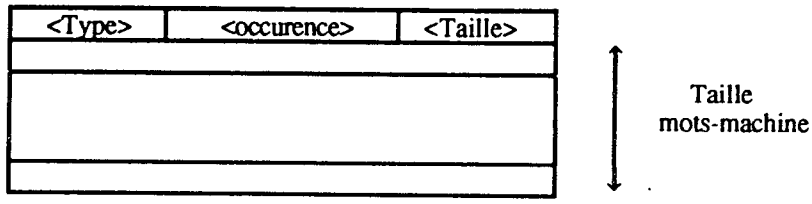


Fig. IV.a Bloc

Le temps pour communiquer un bloc de taille n entre deux nœuds voisins sur l'architecture modèle est donc :

$$T_{com} = \alpha + (n+1).\tau.$$

Remarques

1° Typage d'un bloc

Il est très important de pouvoir typer les objets, de façon à rendre possible l'utilisation des opérateurs surchargés, ainsi que les éventuels changements de type.
ex. : a + b est valide pour tout couple (a,b) de $\mathbb{Z}$ x $\mathbb{Z}$, ou de $\mathbb{Q}$ x $\mathbb{Q}$ ou de $\mathbb{Z}$ x $\mathbb{Q}$...
Ce sont donc ici les *types* de a et b qui permettent de décider de l'algorithme à utiliser pour effectuer leur addition.

Pour rendre possible le typage des blocs, nous avons vu ci-dessus qu'une zone spéciale existe dans le libellé du bloc. Certains types sont prédéfinis et ne peuvent être définis hors de leur domaine d'application. Ils sont énumérés dans le tableau ci-dessous, en indiquant chaque fois le codage correspondant :

Type	Mnémonique du champ type	Valeur binaire
$\mathbb{N}$, $\mathbb{Z}$ standard	type_Z	00000000
$\mathbb{N}$, $\mathbb{Z}$ vectoriel	type_V	00001000
$\mathbb{Q}$	type_Q	00000001
$\mathbb{Z}[X]$	type_ZX	00000010
$\mathbb{Q}[X]$	type_QX	00000011
$\mathbb{Q}(X)$	type_QFX	00000111
Expression	type_expr.	00001111

Fig. IV.b Types prédéfinis

2° Occurence d'un bloc

De façon à ne pas dupliquer de données inutiles, il est essentiel de pouvoir adresser un même bloc à différentes reprises. Pour assurer que la libération d'un bloc n'entraîne pas une perte d'information dans une donnée encore adressée, on associe à chaque bloc son nombre d'occurences. A l'allocation, ce nombre est initialisé à zéro. A chaque duplication, il est incrémenté : il n'est plus alors besoin de copier physiquement le bloc à un autre emplacement mémoire. Lors de la libéraion, un bloc ne peut être libéré que si son nombre d'occurences est nul; sinon, ce nombre est décrémenté.

Nous supposerons désormais définies les fonctions suivantes :

- allouer (t : entier):bloc == retourne un bloc de taille t
- libérer (u : bloc):vide == libère la place occupée par le bloc u
- copier (u : bloc):vide == duplique virtuellement le bloc u (cf 2°)
- taille (u : bloc):entier == retourne la taille du bloc u, autrement dit le nombre de mots-machine contenant sa valeur
- type (u : bloc):entier == retourne le type du bloc u (cf 1°)

Chapitre II

REPRESENTATION INTERNE ET CONVERSIONS

I. Définition de la représentation interne - Format standard

Il existe de nombreux modes de représentation des nombres : sans remonter à l'Antiquité (...), nous pouvons distinguer les représentations "logicielles" suivantes :

♦ Représentation en base ß (ß entier supérieur ou égal à deux) : un nombre est représenté par la suite de ses restes modulo les puissances décroissantes de ß. Cette représentation est unique. Pour plus de commodité, nous noterons par la suite $[a_n,\ldots,a_0]_ß$ l'entier $a = \sum_{i=0}^{n} a_i.ß^i$, avec $0 \leq a_i < ß$ (i=0...n-1).

ex. : $[5E1]_{16}$ est l'écriture en base 16 de l'entier (1+16.(14+16.5))= 1505

Cette définition peut être étendue au cas où ß est négatif; mais les opérations arithmétiques sont alors moins génériques [KNU81e].

Cette représentation est la plus utilisée dans les systèmes de Calcul Formel, en choisissant ß positif. Les opérations arithmétiques sont d'abord définies pour les entiers positifs, puis étendues aux entiers relatifs, en ajoutant une information sur le signe.

♦ Représentation modulaire : un nombre est formé par la suite de ses restes modulo un ensemble de nombres premiers. Avec une telle représentation, les opérations d'addition, de soustraction et de multiplication sont très faciles [KNU81b]. Par contre, comparer deux entiers, ou effectuer une division est difficile, et nécessite une remontée via le théorème des restes chinois [VIL89].
D'autre part, il est nécessaire de connaître une borne pour chaque nombre - et chaque résultat-, de façon à choisir suffisamment de nombres premiers entre eux pour sa représentation. Cette contrainte rend impraticable l'utilisation de l'arithmétique modulaire : dès que l'on a besoin d'un nouveau pseudo-premier, il ne peut plus être supprimé, à moins de faire une remontée de tous les nombres qui en dépendent. Au cours d'un algorithme dans lequel la taille des résultats ne peut être bornée -où lorsque la borne est trop grande-, l'utilisation d'une telle représentation est impossible.

♦ Autres représentations : plus généralement, il existe de nombreuses façons de coder les entiers, de façon à diminuer le coût de certaines opérations (notations carry-save, d'Avizienis [AVI61], [MUL89]). Certaines de ces représentations seront utilisées par la suite.
Il est fondamental de disposer d'une représentation de référence, qui permette la représentation exacte d'un entier quelconque. Cette forme canonique doit être en accord avec le modèle choisi, et doit en outre permettre de :

1° minimiser l'encombrement mémoire

2° communiquer rapidement un entier

3° réaliser facilement toutes les opérations

La représentation standard choisie est la décomposition en base ß, avec $ß = 2^w$ où w est le nombre de bits du mot machine. En effet, cette représentation permet, grâce à la structure bloc de minimiser l'encombrement mémoire pour un entier positif. Le cas des entiers négatifs sera traité ultérieurement, en rajoutant un signe aux entiers positifs. En outre, le choix de ß permet de tirer pleinement parti de l'arithmétique étendue, disponible sur chaque nœud.

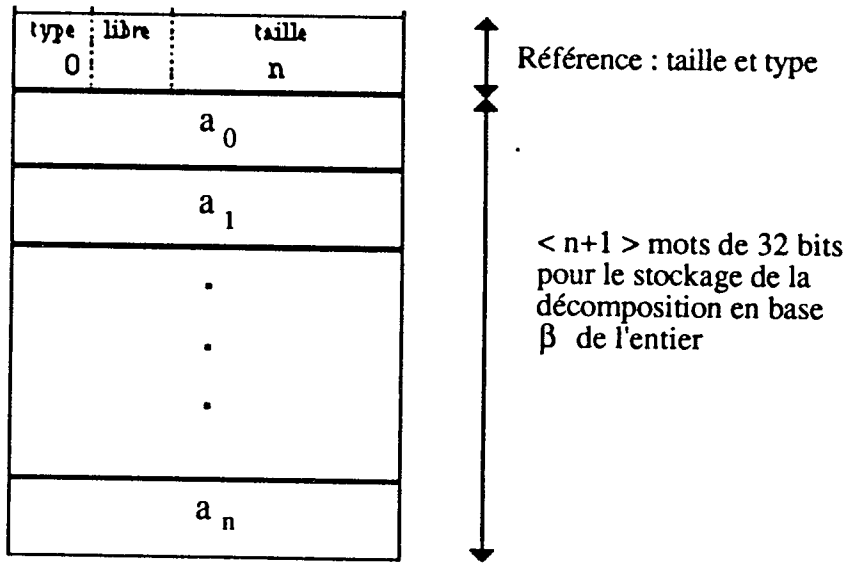


Fig. II.1 Représentation de l'entier $a = \sum_{i=0}^{n} a_i.ß^i$ avec $\begin{cases} 0 \le a_i < ß & (i=0...n-1) \\ 0 < a_n < ß & (\text{si } n \neq 0) \end{cases}$

N.B. : On représente 0 de manière unique : $0 = 0.ß^0$

Notations :

Un *b-chiffre* désigne un entier appartenant à {0,..., b-1}.

Par la suite, nous noterons $A = [a_n, a_{n-1},..., a_0]_b$ l'entier $A = \sum_{i=0}^{n} a_i.b^i$ en représentation en base b sous forme de bloc, et nous le désignerons par le terme *b-entier.*

Nous supposerons en outre définies les fonctions suivantes :

- taille_N (u : b-entier) : entier == retourne la taille du b-entier u, c'est à dire $\lfloor \mathrm{Log}_b u \rfloor + 1$.
- allouer_N (t : entier) : b-entier == retourne un b-entier sous forme de bloc de taille t
- dupliquer_N (t : entier) : b-entier== retourne t (cf. occurence)
- libérer_N (u : b-entier) : vide == libère la place occupée par u
- ajuster_N (u : b-entier) : vide == ramène le b-entier u sous la représentation canonique en base b (b-chiffre de poids fort non nul si u≠0)

Communication :

Soit $a \in \mathbf{N}$ un entier positif quelconque. Le temps pour communiquer a entre deux nœuds voisins est :

$$T_{com}(a) = \alpha + \tau.(\lfloor \mathrm{Log}_\beta a \rfloor + 2) \text{ si } a \neq 0$$

$$T_{com}(0) = \alpha + 2.\tau$$

II. Conversions en format décimal

S'intéresser à des problèmes de grande taille conduit à négliger l'impact des problèmes d'entrée-sortie sur la complexité des algorithmes. C'est pourquoi le choix de la base de numération est guidé par la puissance de calcul maximale et la représentation mémoire la plus dense possible. Il faut néanmoins étudier le problème des conversions qui, bien que rares, sont essentielles pour la donnée d'un problème et pour la visualisation de son résultat.

Convertir $a = \sum_{i=0}^{n} a_i.b^i$ en $A = \sum_{i=0}^{n} a_i.B^i$ peut être essentiellement réalisé de deux façons différentes :

1° On divise a par B en utilisant l'arithmétique en base b, jusqu'à obtenir 0. Les restes des divisions sont les coefficients - poids faibles d'abord - de la décomposition en base B.

2° On calcule $a = (\ldots(a_n.b + a_{n-1}).b + \ldots).b + a_0$ en arithmétique base B. On obtient ainsi la représentation en base B de a.

Ces deux méthodes vont être étudiées pour ($b = \beta$, $B = D$) et ($b = D$, $B = \beta$), où D est une base décimale quelconque (les résultats sont directement généralisables au cas où D n'est pas une puissance de 10). Il est fondamental que tous les calculs soient effectués de préférence en base β, cette base étant la base de calcul machine en arithmétique étendue.

La plus grande puissance de 10 codable en arithmétique étendue est $D = 10^d$ avec $d = \lfloor \mathrm{Log}_{10}(ß) \rfloor$.
ex. : pour $ß = 2^{32}$: $D = 10^9$
On ne considèrera donc comme format décimal que la représentation en base D, celle-ci étant beaucoup plus économique que le codage en base 10, et le passage de la base D à la base 10 étant immédiat.

II.1. Passage de la base ß à la base D

II.1.1. Algorithme direct

On divise a par D en utilisant l'arithmétique en base ß, jusqu'à annuler a. Les restes successifs donnent l'écriture en base ß de l'entier a.
L'algorithme de calcul des A_i est alors :

```
algorithme  ß_vers_D ( a : ß-entier ) → A : D-entier ==
   var      i, n, N : entier
            reste : D-chiffre
   début
      n ← taille_N ( a );
      A ← allouer_N ( ⌊n*Log_D ß⌋ + 1);
      i ← 0;

      Tantque (a ≠ 0 ) faire
            Tantque (a_n = 0) faire  n ← n - 1;
            Pour  j = n...0 (init reste ← 0 ) faire
                  (a_j, reste) ← LONGDIV ( reste, a_j, D);
            A_i ← reste;
            i ← i + 1;
      ajuster_N (A);
   fin
```

Complexité :

Le côut de cet algorithme peut être évalué à partir de :

1° Nombre de passages dans la boucle : $\lceil Log_D(a) \rceil$

2° Coût de chaque passage : seule la division importe.
Au $k^{ième}$ passage, on divise (a/D^{k-1}) par D : pour ce calcul, on effectue $\lceil Log_\beta (a / D^{k-1}) \rceil$ opérations LONGDIV

$$T_{\beta\_vers\_D} = \sum_{k=0}^{\lceil Log_D a \rceil - 1} \lceil Log_\beta (a / D^k) \rceil \tau_{div}$$

D'où le coût de l'algorithme :

$$T_{\beta\_vers\_D} = \frac{\tau_{div}}{2.Log\ \beta\ Log\ D} Log^2 a + O(Log\ a)$$

L'écriture d'un entier précision infinie sous format décimal s'effectue donc en temps quadratique sur le nombre de chiffres de l'entier en base ß.

Occupation mémoire :

1° Pour l'exécution : $Mem = Log_D\ a$

2° Après exécution : la notation décimale n'ayant d'intérêt que pour effectuer une opération d'écriture, la place occupée pour cette représentation peut être libérée dès celle-ci effectuée; elle est *temporaire* : le coût mémoire de l'algorithme est donc nul.

II.1.2. Amélioration de l'algorithme par Divide&Conquer

Il est possible de cumuler les divisions successives par D, en divisant directement par D^k, puis en convertissant successivement le reste et le quotient obtenus.

Soit $a = [a_n \ldots a_0]_\beta$; on suppose que $\lfloor Log_D\ a \rfloor = 2^N$ (pour faciliter le découpage), et soit $D_i = D^{2^i}$ (i=0...N-1) les puissances de D nécessaires au calcul.

On calcule : $a^{(N-1)} = a / D^{2^{N-1}}$ et $a^{(0)} = a \bmod D^{2^{N-1}}$
On peut alors convertir $a^{(N-1)}$ et $a^{(0)}$ en itérant le processus (On les divisera d'abord par $D^{2^{N-2}}$, puis par $D^{2^{N-3}}$, ..., $D^{2^0}=D$).

L'algorithme s'écrit alors récursivement :

algorithme **ß_vers_D_D&C** (a : ß-entier) → A : D-entier ==
var n, N : entier
début
n ← taille_N (a);
N ← $\lfloor \text{Log}_2 n - \text{Log}_2 \text{Log}_\beta D \rfloor + 1$;
/* Calcul des D_i , supposés globaux */
Pour i=1...N -1 (*init* $D_0 \leftarrow D$) *faire*
$D_i \leftarrow$ multiplication (D_{i-1}, D_{i-1});

A ← allouer_N (2^N);
convertir (a, N-1, $[A_{2N-1},\ldots,A_0]_D$);
fin

algorithme **convertir**(a:ß-entier, n:entier, $[d_{2n+1-1},\ldots,d_0]_D$:D-entier) ==
convertir calcule la représentation décimale du ß-entier a dans le D-entier préalablement alloué $[d_{2n+1-1},\ldots,d_0]_D$
var q, r : ß-entier
début
/* $u = \lfloor a / D_n \rfloor$ et $v = a \bmod D_n$ sont obtenus par une division euclidienne effectuée en arithmétique base β */
(q, r) ← division (a, D_n);

si n = 0 *alors* $[d_1, d_0]_D \leftarrow [q, r]_D$
sinon
convertir (r, n-1, $[d_{2n-1},\ldots,d_0]_D$);
convertir (q, n-1, $[d_{2n+1-1},\ldots,d_{2n}]_D$);
fin

Complexité :

La complexité de l'addition, de la multiplication et de la division de deux β-entiers intervenant dans l'algorithme (cf. Chap.II IV), nous les désignerons par :

♦ $A_\beta(n,m)$ le temps pour additionner deux β-entiers ayant respectivement n et m β-chiffres (On notera $A_\beta(n) = A_\beta(n,n)$)
On a : $A_\beta(n,m) = \text{Max}(n, m)$

♦ $M_\beta(n,m)$ le temps pour multiplier deux β-entiers ayant respectivement n et m β-chiffres (On notera $M_\beta(n) = M_\beta(n,n)$)
D'après [SST71] $M_\beta(n,m) = O(n.\text{Log } m.\text{Log Log } m)$

♦ $D_\beta(n,m)$ le temps pour diviser deux ß-entiers ayant respectivement n et m ß-chiffres (On notera $D_\beta(n) = D_\beta(n,n)$)
D'après [COO69] $D_\beta(n,m) = O(n.\text{Log } m.\text{Log Log } m)$

1° Calcul des $D_i=(D^{2^{i-1}})^2$: chaque D_i coûte $M_\beta(2^{i-1})$
Donc au total (i=1...N-1), en utilisant l'inégalité $M(2.n) \geq 2.M(n)$
$O(M_\beta(2^{N-1}))$

2° Appels à *convertir*: à l'étape i, on effectue deux divisions d'un entier
Si C(n) est le temps pour convertir un entier de taille n, on en déduit que: $C(2n) = 2.C(n) + D(2n,n) = 2.C(n) + O(M(n))$

D'où : $C(2^N) = O(M(2^{N-1})+2.M(2^{N-2})+4.M(2^{N-3})+\ldots)$
$= O(n.M(2^{N-1}))$

On en déduit donc le coût de l'algorithme :

$$\boxed{T_{\beta\_vers\_D} = O(M(n)\log(n))}$$

et d'après la valeur de M(n) ci-dessus, l'algorithme convertit un nombre de n ß-chiffres en :

$$\boxed{O(n.\log^2 n.\log\log n)}$$

Remarque :

En utilisant l'algorithme classique de multiplication en n^2, la complexité de l'algorithme est alors $O(n^2.\log(n))$, ce qui est moins bon que l'algorithme direct en $O(n^2)$. Cet algorithme n'est donc interessant qu'avec une multiplication de coût inférieur à n^2. Or, de tels algorithmes ne sont efficaces que pour des entiers suffisamment grands, valeurs pour lesquelles seuls les premiers chiffres sont significatifs.

II.2. Passage de la base D à la base β

II.2.1. Algorithme direct

On évalue $A = \sum_{i=0}^{n} a_i.D^i$ en arithmétique base ß, par le schéma de Horner, ce qui revient à évaluer en arithmétique base ß la suite :

$$\begin{cases} P_0 = A_N \\ P_k = D.P_{k-1} + A_{N-k} \quad k = 1\ldots N \end{cases}$$

En utilisant LONGMUL pour le calcul des P_k, on obtient l'algorithme :

```
algorithme  D_vers_ß ( A : D-entier ) → a : ß-entier ==
  var       n, n_max, N : entier
            retenue : D-chiffre
  début
    | N ← taille_N ( A );
    | a ← allouer_N ( ⌊N*Log_β D⌋ + 1);
    | Pour  i = N-1...0 (init  a_0 ← A_N; n_max ← 0) faire
    |     | Pour  j = 0...n_max (init retenue ← A_i ) faire
    |     |     | (retenue, a_j ) ← LONGMUL (a_j, D, retenue )
    |     | si retenue ≠ 0 alors
    |     |     | n_max ← n_max + 1;
    |     |     | a_n_max ← retenue;
    | ajuster_N (a);
  fin
```

Complexité :

1° Nombre de passages dans la boucle : $\lceil \mathrm{Log}_D(A) \rceil$

2° Coût de chaque passage : seule la multiplication importe.
Au k^{ième} passage, on multiplie $[A/D^{n-k}]_\beta$ par D : pour ce calcul, on effectue $\lceil \mathrm{Log}_\beta (a / D^{n-k}) \rceil$ opérations LONGMUL.

$$T_{D\_vers\_\beta} = \sum_{k=0}^{\lceil \mathrm{Log}_D A \rceil} \lceil \mathrm{Log}_\beta (A / D^{\lceil \mathrm{Log}_D A \rceil - k}) \rceil \, \tau_{mul}$$

D'où le coût de l'algorithme :

$$\boxed{T_{D\_vers\_\beta} = \frac{\tau_{mul}}{2.\mathrm{Log}\ \beta\ \mathrm{Log}\ D} \mathrm{Log}^2 A + O(\mathrm{Log}\ a)}$$

N.B. : La conversion de A de la base D à la base ß par évaluation via le schéma de Horner donne donc la même complexité (au rapport τ_{mul}/τ_{div} près) que l'algorithme permettant de passer de l'écriture de a en base β à l'écriture en base D par divisions successives.

Occupation mémoire :

1° Pour l'exécution : $Mem = \lfloor N.\mathrm{Log}_\beta D \rfloor + 1$

2° Après exécution : le passage de la base D à la base ß se fait *en place* : la place mémoire utilisée au cours de l'algorithme est en fait la place du résultat, soit : $Mem = \lceil \mathrm{Log}_\beta(A) \rceil$

II.2.2. Amélioration de l'algorithme par Divide&Conquer

Plutôt que d'effectuer $Log_D A$ multiplications par D, on peut cumuler les multiplications, en multipliant directement par des puissances de D. Cela revient à modifier la manière de regrouper les termes pour l'évaluation de A (schéma de Horner pour l'algorithme précedent).
En supposant : $Log_D A = N = 2^p-1$, il est possible d'utiliser une technique Divide&Conquer pour l'évaluation de A. Le schéma est alors le suivant :

$$\begin{cases} P_k(A_0,\ldots,A_{2^k-1}) = D^{2^{k-1}}.P_{k-1}(A_{2^{k-1}},\ldots,A_{2^k-1}) + P_{k-1}(A_0,\ldots,A_{2^{k-1}-1}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{pour } k = p\ldots2 \\ P_1(A) = A \end{cases}$$

Le calcul de ce schéma se fait en deux temps :

1°	Calcul de $D_i = D^{2^i}$	$i = 1\ldots p-1$
2°	Calcul de P_i	$i = p\ldots1$

L'algorithme s'écrit alors :

```
algorithme D_vers_ß_D&C ( A : D-entier ) → a : ß-entier ==
  var     N, p : entier
  début
    N ← taille_N(A);
    p ← ⌈Log2 ( N+1 ) ⌉;

    /* Calcul des Di , supposés globaux */
    Pour i=1...p -1 ( init D0 ← D ) faire
          Di ← multiplication ( Di-1, Di-1 );

    a ← évaluer_D&C ( p, [AN,...,A0]D );
  fin

algorithme évaluer_D&C ( p:entier, [A2p-1,...,A0]D:D-entier )
                                                  → a:ß-entier==
   évaluer_D&C calcule le schéma Pk décrit ci-dessus
  var     q, r : ß-entier
  début
    si p = 1 alors a ← A0  /* D<β ⇒ A0 = [A0]D = [A0]β */
    sinon
        a ← addition (évaluer_D&C ( p-1, [A2p-1-1,...,A0]D ),
          multiplication( Dp-1,évaluer_D&C ( p-1, [A2p-1,...,A2p-1]D)))
  fin
```

Complexité :

1° Calcul des $D_i=(D^{2^{i-1}})^2$: chaque D_i coûte $M_\beta(2^{i-1})$
Donc au total (i=1...p-1), en utilisant le fait que $M(2.n) \geq 2.M(n)$
$O(M_\beta(2^{p-1}))$

2° Appels à évaluer_D&C : à l'étape i, on effectue une multiplication, une addition et deux appels à evaluer_D&C.

Si E(n) est le temps pour évaluer un entier de taille n, on en déduit que: $E(2n) = 2.E(n) + A_D(2n) + M_D(n) = 2.E(n) + O(M(n))$
D'où l'on tire : $E(2^p) = O(M(2^{p-1})+2.M(2^{p-2})+4.M(2^{p-3})+\ldots)$
$= O(p.M(2^{p-1}))$

Le coût de l'algorithme est donc:

$$\boxed{T_{D\_vers\_\beta} = O(M(N)\log(N))}$$

et d'après la valeur de M(N), l'algorithme convertit un nombre de N D-chiffres en :

$$\boxed{O(N.\log^2 N.\log\log N)}$$

Remarques

i/ Comme en II.1.2., on constate que cet algorithme n'est intéressant que si l'algorithme de multiplication utilisé a un coût inférieur à $O(n^2)$. Il ne devient donc intéressant que pour des entiers suffisamment grands.

ii/ Basés sur une technique de partage du problème initial en deux sous-problèmes indépendants, ces deux algorithmes de conversions dégagent un parallélisme inhérent particulièrement adapté à une topologie de type hypercube.

Chapitre III

ARITHMETIQUE DANS N
ALGORITHMES STANDARDS

Dans un premier temps, les algorithmes classiques sont présentés : il correspondent aux algorithmes utilisés pour faire les opérations "à la main". Il est intéressant de les étudier de manière précise: ils sont souvent optimaux, notamment dans le cas où les opérandes sont de petites tailles. Ces algorithmes ont été décrits de manière générale dans [KNU81a], et sont ceux utilisés dans la plupart des systèmes de Calcul Formel (bignums de Macsyma [SIR88]). Nous les désignerons par la suite sous l'appellation "algorithmes standards".

I. Comparaisons

D'une manière générale, il est nécessaire de pouvoir tester quel est le plus grand (au sens large ou strict) de deux entiers. Avec la représentation standard en base ß, il suffit de comparer la taille des deux entiers, et en cas d'égalité, de comparer leurs coefficients -poids forts d'abord- : l'algorithme est donc facilement défini à partir de la comparaison de deux ß-chiffres. L'algorithme est alors :

algorithme **gle_ß** (υ, ϖ : ß-chiffre) $\rightarrow$ test : {-1, 0, 1} ==
début
  si $\upsilon = \varpi$ *alors* test $\leftarrow 0$
  sinonsi $\upsilon > \varpi$ *alors* test $\leftarrow 1$
  sinon test $\leftarrow -1$
fin

algorithme **gle_N** (u, v : ß-entier) $\rightarrow$ test : {-1, 0, 1} ==

test prend la valeur $\begin{cases} -1 & \Leftrightarrow \quad a < b \\ 0 & \Leftrightarrow \quad a = b \\ 1 & \Leftrightarrow \quad a > b \end{cases}$

var k : entier
début
  test $\leftarrow$ gle_ß (taille_N(u), taille_N(v));
  k $\leftarrow$ taille_N(u);
  tantque (test = 0) *et* (k≥0) *faire*
    test $\leftarrow$ gle_ß (u_k, v_k);
    k $\leftarrow$ k - 1;
fin

Complexité :

Le pire cas est évidemment le cas d'égalité, dans lequel $\lfloor \mathrm{Log}_\beta u \rfloor + 2$ tests sont nécessaires.
Dans la plupart des cas, le résultat est obtenu dès le premier test.

$$T_{gle\_N}(n,m) \leq O(\mathrm{Inf}(n, m))$$

où $T_{gle\_N}(n, m)$ est le temps pour comparer un entier de n ß-chiffres à un entier de m ß-chiffres.

II. Addition et Soustraction

L'algorithme d'addition consiste à additionner les coefficients de poids faible d'abord, en propageant les retenues éventuelles.

```
algorithme add_N ( u, v : ß-entier ) → r : ß-entier ==
  On suppose u≥v, et on calcule r = u+v
  début
      r ← allouer_N ( taille_N (u) + 1);      /* r≤ 2.u */
      Pour  k = 0...taille_N (v) (init  retenue ← 0)     faire
          | (retenue, rk ) ← LONGADD (uk, vk, retenue )
      Pour  k = taille_N(v)+1...taille_N (u)             faire
          | (retenue, rk ) ← LONGADD (uk, 0, retenue )
      rtaille_N(u)+1 ← retenue;
      ajuster_N (r);
  fin
```

L'algorithme de soustraction (en supposant le résultat positif) est tout à fait similaire :

```
algorithme sub_N ( u, v : ß-entier ) → r : ß-entier ==
  On suppose u≥v, et on calcule r = u-v
  début
      r ← allouer_N ( taille_N (u) );         /* r ≤ u */
      Pour  k = 0...taille_N (v) (init  retenue ← 0)     faire
          | (retenue, rk ) ← LONGDIFF (uk, vk, retenue )
      Pour  k = taille_N(v)+1...taille_N (u)             faire
          | (retenue, rk ) ← LONGDIFF (uk, 0, retenue )
      ajuster_N (r);
  fin
```

Complexité :

$$T_{add}(n, m) = T_{sub}(n, m) = Max\ (n,m).\tau_{add}$$

où $T_{add}(n, m)$ est le temps pour additionner ou soustraire un entier de n ß-chiffres à un entier de m ß-chiffres.

Implantation :

Lorsque l'on obtient une retenue nulle alors que v est déjà parcouru, on peut directement recopier les coefficients de u dans v.

Cette remarque est vraie avec la probabilité $(ß^{k\text{-taille\_N(v)}})^{-1}$ à l'étape k>taille_N(v) : on en déduit que le coût moyen de l'opération est en fait Min (n,m).τ_{add}, aux coûts des déplacements mémoire près.

En outre, le stockage de la retenue peut être évité sur la plupart des processeurs, en la gardant dans un registre (CISC -68000-) ou dans une pile (RISC -T414-).

III. Multiplication

L'algorithme de multiplication classique est efficace dans certains cas (notamment lorsque l'un des opérandes est de petite taille). Cet algorithme est celui utilisé pour les calculs à la main, la seule différence résidant dans le cumul des produits partiels.

```
algorithme mul_N ( u, v : ß-entier ) → r : ß-entier ==
  On suppose u≥v, et on calcule r = u.v
  var       c_+, c_x, t : ß-chiffre
      { c_+ (resp. c_x) est utilisée pour gèrer la retenue lors d'une somme
      (resp. un produit).   t permet de garder un β-chiffre intermédiaire. }
  début
      r ← allouer_N ( taille_N (u) + taille_N(v) + 1);
      Pour  k = 0...taille_N (u)  (init  retenue_x ← 0)  faire          (1)
          (retenue_x, r_k ) ← LONGPROD (u_k, v_0, retenue_x )
      r_{k+1} ← retenue_x;
      retenue_+ ← 0
      Pour  j = 1...taille_N (v)      faire                             (2)
          Pour  k = 0...taille_N (u)  (init  retenue_x ← 0 ) faire
              (retenue_x, t) ←LONGPROD(u_k, v_j, retenue_x ) ;
              (retenue_+, r_{k+j}) ←LONGADD(t, r_{k+j}, retenue_+)
          (retenue_+, r_{taille_N(u)+j+1}) ←
                          LONGADD (retenue_x , 0, retenue_+);           (3)
      ajuster_N (r);
  fin
```

Complexité :

$$T_{mul}(n, m) = n.m.(\tau_{mul} + \tau_{add})$$

où $T_{mul}(n, m)$ est le temps pour multiplier un entier de n ß-chiffres par un entier de m ß-chiffres.

Implantation :

♦ L'instruction (3) est bien valide, et annulle c_+ : en effet
Soient a, b, c trois ß-chiffres : alors a.b + c ≤ ß.(ß - 1), d'où :
◊ soit retenue_x = ß -1 et t=0 d'où retenue_+ = 0
◊ soit retenue_x ≤ ß -2 et retenue_+ ≤ 1
Le stockage de c_+ et de t peut donc être évité.

♦ L'algorithme peut facilement être étendu au calcul de u.v+w, avec w≤u : il suffit alors de recopier w dans r, puis d'intégrer la boucle (1) sur v_0 dans la seconde (2) qui est générique.

♦ Il est intéressant d'éviter le passage dans la boucle (2) lorsque v_j=0. On procède alors à la mise à zéro de $r_{taille\_N(u)+j+1}$.

IV. Division

IV.1. Principe de l'algorithme de Pope-Stein

Soient u et v deux entiers positifs. On cherche q et r tels que u=v.q+r avec 0≤r<b.

On pose $\upsilon = \lfloor Log_\beta u \rfloor$ et $\varpi = \lfloor Log_\beta v \rfloor$, et l'on suppose u>v.

L'algorithme de division de Pope et Stein [POP60] permet de calculer les ß-chiffres de q poids fort d'abord, en "devinant" à chaque étape un nouveau coefficient de q. Il correspond à une généralisation, pour une base quelconque, du calcul de la division à la main en base 10.

A chaque calcul d'un nouveau chiffre du quotient, on est donc ramené à déterminer l'entier θ tel que :

$$\begin{cases} 0 \le \theta < \beta \\ \theta.[v_\varpi, v_{\varpi-1},\ldots,v_0]_\beta \le [u_\upsilon, u_{\upsilon-1},\ldots,u_{\upsilon-\varpi}]_\beta < (\theta+1).[v_\varpi, v_{\varpi-1},\ldots,v_0]_\beta \end{cases}$$

On a :
$$\frac{u_\upsilon.\beta + u_{\upsilon-1}}{v_\varpi + 1} < \theta < \frac{u_\upsilon.\beta + u_{\upsilon-1} + 1}{v_\varpi}$$

D'où l'on tire (comme θ est un entier et $\theta \leq \beta-1$) :

$$\left\lceil \frac{u_{\upsilon}.\beta + u_{\upsilon-1}}{v_{\varpi}+1} \right\rceil \leq \theta \leq \mathrm{Min}\left(\left\lfloor \frac{u_{\upsilon}.\beta + u_{\upsilon-1}+1}{v_{\varpi}} \right\rfloor, \beta-1 \right) = \hat{\theta}$$

On peut se ramener au cas $\frac{\beta}{2} \leq v_{\varpi} < \beta$ par normalisation, en multipliant v (et u) par $\alpha = \left\lfloor \frac{\beta}{v_{\varpi}+1} \right\rfloor$ par exemple.

Dans ce cas, Pope et Stein ont montré que :

$$\theta \leq \hat{\theta} \leq \theta + 2$$

Knuth [KNU81b] a montré que l'on pouvait affiner l'approximation de θ avec $\tilde{\theta}$:

$$\tilde{\theta} = \mathrm{Min}\left(\left\lfloor \frac{u_{\upsilon}.\beta^2 + u_{\upsilon-1}.\beta + u_{\upsilon-2}}{v_{\varpi}.\beta + v_{\varpi-1}} \right\rfloor, \beta-1 \right)$$

N.B. : $\tilde{\theta}$ peut être facilement calculé en arithmétique base ß, à partir du calcul de $\hat{\theta}$, en soustrayant 1 ou 2 à $\hat{\theta}$ grâce à l'itération :

$\tilde{\theta} \leftarrow \hat{\theta}$;
tantque $\left((u_{\upsilon}.\beta + u_{\upsilon-1} - \tilde{\theta}.v_{\varpi}).\beta + u_{\upsilon-2}\right) < \tilde{\theta}.v_{\varpi-1}$ *faire* $\tilde{\theta} \leftarrow \tilde{\theta} - 1$ *finfaire*

On a alors :

$$\boxed{\theta \leq \tilde{\theta} \leq \theta + 1}$$

A partir de la valeur de $\tilde{\theta}$, il est alors possible de calculer :

$$r = [u_{\upsilon}, u_{\upsilon-1},\ldots,u_{\upsilon-\varpi}]\beta - \tilde{\theta}.[v_{\varpi}, v_{\varpi-1},\ldots,v_0]\beta$$

Si $r < 0$, alors on a $\theta = \tilde{\theta} - 1$ et $r \leftarrow r + v$;
Sinon, on a $\theta = \tilde{\theta}$ et r est le reste euclidien.

Si $\varpi=\upsilon$ alors l'algorithme se termine;
Sinon on pose : $u \leftarrow r.\beta^{\upsilon-\omega} + [u_{\upsilon-\varpi-1}, u_{\upsilon-\varpi-2},\ldots,u_0]\beta$ et on réitère l'étape.

Remarque :

Lorsque ß est grand, le cas $\tilde{\theta} = \theta + 1$ a une probabilité très faible de se produire ($<\frac{3}{\beta}$) [COM77][REG84]. La valeur de $\tilde{\theta}$ est alors d'une certaine manière optimale, la valeur exacte de θ ne pouvant, dans certains cas, se décider qu'en examinant les poids faibles de u et v.

IV.2. Normalisation et pseudo-normalisation

Si l'on a normalisé u et v en les multipliant par un α convenable ($\alpha<\beta$), alors le quotient obtenu est bien le bon, mais il faut diviser le reste par α.
Le coût de cette normalisation est donc :

$$T_{norm} = (\upsilon + \varpi).\tau_{mul} + \varpi.\tau_{div}$$

Pseudo-normalisation :
En réalité, pour obtenir la condition de Pope et Stein sur u et v lorsque β est une puissance de 2, il suffit de calculer le décalage σ (multiplication par 2^{σ}) à faire sur v_{ϖ} pour avoir $v_{\varpi} > \frac{\beta}{2}$ (ce qui est très facile et peu coûteux, grâce à la fonction souvent microprogrammée NORMALIZE).

σ est donc défini par :

$$\sigma \leftarrow \lfloor \mathrm{Log}_2 \beta \rfloor - \lfloor \mathrm{Log}_2 v_{\varpi} \rfloor - 1$$

On peut alors effectuer ce décalage σ virtuellement sur les deux coefficients de poids fort de v (ce qui constitue la pseudo-normalisation de v) :

$$[V_0, V_1]\beta \leftarrow (2^{\sigma}.[v_{\varpi}, v_{\varpi-1}, v_{\varpi-2}]\beta) / \beta$$

Puis, à chaque étape, pour le calcul de $\tilde{\theta}$, on répercute ce décalage σ virtuellement sur u pour obtenir les trois coefficients de poids fort de $u.2^{\sigma}$ nécessaires au calcul de $\tilde{\theta}$:

$$[U_2, U_1, U_0]\beta \leftarrow (2^{\sigma}.[u_{\upsilon}, u_{\upsilon-1}, u_{\upsilon-2}, u_{\upsilon-3}]\beta) / \beta$$

Au cours du calcul, le reste est calculé à partir de valeurs non normalisées de u et de v, et en fin d'algorithme le reste et le quotient obtenus sont bien ceux cherchés.

Le coût de cette pseudo-normalisation est donc :

$$T_{\approx norm} = 3.(\upsilon - \varpi).\tau_{shift}(\sigma) + \tau_{norm} + 2.\tau_{shift}(\sigma)$$

Remarque

Sur une mémoire accessible par octets (ce qui est pratiquement toujours le cas), on peut facilement se ramener au cas où $\sigma < 8$, en se plaçant directement sur le premier octet de poids fort de v (décalage virtuel de $\sigma/8$).

On peut alors calculer le shift σ' à faire sur cet octet pour mettre son bit de poids fort à 1 ($\sigma' = \sigma \bmod 8$). Il suffit ensuite, à chaque étape, de se placer directement sur l'octet correspondant de u (décalage virtuel de $\sigma/8$) et d'effectuer le shift σ'.
Cette remarque est de manière évidente généralisable à une mémoire accessible par tranche de k-bits.

Comparaison entre normalisation et pseudo-normalisation

Le rapport ρ entre $T_{\approx norm}$ et T_{norm} est donc (avec $\tau_{mul} \approx \tau_{div}$) :

$$\rho = \frac{T_{\approx norm}}{T_{norm}} = \frac{3.\tau_{shift}(\sigma')}{\tau_{mul}} \cdot \frac{(\upsilon - \varpi)}{(\upsilon + 2.\varpi)}$$

Avec les valeurs de $\tau_{shift}(7)$ et τ_{mul} données dans le tableau I.1 pour le transputer T414 d'Inmos :

$$\rho = \frac{T_{\approx norm}}{T_{norm}} \approx \frac{(\upsilon - \varpi)}{(\upsilon + 2.\varpi)}$$

Conclusion

La pseudo-normalisation est donc toujours moins coûteuse que la normalisation; elle est en outre d'autant plus avantageuse que u et v sont grands et proches. Cette remarque est vraie pour la plupart des processeurs, puisque le côut de $\tau_{shift}(7)$ est généralement bien inférieur à $3.\tau_{mul}$.

IV.3. Algorithme de Pope-Stein avec pseudo-normalisation

L'algorithme de Pope-Stein est décrit ci-dessous, en utilisant une pseudo-normalisation de u et v pour prévoir le chiffre du quotient cherché à chaque étape.

algorithme **div_N** (u, v : ß-entier) → r : ß-entier ==
On suppose u≥v, et on calcule q et r tels que u = v.q + r avec 0≤v<r.

var	υ, ϖ, σ, retenue_+, retenue_-	: entier
	t, retenue_x	: ß-chiffre
	$[V_0, V_1]_\beta$, $[U_2, U_1, U_0]_\beta$	: ß-entier

début

$\upsilon, \varpi \leftarrow$ taille_N (u), taille_N(v);

r, q $\leftarrow$ allouer_N (υ), allouer_N(ϖ); (1)

r $\leftarrow$ dupliquer (u);

$\sigma \leftarrow \lfloor \mathrm{Log}_2 \beta \rfloor - \lfloor \mathrm{Log}_2 v_\varpi \rfloor - 1$; (2)

$[V_0, V_1]_\beta \leftarrow (2^\sigma.[v_\varpi, v_{\varpi-1}, v_{\varpi-2}]_\beta) / ß$;

Pour $k = \upsilon - \varpi \ldots 0$ *faire*

$[U_2, U_1, U_0]_\beta \leftarrow (2^\sigma.[u_\upsilon, u_{\upsilon-1}, u_{\upsilon-2}, u_{\upsilon-3}]_\beta) / ß$ (3)

$$\tilde{\theta} \leftarrow \mathrm{Min}\left(\left\lfloor \frac{U_2.\beta^2 + U_1.\beta + U_0}{V_2.\beta + V_1} \right\rfloor, \beta - 1\right); \quad (4)$$

Pour $j=0 \ldots \varpi$ (*init* retenue_x =0, retenue_- =0) *faire* (5)

(retenue_x, t) $\leftarrow$ LONGPROD($\tilde{\theta}$, v_j, retenue_x);

(retenue_-, r_{k+j}) $\leftarrow$ LONGDIFF(r_{k+j}, t, retenue_-);

(retenue_-, $r_{k+\varpi+1}$)$\leftarrow$LONGDIFF(r_{k+j}, retenue_x, retenue_-);

si (retenue_- $\neq 0$) *alors* (6)

$q_k \leftarrow \tilde{\theta} - 1$; (7)

Pour $j=0 \ldots \varpi$ (*init* retenue_+ = 0) *faire*

(retenue_+,r_{k+j}) $\leftarrow$ LONGADD(r_{k+j}, t, retenue_+);

(retenue_+, $r_{k+\varpi+1}$)$\leftarrow$LONGADD(r_{k+j}, 0, retenue_+);(8)

sinon

$q_k \leftarrow \tilde{\theta}$;

ajuster_N (r), ajuster_N (q); (9)

fin

Implantation et Optimisations :

(1) La duplication de a est inévitable pour pouvoir évaluer les restes successifs à chaque étape. On a donc forcément à l'initialisation l'allocation d'un entier de taille au moins $\lfloor \mathrm{Log}_\beta u \rfloor + 1$, pour obtenir finalement un reste de taille inférieure à $\lfloor \mathrm{Log}_\beta v \rfloor + 1$.

Or le quotient résultat est de taille inférieure à $\lfloor \mathrm{Log}_\beta u \rfloor + 1 - \lfloor \mathrm{Log}_\beta v \rfloor$. Il est donc intéressant d'allouer dès le départ une place égale à $\lfloor \mathrm{Log}_\beta u \rfloor + 3$, pour stocker dans un même bloc quotient et reste - un mot est utilisé en plus pour le stockage du libellé de q-. Nous n'avons pas décrit dans l'algorithme cette optimisation, facilement implantable mais qui complique la lecture de l'algorithme.

Il est à noter que cette allocation contiguë du quotient et du reste n'est à faire que si l'on désire calculer ces deux quantités. Pour un calcul uniquement du quotient (ou du reste), l'algorithme ci-dessus est facilement adaptable.

(2) Pseudo-normalisation de v : elle doit être effectuée à l'aide de NORMALIZE et de SHIFTLEFT.

(3) Pseudo-normalisation de u : réalisée à l'aide de SHIFTLEFT.

(4) Calcul de $\tilde{\theta}$: ce calcul s'effectue en arithmétique étendue en base ß.

(5) Calcul de $r = [u_{k+\varpi}, \ldots, u_{k\upsilon-\varpi}]\beta - \tilde{\theta}.[v_{\varpi}, \ldots, v_0]\beta$

(6) $\tilde{\theta}$ était trop grand de 1 : on met à jour le nouveau coefficient $\tilde{\theta}$-1 de q, et on restitue le vrai reste, qui est en fait r+v. Il faut rappeler ici que ce cas est très rare lorsque ß est grand.

(7) Cette opération ne peut pas entraîner de dépassement de capacité, car si $\tilde{\theta}=0$, alors à coup sûr $q_k = 0$

(8) Cette instruction met retenue_+ à 1 : la retenue va donc s'annuler avec retenue_- qui était aussi à 1. On n'effectue donc pas l'opération.

(9) Le réajustement de r et de q nécessaire à la fin de l'opération doit être réalisé de manière très précise lorque l'allocation des deux quantités se fait dans un seul bloc (cf (1)).

Complexité

Il y a $(\upsilon-\varpi+1)$ étapes (chaque étape correspondant au calcul d'un ß-chiffre du quotient). Nous pouvons négliger le coût de la boucle (6) qui, nous l'avons vu a une probabilité inférieure à $\frac{3}{\beta}$ d'être exécutée [COM77]. Le coût de chaque étape est alors le coût de la mise à jour du reste, autrement dit : $(\varpi+1)(\tau_{mul} + \tau_{add})$ et le coût de la pseudo-normalisation. On a donc :

$$T_{div}(n, m) = (\tau_{mul} + \tau_{add}).(n-m+1).m + 3.\tau_{shift}(7).(n-m+1) + Cte$$

où $T_{div}(n, m)$ est le temps pour diviser un entier de n ß-chiffres par un entier de m ß-chiffres.

Chapitre IV

MULTIPLICATION RAPIDE

D'ENTIERS EN PRECISION INFINIE

La complexité de la division étant la même que celle de la multiplication [COO69], toute amélioration de complexité de l'une de ces opérations entraîne une amélioration de l'autre. La multiplication étant algorithmiquement plus simple à formuler que la division, il est naturel de s'attacher à son amélioration.

Le meilleur algorithme séquentiel connu est celui de Schönhage-Strassen [SST71] : il est basé sur la similarité entre la formule donnant le produit de deux entiers en base ß, et le produit de convolution - calculé par transformée de Fourier discrète rapide : FFT - [KNU81d][AHU74a]. La complexité de cet algorithme est :

$$M_\beta(n,m) = O(n.\text{Log } m.\text{Log Log } m)$$

Cet algorithme ne peut en fait devenir intéressant que pour des tailles d'entiers suffisamment grandes : en effet, il nécessite deux calculs de FFT - directe et inverse - et par conséquent, la constante dans le O est très importante. Cela limite son emploi à des entiers relativement grands (plus de mille chiffres décimaux), même si certaines améliorations peuvent être apportées, notamment en effectuant certains calculs dans des corps finis bien choisis [HER89].

Nous nous intéressons ici à l'algorithme de Karatsuba. Une généralisation de cet algorithme adaptée au cas où les opérandes ne sont pas de même taille est étudiée.

I. Algorithme de Karatsuba

I.1. Présentation et complexité

L'algorithme de Karatsuba [KAO62] permet, par une technique *Diviser Pour Régner*, d'améliorer la complexité de la multiplication, et ce même pour des entiers de taille modeste.

Soient $u = [u_n, u_{n-1}, \ldots, u_0]_\beta$ et $v = [v_n, v_{n-1}, \ldots, v_0]_\beta$ deux β-entiers de même taille, supposée de la forme $n = 2^p-1$.

Posons : $\begin{cases} u_H = [u_{2^p-1}, u_{2^p-2}, \ldots, u_{2^{p-1}}]_\beta \\ u_L = [u_{2^{p-1}-1}, u_{2^{p-1}-2}, \ldots, u_0]_\beta \end{cases}$ et $\begin{cases} v_H = [v_{2^p-1}, v_{2^p-2}, \ldots, v_{2^{p-1}}]_\beta \\ v_L = [v_{2^{p-1}-1}, v_{2^{p-1}-2}, \ldots, v_0]_\beta \end{cases}$

Le produit u.v peut s'écrire :

$$u.v = (u_H.\beta^{2^{p-1}} + u_L).(v_H.\beta^{2^{p-1}} + v_L)$$
$$= (u_H.v_H).\beta^{2^p} + ((u_H-u_L).(v_L-v_H) + u_H.v_H + u_L.v_L)\,\beta^{2^{p-1}} + u_L.v_L$$

Il est donc possible d'effectuer une multiplication de deux ß-entiers de taille n avec trois multiplications de taille n/2, de deux additions de taille n/2 (u_H - u_L et v_L - v_H) et de quatre additions de taille n.
On a donc, si $T_{kar}(p)$ est le temps mis pour multiplier deux nombres de taille $n = 2^p-1$ par l'algorithme de Karatsuba :

$$T_{kar}(1) = \tau_{mul}$$
$$T_{kar}(n = 2^{p-1}) = 3.T_{kar}(n/2) + 5.n.\tau_{add}$$

D'où l'on tire : $T_{kar}(2^{p-1}) = 3^{p-1}.\tau_{mul} + 5.2^p.\left(\left(\frac{3}{2}\right)^{p-1}-1\right).\tau_{add}$

Et donc : $T_{kar}(n) = n^{Log_2 3}.\tau_{mul} + 10.\tau_{add}.(n^{Log_2 3} - n)$

$$\boxed{T_{kar}(n) = (\tau_{mul} + 10.\tau_{add}).n^{Log_2 3} - 10.n.\tau_{add}}$$

Remarque : Une évaluation plus précise de la complexité tenant compte du nombre de comparaisons effectuées est donné dans [ZIM89].

I.2. Implantation de l'algorithme - Etude du coût mémoire

Une étude de l'algorithme permet de borner la place mémoire nécessaire à son exécution.

Proposition

Soient u et v deux entiers de n β-chiffres, où n est une puissance de deux. Alors 3.n β-chiffres suffisent à l'évaluation de l'agorithme de Karatsuba appliqué à u et v.

Preuve :
Au plus haut niveau, après la découpe de u et de v en respectivement $[u_H,u_L]$ et $[v_H,v_L]$, on procède au calcul de :

- $S_0 = u_H-u_L$ .....n/2 β-chiffres nécessaires
- $S_1 = v_L-v_H$ .....n/2 β-chiffres nécessaires
- $P_0 = S_0.S_1$.....n β-chiffres nécessaires
- $P_1 = u_H.v_H$ .....n β-chiffres nécessaires

- $P_2 = u_L.v_L$ .....n β-chiffres nécessaires
- R = u.v en sommant P_0,P_1,P_2...2.n β-chiffres nécessaires

Soit R le résultat de l'opération finale. R peut être alloué avant le début du calcul, et la place qu'il occupe est alors disponible pour le stockage de valeurs temporaires.
Par ailleurs, il est nécessaire de disposer au moins de 3.n β-chiffres pour le stockage de P_0,P_1,P_2. Il est donc nécessaire de disposer au moins de n β-chiffres en plus du résultat R. Soit T un entier de n β-chiffres pré-alloué.
Il est alors possible d'effectuer toutes les opérations sans avoir besoin de mémoire supplémentaire.

L'algorithme suivant propose une implantation. Il est à noter que l'un des opérandes (u) est perdu, et qu'il est donc nécessaire d'effectuer l'algorithme sur une valeur de u dupliquée.
Soient : R_H (resp. R_L) les n β-chiffres de poids fort (resp. faible) de R
R_{Hh} (resp. R_{Hl}) les n β-chiffres de poids fort (resp. faible) de R_H
R_{Lh} (resp. R_{Ll}) les n β-chiffres de poids fort (resp. faible) de R_L
T la zone de calculs intermédiaires de n β-chiffres pré-allouée

1°/ u_H-u_L ← R_{Ll}
2°/ v_L-v_H ← R_{Lh}
3°/ $R_{Ll}.R_{Lh}$ ← T *avec* R_{Hl} *comme zone de calculs intermédiaires*
4°/ $u_L.v_L$ ← R_L *avec* R_{Hl} *comme zone de calculs intermédiaires*
5°/ $u_H.v_H$ ← R_H *avec* u_H *comme zone de calculs intermédiaires*
6°/ $T+u_L.v_L$ ← T
7°/ $T+u_H.v_H$ ← T
8°/ $T+[R_{Lh},R_{Hl},R_{Hh}]$ ← $[R_{Lh},R_{Hl},R_{Hh}]$

II. Algorithme de Karatsuba mixte

II.1. Présentation de l'algorithme

Il est essentiel de remarquer que le contrôle nécessité par cet algorithme (utilisation d'une pile pour les différents appels récursifs, accès à différentes parties des entiers) est très coûteux. Ce contrôle augmente d'un facteur non négligeable la constante du terme en $n^{Log_2 3}$. La complexité pratique de l'algorithme de Karatsuba est donc beaucoup plus importante que la complexité théorique.
Par contre, l'algorithme standard présenté en IV.1.d ne demande quasiment aucun contrôle; en outre, les accès se font par incrément constant de la taille du mot machine. Nous avons vu que la complexité de cet algorithme est :

$$T_{mul}(n,m) = (\tau_{mul} + \tau_{add}).n.m$$

Par ailleurs, l'algorithme de Karatsuba n'est bien adapté qu'au cas où les deux opérandes sont de même taille, et de plus lorsque cette taille est de la forme 2^p-1. Dans le cas général, il est donc intéressant d'utiliser l'algorithme standard dès que la taille d'un des opérandes à multiplier au cours de l'algorithme devient inférieure à une certaine constante N_{kar} dépendante de la machine.

Nkar est définie par : $\begin{cases} T_{mul}(N_{kar}, N_{kar}) \leq T_{kar}(N_{kar}) \\ T_{mul}(N_{kar}+1, N_{kar}+1) > T_{kar}(N_{kar}+1) \end{cases}$

N_{kar} est donc le plus petit entier n tel que :

$$(\tau_{mul} + 10.\tau_{add}).n^{Log_2 3} - 10.n.\tau_{add} - (\tau_{mul} + \tau_{add}).n^2 \leq 0$$

En négligeant le terme linéaire, on en déduit que N_{kar} est plus grand que le plus petit entier n tel que : $(\tau_{mul} + 15.\tau_{add}).n^{Log_2 3} \leq (\tau_{mul} + \tau_{add}).n^2$
D'où l'on tire :

$$N_{kar} \geq \left(\frac{\tau_{mul}+10.\tau_{add}}{\tau_{mul}+\tau_{add}}\right)^{\frac{1}{2 - Log_2 3}}$$

II.2. Application au Transputer T414

En utilisant les valeurs de τ_{mul} et τ_{add} données dans le tableau II.1, on en déduit :

$$N_{kar} \geq \left(\frac{152}{53}\right)^{\frac{1}{2 - Log_2 3}} > 12$$

Il apparaît donc que l'algorithme de Karatsuba n'est intéressant que pour des entiers déjà raisonnablement grands (au moins 130 chiffres décimaux).

Détermination expérimentale :

Le calcul précédent ne tient pas compte des lectures et écritures mémoire ainsi que du coût de contrôle de l'algorithme. Le resultat donné ne fournit donc qu'une borne inférieure pour N_{kar}.
La détermination de N_{kar} peut être très facilement faite de manière expérimentale par comparaison des temps obtenus par l'algorithme standard et par l'algorithme mixte décrit ci-dessous, et ce en faisant varier N_{kar}.

La valeur expérimentale trouvée est alors : $N_{kar} = 24$

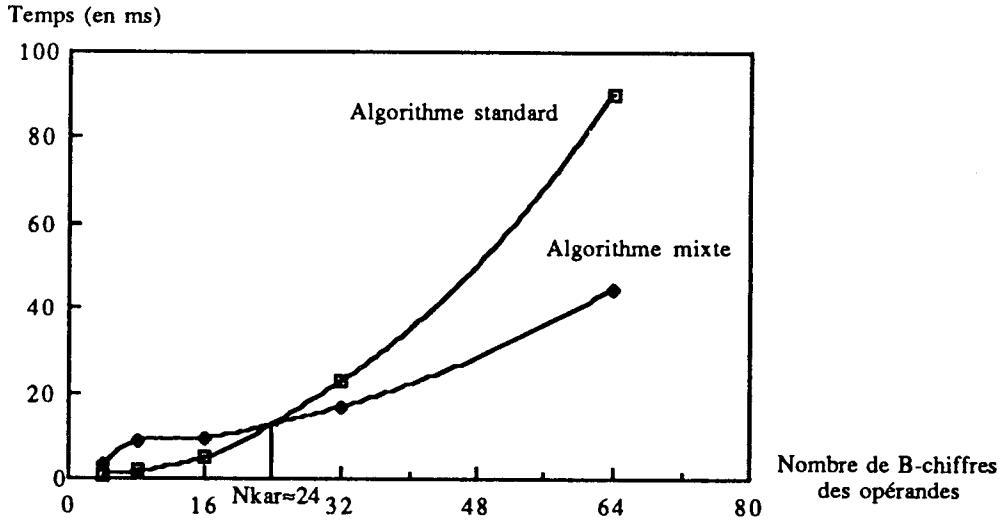


Fig 1 Calcul de la constante N_{kar} sur le transputer T414

Il apparaît donc que l'algorithme de Karatsuba n'est intéressant que pour des entiers déjà raisonnablement grands (environ 250 chiffres décimaux).

II.3. Implantation et comparaison des algorithmes

L'algorithme de Karatsuba mixte a été implanté sur le transputer T414, et des mesures ont été effectuées pour différentes valeurs de N_{kar}. Nous donnons ici les courbes correspondant à ces mesures, ainsi qu'une comparaison entre l'algorithme standard, le meilleur algorithme de Karatsuba mixte, et l'algorithme de Schönhage-Strassen qui a été implanté sur le transputer par Yvan Herreros [HER88].

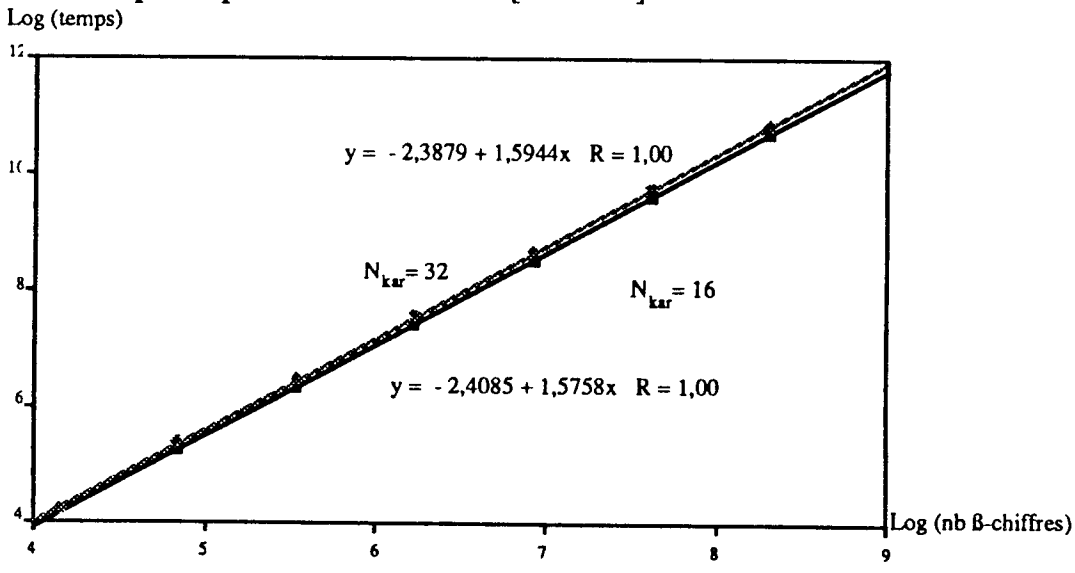


Fig. 2 Vérification de la complexité théorique : $Log_2 3 \approx 1{,}585$

La courbe ci-dessus permet de vérifier que le coût expérimental est :

$$T(n) \approx 9.n^{1,58}.10^{-5} \text{ s}$$

Ce qui est en accord avec la complexité théorique.

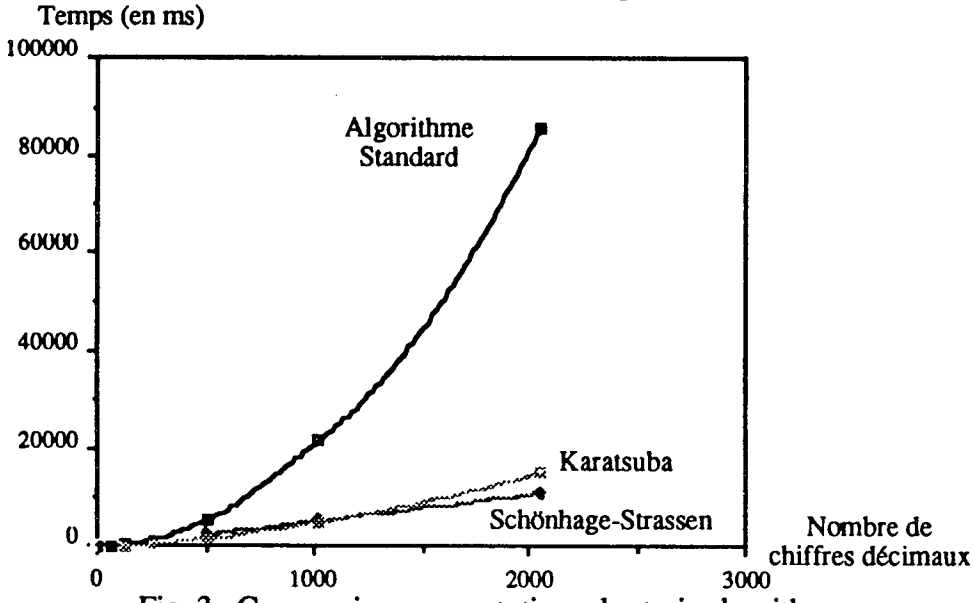


Fig. 3 Comparaison asymptotique des trois algorithmes

III. Algorithme de Karatsuba mixte généralisé

Nous nous intéressons maintenant à la multiplication d'un nombre u de n β-chiffres par un nombre v ayant p β-chiffres.

III.1. Présentation de l'algorithme - Calcul de la découpe optimale

Soit d un entier positif quelconque. En utilisant un schéma type Karatsuba, nous pouvons écrire :

$$u.v = \Pi_1.\beta^{2.d} + (\Pi_3-\Pi_1-\Pi_2).\beta^d + \Pi_2 \qquad (1)$$

$$\text{avec} \quad \begin{cases} \Pi_1 = (u / \beta^d).(v / \beta^d) \\ \Pi_2 = (u \,\%\, \beta^d).(v \,\%\, \beta^d) \\ \Pi_3 = (u / \beta^d + u \,\%\, \beta^d).(v / \beta^d + v \,\%\, \beta^d) \end{cases} \qquad (2)$$

Il est donc possible quelles que soient les tailles de u et v d'obtenir leur produit à partir de trois sous-multiplications. Le problème est de choisir l'entier d qui minimise le côut de l'opération totale.

Soit ϕ (n,p) le coût de la multiplication de u par v, et soit $\varphi_{n,p}$ (d) le coût pour l'operation (1) avec découpe en d. Nous pouvons écrire d'après (2) :

$$\varphi_{n,p}(d) = \phi(n-d, p-d) + \phi(\max(n-d,d), \max(p-d,d)) + \phi(d,d) \qquad (3)$$

Soit d*(n,p) la découpe optimale qui minimise ϕ (n,p). La fonction d* est donc caractérisée par :

$$\boxed{\varphi_{n,p}(d^*_{n,p}) = \underset{d\in N^*}{\text{Min}} \left\{\phi(n\text{-}d,p\text{-}d) + \phi(\text{Max}(n\text{-}d,d),\text{Max}(p\text{-}d,d)) + \phi(d,d)\right\}} \quad (4)$$

Proposition 1

Si ϕ est convexe et super-linéaire, alors :

$$\exists (n_0, p_0) / \forall (n,p) \quad n>n_0 \text{ et } p> p_0 \Rightarrow d^*(n,p) = \frac{Max\,(n,\,p)}{2}$$

Preuve :

ϕ super-linéaire équivaut à : $\dfrac{\phi(n,p)}{n+p} \underset{n,p\to\infty}{\longrightarrow} \infty$

Sous les hypothèses de la proposition, on obtient donc que la somme des trois valeurs : ϕ(n-d,p-d), ϕ(max(n-d,d),max(p-d,d)) et ϕ(d,d) est du même ordre que la plus grande de ces trois valeurs.
On a donc asymptotiquement :

$$\varphi_{n,p}(d^*_{n,p}) = \underset{d\in N^*}{\text{Min}}\ \text{Max}\ \left\{\phi(n\text{-}d,p\text{-}d);\ \phi(\text{Max}(n\text{-}d,d),\text{Max}(p\text{-}d,d));\ \phi(d,d)\right\}$$

Supposons p ≤ n. Trois cas sont alors à distinguer :

1°/ si d ∈ {0,...,p/2} : on a : n-d ≥ p-d ≥ d. D'où l'on tire :

Max {ϕ(n-d, p-d), ϕ(max(n-d,d), max(p-d,d)), ϕ(d,d)} = ϕ(n-d, p-d)

Or, dans {0,...,p/2}, la valeur minimale de ϕ(n-d, p-d) est atteinte pour d = p/2, et vaut : ϕ(n - p/2, p/2)

2°/ si d ∈ {p/2+1,...,n/2} :

On peut ici distinguer le cas d ≥ p du cas d < p.
Mais, dans les deux cas on a : n-d ≥ d ≥ p-d. D'où l'on tire :

Max {ϕ(n-d, p-d), ϕ(max(n-d,d), max(p-d,d)), ϕ(d,d)} = ϕ(n-d,d)

Or, sur {p/2,...,n/2}, ϕ(n-d, d) est minimal lorsque l'écart entre (n-d) et d est minimal car le problème est symétrique par rapport aux deux quantités (n-d) et d, et la fonction ϕ est convexe.

Autrement dit, la valeur minimale de ϕ(n-d, d) est atteinte pour d=n/2, et vaut : ϕ(n/2, n/2)

3°/ si $d \in \{n/2+1,\ldots,+\infty\}$: on a : $d \geq n-d \geq p-d$. D'où l'on tire :

$$\text{Max}\ \{\phi(n-d, p-d), \phi(\max(n-d,d), \max(p-d,d)), \phi(d,d)\} = \phi(d, d)$$

Or, sur $\{n/2+1,\ldots,+\infty\}$, la valeur minimale de $\phi(d, d)$ est atteinte pour $d=n/2+1$, et vaut : $\phi(n/2+1, n/2+1)$

La plus petite valeur est donc obtenue dans le cas (2), et on a : $d^*(n,p) = \left\lfloor \frac{n}{2} \right\rfloor$

La valeur correspondante de $\varphi_{n,p}$ ($d^*(n,p)$) est alors obtenue en remplaçant d par sa valeur dans (4) :

$$\varphi_{n,p}(d^*(n,p)) = \phi\left(\left\lceil \frac{n}{2} \right\rceil, p-\left\lfloor \frac{n}{2} \right\rfloor\right) + \phi\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right) + \phi\left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor\right) \quad (5)$$

Corollaire

Une technique de type Karatsuba permet de multiplier deux entiers de n et p β-chiffres en un nombre $M_{kar}(n,p)$ de multiplications dans U qui vérifie :

$$\begin{cases} M_{kar}(n,1) = n.\tau_{mul} \\ M_{kar}(n,p) = M_{kar}\left(\left\lceil \frac{n}{2} \right\rceil, p-\left\lfloor \frac{n}{2} \right\rfloor\right) + M_{kar}\left(\left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor\right) + M_{kar}\left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor\right) \end{cases}$$

Preuve :

Avec la conjecture que la multiplication est une fonction convexe (hypothèse tout à fait naturelle sur le modèle de calcul choisi), la récurrence donnée ci-dessus se déduit directement, à partir de (3) et (5). $M_{kar}(n,1)$ est obtenu en multipliant par l'algorithme standard les deux entiers.

III.2. Complexité de l'algorithme de Karatsuba généralisé

Proposition 2

La complexité $M_{kar}(n,p)$ de l'algorithme type Karatsuba, basé sur la découpe optimale $d^*(n,p)$, est :

$$M_{kar}(n,p) = O(\, n.p^{Log_2 3 - 1}\,)$$

Preuve :

Lemme : $M_{kar}(2^d,2^d) = 3^d.\tau_{mul}$

Preuve:

$M_{kar}(2^d,2^d) = 2.M_{kar}(2^d-2^{d-1},2^{d-1}) + M_{kar}(2^{d-1},2^{d-1}) = 3.M_{kar}(2^{d-1},2^{d-1})$

D'où l'on tire le résultat.

Supposons $n \geq p \geq n/2$. On a alors:

$$M_{kar}(n,p) = M_{kar}(n/2,p-n/2) + 2.M_{kar}(n/2,n/2)$$
$$= O(M_{kar}(n/2,n/2)) + M_{kar}(n/2,p-n/2)$$

Or, $(p-n/2 < n/2)$ par hypothèse. Donc, la fonction de coût de la multiplication étant croissante, on a : $M_{kar}(n/2,p-n/2) \leq M_{kar}(n/2,n/2)$

D'où l'on déduit : $M_{kar}(n,p) = O(M_{kar}(n/2,n/2))$
Et en utilisant le lemme : $M_{kar}(n,p) = O(n^{Log_2 3})$

Supposons $n \geq p$ et $p \leq n/2$. On a alors:

$$M_{kar}(n,p) = 2.M_{kar}(n/2,p)$$

Posons $n = 2^d$ et $p = 2^e$. On a alors : $M_{kar}(2^d,2^e) = 2.M_{kar}(2^{d-1},2^e)$
et par une récurrence immédiate : $M_{kar}(2^d,2^e) = 2^{d-e}.M_{kar}(2^e,2^e)$

Et finalement :

$$M_{kar}(n,p) = \frac{n}{p} . p^{Log_2 3} = n.p^{Log_2 3 - 1}$$

Cette dernière formule étant aussi vérifiée dans le cas $n \geq p \geq n/2$ étudié ci-dessus, on a démontré le résultat énoncé.

Conclusion Générale

L'algorithme de Karatsuba généralisé a une complexité théorique inférieure à l'algorithme standard, même lorsque les opérandes sont de tailles distinctes

Remarques

1° Les remarques faites sur l'algorithme de Karatsuba classique à propos de l'implantation et du coût mémoire sont encore valables : il suffit de considérer le plus grand des deux entiers pour allouer la place suffisante.

2° Par analogie à la constante N_{kar}, il est essentiel de calculer les valeurs critiques de n et p pour lesquelles l'algorithme standard est plus rapide.

Chapitre V

DIVISION RAPIDE D'ENTIERS EN PRECISION INFINIE

Le but de ce paragraphe est de montrer que la division d'entiers en précision infinie a la même complexité que la multiplication. Cela ne peut être déduit de l'algorithme Pope et Stein, car les multiplications effectuées dans cet algorithme sont des produits de ß-entiers par des ß-chiffres. Or, nous savons que, dans ce cas, l'algorithme de multiplication standard réalise l'optimal.

L'algorithme proposé réalise un intéressant compromis : d'une part, il a le même ordre de complexité que l'algorithme de Cook pour la division d'un entier de 2.n ß-chiffres par un autre de n ß-chiffres [AHU74b]. D'autre part, il est particulièrement adapté à la division de deux nombres quelconques : dans ce cas sa complexité est meilleure que celle de l'algorithme de Cook. Si m est le nombre de chiffres du dividende et n celui du diviseur (m>n), alors la complexité de cet algorithme est :

$$D_\beta(m, n) = (5 + \frac{m+1}{2.n-1}).M_\beta(n, n) + M_\beta(m, m-n) + M_\beta(n, m-n) + O(n,m)$$

Par ailleurs, bien que construit sur un itération de Newton, cet algorithme calcule les itérés successifs sans troncature, et est par suite minimal: tous les chiffres calculés sont pleinement exploités pour obtenir le résultat.

Soit u, v deux entiers; on suppose u>v et on désire calculer q = u/v.

$$q = \lfloor u / v \rfloor \quad \Leftrightarrow \quad q = \lfloor u.\rho \rfloor \quad \text{avec } \rho = \frac{1}{v}$$

L'idée de base proposée par Cook [COO66] est d'approcher ρ par son réciproque $\tilde{\rho}$ défini par :

Définition 1

On appelle réciproque de Cook de v en base β l'entier r défini par :

$$r = \lfloor \beta^{2.n+1} / v \rfloor \quad avec \quad n = \lfloor Log_\beta v \rfloor$$

En fait, pour calculer la valeur de q, il est clair que le choix du nombre α_u de β-chiffres de ρ est lié au nombre de β-chiffres de u. Il faut donc généraliser la notion de réciproque au calcul d'un certain nombre α de chiffres significatifs de l'inverse rationnel d'un entier.

Définition 2 *(généralisation du réciproque de Cook)*

On appelle α-réciproque en base β d'un entier v l'entier $\widetilde{\rho_\alpha}$ défini par :

$$\widetilde{\rho_\alpha} = \left\lfloor \frac{\beta^\alpha}{v} \right\rfloor$$

Proposition 1

Soit $u=[u_p,\ldots,u_0]_\beta$ *et* $v=[v_n,\ldots,v_0]_\beta$ *deux entiers, avec* $v_n \neq 0$ *et* $u_p \neq 0$. *Alors :*

$$\left\lfloor \frac{u}{v} \right\rfloor = \left\lfloor u.\widetilde{\rho_{p+1}}.\beta^{-p-1} \right\rfloor + c \quad \text{avec} \quad c \in \{0,1\} \text{ terme correcteur}$$

où $\widetilde{\rho_{p+1}}$ *est le (p+1)-réciproque base* β *de v.*

Preuve :

On a : $\widetilde{\rho_{p+1}} = \left\lfloor \frac{\beta^{p+1}}{v} \right\rfloor$ et soit $\varepsilon = \frac{\beta^{p+1}}{v} - \left\lfloor \frac{\beta^{p+1}}{v} \right\rfloor$ $(0 \leq \varepsilon < 1)$

D'où l'on tire : $\left\lfloor \frac{u}{v} \right\rfloor = \left\lfloor u.\widetilde{\rho_{p+1}}.\beta^{-p-1} + u.\varepsilon.\beta^{-p-1} \right\rfloor$ et $0 \leq u.\varepsilon.\beta^{-p-1} < 1$

Par propriété de la partie entière d'un nombre positif, on en déduit donc le résultat.

Remarques :

Le choix de α_u est dans un certain sens *optimal*, car la p ème β-décimale de $\frac{1}{v}$ intervient de manière *directe* dans le calcul du plus petit β-chiffre de (u/v).

Par ailleurs, les n premières β-décimales de $\frac{1}{v}$ sont nulles, donc seules les (p-n) premières β-décimales de $\frac{1}{v}$ sont à calculer.

I. Calcul du α-réciproque par une itération type Newton

Soit v un entier positif (supposé plus grand que 2) quelconque. On s'intéresse ici au calcul du α-réciproque $\rho = \frac{\beta^\alpha}{v}$ de v.

En fait, de façon à éviter la division par v, ρ peut être considéré comme la solution de l'équation non linéaire - du type $f(\rho) = 0$ - suivante :

$$\frac{1}{\rho} - \frac{v}{\beta^\alpha} = 0$$

ρ peut donc être considéré comme le point fixe de l'itération de Newton associée à cette équation. Cette itération est obtenue en considérant le développement limité de f à l'ordre 1 au voisinage d'une racine y de f :

$$y = x - \frac{f(x)}{f'(x)} + O(x-y)^2$$

D'où l'on tire : $y = 2.x - x^2.v.\beta^{-\alpha} + O(x-y)^2$

ρ est donc point fixe de l'itération :

$$\rho_{n+1} = 2.\rho_n - \rho_n^2.v.\beta^{-\alpha}$$

Soit $\varepsilon_n = \rho_n - \rho$ l'erreur commise à chaque étape. On a :

$$\varepsilon_n = -\frac{v}{\beta^\alpha}.\varepsilon_{n-1}^2$$

Soit ρ_0 une valeur approchée de ρ et $\varepsilon_0 = \rho_0 - \rho$. On a alors :

$$\boxed{\varepsilon_n = (-1)^n.\left(\frac{v}{\beta^\alpha}\right)^n.\varepsilon_0^{2.n}}$$

L'itération de Newton convergeant quadratiquement si ε_0 est "assez" petit, on obtient donc le "double de précision" sur ρ à chaque étape. Mais les itérés sont rationnels : or on aimerait calculer dans les entiers, puisque le résultat $\widetilde{\rho_\alpha}$ cherché est un entier.

Il s'agit donc de construire à partir de cette itération dans $\mathbb{Q}$, une itération dans $\mathbb{Z}$ qui converge vers $\widetilde{\rho_\alpha}$ en un nombre fixé d'itérations.
L'idée de base est à chaque étape d'effectuer les calculs modulo une certaine borne. Puis, pour l'étape suivante, de doubler cette borne, de façon à doubler le nombre de chiffres significatifs obtenus - caractéristique d'une convergence quadratique -.

Le schéma général de l'itération (I) peut alors s'écrire :

$$\left|\begin{array}{ll}
\begin{cases} B_0 \\ r_0 \end{cases} & \begin{array}{l} \text{borne initiale pour les calculs} \\ \text{approximation de } (B_0/v) \end{array} \\
B_n = B_{n-1}^2 & \text{la précision est doublée à chaque étape} \\
(\widehat{I_n}) \begin{cases} y_0^{(n+1)} = 2.r_n.B_n - r_n^2.v \\ y_k^{(n+1)} = 2.y_{k-1}^{(n+1)} - \dfrac{(y_{k-1}^{(n+1)})^2.v}{B_{n+1}} \end{cases} & \begin{array}{l} y_0^{(n+1)} \text{ est une approximation de } (B_{n+1}/v) \\ k=1\ldots K \begin{cases} \text{on ajuste } y_0^{(n+1)} \text{ par} \\ \text{une itération de Newton} \end{cases} \end{array} \\
r_{n+1} = \lfloor y_K^{(n+1)} \rfloor & \text{résultat entier final de l'étape}
\end{array}\right.$$

Soit $\delta_n = \frac{B_n}{v} - r_n$ l'erreur commise sur le $n^{ième}$ itéré; on pose : $\delta_0 = \frac{B_0}{v} - r_0$.

En pratique, la constante K de l'itération $\widehat{I_n}$ n'est pas fixée : on boucle sur cette itération tant que la différence entre deux termes $y_n^{(k)}$ consécutifs est supérieure à une certaine valeur - en général on choisit B_{n-1} comme critère d'arrêt -. L'analyse de la complexité est alors difficile.

II. Le problème de la convergence

Ici, il s'agit d'assigner à la constante K la plus petite valeur possible. Il faut donc déterminer le plus petit entier K qui assure la convergence, et choisir

les valeurs initiales B_0 et r_0 qui permettent d'obtenir un certain nombre de β-chiffres de ρ dans r_0.

Nous considérons dans la suite deux types de convergences pour cette itération : "sur-quadratique" et "sous-quadratique".

Définition 3 :
L'itération I converge sur-quadratiquement si et seulement si :

$$\begin{cases} \exists\, \alpha > 1, \exists\, N_0 / & \forall\, n > N_0 & \delta_{n+1} < \alpha.\delta_n^2 \\ \forall\, \beta \leq 1, \forall\, N_0 / & \exists\, n > N_0 & \delta_{n+1} > \beta.\delta_n^2 \end{cases}$$

Définition 4 :
L'itération I converge sous-quadratiquement si et seulement si :

$$\exists\, \alpha \leq 1, \exists\, N_0 / \quad \forall\, n > N_0 \quad \delta_{n+1} < \alpha.\delta_n^2$$

Deux cas peuvent se produire :

- si la convergence est sur-quadratique, la justesse d'un certain nombre de chiffres à une étape ne garantit pas la justesse de ces mêmes chiffres à l'étape suivante.
- par contre, si la convergence est sous-quadratique, alors l'exactitude d'un certain nombre de chiffres à une étape garantit la justesse de ces mêmes chiffres pour les étapes suivantes.

III. Sur-convergence et itération approchée

Il est clair que si l'itération (I) est sur-quadratique, alors, à chaque étape, l'erreur entre $\frac{B_n}{v}$ et r_n croît, même lorsque les conditions initiales sont telles que l'itération permet d'obtenir une approximation $\frac{r_n}{B_n}$ de plus en plus précise de $\frac{1}{v}$.

Lemme 1 :
$K \geq 1$ est une condition nécessaire pour que la convergence soit sous-quadratique.

Preuve :

Si K = 0. On a alors : $\delta_{n+1} < v.\delta_n^2 + 1$, et la convergence est sur-quadratique (puisque v>1).

Si l'on choisit B_0 tel que $B_0 > v$ et r_0 tel que $|\delta_0| \leq 1$ (cas optimal pour l'approximation entière d'un rationnel), on obtient :

$$\delta_n = v^{2^n-1}.\delta_0^{2^n} < B_0^n$$

Or, pour $n \geq 2$:

$$r_n = \frac{B_0^{2^n}}{v} - \delta_n > B_0^{2^n-1} - B_0^n > B_0^{2^n-2}$$

On en déduit donc qu'à chaque étape, au moins les $(2^n - 2 - n)$ premiers B_0-chiffres de r_n sont exacts.

Proposition 2.1 :

Soit r_n défini par :

$$\begin{cases} B_0 = \beta^{(2.t+1)} \text{ avec } t = \lfloor \mathrm{Log}_\beta v \rfloor \text{ et } r_0 = \left\lfloor \frac{B_0}{v} \right\rfloor \\ \begin{cases} B_n = B_{n-1}^2 \\ r_{n+1} = 2.r_n.B_n - r_n^2.v \end{cases} \text{ pour } n \geq 1 \end{cases}$$

Pour toute valeur de n, r_n vérifie : $\left| \left\lfloor \frac{r_n}{B_0^{2^n-2-n}} \right\rfloor - \frac{B_0^{2^n+2+n}}{v} \right| \leq 1$

Preuve

Elle est immédiate à partir des remarques ci-dessus, en tenant compte que les $(2^n - 2 - n)$ B_0-chiffres de $\frac{B_{n+1}}{v}$ correspondent à $\frac{B_0^{2^n+2+n}}{v}$.

Les schémas ci-dessous montrent le comportement de r_{n+1} à partir de r_n lors d'une iteration, à partir de l'approximation qui est effectuée.

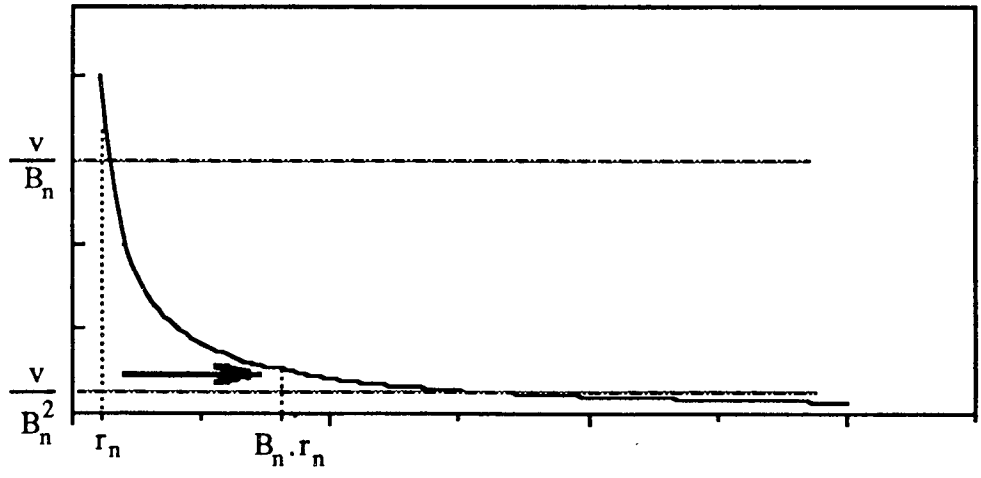


Fig. 1 Approximation grossière de $\left\lfloor \frac{B_n^2}{v} \right\rfloor$ à partir de r_n

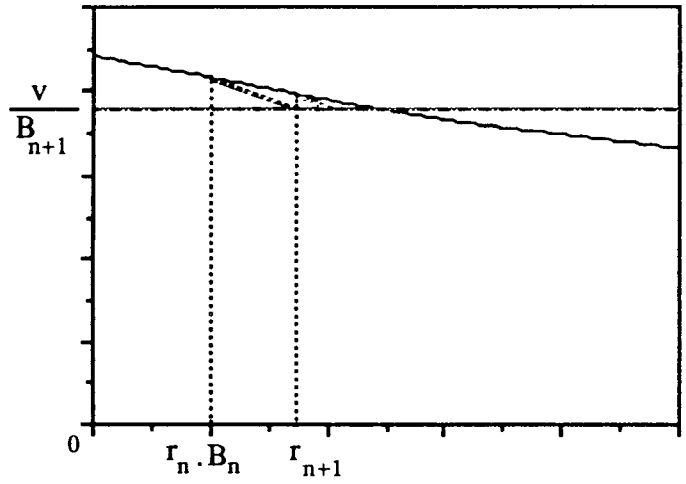


Fig. 2 Calcul de r_{n+1} à partir d'une itération type Newton

IV. Sous-convergence et itération exacte

Il est clair que si l'itération (I) est sous-quadratique, et que les conditions initiales sont telles que δ_0 est majoré par 1, alors pour toute valeur de n, δ_n sera majoré par 1, ce qui assure des approximations successives de $\lfloor \frac{B_n}{v} \rfloor$ "exactes" (l'erreur étant majorée par 1, l'approximation est quasi-optimale).

Il s'agit donc d'étudier pour quelles valeurs de K la convergence est sous-quadratique, puis d'étudier des conditions initiales satisfaisantes.

Lemme 2 :
K=1 et est une condition suffisante pour que la convergence soit sous-quadratique.

Preuve :
Si K = 1, on a alors :

$$\delta_{n+1} = \frac{B_n^2}{v} - 2.y_0^{(n+1)} + \frac{\left(y_0^{(n+1)}\right)^2.v}{B_n^2} + \left(\left\lceil \frac{\left(y_0^{(n+1)}\right)^2.v}{B_n^2} \right\rceil - \frac{\left(y_0^{(n+1)}\right)^2.v}{B_n^2} \right)$$

D'où l'on tire : $$\delta_{n+1} < \frac{v}{B_n^2} . \left(\frac{B_n^2}{v} - y_0^{(n+1)} \right)^2 + 1$$

Et, en remplaçant $y_0^{(n+1)}$ par sa valeur :

$$\delta_{n+1} < \frac{v^3}{B_n^2} .\delta_n^4 + 1$$

Avec un bon choix des conditions initiales, la convergence est alors au moins sous-quadratique.

Adéquation des conditions initiales pour une itération exacte

Supposons que l'on veuille assurer une itération "exacte", c'est à dire telle qu'à toute étape r_n soit égal à $\left\lfloor \frac{B_n}{v} \right\rfloor$.

Partant du calcul du réciproque de Cook de v, on obtient les conditions initiales :

$$\begin{cases} B_0 = \beta^{(2.t+1)} \text{ avec } t = \lfloor \mathrm{Log}_\beta v \rfloor \\ r_0 = \left\lfloor \frac{B_0}{v} \right\rfloor \quad \text{et} \quad |\delta_0| < 1 \end{cases}$$

Il est donc clair que, pour tout n, $\delta_n < 2$. On en déduit alors - r_n et $\left\lfloor \frac{B_n}{v} \right\rfloor$ étant entiers - :

$$\left| r_n - \left\lfloor \frac{B_n}{v} \right\rfloor \right| \leq 1$$

L'approximation r_n de la valeur $\left\lfloor \frac{B_n}{v} \right\rfloor$ est donc de *précision fixe*.

Proposition 2.2 :

Soit r_n défini par :

$$\begin{cases} B_0 = \beta^{(2.t+1)} \text{ avec } t = \lfloor \mathrm{Log}_\beta v \rfloor ; \text{ et soit } B_n = B_{n-1}^2 \\ r_0 = (B_0/v) \qquad r_0 \text{ est le 1-réciproque de Cook de v} \\ \begin{cases} r_{n+1/2} = 2.r_n.B_n - r_n^2.v & \text{itération entière} \\ r_{n+1} = 2.r_{n+1/2} - \left\lfloor \frac{r_{n+1/2}^2.v}{B_{n+1}} \right\rfloor & \text{correction} \end{cases} \end{cases}$$

Pour toute valeur de n, r_n vérifie :

$$\left| r_n - \left\lfloor \frac{B_n}{v} \right\rfloor \right| \leq 1$$

L'algorithme de passage d'un itéré au suivant est représenté ci-dessous.

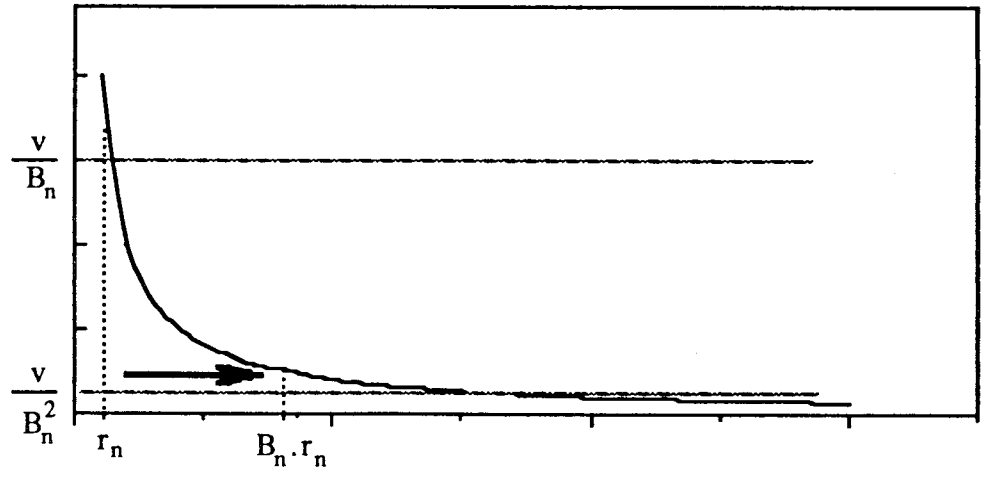


Fig. 3 Approximation grossière de $\left\lfloor \frac{B_n^2}{v} \right\rfloor$ à partir de r_n
(identique à l'itération précédente : fig.1)

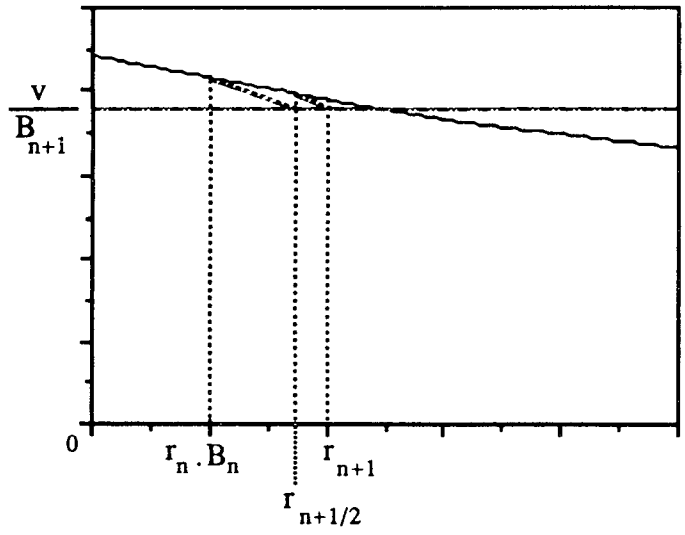


Fig. 4 Calcul de r_{n+1} à partir de deux itérations type Newton

V. Itération type Newton avec correction en temps constant

Nous avons vu qu'une convergence sous-quadratique est nécessaire si l'on veut à chaque étape une approximation à précision fixe de $\left\lfloor \frac{B_n}{v} \right\rfloor$. Il est cependant clair que la conclusion 2 - et la correction par une itération type Newton- impose un calcul complexe d'une part, et d'autre part donne un résultat très -trop- fort : à partir d'une convergence sur-quadratique, on obtient une convergence sous-quadrique, pour n'utiliser finalement que le fait que la convergence est sous-quadratique !

Nous nous intéressons donc maintenant à l'itération de Newton entière décrite précédemment (§III) :

$$\begin{cases} B_0 = \beta^{(2.t+1)} \text{ avec } t = \lfloor \mathrm{Log}_\beta v \rfloor \text{ et } r_0 = \left\lfloor \frac{B_0}{v} \right\rfloor \\ \begin{cases} B_n = B_{n-1}^2 \\ r_{n+1} = 2.r_n.B_n - r_n^2.v \end{cases} \quad \text{pour } n \geq 1 \end{cases}$$

Soit $\Delta_n = B_n - r_n.v$ le *reste* obtenu par l'approximation r_n de $\left\lfloor \frac{B_n}{v} \right\rfloor$.
Il est clair que l'on a :

$$0 \leq \delta_n < 1 \quad \Leftrightarrow \quad 0 \leq \Delta_n < v$$

Or, à chaque étape, Δ_n est donné par la récurrence :

$$\Delta_{n+1} = \Delta_n^2$$

Supposons $r_n = \left\lfloor \frac{B_n}{v} \right\rfloor$. On a alors : $0 \leq \Delta_n < v$. On en déduit donc : $\Delta_{n+1} < v^2$.

Le terme correcteur τ_{n+1} à imputer à r_{n+1} pour obtenir $\Delta_{n+1} < v$ peut donc être évalué en divisant Δ_{n+1} par v.
Posons $\tau_{n+1} = (\Delta_{n+1} / v)$. Il vient :

$$0 \le B_{n+1} - (r_{n+1} + \tau_{n+1}).v < v$$

$(r_{n+1} + \tau_{n+1})$ est donc le quotient euclidien exact $\left\lfloor \frac{B_{n+1}}{v} \right\rfloor$.

Proposition 3 :

Soit r_n défini par :

$$\begin{cases} B_0 = \beta^{(2.t+1)} \quad \text{avec } t = \lfloor \mathrm{Log}_\beta v \rfloor ; & \text{et soit } B_n = B_{n-1}^2 \\ r_0 = (B_0/v) & r_0 \text{ est le 1-réciproque de Cook de v} \\ \begin{cases} r_{n+1/2} = 2.r_n.B_n - r_n^2.v & \text{itération entière} \\ r_{n+1} = r_{n+1/2} + \left\lfloor \frac{B_{n+1} - r_{n+1/2}.v}{v} \right\rfloor & \text{correction} \end{cases} \end{cases}$$

Pour toute valeur de n, r_n vérifie : $r_n = (B_n / v)$

Preuve

La construction de l'itération est une démonstration de la proposition. Nous donnons ici une démonstration directe, par récurrence.

De par les conditions initiales, on a : $r_0 = \left\lfloor \frac{B_0}{v} \right\rfloor$.
Supposons que r_n est le quotient de la division euclidienne de B_n par v.
Soit $\Delta_n = B_n - r_n.v$: Δ_n est alors le reste euclidien de la division de B_n par v.

$$\begin{aligned} \Delta_{n+1} &= B_{n+1} - r_{n+1}.v \\ &= B_{n+1} - 2.r_n.B_n.v - r_n^2.v^2 - v.\left\lfloor \frac{B_{n+1} - 2.r_n.B_n.v - r_n^2.v^2}{v} \right\rfloor \\ &= (B_n - r_n.v)^2 - v.\left\lfloor \frac{(B_n - r_n.v)^2}{v} \right\rfloor \\ &= v.\left(\frac{\Delta_n^2}{v} - \left\lfloor \frac{\Delta_n^2}{v} \right\rfloor \right) \end{aligned}$$

D'où l'on tire : $0 \le \Delta_{n+1} < v$
Comme Δ_{n+1} est un entier, on en déduit que Δ_{n+1} est le reste euclidien de la division de B_{n+1} par v.

Par suite, r_{n+1} est bien le quotient de la division euclidienne de B_{n+1} par v.

VI. Analyse du calcul : recentrage de l'itération

Différentes remarques permettent de simplifier le calcul de l'itération précédente. Nous nous attachons à les développer ici.

A chaque pas de l'itération précédente, on a : $0 \leq \frac{B_n}{v} - r_n < 1$

A la $(n+1)^{ième}$ itération, seuls les $\lfloor \mathrm{Log}_\beta B_n \rfloor$ ß-chiffres de poids faible de r_{n+1} sont donc modifiés - si l'on ne compte pas une éventuelle propagation de retenue, toujours possible mais de coût linéaire -.

Il est donc suffisant de considérer seulement ces ß-chiffres de poids faible, ce qui revient à "recentrer" l'itération à droite de zéro.

Posons : $\lambda_{n+1/2} = r_{n+1/2} - r_n.B_n$

$\lambda_{n+1/2}$ peut être considéré comme l'approximation par une itération de Newton des B_n chiffres de poids faible de r_{n+1}. On a translaté les chiffres inconnus à déterminer près de zéro.

En remplaçant $r_{n+1/2}$ par son expression, $\lambda_{n+1/2}$ s'écrit :

$$\lambda_{n+1/2} = r_n.\ (B_n - r_n.v\ )$$

Posant $\Delta_n = B_n - r_n.v$, on obtient : $\lambda_{n+1/2} = r_n.\Delta_n$

(Δ_n est forcément connu lorsque l'on effectue la correction présentée au paragraphe précédent).

Par analogie, posons $\Delta_{n+1/2} = B_{n+1} - r_{n+1/2}.v$; il vient :

$$\Delta_{n+1/2} = \Delta_n.B_n - \lambda_{n+1/2}$$

$\Delta_{n+1/2}$ est donc très facile à évaluer, puisqu'il correspond à la soustraction de Δ_n -décalé de B_n β-chiffres- et de $\lambda_{n+1/2}$.

A partir de la valeur de $\Delta_{n+1/2}$, il est possible de construire τ_{n+1}, correction à apporter à $r_{n+1/2}$ pour obtenir r_{n+1}, ainsi que Δ_{n+1} reste euclidien de B_{n+1} divisé par v.

En effet : $$\tau_{n+1} = \left\lfloor \frac{B_{n+1} - r_{n+1/2}.v}{v} \right\rfloor = \left(\Delta_{n+1/2} / v\right)$$

Et : $$\Delta_{n+1} = B_{n+1} - r_{n+1}.v = \left(\Delta_{n+1/2} \bmod v\right)$$

Proposition 4 :

Soit r_n défini par :

$$\begin{cases} B_0 = \beta^{(2.t+1)} \quad \text{avec } t = \lfloor \text{Log}_\beta v \rfloor ; \quad \text{et soit } B_n = B_{n-1}^2 \\ r_0 = (B_0 / v) \quad \text{et } \Delta_0 = (B_0 \,\%\, v) \\ \begin{cases} \lambda_{n+1/2} = \Delta_n . r_n \\ \Delta_{n+1/2} = \Delta_n^2 \end{cases} \\ \begin{cases} \tau_{n+1} = \Delta_{n+1/2} / v \quad \text{et} \quad \Delta_{n+1} = \Delta_{n+1/2} \bmod v \\ r_{n+1} = r_n . B_n + \lambda_{n+1/2} + \tau_{n+1} \end{cases} \end{cases}$$

Pour toute valeur de n, r_n et Δ_n vérifient : $r_n = (B_n / v)$ et $\Delta_n = (B_n \bmod v)$

Preuve

Elle se déduit directement de la conclusion 3, en utilisant les remarques ci-dessus. Nous proposons ici une démonstration directe, par récurrence.

De par les conditions initiales, on a : $r_0 = \left\lfloor \frac{B_0}{v} \right\rfloor$, et Δ_0 est le reste euclidien de la division de B_0 par v.

Supposons que r_n et Δ_n sont respectivement le quotient et le reste de la division euclidienne de B_n par v.
Montrons que Δ_{n+1} est alors le reste euclidien de la division de B_{n+1} par v.

$$\begin{aligned} \Delta_{n+1} &= \Delta_{n+1/2} \bmod v \\ &= \Delta_n^2 - \tau_{n+1} . v \\ &= B_{n+1} - 2 . r_n . B_n . v + r_n^2 . v^2 - v . \tau_{n+1} \\ &= B_{n+1} - v . (r_n . B_n + r_n . (B_n - r_n . v) + \tau_{n+1}) \end{aligned}$$

Or, par hypothèse de récurrence : $\Delta_n = B_n - r_n . v$
Comme $r_n . \Delta_n = \lambda_{n+1/2}$, et de par la valeur de r_{n+1}, on obtient :

$$B_{n+1} = r_{n+1} . v + \Delta_{n+1}$$

Par ailleurs, comme Δ_{n+1} est le reste d'une division euclidienne par v, Δ_{n+1} est entier et $0 \le \Delta_{n+1} < v$.
On a donc : $B_{n+1} = r_{n+1} . v + \Delta_{n+1}$ avec $0 \le \Delta_{n+1} < v$
On en déduit que Δ_{n+1} est le reste euclidien de la division de B_{n+1} par v.
Par suite, r_{n+1} est donc le quotient de la division euclidienne de B_{n+1} par v

VII. Etude du calcul de l'itération : Réciproque de Cook

VII.1. Conditions initiales

Nous avons vu que les conditions initiales de l'itération précédente dépendent du calcul du réciproque de Cook de v.

En fait, la démonstration montre qu'il n'est pas nécessaire de partir du réciproque de v; la seule condition nécessaire est que r_0 et B_0 soient tels que :

$$r_0 = \left\lfloor \frac{B_0}{v} \right\rfloor$$

Supposons que β soit un carré parfait, et soit $\beta^{1/2}$ sa racine.

Posons : $B_0 = \beta^t.\beta^{1/2}$ où $t = \lfloor \mathrm{Log}_\beta v \rfloor$

On a alors : $0 \le \left\lfloor \frac{B_0}{v} \right\rfloor \le \beta^{1/2}$

$\left\lfloor \frac{B_0}{v} \right\rfloor$ peut donc être calculé en temps linéaire par rapport à t : ce calcul est donc de coût "négligeable".

Conclusion : *on peut s'affranchir du calcul du réciproque de Cook de v dans l'estimation des conditions initiales de l'itération (4).*

VII.2. Division par v et réciproque de Cook

Il reste néanmoins qu'à chaque étape de l'itération (4), une division d'un entier inférieur au carré de v par v est à effectuer. D' après § II, cette division peut-être à chaque étape effectuée à l'aide d'une multiplication par le réciproque de v : le calcul du réciproque de Cook est donc inévitable.

Notre propos est ici de dégager un algorithme itératif de calcul du réciproque base β d'un entier quelconque. Cet algorithme peut être vu comme l'extension à une base β quelconque du calcul du réciproque base 2, décrit par Cook [COO66][AHU74b].

Soit $v = [v_n, v_{n-1}, \ldots, v_0]_\beta$ un β-entier supérieur à 2, avec $v_n \ge 1$. Nous supposerons en outre $n = 2^p - 1$ (pour plus de commodité, et sans restriction).

Définition :
On appelle réciproque de Cook de v l'entier r défini par :

$$r = (B_0 / v) \quad \text{avec} \quad B_0 = \beta^{2.n+1} \quad \text{et} \quad n = \lfloor \mathrm{Log}_\beta v \rfloor$$

Soit $r^{(k)}$ le réciproque de $v^{(k)} = [v_n, v_{n-1}, \ldots, v_{n-2^k+1}]_\beta$.
Le réciproque de v est alors $v^{(p)}$.

Par récurrence, on suppose connu $v^{(k)}$ et l'on cherche à construire $v^{(k+1)}$.

Posons $\Delta^{(k)} = \beta^{2^{k+1}-1} - r^{(k)}.v^{(k)}$: comme $r^{(k)}$ est le réciproque de $v^{(k)}$, on en déduit que : $0 \le \Delta^{(k)} < v^{(k)}$

La connaissance de $r^{(k)}$ permet de donner une première approximation de $r^{(k+1)}$. En effet, soit $C^{(k)} = v^{(k+1)} - v^{(k)}.\beta^{2^k}$. On a alors : $0 \le C^{(k)} < \beta^{2^k}$.

A l'aide de $C^{(k)}$, $r^{(k+1)}$ peut s'écrire :

$$r^{(k+1)} = \left\lfloor \frac{\beta^{2^{k+2}-1}}{v^{(k+1)}} \right\rfloor = \left\lfloor \frac{\beta^{2^{k+2}-1}}{\beta^{2^k}.v^{(k)} + C^{(k)}} \right\rfloor = \left\lfloor \beta^{2^k}.\frac{\beta^{2^{k+1}-1}}{v^{(k)} + C^{(k)}.\beta^{-2^k}} \right\rfloor$$

Compte-tenu de l'encadrement sur $C^{(k)}$, on déduit :

$$\beta^{2^k}.\left\lfloor \frac{\beta^{2^{k+1}-1}}{v^{(k)}+1} \right\rfloor \le r^{(k+1)} \le \beta^{2^k}.r^{(k)}$$

$\beta^{2^k}.r^{(k)}$ est donc une approximation - par valeur supérieure - de $r^{(k+1)}$: cette approximation est très facile à obtenir, mais il faut l'affiner.

De façon à éviter une division par $v^{(k)}$, rarement agréable -...-, $r^{(k)}$ peut être considéré comme la partie entière de la solution rationnelle de l'équation :

$$\frac{1}{x} - \frac{v^{(k+1)}}{\beta^{2^{k+2}-1}} = 0$$

L'itération de Newton associée à cette équation est (§ III.) :

$$x_{p+1} = 2.x_p - \frac{x_p^2.v^{(k+1)}}{\beta^{2^{k+2}-1}}$$

Posons : $x_0 = \beta^{2^k}.r^{(k)}$

Le premier itéré est : $x_1 = 2.\beta^{2^k}.r^{(k)} - r^{(k)2}.v^{(k+1)}.\beta^{-2^{k+1}+1}$

Comme pour le calcul du α-réciproque, il est difficile d'estimer l'erreur entre les itérés successifs et le réciproque -entier- cherché. Nous considérons donc l'erreur avec le réciproque *rationnel* -exact-.

Soit $\varepsilon = \frac{\beta^{2^{k+2}-1}}{v^{(k+1)}} - x_1$

En remplaçant x_1 par sa valeur, on obtient :

$$\varepsilon = \frac{v^{(k+1)}}{\beta^{2^{k+1}-1}}.\left(r^{(k)2} - 2.r^{(k)}.\frac{\beta^{3.2^k-1}}{v^{(k+1)}} + \frac{\beta^{6.2^k-2}}{v^{(k+1)2}} \right)$$

On en déduit :

$$\varepsilon = \frac{v^{(k+1)}}{\beta^{2^{k+1}-1}} \cdot \left(r^{(k)} - \frac{\beta^{2^{k+1}-1}}{\frac{v^{(k+1)}}{\beta^{2^k}}} \right)^2$$

Or : $$\frac{v^{(k+1)}}{\beta^{2^k}} = v^{(k)} + C^{(k)}.\beta^{-2^k}$$

Et $$v^{(k+1)} < \beta^{2^{k+1}-1}$$

On obtient ainsi un encadrement de l'erreur commise ε :

$$0 \le \varepsilon < \left(r^{(k)} - \frac{\beta^{2^{k+1}-1}}{v^{(k)} + C^{(k)}.\beta^{-2^k}} \right)^2$$

Or :

$$\left(r^{(k)} - \frac{\beta^{2^{k+1}-1}}{v^{(k)} + C^{(k)}.\beta^{-2^k}} \right) = \left(r^{(k)} - \frac{\beta^{2^{k+1}-1}}{v^{(k)}} \right) + \frac{\beta^{2^k}}{v^{(k)}}.\beta^{2^k-1} \cdot \left(1 - \frac{1}{1 + C^{(k)}.\beta^{-2^k}.v^{(k)-1}} \right)$$

$r^{(k)}$ est le réciproque de $v^{(k)}$; on a donc : $$-1 < r^{(k)} - \frac{\beta^{2^{k+1}-1}}{v^{(k)}} \le 0$$

Et, par ailleurs, comme $0 \le C^{(k)}.\beta^{-2^k}.v^{(k)-1} < 1$, on a :

$$0 \le \left(1 - \frac{1}{1 + C^{(k)}.\beta^{-2^k}.v^{(k)-1}} \right) \le \frac{C^{(k)}.\beta^{-2^k}}{v^{(k)}}$$

Nous avons supposé v normalisé, c'est à dire $v_n \ge 1$. On en déduit :

$$\frac{C^{(k)}}{\beta.v^{(k)}} < 1 \quad \text{et} \quad \frac{\beta^{2^k}}{v^{(k)}} < 1$$

Finalement, on obtient l'encadrement de l'erreur ε :

$$\boxed{0 \le \frac{\beta^{2^{k+2}-1}}{v^{(k+1)}} - x_1 < 4}$$

Il est alors très facile d'estimer l'erreur entre le réciproque $r^{(k+1)}$ de $v^{(k+1)}$ et la partie entière supérieure de x_1. On obtient l'encadrement :

$$\boxed{-1 \le r^{(k+1)} - \lceil x_1 \rceil \le 3}$$

Lemme 5.1 : Détermination approchée de $r^{(k+1)}$

Soit $r^{(k)}$ le réciproque de $v^{(k)}$, et $r^{(k+1)}$ le réciproque de $v^{(k+1)}$.
Soit ρ l'entier défini à partir de $r^{(k)}$ par :

$$\rho = 2.\beta^{2^k}.r^{(k)} - \left\lfloor \frac{r^{(k)2}.v^{(k+1)}}{\beta^{2^{k+1}-1}} \right\rfloor$$

Alors ρ vérifie :

$$-1 \leq r^{(k+1)} - \rho \leq 3$$

N.B. : Aho, Hopcroft et Ullman [AHU74] montrent, uniquement dans le cas où $\beta=2$, que l'on a $0\leq\varepsilon\leq6$; la borne donnée ici est donc plus fine.

Preuve

Elle se déduit directement de l'encadrement de $r^{(k+1)} - \lceil x_1 \rceil$ qui précède, à partir de la remarque :

$$\lceil x_1 \rceil = \left\lceil 2.\beta^{2^k}.r^{(k)} - \frac{r^{(k)2}.v^{(k+1)}}{\beta^{2^{k+1}-1}} \right\rceil = 2.\beta^{2^k}.r^{(k)} - \left\lfloor \frac{r^{(k)2}.v^{(k+1)}}{\beta^{2^{k+1}-1}} \right\rfloor$$

Correction

A partir de la valeur de ρ, il faut pouvoir être capable de calculer la valeur de $r^{(k+1)}$. Cela peut être réalisé grâce au lemme :

Lemme 5.2 : Détermination exacte de $r^{(k+1)}$

Soit ρ l'entier défini dans le lemme1, et soit ($\mathcal{A}$) l'algorithme suivant :

($\mathcal{A}$)

$c \leftarrow 0$; $u \leftarrow (\rho+1).v^{(k+1)}$;
$B_L \leftarrow -2$; $B_H \leftarrow 2$;
Tantque ($B_H \neq B_L$) *faire*
  si ($u \leq \beta^{2^{k+2}-1}$) *alors*
    $B_L \leftarrow c$; $c \leftarrow c+1$; $u \leftarrow u + v^{(k+1)}$;
  sinon
    $B_H \leftarrow c-1$; $c \leftarrow c-1$; $u \leftarrow u - v^{(k+1)}$;

L'algorithme ($\mathcal{A}$) calcule $c \in \{-2,\ldots,2\}$ tel que : $r^{(k+1)} = \rho + c + 1$

Preuve

Elle se déduit directement de la remarque suivante :

Soit x un entier. On a : $x = r^{(k+1)} \Leftrightarrow \begin{cases} x.v^{(k+1)} \leq \beta^{2^{k+2}-1} \\ (x+1).v^{(k+1)} > \beta^{2^{k+2}-1} \end{cases}$

Posons $x = (\rho+1)$.
D'après le lemme 1, $r^{(k+1)} = x + c$ avec $c \in I=\{-2,1,0,1,2\}$.
L'algorithme ($\mathcal{A}$) effectue en fait une recherche dichotomique sur l'intervalle I, pour déterminer l'unique $c \in I$ tel que:

$$\begin{cases} (x+c).v^{(k+1)} \leq \beta^{2^{k+2}-1} \\ (x+c+1).v^{(k+1)} > \beta^{2^{k+2}-1} \end{cases}$$

Il est donc clair qu'une fois l'algorithme ($\mathcal{A}$) exécuté, on a : $r^{(k+1)} = \rho+1+c$

Corollaire 5.3

La valeur exacte de $r^{(k+1)}$ peut être évaluée à partir de la valeur de ρ avec au plus une multiplication, trois tests et trois additions.
En négligeant les temps d'additions et de tests devant le temps de la multiplication, on en déduit que la complexité de cette correction est :

$$M\beta(2^{k+1}, 2^{k+1})$$

Preuve

Elle se déduit de l'algorithme (A), en considérant le nombre d'opérations exécutées. L'algorithme effectuant une recherche dichotomique parmi 5 valeurs, le pire cas nécessite $\lfloor \text{Log}_2 5 \rfloor + 1 = 3$ passages dans la boucle.
Par ailleurs, la seule multiplication nécessitée est celle de ρ par $v^{(k+1)}$.
Or, on a : $\text{Taille}(v^{(k+1)}) = 2^{k+1}$ et $2^k \leq \text{Taille}(\rho) \leq 2^{k+1}$
On en déduit que le coût de ce produit est borné par $M\beta(2^{k+1}, 2^{k+1})$.

Finalement, le coût maximal pour l'évaluation de cette boucle est donc :

$$3.A\beta(2^{k+1}, 2^{k+1}) + M\beta(2^{k+1}, 2^{k+1})$$

En négligeant le temps d'addition devant celui de multiplication (le premier étant linéaire, et le second étant sur-linéaire), on obtient le résultat énoncé.

Proposition 5 : Calcul du réciproque de Cook en base β

Soit $v = [v_n,\ldots,v_0]_\beta$ un β-entier quelconque tel que $n = 2^p-1$ et $v_n \neq 0$.
Soit ($\mathcal{R}$) l'algorithme suivant :

($\mathcal{R}$)

$r^{(0)} \leftarrow \beta / v_n$ (Cette opération est réalisée grâce à ldiv) (1)

Pour k = 0, 1, ..., p-1 faire

$v^{(k+1)} \leftarrow [v_n, v_{n-1}, \ldots, v_{n-2^{k+1}+1}]_\beta$

$r^{(k+1)} \leftarrow 2.\beta^{2^k}.r^{(k)} - \left\lfloor \frac{r^{(k)2}.v^{(k+1)}}{\beta^{2^{k+1}-1}} \right\rfloor$ (2)

$c \leftarrow 0$; $u \leftarrow (r^{(k+1)}+1).v^{(k+1)}$;

$B_L \leftarrow -2$; $B_H \leftarrow 2$;

Tantque ($B_H \neq B_L$) faire (3)

si ($u \leq \beta^{2^{k+2}-1}$) alors

$B_L \leftarrow c$; $c \leftarrow c + 1$; $u \leftarrow u + v^{(k+1)}$;

sinon

$B_H \leftarrow c - 1$; $c \leftarrow c - 1$; $u \leftarrow u - v^{(k+1)}$;

$\rho \leftarrow r^{(p)}$

L'algorithme ($\mathcal{R}$) calcule ρ, réciproque de Cook de v.

Preuve

La démonstration peut se faire par induction, en utilisant les deux lemmes précédents.

Montrons que, pour tout k = 0,...,p : $r^{(k)}$ est le réciproque de $v^{(k)}$.
Au rang k=0, $r^{(0)}$ est bien, d'après (1), le réciproque de $v^{(0)}$.
Supposons la propriété vraie pour k<p.
D'après le lemme 1, après l'étape (2), on a :

$$-1 \leq \left\lfloor \frac{\beta^{2^{k+2}-1}}{v^{(k+1)}} \right\rfloor - r^{(k+1)} \leq 3$$

D'après le lemme 2, on en déduit qu'après la boucle (3) - qui est en fait l'algorithme ($\mathcal{A}$) -, on a :

$$r^{(k+1)} = \left\lfloor \frac{\beta^{2^{k+2}-1}}{v^{(k+1)}} \right\rfloor$$

$r^{(k+1)}$ est donc le réciproque de Cook de $v^{(k+1)}$, et par induction jusqu'à l'ordre p, on en déduit que ρ est le réciproque de Cook de v.

Remarque : Recentrage de l'itération

Comme pour le calcul du α-réciproque, il est possible d'effectuer les opérations sur des entiers de taille moindre, en recentrant l'itération à droite de zéro. Néanmoins, et à la différence du calcul du α-réciproque, cette translation ne permet pas d'éliminer le pire cas dans lequel tous les chiffres déjà calculés sont modifiés.

VIII. Un algorithme de division

Les différentes propositions énoncées dans cette partie permettent de définir un algorithme de calcul du quotient et du reste euclidien de deux entiers u et v. On suppose u > v > 2. Cet algorithme se décompose en deux temps :

1/ Calcul de ρ, $(\lfloor \mathrm{Log}_\beta u \rfloor+1)$-réciproque de v.

Ce calcul s'effectue à l'aide de l'itération décrite dans la proposition 4. Les conditions initiales pour cette itération sont évaluées à l'aide de la proposition 5.

2/ Approximation du quotient q_1 : $q_1 = (\rho.u) / \beta^{\lfloor \mathrm{Log}_\beta u \rfloor+1}$

Ce calcul s'effectue à l'aide d'une simple multiplication.

3/ Ajustement du quotient : $q = q_1 + c$ avec $c \in \{0,1\}$.

Le calcul de q est effectué à partir de la comparaison entre le reste r_1 ($r_1 = u - q_1.v$) et v : si $r_1 \geq v$, alors $q = q_1+1$ et $r = r_1-v$; sinon, $q = q_1$ et $r = r_1$.

VIII.1. Conditions initiales et réciproque de Cook

Cet algorithme est basé sur la proposition 5. Pour pouvoir appliquer cette proposition, il faut supposer que $\lfloor \mathrm{Log}_\beta v \rfloor$ est de la forme 2^p-1. Lorsque cela n'est pas vérifié, il suffit de "normaliser" v, en le décalant d'un nombre suffisant de β-chiffres.

Soit t le nombre de β-chiffres de v, et soit p l'entier défini par :

$$p = \lceil \mathrm{Log}_2 t \rceil$$

Le réciproque de Cook C de v est défini par : $$C_v = \left\lfloor \frac{\beta^{2.t-1}}{v} \right\rfloor$$

Posons $w = v.\beta^{2^p-t}$. On a alors :

$$C_v = \left\lfloor \frac{\beta^{2.t-1}}{v} \right\rfloor = \left\lfloor \frac{\beta^{2.2^p-1}}{w} . \frac{1}{\beta^{2^p-t}} \right\rfloor = C_w / \beta^{2^p-t}$$

Le réciproque de Cook de v est donc obtenu en prenant les premiers β-chiffres significatifs du réciproque de Cook de w.

algorithme **Cook-réciproque** (v : β-entier) → r : β-entier ==
début
t ← taille_N (v); p ← ⌈Log₂ t ⌉;

Normalisation
$w \leftarrow v.\beta^{2^p-t}$; n ← 2^p-1;

Réciproque de Cook de w
r ← β / w_n ; (1)
Pour k = 0, 1, ..., p-1 faire
partie_w ← $[w_n, w_{n-1}, \ldots, w_{n-2^{k+1}+1}]_\beta$; (2)
r ← $2\beta^{2^k}.r - \left\lfloor \frac{r^2.\text{partie\_w}}{\beta^{k+1}} \right\rfloor$; (3)
c ← 0 ; u ← (r+1).partie_w ; (4)
B_L ← -2 ; B_H ← 2 ;
Tantque ($B_H \neq B_L$) faire
si ($u \leq \beta^{2^{k+2}-1}$) alors
B_L ← c; c ← c + 1; u ← u + partie_w ;
sinon
B_H ← c -1; c ← c - 1; u ← u - partie_w ;

Dénormalisation
r ← r / β^{2^p-t};
fin

VIII.2. Calcul du α-réciproque de v

On s'intéresse ici au calcul des α premiers chiffres significatifs de v^{-1}.
Nous avons vu que la proposition 4 permet le calcul des $2^n.(2.\lfloor \text{Log}_\beta v \rfloor+1)$ premiers ß-chiffres de v^{-1}, pour tout entier n.

Pour calculer α ß-décimales de v^{-1}, il suffit donc d'effectuer l'itération jusqu'au rang n_0 défini par :

$$n_0 = \left\lceil \mathrm{Log}_2 \left(\frac{\alpha}{2.\lfloor \mathrm{Log}_\beta v \rfloor + 1} \right) \right\rceil$$

L'algorithme s'écrit alors :

algorithme **α-réciproque** (v : ß-entier, α : entier) $\rightarrow$ r : ß-entier ==
début

$t \leftarrow$ taille_N (v); $p \leftarrow \lceil \mathrm{Log}_2\, t \rceil$;

$n_0 \leftarrow \left\lceil \mathrm{Log}_2 \left(\frac{\alpha}{2.t - 1} \right) \right\rceil$;

Initialisation

inv_Cook_v $\leftarrow$ Cook-réciproque (v); (1)

$\Delta \leftarrow$ inv_Cook_v ; $r \leftarrow \beta^{2.t-1} - v.\text{inv\_Cook\_v}$; (2)

α-réciproque de v

Pour n = 1, 2, ..., n_0 faire

$\lambda \leftarrow \Delta.r$; (3)

$\Delta \leftarrow \Delta^2$; (4)

$\tau \leftarrow (\Delta.\text{inv\_Cook\_v}) / \beta^{2.t-1}$; $\Delta \leftarrow \Delta - v.\tau$; (5)

$r \leftarrow r.\beta^{2^n.(2.t-1)} + \lambda + \tau$; (6)

fin

VIII.3. La division euclidienne de u par v

L'algorithme de division de u par v peut alors être écrit à l'aide du calcul du ($\lfloor \mathrm{Log}_\beta u \rfloor$+1)-réciproque de v. Les calculs du quotient et du reste sont effectués simultanément (le reste est nécessaire à l'ajustement du quotient - cf. §II.).

algorithme **division_Newton** (u, v : β-entier) → q, r : β-entier ==
début
$T_u \leftarrow$ taille_N (u); $T_v \leftarrow$ taille_N (v);

Calcul du ($\lfloor \text{Log}_\beta u \rfloor$+1)-réciproque de v et approche du quotient
$\rho \leftarrow \alpha$-réciproque (v, $T_u + 1$); (1)
$q \leftarrow \lfloor u.\rho.\beta^{-T_u-1} \rfloor$; (2)

Calcul du reste et Ajustement du quotient
$r \leftarrow u - q.v$; (3)
si ($r \geq v$) alors
    $r \leftarrow r - v$;
    $q \leftarrow q+1$;
fin

IX. Complexité

La complexité de l'algorithme de division est liée à la complexité du calcul du réciproque de Cook et du α-réciproque d'un entier.
Soient u et v deux entiers quelconques ayant respectivement m et n β-chiffres.

Nous noterons :

- RC_β(n) la complexité du calcul de *Cook-réciproque (v)*
- $R_\beta(\alpha, n)$ la complexité du calcul de *α-réciproque (α , v)*
- D_β(m, n) la complexité de *division_Newton (u, v)*

Nous supposerons en outre que la multiplication (resp. l'addition) de u et v est $M_\beta(m,n)$ (resp. $A_\beta(m,n)$), et que la fonction $M_\beta(m,n)$ est symétrique et vérifie :

$$M_\beta(2.m, 2.n) = 2.M_\beta(m, n) + O(m,n)$$

Cette supposition induit donc :

$$\sum_{k=0}^{p-1} M_\beta(2^k, 2^k) \approx M_\beta(2^p, 2^p)$$

Proposition 6

$$RC_\beta (n) = 5.M_\beta(n,n) + O(n)$$

Preuve

Nous considérons l'algorithme Cook-réciproque appliqué à un nombre ayant n β-chiffres. Posons $p = \lceil \mathrm{Log}_2\, n \rceil$.
L'étape (1) -qui correspond à l'initialisation- coûte une opération de division longue (*ldiv*).
L'étape (2) a un coût nul.

La complexité de l'algorithme correspond donc au coût de l'évaluation des instructions de boucle (3) et (4).

Lemme

Après l'étape k de l'algorithme Cook-réciproque, on a : $|r| \leq \beta^{2^{k+1}}$

Preuve

D'après la proposition 5, après le $k^{ième}$ passage dans la boucle, r calculé est le réciproque de $[w_n, w_{n-1}, \ldots, w_{n-2^{k+1}+1}]_\beta$.
Donc, après l'étape k de la boucle, on a :

$$r = \left\lfloor \frac{\beta^{2^{k+2}-1}}{[w_n, w_{n-1}, \ldots, w_{n-2^{k+1}+1}]_\beta} \right\rfloor$$

Or, w étant normalisé : $w_n \geq 1$

On en déduit : $|r| \leq \beta^{2^{k+1}}$

De ce lemme, on déduit que le coût T_3 de l'instruction (3) à l'étape k est :

$$T_3 = M_\beta(2^{k+1}, 2^{k+1}) + M_\beta(2^k, 2^k) + A_\beta(2^{k+1}, 2^{k+1})$$

D'après le corollaire 5.3, et en utilisant le lemme ci-dessus, on déduit que le coût T_4 de l'instruction (4) à l'étape k est - en négligeant les temps de comparaison devant les temps des opérations arithmétiques - :

$$T_4 = M_\beta(2^{k+1}, 2^{k+1}) + 3.A_\beta(2^{k+1}, 2^{k+1})$$

Le coût final de l'algorithme *Cook-réciproque* est donc :

$$RC_\beta(2^p) = \tau_{div} + \sum_{k=0}^{p-1} 2.M_\beta(2^{k+1}, 2^{k+1}) + M_\beta(2^k, 2^k) + 4.A_\beta(2^{k+1}, 2^{k+1})$$

Or, avec la supposition faite sur M(n,p), en négligeant le temps d'addition qui est linéaire devant celui de multiplication :

$$2.M_\beta(2^{k+1}, 2^{k+1}) + M_\beta(2^k, 2^k) + 4.A_\beta(2^{k+1}, 2^{k+1}) \approx 5.M_\beta(2^k, 2^k)$$

D'où l'on tire : $RC_\beta(2^p) \approx \sum_{k=0}^{p-1} 5.M_\beta(2^k, 2^k) \approx 5.M_\beta(2^p, 2^p)$

Proposition 7

$$R_\beta(\alpha, n) = (5 + \frac{\alpha}{2.n - 1}).M_\beta(n, n) + O(n)$$

Preuve

Nous considérons l'algorithme α-réciproque appliqué à un nombre ayant n β-chiffres. Posons $p = \lceil \mathrm{Log}_2 n \rceil$.
Le calcul du réciproque de Cook - étape (1) - induit le calcul du reste effectué à l'étape (2). Le coût total de l'initialisation est donc $RC_\beta(n)$ - et non $2.RC_\beta(n)$ comme le laisse supposer l'algorithme -.

L'itération qui permet d'évaluer le α-réciproque comprend n_0 étapes, avec :

$$n_0 = \left\lceil \mathrm{Log}_2\left(\frac{\alpha}{2.n - 1}\right) \right\rceil$$

Après l'étape k, d'après la proposition 4, r est borné par : $r < \beta^{2^k.n}$
En outre, après chaque itération, Δ est borné par v (d'après la proposition 4, Δ est en fait le reste d'une division euclidienne par v, et est donc borné par v).

Les coûts T_3, T_4, T_5 et T_6 des instructions respectives (3), (4), (5) et (6) sont donc :

$$T_3 = M_\beta(n, n.2^k) \approx M_\beta(n.2^{k-1}, n.2^{k-1}) \approx 2^{k-1}.M_\beta(n, n)$$
$$T_4 = M_\beta(n, n)$$
$$T_5 = M_\beta(2.n, n)$$
$$T_6 = A_\beta(2^{k+1}.n, (2^k+1).n) \approx 2^{k+1}.A_\beta(n, n)$$

T_6 est en fait négligeable devant T_3, car l'étape (6) correspond à une recopie de donnée : l'addition ne correspond ici qu'à une propagation de retenue, dont l'amortissement a une forte probabilité d'être très rapide (cf. la construction de l'itération (4).

On en déduit donc que le coût global de la boucle est : $T \approx 2^{n_0}.M_\beta(n, n)$
De cette expression, on tire facilement la valeur de $R_\beta(\alpha, n)$.

Proposition 8

$$D_\beta(m, n) = (5 + \frac{m+1}{2.n-1}).M_\beta(n, n) + M_\beta(m, m-n) + M_\beta(n, m-n) + O(n,m)$$

Preuve

La complexité de l'algorithme *division_Newton* est donné par les complexités T_1, T_2 et T_3 des étapes (1), (2) et (3) de l'algorithme.

D'après la proposition (7) :

$$T_1 = (5 + \frac{m+1}{2.n-1}).M_\beta(n, n) + O(n)$$

Le coût de l'étape (2) est celui de la multiplication de u par ρ, soit :

$$T_2 = M_\beta(m, m-n)$$

L'étape (3) correspond à la correction éventuelle du quotient par l'évaluation du reste :

$$T_3 = M_\beta(n, m-n) + A_\beta(m, m)$$

En négligeant les temps d'addition ($A_\beta(m, m) \ll T_2$), on en déduit l'expression de $D_\beta(m, n)$.

X. Conclusion - Validité de l'algorithme

Il est intéressant de comparer la complexité de l'algorithme précédent à celle de l'algorithme classique de Pope-Stein (§ IV.1.d). Soit $T_{div}(m, n)$ la complexité de cet algorithme pour la division d'un entier de m β-chiffres par un autre de n β-chiffres.
Il est à noter que les deux algorithmes remplissent la même fonction, à savoir qu'ils permettent chacun de connaître le quotient mais aussi, indissociablement, le reste de la division euclidienne.

Nous rappelons ici ces deux complexités :

$$D_\beta(m, n) = (5 + \frac{m+1}{2.n-1}).M_\beta(n, n) + M_\beta(m, m-n) + M_\beta(n, m-n) + O(n,m)$$

$$T_{div}(m, n) = (\tau_{mul} + \tau_{add}).(n-m+1).m + O(n,m)$$

De façon à comparer asymptotiquement ces complexités, nous supposerons que [SST71] :

$$M_\beta(m, n) = O((n+m).\text{Log}(n+m).\text{Log Log}(n+m))$$

On obtient alors :

$$D_\beta(m, n) = O[(m+10.n).Log(n).LogLog(n)+m.Log(m).Log\ Log(m) + (2.m-n).Log(2.m-n).Log\ .Log(2.m-n)\ ]$$
$$T_{div}(m, n) = O\ [\ m.(m-n)\ ]$$

Il apparaît donc clairement que l'algorithme de division newtonienne est le meilleur asymptotiquement. De façon à visualiser plus clairement ce résultat, posons $\tau = \frac{m}{n}$.

Considérons les quatre cas critiques : $1 \approx \tau \ll n$, $1 \ll \tau \ll n$, $\tau \approx n$ et $\tau \gg n$.

- si $1 \approx \tau \ll n$, :

$$D_\beta(\tau.n, n) = O[\ n.Log(n).LogLog(n)\ ]$$
$$T_{div}(\tau.n, n) = O[\ \tau.n\ ]$$

- si $\tau \ll n$:

$$D_\beta(\tau.n, n) = O[\ n.Log(n).LogLog(n)\ ]$$
$$T_{div}(\tau.n, n) = O[\ n^2\ ]$$

- si $\tau \approx n$:

$$D_\beta(\tau.n, n) = O[\ n^2.Log(n).LogLog(n)\ ]$$
$$T_{div}(\tau.n, n) = O[\ n^4\ ]$$

- si $\tau \gg n$:

$$D_\beta(\tau.n, n) = O[\ \tau.n.Log(\tau.n).LogLog(\tau.n)\ ]$$
$$T_{div}(\tau.n, n) = O[\ \tau^2.n^2\ ]$$

Finalement, nous pouvons faire la conclusion :

L'algoritme de division Newtonienne est asymptotiquement plus intéressant que l'algorithme de division de Pope-Stein, sauf dans le cas où les deux opérandes sont du même ordre de grandeur.

Remarque

La complexité a été étudiée dans le cas où l'on dispose d'une multiplication rapide : il est clair que cette supposition limite l'emploi de la division Newtonienne à des nombres suffisament larges pour pouvoir apprécier la qualité de cette multiplication rapide -sur laquelle tout repose- par rapport à une multiplication classique.

Chapitre VI

CALCUL DU PGCD DE DEUX ENTIERS

Le calcul du plus grand commun diviseur (pgcd) de deux entiers est fondamental en arithmétique puisqu'il intervient aussi bien dans des problèmes entiers (ex. élimination pour le calcul de formes normales) que dans l'arithmétique des rationnels (mise sous forme canonique). Le coût de cette opération étant plus important que celui des autres opérations arithmétiques classiques, le temps passé en calcul de pgcd lors d'évaluations d'expressions dans les rationnels est très important. L'étude et l'implantation de "bons" algorithmes est donc -comme toujours...- essentielle.

I. LES ALGORITHMES CLASSIQUES

I.1. Introduction : Pgcd et Coefficients de Bezout

Théorème
Soient a et b deux entiers, il existe un entier positif qui divise à la fois a et b et tel que tout diviseur commun à a et b divise d; cet entier est appelé le pgcd de a et b et est noté (a,b).
De plus, il existe deux entiers u et v tels que :

$$d = u.a + b.v$$

Dans la suite, nous désignerons par *coefficients de Bezout* deux entiers u et v qui vérifient cette relation.

Cas particuliers :
Dans le cas où l'un des deux entiers est nul, on a :

- si u est un entier positif non nul alors : (u,0) = u.
- (0,0) est indéfini.

Historiquement, l'algorithme d'Euclide (probablement découvert bien avant Euclide) constitue l'un des plus vieux algorithmes connus (et même le plus vieux au dire de Knuth...).

Même si l'algorithme d'Euclide est connu depuis très longtemps, la recherche d'un meilleur algorithme ne date pas de l'ère des ordinateurs, puisque en 1846, A. Dupré [DUP46] écrit :
"Le nombre d'opérations à effectuer pour obtenir le plus grand commun diviseur entre deux nombres entiers A > a pouvant être considérable lorsque le plus petit des deux est fort grand, la recherche d'une limite propre à rassurer les calculateurs a attiré l'attention des analystes."

Il semble donc que les calculateurs de l'époque, même si ils s'apparentaient plus à Ramanujan qu'à Cray2, avaient pourtant eux aussi des limites...

Différentes améliorations ont été trouvées à l'algorithme d'Euclide, qui, sans changer son ordre de complexité, ont permis de réduire son coût.
La technique de Lehmer [LEH38] s'inscrit dans ce contexte et est encore la plus utilisée dans les systèmes de calcul formel.

Plus récemment, l'algorithme binaire [STE67] permet de calculer un pgcd sans divisions, uniquement en utilisant des soustractions et des décalages dans la représentation en base deux des entiers. L'ordre de cet algorithme est le même que celui d'Euclide.

Avec la découverte d'algorithmes de multiplication rapide [KAO62] [SST71], Schönhage [SCH71] a proposé en 1971 un algorithme avec un ordre meilleur que celui d'Euclide. C'est actuellement le meilleur algorithme séquentiel connu. Il a été généralisé à n'importe quel domaine euclidien par Mœnck [MŒ73]. Cet algorithme peut être considéré comme une version *Diviser Pour Régner* de l'algorithme d'Euclide.

Enfin, Brent et Kung [BKU83] ont proposé un réseau systolique -basé sur une variante de l'algorithme binaire- qui permet le calcul en un temps linéaire de la taille des données avec un nombre linéaire de cellules.

Nous proposons ici une généralisation par une technique *Diviser Pour Régner* de l'algorithme de Lehmer : l'avantage de cet algorithme est double, puisqu'il permet d'une part d'être en parfaite adéquation avec l'arithmétique interne du calculateur, et d'autre part de tirer parti de multiplications rapides (efficaces, nous l'avons vu, lorsque les opérandes ont plus de 250 chiffres). L'algorithme obtenu est comparé avec l'algorithme de Schönhage.

Les algorithmes décrits ici ne sont pas présentés dans leur ordre chronologique, mais plutôt dans un ordre de "dépendance algorithmique" :

- L'algorithme binaire (le plus "primitif")
- L'algorithme d'Euclide comme un raffinement de l'algorithme binaire
- L'algorithme de Lehmer comme une implantation raffinée de l'algorithme d'Euclide
- L'algorithme de Schönhage comme une adaptation *Diviser Pour Régner* de l'algorithme d'Euclide
- L'algorithme de Lehmer généralisé comme une adaptation *Diviser Pour Régner* de l'algorithme de Lehmer

I.2. L'algorithme binaire

Paradoxalement, cet algorithme, qui tire parti de propriétés élémentaires du pgcd, est relativement récent [STE61]. Sa plus grande caractéristique est d'être particulièrement adapté au calcul en base deux.

Soient u et v deux entiers tels que $u > v \geq 0$. Les propriétés suivantes sont vérifiées de manière évidente :

$$\begin{cases} (P0)\ (u, 0) = u \\ (P1)\ (u, v) = (u-v, v) \\ (P2)\ (u, v) = 2.(u/2, v/2) & \text{si } u \equiv v \equiv 0 \quad [2] \\ (P3)\ (u, v) = (u/2, v) & \text{si } u \equiv v+1 \equiv 0 \quad [2] \\ (P3')\ (u, v) = (u, v/2) & \text{si } u+1 \equiv v \equiv 0 \quad [2] \end{cases}$$

D'où l'algorithme binaire :

Théorème 1 *Algorithme binaire*

Soient u_0 et v_0 deux entiers vérifiant $u_0 > v_0 \geq 0$ et soit ($\mathcal{A}$) l'algorithme :

($\mathcal{A}$)

$\begin{bmatrix} u \\ v \\ g \end{bmatrix} \leftarrow \begin{bmatrix} u_0 \\ v_0 \\ 1 \end{bmatrix}$;

Tantque $(v \neq 0)$ *faire*

  selon (u mod 2 , v mod 2) :

    $(0\ ,\ 0) \Rightarrow \begin{bmatrix} u \\ v \\ g \end{bmatrix} \leftarrow \begin{bmatrix} u/2 \\ v/2 \\ 2.g \end{bmatrix}$;

    $(0\ ,\ 1) \Rightarrow \begin{bmatrix} u \\ v \\ g \end{bmatrix} \leftarrow \begin{bmatrix} u/2 \\ v \\ g \end{bmatrix}$;

    ou $\left.\begin{matrix} (1\ ,\ 1) \\ (1\ ,\ 0) \end{matrix}\right\} \Rightarrow \begin{bmatrix} u \\ v \\ g \end{bmatrix} \leftarrow \begin{bmatrix} u-v \\ v \\ g \end{bmatrix}$;

  fselon

  si $(u < v)$ ***alors*** $\begin{bmatrix} u \\ v \end{bmatrix} \leftarrow \begin{bmatrix} v \\ u \end{bmatrix}$ ***fsi***

ftantque

$\delta \leftarrow g.u$;

Après exécution de l'algorithme ($\mathcal{A}$), on a : $\delta = (u_0, v_0)$

Preuve :
L'utilisation des propriétés énoncées ci-dessus montre qu'après chaque passage dans la boucle, on a :

$$(u_0,v_0) = g.(u,v)$$

La décroissance stricte de la quantité (u+v) (minorée par 1) à chaque passage dans la boucle prouve que l'algorithme s'arrête en un temps fini.
En sortie de la boucle, on obtient donc : $(u_0,v_0) = g.(u,0)$
D'où δ est le pgcd de u_0 et v_0.

Remarque : cet algorithme ne nécessite aucune division, d'où son succès...
Il est en outre directement adaptable au calcul de coefficients de Bezout.

I.3. L'algorithme d'Euclide

Bien qu'antérieur, il peut être considéré comme un raffinement de l'algorithme binaire. Il est basé sur le corollaire (C1) de la propriété (P1) :

(C1) $a > b \geq 0$ $\qquad (a,b) = (b, a \bmod b)$

Associé à la condition d'arrêt (P0), on obtient immédiatement l'algorithme d'Euclide :

Théorème 2 *Algorithme d'Euclide*

Soient u_0 et v_0 deux entiers vérifiant $u_0 > v_0 \geq 0$, et soit ($\mathcal{E}$) l'algorithme :

($\mathcal{E}$)

$\begin{bmatrix} u \\ v \end{bmatrix} \leftarrow \begin{bmatrix} u_0 \\ v_0 \end{bmatrix}$; $\quad M \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;
Tantque $(v \neq 0)$ *faire*
  $\begin{bmatrix} u \\ v \end{bmatrix} \leftarrow \begin{bmatrix} v \\ u \bmod v \end{bmatrix}$;
  $M \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor \frac{u}{v} \rfloor \end{bmatrix}.M$
ftantque
$\delta \leftarrow g.u$; $\quad [\alpha\ \beta] \leftarrow [1\ 0].M$;

Après exécution de l'algorithme ($\mathcal{E}$), on a : $\quad \begin{cases} \delta = (u_0,v_0) \\ \alpha.u_0 + \beta.v_0 = \delta \end{cases}$

Preuve : L'utilisation du corollaire (C1) énoncé ci-dessus montre que la boucle tantque conserve le pgcd. Après chaque passage, on a bien :

$$(u_0,v_0) = (u,v) \quad \text{et} \quad M_{1,1}.u_0 + M_{1,2}.v_0 = u$$

La décroissance stricte de u et v (minorée par 1) à chaque passage dans la boucle prouve que l'algorithme s'arrête en un temps fini.
En sortie de la boucle, on obtient donc : $(u_0,v_0) = (u,0)$

D'où : $$\delta = (u_0,v_0)$$

et α et β sont des coefficients de Bezout associés.

Complexité

Le pire cas pour l'algorithme d'Euclide a été étudié par Lamé[LAM44]. Il est obtenu lorsque, à taille bornée, les opérandes maximisent le nombre de divisions à effectuer. Cela revient à dire que les quotients successifs obtenus valent tous 1 : les itérés successifs forment alors la suite de Fibonacci :

$$\begin{cases} F_{-2} = 1 \quad F_{-1} = 1 \\ F_k = F_{k-1} + F_{k-2} \quad k=0\ldots n \end{cases}$$

On en déduit que les deux plus petits entiers u>v>0 tels que l'algorithme d'Euclide appliqué à u et v nécessite N divisions sont :

$$\begin{cases} u = F_N \\ v = F_{N-1} \end{cases}$$

Le nombre de divisions est donc toujours inférieur à $5.\log_{10} u + O(1)$ [LAM44].

Après l'étude du pire cas, il est fondamental de connaître le nombre *moyen* de divisions lorsque u et v sont choisis aléatoirement. Knuth [KNU81f] a montré le théorème suivant, en en donnant par ailleurs une étude approchée relativement convaincante.

Théorème 3
Le nombre de divisions (et donc d'étapes) dans l'algorithme d'Euclide évolue en moyenne (i.e. lorsque u et v sont tirés aléatoirement selon une loi uniforme parmi {1,2,...,n}) comme le logarithme du nombre de chiffres du plus petit des opérandes.

Autement dit, à nombre de chiffres fixé, la complexité moyenne est proprotionnelle au logarithme du coût du pire cas.

Pour un résultat plus parlant, Collins [COL74] a montré que le temps moyen d'exécution de l'algorithme d'Euclide sur deux entiers u et v (en utilisant les algorithmes standards pour les calculs en précision infinie) est :

$$\left[1 + \text{Log}\left(\frac{\text{Max}(u,v)}{(u,v)}\right)\right].\text{Log}\,[\text{Min}(u,v)]$$

Remarque Importante :

Les quotients obtenus à chaque étape de l'algorithme sont très petits en général, et par suite la décroissance est lente : dans 99,85% des cas, ils sont inférieurs à 1000 - après la première étape, les entiers étant tirés selon une loi uniforme-. Ce résultat a été formalisé par D.E. Knuth [KNU81f].
L'algorithme d'Euclide est donc très bien adapté aux algorithmes standards de calcul en précision infinie. Les quotients étant faibles, les divisions effectuées à chaque étape avec l'algorithme de Pope-Stein sont "optimales".
Les algorithmes séquentiels asymptotiquement meilleurs connus aujourd'hui sont tous basés sur des algorithmes de multiplication rapide : il est remarquable que si les divisions effectuées dans ces algorithmes étaient réalisées par un algorithme de Pope-Stein, alors ils seraient plus coûteux que l'algorithme d'Euclide.

I.4. Algorithme d'Euclide et Fractions Continues

Définition : *Soit $x \geq 1$ un réel. On appelle fraction continue de x la représentation $<b_1,b_2,\ldots>$ où les b_k sont donnés par l'itération :*

$$(1)\quad \begin{cases} x_1 = x \\ b_k = \lfloor x_k \rfloor \geq 1 \\ x_k = b_k + \dfrac{1}{x_{k+1}} \Leftrightarrow x_{k+1} = \dfrac{1}{x_k - b_k} \end{cases} \qquad (k = 1,2,\ldots)$$

Théorème 4 *Fraction continue d'un rationnel et algorithme d'Euclide*

Si $x \in \mathbb{Q}$, posons : $x = \dfrac{p_0}{p_1}$ *avec $p_0 \geq p_1$ entiers positifs.*

La suite $(b_k)_{k=1,2,\ldots}$ est alors finie et correspond aux différents quotients obtenus en appliquant l'algorithme d'Euclide à p_0 et p_1.

L'itération s'arrête au pas (t+1) avec : $\begin{cases} p_t = (p_0,p_1) \\ p_{t+1} = 0 \end{cases}$

Preuve :

Posons $x_k = \dfrac{p_{k-1}}{p_k}$ (k=1,2,...). L'itération (1) s'écrit alors :

$$(2)\quad \begin{cases} p_{k+1} = p_{k-1} - b_k \cdot p_k \\ b_{k+1} = \left\lfloor \dfrac{p_k}{p_{k+1}} \right\rfloor \end{cases}$$

D'où l'on déduit directement que les p_k sont les restes successifs obtenus en appliquant l'algorithme d'Euclide à p_0 et p_1, et, par suite, que les coefficients b_k sont les quotients successifs correspondants.

Exemple :

$$
\begin{aligned}
(7968,6573) &= (6573,1395) && \text{quotient} = 1\\
&= (1395,993) && \text{quotient} = 4\\
&= (993,402) && \text{quotient} = 1\\
&= (402,189) && \text{quotient} = 2\\
&= (189,24) && \text{quotient} = 2\\
&= (24,21) && \text{quotient} = 7\\
&= (21,3) && \text{quotient} = 1\\
&= 3 && \text{quotient} = 7
\end{aligned}
$$

et $\dfrac{7968}{6573} = < 1,4,1,2,2,7,1,7 >$

$$
= 1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{7+\cfrac{1}{1+\frac{1}{7}}}}}}}
$$

I.5. Propriétés fondamentales *(...du moins pour la suite)*

Théorème 5 *Borne sur les coefficients de Bezout*

Soient $u>v\geq 1$ deux entiers et soient (α,β) les coefficients de Bezout fournis en appliquant l'algorithme d'Euclide à u et v (cf théorème 2 §I-3).

Alors : $\begin{cases} |\alpha| < \dfrac{v}{(u,v)} \\ |\beta| < \dfrac{u}{(u,v)} \end{cases}$ *et* $\alpha.\beta \leq 0$

Preuve : par récurrence sur le nombre de divisions effectuées lors de l'algorithme d'Euclide appliqué à u et v.
On démontre d'abord le lemme suivant :

Lemme *Soient* $u' = \frac{u}{(u,v)}$ *et* $v' = \frac{v}{(u,v)}$

Alors les coefficients de Bezout α' *et* β' *fournis en appliquant l'algorithme d'Euclide à* u' *et* v' *sont respectivement les coefficients de Bezout* α *et* β *fournis en appliquant l'algorithme d'Euclide à* u *et* v.

Sous-Preuve

Comme $\frac{v}{u} = \frac{v'}{u'}$, les fractions continues de $\frac{v}{u}$ et de $\frac{v'}{u'}$ sont identiques. Par suite, les quotients obtenus respectivement dans l'algorithme d'Euclide appliqué à (u,v) et (u',v') sont les mêmes, et les coefficients de Bezout associés sont donc les mêmes.
Fin-Sous-Preuve

On s'intéresse maintenant aux différentes étapes (i.e. divisions) de l'algorithme d'Euclide.
Soient $u_N > v_N \geq 1$ deux entiers premiers entre eux tels que l'algorithme d'Euclide appliqué à u_N et v_N nécessite exactement N divisions euclidiennes.

On note u_k et v_k les coefficients obtenus à la $(N-k)^{ième}$ étape, c'est à dire :

$$\begin{bmatrix} u_{k-1} \\ v_{k-1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -\left\lfloor \frac{u_k}{v_k} \right\rfloor \end{bmatrix} \cdot \begin{bmatrix} u_k \\ v_k \end{bmatrix} \qquad k=N \ldots 1$$

Comme l'algorithme d'Euclide prend exactement N pas, et que u_N et v_N sont premiers entre eux, on a : $\begin{cases} u_0=1 \\ v_0=0 \end{cases}$

On note α_k et β_k les coefficients définis par l'itération :

$$\begin{cases} \alpha_0=1 \\ \beta_0=0 \end{cases} \quad \text{et} \quad \begin{bmatrix} \alpha_k \\ \beta_k \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -\left\lfloor \frac{u_{N-k+1}}{v_{N-k+1}} \right\rfloor \end{bmatrix} \cdot \begin{bmatrix} \alpha_{k-1} \\ \beta_{k-1} \end{bmatrix} \quad k=N \ldots 1$$

N.B. α_N et β_N sont les coefficients de Bezout délivrés par l'algorithme d'Euclide appliqué à u_N et v_N.

Propriété $\forall N \geq 1$, *on a :* $\begin{cases} \alpha_N . \beta_N \leq 0 \\ |\alpha_N| < v_N \\ |\beta_N| < u_N \end{cases}$

Sous-Preuve : Par récurrence

La propriété est vérifiée pour N=1, puisqu'on a : $\begin{cases} u_1 \text{ donné} > v_1 \\ v_1=1 \end{cases}$ et $\begin{cases} \alpha_1=0 \\ \beta_1=1 \end{cases}$

Supposons la propriété vraie à l'ordre N-1. On a :

$$\begin{cases}\alpha_N=\beta_{N-1}\\ \beta_N=\alpha_{N-1} - \left|\frac{u_N}{v_N}\right|.\beta_{N-1}\end{cases}$$

Donc $\alpha_N.\beta_N = \alpha_{N-1}.\beta_{N-1} - \left|\frac{u_N}{v_N}\right|.(\beta_{N-1})^2 \le 0$ car par récurrence $\alpha_{N-1}.\beta_{N-1} \le 0$

De plus : $$\begin{bmatrix}u_N\\ v_N\end{bmatrix} = \begin{bmatrix}0 & 1\\ 1 & -\left|\frac{u_N}{v_N}\right|\end{bmatrix}^{-1}.\begin{bmatrix}u_{N-1}\\ v_{N-1}\end{bmatrix} = \begin{bmatrix}\left|\frac{u_N}{v_N}\right|.u_{N-1}+v_{N-1}\\ u_{N-1}\end{bmatrix}$$

D'où l'on tire : $u_{N-1} = v_N$ et par suite $|\alpha_N| = |\beta_{N-1}| < u_{N-1} = v_N$

Par ailleurs :

$$|\beta_N| = |\alpha_{N-1} - \left|\frac{u_N}{v_N}\right|.\beta_{N-1}| = |\alpha_{N-1}| + \left|\frac{u_N}{v_N}\right|.|\beta_{N-1}| \qquad (\text{car } \alpha_{N-1}.\beta_{N-1}\le 0)$$
$$< v_{N-1} + \left|\frac{u_N}{v_N}\right|.u_{N-1} = u_N$$

La propriété est donc démontrée.
Fin-Sous-Preuve

En utilisant le lemme précédent, on en déduit que le théorème 4 est démontré.
Fin-Preuve

Propriété 6

Soit $M^{(k)}$ la matrice des coefficients de Bezout après la k^ième étape de l'algorithme d'Euclide appliqué à a et b, avec a≥b≥1. Alors on a :

$$\begin{cases}(i) & M^{(k)}_{i,1} > 0 \Leftrightarrow M^{(k)}_{i,2} < 0 & i=1,2\\ (ii) & \left|M^{(k)}_{i,1}\right| \le \left|M^{(k)}_{i,2}\right| & i=1,2\\ (iii) & M^{(k)}_{1,1} > 0 \Leftrightarrow M^{(k)}_{2,2} > 0 & \end{cases}$$

Preuve :

Soient $a^{(k)}$ et $b^{(k)}$ les itérés de a et b obtenus après la k^ième étape de l'algorithme d'Euclide appliqué à a et b. On a, par construction de $M^{(k)}$:

$$\begin{bmatrix}a^{(k)}\\ b^{(k)}\end{bmatrix} = M^{(k)}.\begin{bmatrix}a\\ b\end{bmatrix}$$

Comme l'algorithme d'Euclide entraîne une décroissance stricte des suites $a^{(k)}$ et $b^{(k)}$, on en déduit (i).

Comme de plus $a^{(k)}\ge 0$, on a : $a^{(k)} = \left|\,|M^{(k)}_{1,1}|.a - |M^{(k)}_{1,2}|.b\right|$

Si $|M^{(k)}_{1,1}| > |M^{(k)}_{1,2}|$, comme a ≥ b, on a :

$$\left|\,|M^{(k)}_{1,1}|.a - |M^{(k)}_{1,2}|.b\right| \ge (|M^{(k)}_{1,1}| - |M^{(k)}_{1,2}|).a \ge a$$: ce qui est absurde

On en déduit donc (ii) (le cas i=2 découlant du même raisonnement).

(iii) se déduit directement de l'inégalité (i) associée à la décroissance des itérés $a^{(k)}$ et $b^{(k)}$.

Propriété 7

Soient $\alpha > \beta \geq \gamma > \delta \geq 0$ *quatre entiers. Soient* $q_1, q_2, \ldots, q_N$ *définis par :*

$$\begin{cases} \dfrac{\alpha}{\delta} = \langle q_0, q_1, \ldots, q_{N-1}, a_N \ldots \rangle \\ \dfrac{\beta}{\gamma} = \langle q_0, q_1, \ldots, q_{N-1}, b_N \ldots \rangle \end{cases} \quad \text{avec } N \text{ tel que } a_N \neq b_N$$

et soient les séquences définies par :

$$\begin{cases} \begin{bmatrix} \alpha_0 & \beta_0 \\ \gamma_0 & \delta_0 \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} \\ \begin{bmatrix} \alpha_{n+1} & \beta_{n+1} \\ \gamma_{n+1} & \delta_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_n \end{bmatrix} \begin{bmatrix} \alpha_n & \beta_n \\ \gamma_n & \delta_n \end{bmatrix} \end{cases} \qquad n = 0 \ldots N-1$$

Alors: $\begin{cases} et \ \alpha_{2k} > \beta_{2k} \geq \gamma_{2k} > \delta_{2k} \geq 0 \\ \beta_{2k+1} > \alpha_{2k+1} \geq \delta_{2k+1} > \gamma_{2k+1} \geq 0 \end{cases} \quad k = 0,1,\ldots,\lfloor (N-1)/2 \rfloor$

Preuve : par récurrence

Pour k = 0, on a bien $\alpha_0 > \beta_0 \geq \gamma_0 > \delta_0 \geq 0$

De plus, $\alpha_1 = \delta_0 < \gamma_0 = \beta_1$ et $\gamma_1 = \beta_0 - q_0 \cdot \gamma_0 < \alpha_0 - q_0 \cdot \delta_0 = \delta_1$.

Or $\gamma_1 - \beta_1 = \beta_0 - (q_0+1) < 0$ car $q_0 = \lfloor \beta_0 / \gamma_0 \rfloor$

La propriété est donc bien vérifiée pour k = 0.

Le même type de calcul permet de montrer que l'ordre k implique l'ordre k+1.

II. L'ALGORITHME DE LEHMER

II.1. Introduction

On suppose qu'on travaille dans une base de calcul $\beta = 2^{\gamma}$.

Le calcul du pgcd de deux entiers précision infinie par la méthode d'Euclide nécessite de nombreuses divisions, dont le coût est important.

La méthode de Lehmer[LEH38][KNU81f] permet de remplacer les divisions en précision infinie par des divisions sur des entiers-machine.

Elle est basée sur deux remarques importantes :

1°/ L'algorithme d'Euclide est basé sur des divisions successives de nombres de tailles semblables: les quotients qui apparaissent sont donc souvent très petits.

2°/ On peut facilement regrouper entre elles différentes étapes de l'algorithme d'Euclide.

Soient a et b ($1 \leq b \leq a$) deux entiers dont on veut calculer le pgcd.
Supposons que l'on connaisse deux réels A et B tels que : $A \leq \left|\frac{b}{a}\right| \leq B$
Si les développements en fractions continues de A et B : $A = <A_1,A_2,\ldots>$ et $B = <B_1,B_2,\ldots>$, coincident jusqu'à l'ordre n (i.e. $A_i = B_i$ i=1,...,n) alors le développement en fractions continues de $\left|\frac{b}{a}\right|$ s'écrit $<A_1,A_2,\ldots A_n,x_{n+1},\ldots>$.
On en déduit que les n premiers quotients obtenus en appliquant l'algorithme d'Euclide à a et b sont respectivement : $A_1,A_2,\ldots$ et A_n.

Considérons les représentations en base ß de a et b :
$a = [a_n\, a_{n-1} \ldots a_0]_ß$ $(0 \leq a_i < ß)$ et $b = [b_n\, b_{n-1} \ldots b_0]_ß$ $(0 \leq b_i < ß)$

Posons $A = \frac{a_n}{b_n + 1}$ et $B = \frac{a_n + 1}{b_n}$. On a bien $A \leq \left|\frac{b}{a}\right| \leq B$.

La partie commune des développements en fractions continues de A et B peut être obtenue en appliquant l'algorithme d'Euclide à (a_n, b_n+1) d'une part et (a_n+1, b_n) d'autre part, et ce tant que les quotients $q_1,\ldots,q_p$ obtenus à chaque étape pour chacun des deux pgcd sont égaux.
Pour simuler les n premières étapes de l'algorithme d'Euclide sur a et b, il suffit alors d'appliquer la transformation :

$$\begin{bmatrix} a \\ b \end{bmatrix} \leftarrow M \begin{bmatrix} a \\ b \end{bmatrix}$$

où la matrice M est définie par : $M = \begin{bmatrix} 0 & 1 \\ 1 & -q_p \end{bmatrix} \cdots\cdots \begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix}$

On peut alors réitérer le processus, en considérant les nouvelles valeurs ainsi obtenues de a et b.

Ainsi, les p premières divisions (coûteuses puisqu'effectuées en précision infinie) ont pu être remplacées par 2.p divisions arithmétiques machine (à opérandes bornés par ß), et 4 multiplications en précision infinie.
Cette optimisation est donc particulièrement bien adaptée à des calculs machine.

Après avoir présenté l'algorithme, nous étudierons différentes optimisations essentielles pour une implantation efficace. Une intégration sur silicium de cet algorithme est en cours d'étude [GUY90].

II.2. Algorithme

Soient $a \geq b \geq 0$ deux entiers dont on veut calculer le pgcd.

Entrée : $a = [a_n\ a_{n-1} \dots a_0]_\beta\ (0 \le a_i < \beta)$ et $b = [b_n\ b_{n-1} \dots b_0]_\beta\ (0 \le b_i < \beta)$
Sortie : $d = (a,b)$ et $(u$ et $v)$ tel que : $u.a + v.b = d$

1/ Initialisation $M \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$; {M est la matrice qui contiendra les coefficients}
{de Bezout.Elle permet d'enregistrer toutes les}
{manipulations faites sur a et b. }

2/ Boucle sur l'algorithme d'Euclide

tantque $(b \neq 0)$ **faire**

(2.1) $m \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$; {m permet de cumuler les opérations à faire en précision infinie}

si $(\lfloor \text{Log}_\beta\ a \rfloor == \lfloor \text{Log}_\beta\ b \rfloor)$ **alors**

(2.2) $a_{inf} \leftarrow a_n$, $b_{inf} \leftarrow b_n$ **avec** $n = (\text{Log}_\beta\ a)$;
$a_{sup} \leftarrow a_{inf} + 1$, $b_{sup} \leftarrow b_{inf} + 1$;
{ On effectue Euclide d'une part sur (a_{sup}, b_{inf}) et d'autre part sur (a_{inf}, b_{sup})}
{ jusqu'à obtenir deux quotients différents entre les deux séquences }

(2.3) **tantque** $((b_{inf} \neq 0)$ et $(b_{sup} \neq 0))$ et
$((q_{sup} == q_{inf})$ **avec** $q_{sup} = a_{sup}/b_{inf}$, $q_{inf} = a_{inf}/b_{sup}$;)
faire

$$m \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q_{inf} \end{bmatrix} . m ; \qquad (2.3.1)$$

$$\begin{bmatrix} a_{inf} & a_{sup} \\ b_{sup} & b_{inf} \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q_{inf} \end{bmatrix} . \begin{bmatrix} a_{inf} & a_{sup} \\ b_{sup} & b_{inf} \end{bmatrix} ; \qquad (2.3.2)$$

finfaire

fsi

{ On met à jour a et b en précision infinie }

(2.4) **si** $(m_{1,2} == 0)$ **alors**
{ Il a été impossible de simuler Euclide avec Lehmer }

$$M \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -\frac{a}{b} \end{bmatrix} . M ; \qquad (2.4.1)$$

$$\begin{bmatrix} a \\ b \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -\frac{a}{b} \end{bmatrix} . \begin{bmatrix} a \\ b \end{bmatrix} ; \qquad (2.4.2)$$

(2.5) **sinon**

$$\begin{bmatrix} a \\ b \end{bmatrix} \leftarrow m . \begin{bmatrix} a \\ b \end{bmatrix} ; \qquad (2.5.1)$$

$$M \leftarrow m.M ; \qquad (2.5.2)$$

fsi

finfaire

3/ Sortie : $d \leftarrow a$; $u \leftarrow m_{1,1}$; $v \leftarrow m_{1,2}$;

II.3. Particularités et optimisations

L'algorithme de Lehmer permet de cumuler les manipulations - coûteuses - à faire en précision infinie. Il est nécessaire de voir quelles optimisations peuvent être apportées aux différentes étapes de l'algorithme.

(2.1) m est la matrice qui permet de cumuler les opérations en précision infinie détectées par la méthode de Lehmer. m est modifiée lorsque a et b ne sont pas de même taille (très rare après le premier calcul de reste), ou lorsque le premier test donne $q_{sup} \neq q_{inf}$ (encore plus rare cf (2.2)).

(2.2) Pour cette étape, il faut choisir n le plus grand possible (pour avoir la plus grande précision sur l'encadrement de a/b), tout en gardant a_n et b_n (poids forts de a et b) inférieurs à ß (base machine). Pour, cela on modifie l'algorithme en effectuant une normalisation :

$$n \leftarrow \lfloor \mathrm{Log}_2\, a \rfloor ;$$
$$a_{inf} \leftarrow a / 2^{n-\gamma} ;$$
$$b_{inf} \leftarrow b / 2^{n-\gamma} ;$$

Ces opérations sont facilement réalisables en utilisant des décalages sur $(a_n a_{n-1})$ et $(b_n b_{n-1})$.
Il faut toutefois faire attention à l'overflow sur le calcul de $a_{inf}+1$, donc choisir $a_{inf} \leq$ ß-2 (en divisant a_{inf} et b_{inf} par 2 -décalage- lorsque a_{inf} = ß-1).
Cet overflow aurait pu être systématiquement évité en choisissant $\lceil \mathrm{Log}_2\, a \rceil$ comme valeur de n (mais perte d'information de 1 bit).

(2.3) On peut très bien avoir b_{inf} ou b_{sup} nul. Il suffit de choisir :

$$\begin{cases} b_{inf} > 2 \\ 0 \leq r_{sup} < b_{inf} \\ q_{inf} = r_{sup} - 1 \\ a_{inf} = q_{inf} \cdot (b_{inf} + 1) \end{cases}$$

et on a alors :

$$\begin{cases} a_{inf} = (b_{inf} + 1) \cdot q_{inf} \\ (a_{inf} + 1) = b_{inf} \cdot q_{inf} + r_{sup} \quad \text{avec } 0 \leq r_{sup} < b_{inf} \end{cases}$$

A l'itération suivante, on aura ainsi $b_{sup} = 0$.
Il est cependant impossible d'avoir b_{inf} et b_{sup} nuls simultanément, comme le prouve le lemme suivant.

Lemme : $\left|b_{sup}^{(k+1)}\right| + \left|b_{inf}^{(k+1)}\right| \neq 0 \quad \forall k>0$

Preuve:

Supposons $\left|b_{sup}^{(k+1)}\right| = \left|b_{inf}^{(k+1)}\right| = 0$. On a alors $\begin{cases} a_{sup}^{(k)} = q_{inf}^{(k)} \cdot b_{inf}^{(k)} \\ a_{inf}^{(k)} = q_{inf}^{(k)} \cdot b_{inf}^{(k)} \end{cases}$.

D'où l'on déduit : $\left(a_{sup}^{(k)} - a_{inf}^{(k)}\right) = \ . q_{inf}^{(k)} . \left(b_{inf}^{(k)} - b_{sup}^{(k)}\right)$

Finalement : $\left(a_{sup}^{(k)} - a_{inf}^{(k)}\right) . \left(b_{inf}^{(k)} - b_{sup}^{(k)}\right) \geq 0$

Or, d'après le théorème 7 (§I.5), on a :

$$\begin{cases} a_{sup}^{(2k)} > a_{inf}^{(2k)} > b_{sup}^{(2k)} > b_{inf}^{(2k)} \geq 0 \\ a_{inf}^{(2k+1)} > a_{sup}^{(2k+1)} > b_{inf}^{(2k+1)} > b_{sup}^{(2k+1)} \geq 0 \end{cases} \Rightarrow \left(a_{sup}^{(k)} - a_{inf}^{(k)}\right) . \left(b_{inf}^{(k)} - b_{sup}^{(k)}\right) \leq 0$$

Donc, nécessairement : $b_{inf}^{(k)} = b_{sup}^{(k)}$. Et, par suite : $a_{sup}^{(k)} = a_{inf}^{(k)}$, ce qui est absurde. $b_{inf}^{(k)}$ et $b_{sup}^{(k)}$ sont donc non tous deux nuls.

Finalement, b_{inf} et b_{sup} étant non tous deux nuls, l'un au moins des deux quotients q_{inf} ou q_{sup} est forcément plus grand que 1.

Remarque : sur la plupart des processeurs, une division par 0 donne 0 en résultat. Donc, dans le cas où $b_{inf}=0$ ou $b_{sup}=0$, on obtiendra $q_{inf}=q_{sup}$. On évite ainsi deux tests à chaque itération de l'algorithme.

(2.3.1) D'après le théorème 5 (§I.5), tous les coefficients de la matrice m restent bornés par les entrées (a_{inf}, b_{inf}), donc les coefficients de la matrice m peuvent être tous calculés avec des opérations en simple précision -ce qui est fondamental-.

(2.3.2) Il est à noter qu'on peut ici très bien économiser une multiplication, en transformant un peu l'algorithme ([KNU81f] p. 329-330). En fait, il est souvent plus intéressant - mais cela dépend évidemment de la machine considérée- d'effectuer un calcul simultané du reste et du quotient dans (2.3). Ainsi on peut à la fois tester $q_{inf}==q_{sup}$ et mettre à jour facilement les restes b_{inf} et b_{sup}.

(2.4) Ce test prend la valeur vrai s'il a été impossible d'estimer sûrement le premier quotient. Nous allons tenter d'estimer la probabilité de cet évènement. Autrement dit, en utilisant les choix de a_{inf} et b_{inf} décrits en (2.2), le problème peut se formuler ainsi : Quelle est la probabilité pour que $\left\lfloor \frac{a+1}{b} \right\rfloor \neq \left\lfloor \frac{a}{b+1} \right\rfloor$, sachant que $m \leq a \leq M$ et $1 \leq b \leq a$?

L'analyse de ce problème est effectuée dans le paragraphe suivant. La proposition (1) montre que cette probabilité est inférieure à 2^{-14} lorsque $\gamma = 32$ (cadre de l'implantation).
La probabilité que le test (2.4) prenne la valeur vrai est donc excessivement faible.

(2.5) Cette étape consiste en la mise à jour de a, b et de M en précision infinie. Différentes remarques permettent d'optimiser ces manipulations :

(2.5.1) Calcul de a et b :
On dispose des coefficients de m, qui sont des entiers "*courts*" (cf (2.3.1)) et on effectue les opérations :

$$\begin{cases} a^{(k+1)} \leftarrow m_{11}.a^{(k)} + m_{12}.b^{(k)} \\ b^{(k+1)} \leftarrow m_{21}.a^{(k)} + m_{22}.b^{(k)} \end{cases}$$

Ces opérations peuvent être réalisées *en place* - i.e. sans allocation de mémoire supplémentaire -, car $a^{(k+1)} < a^{(k)}$ et $b^{(k+1)} < b^{(k)}$.
De plus, en utilisant les résultats du théorème 6 (§I.5), on a :

$$\left(m_{11}^{(k)} \geq 0\right) \Leftrightarrow \left(m_{22}^{(k)} \geq 0\right) \Leftrightarrow \left(m_{12}^{(k)} < 0\right) \Leftrightarrow \left(m_{21}^{(k)} < 0\right)$$

Ainsi, en supposant que l'on dispose des opérations de base décrites précédemment (Partie 2 - Chapitre I), la mise à jour de a et b peut être réalisée grâce à la boucle suivante :

c_X <op> y est utilisée pour le stockage de la retenue générée lors du calcul de x <*op*> y
r_X <op> y est utilisée pour le stockage du résultat du calcul de x <*op*> y

Initialisation : $c_{m_{11}*a} \leftarrow c_{m_{12}*b} \leftarrow c_{m_{21}*a} \leftarrow c_{m_{22}*b} \leftarrow c_a \leftarrow c_b \leftarrow 0$;

si $\left(m_{11}^{(k)} \geq 0\right)$ *alors*
$m_{12} \leftarrow -m_{12}$, $m_{21} \leftarrow -m_{21}$;
Pour i = 0...$\log_\beta$ a *faire*
$(c_{m_{11}*a}, r_{m_{11}*a}) \leftarrow$ LONGPROD (a[i], m_{11}, $c_{m_{11}*a}$);
$(c_{m_{12}*b}, r_{m_{12}*b}) \leftarrow$ LONGPROD (b[i], m_{12}, $c_{m_{12}*b}$);
$(c_{m_{21}*a}, r_{m_{21}*a}) \leftarrow$ LONGPROD (a[i], m_{21}, $c_{m_{21}*a}$);
$(c_{m_{22}*b}, r_{m_{22}*b}) \leftarrow$ LONGPROD (b[i], m_{22}, $c_{m_{22}*b}$);
$(c_a, a[i]) \leftarrow$ LONGDIFF ($r_{m_{11}*a}$, $r_{m_{12}*b}$, c_a);
$(c_b, b[i]) \leftarrow$ LONGDIFF ($r_{m_{22}*b}$, $r_{m_{21}*a}$, c_b);
finpour

sinon

$m_{11} \leftarrow -m_{11}, \quad m_{22} \leftarrow -m_{22}$;

Pour $i = 0 \ldots \log_{\beta} a$ *faire*

$(c_{m_{11}*a}, r_{m_{11}*a}) \leftarrow \text{LONGPROD}(a[i], m_{11}, c_{m_{11}*a})$;

$(c_{m_{12}*b}, r_{m_{12}*b}) \leftarrow \text{LONGPROD}(b[i], m_{12}, c_{m_{12}*b})$;

$(c_{m_{21}*a}, r_{m_{21}*a}) \leftarrow \text{LONGPROD}(a[i], m_{21}, c_{m_{21}*a})$;

$(c_{m_{22}*b}, r_{m_{22}*b}) \leftarrow \text{LONGPROD}(b[i], m_{22}, c_{m_{22}*b})$;

$(c_a, a[i]) \leftarrow \text{LONGDIFF}(r_{m_{12}*b}, r_{m_{11}*a}, c_a)$;

$(c_b, b[i]) \leftarrow \text{LONGDIFF}(r_{m_{21}*a}, r_{m_{22}*b}, c_b)$;

finpour

finsi

Le passage par les instructions (2.5) implique $\lfloor \text{Log}_{\beta} a \rfloor = \lfloor \text{Log}_{\beta} b \rfloor$.
Finalement, le coût arithmétique de cette mise à jour est donc :

$$T^{(1)}_{\text{Lehmer}} = (4.\tau_{\text{mul}} + 2.\tau_{\text{add}}).\lfloor \text{Log}_{\beta} a \rfloor \qquad (1)$$

Dans les mêmes conditions pour a et b, le coût d'une étape de l'algorithme d'Euclide est celui de la division de a par b. Comme a et b ont le même nombre de ß-chiffres, le coût de cette division revient au coût de la multiplication de b par le quotient (inférieur à ß), puis de la soustraction du résultat à a. Le côut arithmétique de l'étape est donc :

$$T^{(1)}_{\text{Euclide}} = (\tau_{\text{mul}} + \tau_{\text{add}}).\lfloor \text{Log}_{\beta} a \rfloor \qquad (2)$$

D'où la conclusion :

Conclusion 1

En supposant $\tau_{mul} \gg \tau_{add}$, et sans tenir compte du calcul des coefficients de Bezout associés, le coût arithmétique d'une étape de l'algorithme de Lehmer est 4 fois supérieur au coût arithmétique d'une étape de l'algorithme d'Euclide.

Par conséquent, l'algorithme de Lehmer ne devient intéressant que si le nombre moyen de divisions en précision infinie, cumulées à chaque étape, est supérieur à 4.

(2.5.2) Cette opération consiste en la mise à jour des coefficients de Bezout associés, et elle n'est donc à effectuer que lorsque l'on désire ces coefficients.
Le calcul de cette mise à jour s'effectue de la même manière que l'opération (2.5.1) présentée ci-dessus.

Le coût arithmétique du calcul est alors :

$$T^{(2)}_{Lehmer} = (4.\tau_{mul} + 2.\tau_{add}).\lfloor Log_{\beta} a \rfloor \quad (3)$$

Dans l'algorithme d'Euclide, la matrice m à chaque étape est : $m = \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix}$

Le coût arithmétique de la mise à jour des coefficients de Bezout est alors :

$$T^{(2)}_{Euclide} = (\tau_{mul} + \tau_{add}).\lfloor Log_{\beta} a \rfloor \quad (4)$$

On en déduit donc :

Conclusion 2

En supposant $\tau_{mul} \gg \tau_{add}$, et en tenant compte du calcul des coefficients de Bezout associés, le coût arithmétique d'une étape de l'algorithme de Lehmer est 4 fois supérieur au coût arithmétique d'une étape de l'algorithme d'Euclide.

Par conséquent, avec ou sans calcul des coefficients de Bezout, l'algorithme de Lehmer ne devient intéressant que si le nombre moyen de divisions en précision infinie, cumulées à chaque étape, est supérieur à 4.

II.4. Nombres d'étapes cumulées par l'algorithme

Le gain apporté par l'algorithme de Lehmer est lié au nombre de pas moyen cumulés à chaque étape.

Dans un premier temps, la probabilité pour que $\left\lfloor \frac{a+1}{b} \right\rfloor \neq \left\lfloor \frac{a}{b+1} \right\rfloor$, sachant $m \leq a \leq M$ et $1 \leq b \leq a$ est estimée. Cette probabilité permet de quantifier les cas où le recours à l'algorithme d'Euclide (et le calcul direct du reste) est inévitable.

Puis, le nombre moyen d'étapes cumulées à chaque pas de l'algorithme de Lehmer est estimé numériquement.

II.4.1. Cas où le calcul du quotient euclidien est inévitable

Lemme 1:

Soient m et M deux entiers tels que $1 < m \leq M$, et soient a et b deux entiers :

a est choisi selon une loi uniforme dans {m..M}

b est choisi uniformément dans {1,...a}.

Alors la probabilité P pour que a et b soient tels que $\left\lfloor \frac{a+1}{b} \right\rfloor \neq \left\lfloor \frac{a}{b+1} \right\rfloor$ vérifie :

$$P < \frac{\sum_{b=1}^{M} \sqrt{m} + \sqrt{M} + \frac{M^2 - m^2}{2.b^2}}{M.[M - Max(m, b+1)]} \quad (1)$$

Preuve :

De façon à rendre ce problème linéaire, fixons b entre 1 et M et étudions les valeurs de a ($\mathrm{Max}\{b, m\} \le a \le M$) pour lesquelles :

$$\left\lfloor \frac{a+1}{b} \right\rfloor > \left\lfloor \frac{a}{b+1} \right\rfloor \quad (E)$$

On supprime le cas a=b, car (E) est alors vérifiée. Trois cas sont alors à distinguer :

- Si $a \ge b^2-1$: alors :
  $\frac{a+1}{b} - \frac{a}{b+1} \ge 1 \Leftrightarrow \left\lfloor \frac{a+1}{b} \right\rfloor > \left\lfloor \frac{a}{b+1} \right\rfloor$: (E) est vérifiée.

- Si $a < 2.b - 1$: alors :
  $\left\lfloor \frac{a+1}{b} \right\rfloor = \left\lfloor \frac{a}{b+1} \right\rfloor = 1$: (E) n'est pas vérifiée.

- Cas général : Soit $K = \left\{ \begin{array}{c} a \in \{m,\ldots,M-1\} \\ (a,b) \in N / b \in \{m,\ldots,M-1\} \\ a < b \end{array} \right\}$

On a alors : (E) : $\left\lfloor \frac{a+1}{b} \right\rfloor > \left\lfloor \frac{a}{b+1} \right\rfloor \Leftrightarrow a < (b+1)\left\lfloor \frac{a+1}{b} \right\rfloor$ avec $(a,b) \in K$

Soient $(q,r) \in N^2$ définis par : $a + 1 = b.q + r$ $(q \ge 1)$. (E) s'écrit alors :

$$(E) \quad \Leftrightarrow \quad r < q - 1$$

Soit $K = \left\{ (q,r) \in N^2 / \left(\begin{array}{c} 0 \le r < b \\ q \ge 1 \\ m < bq + r \le M \end{array} \right) \right\}$ et $\overline{K} = \left\{ (q,r) \in K_b / r < q-1 \right\}$

Soit $K_b = \left\{ (q,r) \in N^2 / \left(\begin{array}{c} 0 \le r < b \\ q \ge 1 \\ m < bq + r \le M \end{array} \right) \right\}$ et $\overline{K_b} = \left\{ (q,r) \in K_b / r < q-1 \right\}$

Il s'agit alors de dénombrer les ensembles K_b et $\overline{K_b}$.

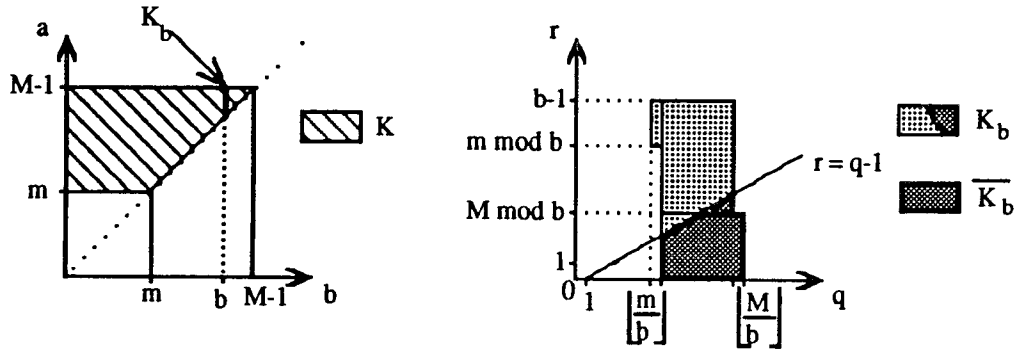


Fig.1 Représentation graphique des ensembles K_b et $\overline{K_b}$

$$\text{card } K_b = \text{card } \{a \in N / \text{Max}(m,b+1) \le a < M\} = M - \text{Max}\{m, b+1\} \quad (2)$$

$$\text{card } \overline{K_b} = \text{Max}\left\{\text{Min}\left(\left\lfloor\frac{m}{b}\right\rfloor - 1,\ b-1\right) - (m \bmod b), 0\right\} + \text{Min}\left\{M \bmod b, \left\lfloor\frac{M}{b}\right\rfloor - 1\right\} + \sum_{i=\left\lfloor\frac{m}{b}\right\rfloor+1}^{\left\lfloor\frac{M}{b}\right\rfloor-1} (i-1) \quad (3)$$

Seule une majoration de $\overline{K_b}$ suffit.

Or : $$\text{Max}\left\{\text{Min}\left(\left\lfloor\frac{m}{b}\right\rfloor - 1,\ b-1\right) - (m \bmod b), 0\right\} \le \text{Min}\left\{\frac{m}{b}, b\right\} - 1 < \sqrt{m} \quad (4)$$

et : $$\text{Min}\left\{M \bmod b, \left\lfloor\frac{M}{b}\right\rfloor - 1\right\} \le \text{Min}\left\{b, \frac{M}{b}\right\} < \sqrt{M} \quad (5)$$

Par ailleurs : $$\sum_{i=\left\lfloor\frac{m}{b}\right\rfloor+1}^{\left\lfloor\frac{M}{b}\right\rfloor-1} (i-1) = \left(\left\lfloor\frac{M}{b}\right\rfloor - 2\right)\left(\left\lfloor\frac{M}{b}\right\rfloor - 1\right) - \left\lfloor\frac{m}{b}\right\rfloor\left(\left\lfloor\frac{m}{b}\right\rfloor + 1\right) < \frac{M^2 - m^2}{b^2} \quad (6)$$

De ces différentes majorations (4) (5) et (6), on déduit la majoration de (3) :

$$\text{card } \overline{K_b} < \sqrt{m} + \sqrt{M} + \frac{M^2 - m^2}{2b^2} \quad (7)$$

Soit P_b la probabilité pour que a, choisi selon une loi uniforme dans K_b, appartienne à $\overline{K_b}$. De (2) et (7) on déduit la majoration de P_b :

$$P_b = \frac{\text{card}(\overline{K_b})}{\text{card}(K_b)} < \frac{\sqrt{m} + \sqrt{M} + \dfrac{M^2 - m^2}{2b^2}}{M - \text{Max}(m, b+1)} \quad (8)$$

Soit alors P la probabilité pour que (a,b), choisi selon une loi uniforme dans K, vérifie (E).

On a : $$P = \frac{\sum_{b=1}^{M} P_b}{M} \qquad (9)$$

Remplaçant l'expression de P_b par la majoration (8), on obtient le résultat (1) du lemme.

Dans le cadre du problème la base de calcul ß choisie est une puissance de deux. En supposant que le plus grand des deux entiers (a) est normalisé (supérieur à la moitié de la base de calcul), le lemme précédent peut être appliqué pour estimer la probabilité où le calcul direct du reste euclidien est inévitable.

Proposition 1

Soit a un 2^γ-chiffre normalisé (i.e. $a \geq 2^{\gamma-1}$), choisi selon une loi uniforme dans $\{2^{\gamma-1},\ldots, 2^\gamma-1\}$.
Et soit b un entier choisi selon une loi uniforme dans $\{1,\ldots,a\}$.
Alors la probabilité P pour que a et b soient tels que $\left\lfloor \frac{a+1}{b} \right\rfloor \neq \left\lfloor \frac{a}{b+1} \right\rfloor$ *est majorée par :*

$$P < \frac{3.2^{\frac{\gamma}{2}}}{M} < 2^{2-\frac{\gamma}{2}} \qquad (10)$$

Preuve :

On se place désormais dans le cadre de notre problème: $m = 2^{\gamma-1}-1$ et $M = 2^\gamma-2$. En utilisant la majoration (1) du lemme précédent, on distingue les trois cas suivants :

- $b \geq m+1$: on a alors $M < 2b - 1$: donc (E) n'est pas vérifiée : $P_b = 0$
- $1 \leq b < \left\lfloor \sqrt{2^{\gamma-1} - 1} \right\rfloor$: on a alors : $a < b^2-1$: donc (E) est vérifiée : $P_b = 1$
- $\left\lfloor \sqrt{2^{\gamma-1} - 1} \right\rfloor \leq b < 2^{\gamma-1}$: $$\sum_{k=2^{\frac{\gamma-1}{2}}}^{2^{\gamma-1}} P_b < 2^{\gamma-1}\frac{\sqrt{M}+\sqrt{m}}{M-m} + \frac{(M+m)}{2}\sum_{k=2^{\frac{\gamma-1}{2}}}^{2^{\gamma-1}} \frac{1}{k^2}$$

En majorant la somme par l'intégrale, on obtient :

$$\sum_{k=2^{\frac{\gamma-1}{2}}}^{2^{\gamma-1}} \frac{1}{k^2} < \int_{2^{\frac{\gamma-1}{2}}-1}^{2^{\gamma-1}} \frac{1}{t^2}\,dt < \frac{1}{2^{\frac{\gamma-1}{2}}}$$

Tenant compte des valeurs de m et M, on obtient finalement le résultat (10).

Application au modèle arithmétique : $\beta = 2^{32}$

Dans le cadre de l'implantation, on a : $\gamma = 32$.
On obtient ainsi comme borne sur P : $P < 2^{-14}$

Il est donc très rare que l'on ne puisse prévoir le premier quotient de l'algorithme d'Euclide à chaque étape de l'algorithme de Lehmer.

II.4.2. Estimation du nombre moyen d'étapes cumulées

Soient u et v deux entiers, tels que : $2^{\gamma} > u \geq 2^{\gamma-1} \geq v \geq 1$

Le problème peut être ainsi formulé :

Quel est le nombre de pas pour lequel l'algorithme d'Euclide appliqué respectivement à (u+1, v) et (u, v+1) donne les mêmes quotients ?

C'est à dire, en utilisant les fractions continues :
Quel est le nombre de termes de tête identiques dans le développement en fraction continue de $\frac{u+1}{v}$ et $\frac{u}{v+1}$?

Ce nombre moyen de pas cumulés est intéressant pour différentes raisons :

- il donne une information moyenne sur le coefficient d'accélération de la méthode de Lehmer

- il permet de choisir la base de calcul ß qui maximise ce coefficient d'accélération - en tenant compte de l'augmentation du coût des opérations de base lorsque ß augmente -.

N'étant pas parvenu à résoudre ce problème, nous en présentons une étude statistique, réalisée sur Sun - et utilisant le générateur aléatoire loi uniforme implanté sur la machine-.

Pour γ variant de 1 à 30, a est tiré selon une loi uniforme entre $2^{\gamma-1}$ et 2^{γ} et b est tiré selon une loi uniforme entre 0 et a. Pour chaque valeur de γ, plusieurs milliers de valeurs pour u et v ont été tirées.

Les résultats obtenus sont résumés sur la figure ci-dessous :

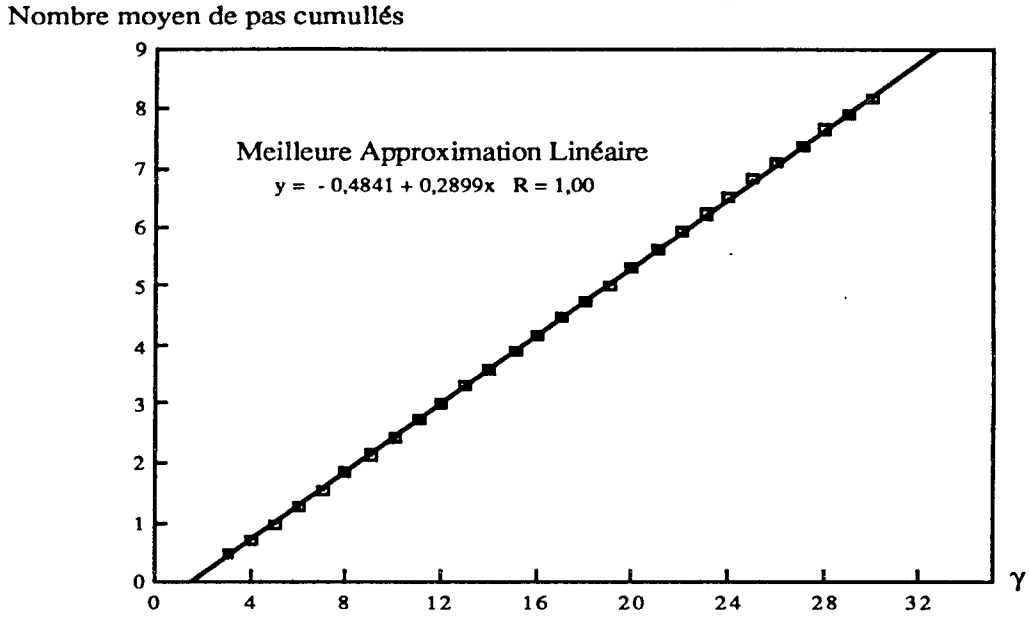


Fig. 1 Nombre de pas cumulés lors de l'algorithme de Lehmer en fonction de la base de calcul $ß = 2^{\gamma}$

Conclusion *Le nombre d'étapes cumulées semble donc croître linéairement en fonction du logarithme de la base de calcul ß choisie.*

Application au choix optimal de la base de calcul pour l'algorithme de Lehmer :

Soit B la base de calcul machine. Si ß est supérieure à B, le coût pour effectuer les calculs en base ß à partir de l'arithmétique en base B est une fonction super-linéaire du rapport des logarithmes de ß et B (cf Partie II, Chapitres précédents).
Or, d'après la conclusion précédente, le nombre d'étapes cumulées par l'algorithme de Lehmer (et par suite le coût du calcul du pgcd) est une fonction linéaire du rapport des logarithmes de ß et B.

Donc, deux cas sont à distinguer pour le choix de ß :

- si ß < B : alors, pour un même coût de calcul, le nombre d'étapes cumulées est augmenté en choisissant ß=B
- si ß > B : alors, le facteur d'augmentation du coût des calculs en base ß -par rapport au coût des calculs en base B- est supérieur au facteur d'accélération de Lehmer - dû au nombre d'étapes cumulées -.

Conclusion *Le meilleur choix de la base de calcul est ß=B.*

Cette conclusion est illustrée clairement par l'exemple suivant.

Soit τ_{add} -τ_{mul}- le coût d'une addition -multiplication- en base B.
Et soit $T_{add}(ß)$ -$T_{mul}(ß)$- le coût d'une addition -multiplication- en base ß.

Alors : $\begin{cases} T_{add}(ß) \geq \dfrac{Log\ (ß)}{Log\ (B)}.\tau_{add} \\ T_{mul}(ß) \gg \dfrac{Log\ (ß)}{Log\ (B)}.\tau_{mul} \end{cases}$

Si $ß=B^2$ (double précision), on a ainsi :

$$\begin{cases} T_{add}(ß) \geq 2.\tau_{add} \\ T_{mul}(ß) \gg 2.\tau_{mul} \end{cases}$$

Comme le gain apporté est supposé linéaire en fonction de γ, on en déduit que le nombre d'étapes d'Euclide cumulées en base ß est exactement deux fois le nombre de pas cumulés en base B.
La perte de temps en calcul est donc supérieure au gain de temps en nombre de pas cumulés : le calcul en base B est alors plus rapide que celui en base ß.

II.5. Complexité

Comme le nombre d'étapes cumulées est borné, l'algorithme de Lehmer a le même ordre de complexité que l'algorithme d'Euclide, à savoir [COL74]:

$$\left[1 + \log\left(\frac{\max(u,v)}{(u,v)}\right)\right].\log\left(\min(u,v)\right)$$

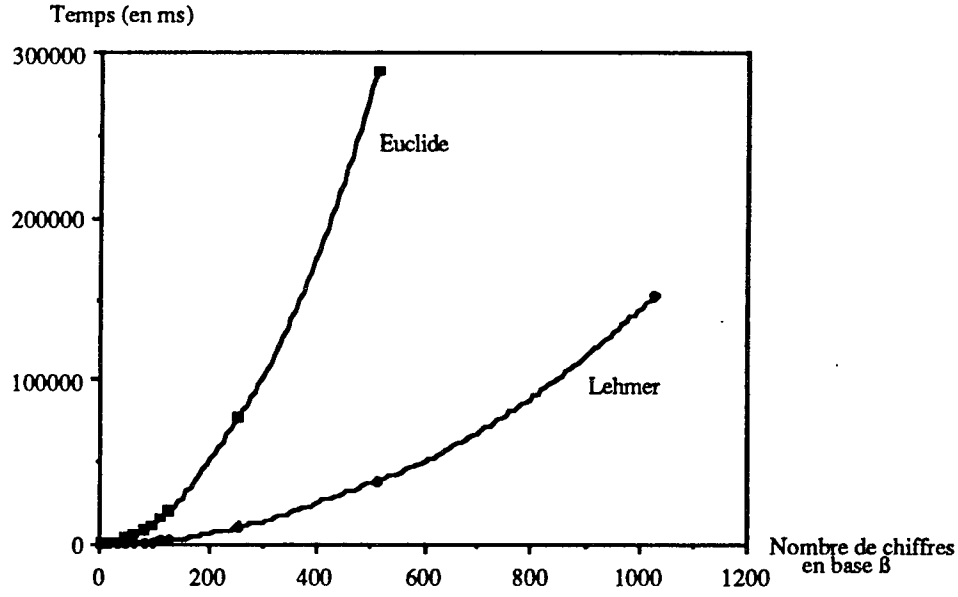
Nous avons vu précédemment que le coût arithmétique d'une étape de l'algorithme de Lehmer est approximativement quatre fois celui d'une étape de l'algorithme d'Euclide.
Soit τ le nombre moyen de divisions précision infinie cumulées à chaque étape.
Le facteur d'accélération apporté par l'algorithme de Lehmer sur l'algorithme d'Euclide est donc approximativement : $\tau/4$

Application au modèle arithmétique

Pour $B=ß=2^{31}$, l'étude précédente montre qu'environ 8,5 divisions en précision infinie sont cumulées en moyenne.
Le facteur d'accélération attendu est donc supérieur à 2.
En fait, l'algorithme de Lehmer étant particulièrement adapté à une implantation très fine (par exemple la mise à jour des coefficients de Bezout (2.5.1)), le facteur expérimental est bien supérieur à cette valeur théorique.

II.6. Résultats expérimentaux

Les mesures ont été faites sur deux nombres tirés aléatoirement ayant le même nombre de chiffres en base ß.

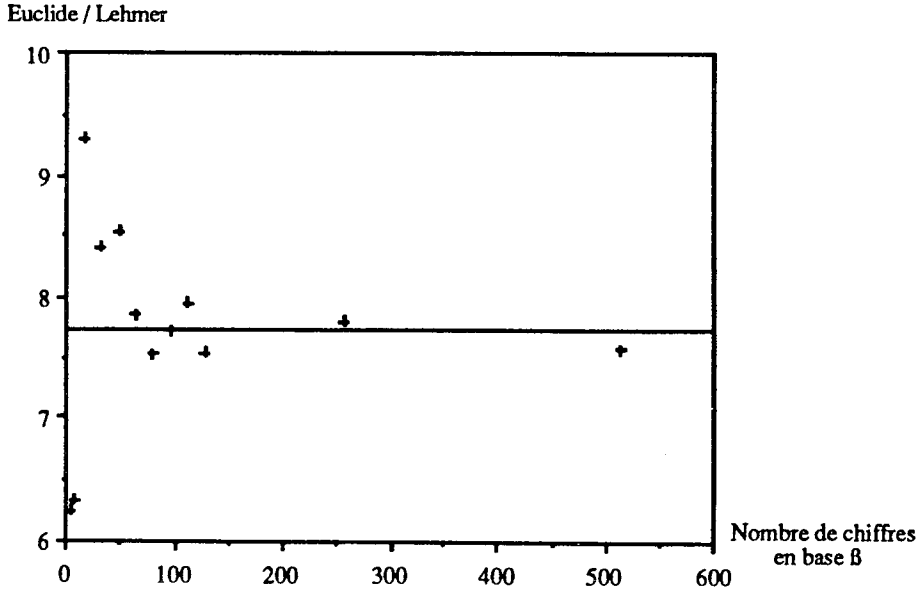


Comparaison des temps de l'algorithme d'Euclide et de celui de Lehmer

Les complexités expérimentales mesurées par meilleure approximation logarithmique sont :

Euclide	$T(n) \approx 3{,}17.n^{1{,}810}$	R = 1
Lehmer	$T(n) \approx 0{,}41.n^{1{,}814}$	R = 1

Les mesures montrent que l'algorithme de Lehmer est bien du même ordre que l'algorithme d'Euclide, mais qu'il est meilleur d'un facteur proche de 8, comme le précise la figure suivante :



Rapport des temps de l'algorithme d'Euclide sur celui de Lehmer

Remarque

Un avantage considérable que présente l'algorithme de Lehmer est d'être toujours meilleur que celui d'Euclide (même si le facteur de proportionnalité n'est plus aussi grand pour les petites tailles).

III. L'ALGORITHME DE LEHMER GENERALISE

III.1. Présentation

Le paragraphe précédent montre qu'il est possible de cumuler un certain nombre d'étapes dans l'algorithme d'Euclide. Lorsque le quotient ne peut plus être directement évalué à partir des valeurs des poids forts des opérandes, les deux entiers sont mis à jour.
Pourtant, les multiplications effectuées dans cet algorithme sont *déséquilibrées* : on multiplie chaque fois un entier précision infinie par un unique ß-chiffre. L'algorithme de Lehmer n'est donc pas adapté à l'utilisation de procédures de multiplication rapide. Le but de l'algorithme présenté ici est de cumuler entre elles plusieurs étapes de l'algorithme de Lehmer, de façon à obtenir des multiplications équilibrées.
Cela peut être réalisé en ne mettant à jour que les chiffres qui sont nécessaires à l'évaluation d'un nouveau quotient (calcul qui n'est basé que sur les premiers bits des opérandes).

Soit $u = [u_{2.n-1},\ldots,u_0]$ et $v = [v_{2.n-1},\ldots,v_0]$ deux ß-entiers ayant 2.n ß-chiffres dont on désire calculer le pgcd. On suppose $u<v$.

Définition 1
On appelle matrice de Bezout de rang i (i entier positif) associée à u et v - notée $\mathcal{B}^{(i)}(u,v)$ - la matrice définie itérativement par :

$$\begin{cases} \mathcal{B}^{(0)}(u,v) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \mathcal{B}^{(k)}(u,0) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \text{si } v \neq 0 \text{ alors } \mathcal{B}^{(k)}(u,v) = \mathcal{B}^{(k-1)}(v,\ u - q.v).\begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \text{ où } q = \left\lfloor \frac{u}{v} \right\rfloor \end{cases}$$

Remarque :

On a : $$\mathcal{B}^{(i)}(u,v) = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix}.\begin{bmatrix} 0 & 1 \\ 1 & -q_{i-1} \end{bmatrix}.\ldots\begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix}$$

où les q_k (k=1...i) sont les quotients successifs obtenus en appliquant l'algorithme d'Euclide à u et v; on a donc :

$$\frac{v}{u} = < q_1, q_2, \ldots q_i, \ldots >$$

Définition 2

On appelle matrice de Lehmer associée aux entiers positifs u_1, v_1, u_2, et v_2 - notée $\mathcal{M}_{Lehmer}(u_1, v_1, u_2, v_2)$ - la matrice $\mathcal{B}^{(i)}(u_1, v_1)$ où i est le plus grand entier positif tel que :

$$\mathcal{B}^{(k)}(u_1, v_1) = \mathcal{B}^{(k)}(u_2, v_2) \quad \forall k \,/\, 0 \leq k \leq i$$

Notation

i est appelé dans la suite *indice de Lehmer* associé à $\mathcal{M}_{Lehmer}(u_1, v_1, u_2, v_2)$.

Remarque :

Soit $\frac{v_1}{u_1} = <0, f_1, \ldots, f_k, \ldots>$ et $\frac{v_2}{u_2} = <0, f'_1, \ldots, f'_k, \ldots>$.

L'indice de Lehmer i vérifie : $\begin{cases} f'_k = f_k & \forall \quad k \leq i \\ f'_{i+1} \neq f_{i+1} \end{cases}$

Autrement dit, l'indice de Lehmer est le nombre de termes communs à toutes les fractions continues des réels de l'intervalle $\left[\frac{v_1}{u_1}, \frac{v_2}{u_2} \right]$.

Proposition 1

Soient u et v deux entiers. Et soient u_H, u_L, v_H, v_L quatre entiers positifs vérifiant : $\exists k_1, k_2 \in \mathbf{N} \,/\, \begin{cases} u_H.k_1 \geq u \text{ et } v_L.k_1 \leq v \\ u_L.k_2 \leq u \text{ et } v_H.k_2 \geq v \end{cases}$

Alors : $\mathcal{B}^{(i)}(u, v) = \mathcal{M}_{Lehmer}(u_H, v_L, u_L, v_H)$

où i est l'indice de Lehmer associé à $\mathcal{M}_{Lehmer}(u_H, v_L, u_L, v_H)$

Preuve :

Elle est immédiate en raisonnant par récurrence sur l'indice de Lehmer associé à $\mathcal{M}_{Lehmer}(u_H, v_L, u_L, v_H)$.

Du fait des encadrements de u et v : $\frac{u_H}{v_L} \leq \frac{u}{v} \leq \frac{u_L}{v_H}$

Si $\frac{u_H}{v_L} \neq \frac{u_L}{v_H}$ alors $\mathcal{M}_{Lehmer}(u_H, v_L, u_L, v_H) = Id$, et l'indice de Lehmer associé i=0 convient.

Sinon, soit $q = \frac{u_H}{v_L} = \frac{u_L}{v_H}$. On a alors $\frac{u}{v} = q$.

De plus, de manière évidente :

$$\begin{cases} k_2.v_H \geq v \text{ et } k_2.(u_L - q.v_H) \leq (u - q.v) \\ k_1.v_L \leq v \text{ et } k_1.(u_H - q.v_L) \geq (u - q.v) \end{cases}$$

Et, par induction : $i = 1 + j$ où j est l'indice de Lehmer associé à $\mathcal{M}_{Lehmer}(v_H, u_L - q.v_H, v_L, u_H - q.v_L)$.

Proposition 2

Soient u_1, v_1, u_2, et v_2 quatre entiers positifs tels que :.

$$\begin{cases} u_1 \geq u_2 \\ v_1 \leq v_2 \end{cases}$$

Soient u_H, u_L, v_H, v_L quatre entiers positifs vérifiant :

$$\exists k_1,k_2 \in \mathbf{N} \ / \quad \begin{cases} u_H.k_1 \geq u_1 \text{ et } v_L.k_1 \leq v_1 \\ u_H.k_2 \leq u_2 \text{ et } v_H.k_2 \geq v_2 \end{cases}$$

Alors $\exists A \in \mathcal{M}_{2,2}(\mathbf{Z})$ *tel que :*

$$\mathcal{M}_{Lehmer}(u_1, v_1, u_2, v_2) = A.\mathcal{M}_{Lehmer}(u_H, v_L, u_L, v_H)$$

Preuve : elle est immédiate en utilisant la proposition précédente.
La matrice $\mathcal{M}_{Lehmer}(u_H, v_L, u_L, v_H)$ correspond en effet au cumul de toutes les étapes de l'algorithme d'Euclide - c'est à dire les quotients successifs - communes au quatre couples (u_1, v_1), (u_2, v_2), (u_H, v_L) et (u_L, v_H).
Elle constitue donc un facteur de $\mathcal{M}_{Lehmer}(u_1, v_1, u_2, v_2)$, qui correspond au cumul des étapes communes au deux seuls couples (u_1, v_1) et (u_2, v_2),

---

Proposition 3

Soient $u_1 \geq v_1$, $u_2 \geq v_2$ quatre entiers positifs. Alors :

$$\max_{\substack{i=1,2 \\ j=1,2}} \{ |[\mathcal{M}_{Lehmer}(u_1, v_1, u_2, v_2)]_{i,j}| \} \leq \max \{ |u_1|, |u_2| \}$$

Preuve : elle se déduit directement du théorème 5 (§I.5).

Principe de l'algorithme :

Soient $u = [u_n \ldots u_0]_\beta$ et $v = [v_n \ldots v_0]_\beta$ deux ß-entiers, avec $u \geq v > 0$. On suppose que l'on connaît un encadrement de $\frac{u}{v}$: $\frac{u_H}{v_L} \leq \frac{u}{v} \leq \frac{u_L}{v_H}$

On peut alors calculer la matrice de Lehmer M associée à (u_H, v_L) et (u_L, v_H):

$$M = \mathcal{M}_{Lehmer}(u_H, v_L, u_L, v_H)$$

M correspond au i premiers termes du développement en fraction continue de $\frac{v}{u}$ avec i indice de Lehmer associé à M.

Soient u' et v' les itérés correspondants de u et v : $\begin{bmatrix} u' \\ v' \end{bmatrix} = M.\begin{bmatrix} u \\ v \end{bmatrix}$

L'algorithme consiste à déterminer quatre entiers (u'_L,u'_H) et (v'_L,v'_H) pour obtenir un encadrement de u' et v', mais sans calculer u' et v'.

Soit n_0 tel que : $\beta^{n_0} > \text{Max}\ \{\ |M_{1,1}|, |M_{1,2}|, |M_{2,1}|, |M_{2,2}|\ \} \geq \beta^{n_0-1}$

En utilisant la décroissance de l'algorithme d'Euclide, on a : $u' \leq u$ et $v' \leq v$. Posons $u' = [u'_n \ldots u'_0]_\beta$ et $v' = [v'_n \ldots v'_0]_\beta$. La décroissance de u et v étant logarithmique, on a dans le cas général : $u'_n=\ldots=u'_{n-n_0+2}=v'_n=\ldots=v'_{n-n_0+2}=0$, et $[u'_{n-n_0+1},u'_{n-n_0}]_\beta \neq 0$, $[v'_{n-n_0+1},v'_{n-n_0}]_\beta \neq 0$.

Posons: $\hat{u}' = [u'_n \ldots u'_{n-n_0}]_\beta$ et $\hat{v}' = [v'_n \ldots v'_{n-n_0}]_\beta$

Soient $\bar{u}$, $\bar{v}$ et $\bar{u}'$, $\bar{v}'$ définis par : $\begin{cases} \bar{u} = [u_n \ldots u_{n-\mu}]_\beta \\ \bar{v} = [v_n \ldots v_{n-\mu}]_\beta \end{cases}$ et $\begin{bmatrix} \bar{u}' \\ \bar{v}' \end{bmatrix} = M.\begin{bmatrix} \bar{u} \\ \bar{v} \end{bmatrix}$

Il s'agit donc de déterminer μ le plus petit possible qui permette d'obtenir un encadrement de très faible amplitude de $\hat{u}'$ et $\hat{v}'$.

On a : $$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} \bar{u}' \\ \bar{v}' \end{bmatrix}.\beta^{n-\mu} + M.\begin{bmatrix} u-\bar{u}'.\beta^{n-\mu} \\ v-\bar{v}'.\beta^{n-\mu} \end{bmatrix} \qquad (1)$$

et aussi : $$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} \hat{u}' \\ \hat{v}' \end{bmatrix}.\beta^{n-n_0} + \begin{bmatrix} [u'_{n-n_0-1} \ldots u'_0]_\beta \\ [v'_{n-n_0-1} \ldots v'_0]_\beta \end{bmatrix} \qquad (2)$$

Pour approcher $\hat{u}'$ et $\hat{v}'$uniquement avec $\bar{u}'$ et $\bar{v}'$, on obtient à partir de (1) et (2) la condition sur μ : $0 \leq |\ M_{1,1}.(u-\bar{u}.\beta^{n-\mu})+M_{1,2}.(v-\bar{v}.\beta^{n-\mu})\ | < \beta^{n-n_0}$ (3)

Or du fait de la borne sur M et du choix de $\bar{u}$ et $\bar{v}$, on a :

$$|\ M_{1,1}.(u-\bar{u}.\beta^{n-\mu})+M_{1,2}.(v-\bar{v}.\beta^{n-\mu})\ | < \beta^{n+n_0-\mu} \qquad (4)$$

En remplaçant dans (3), on obtient que la condition ($\mu \geq 2.n_0$) convient. Par suite, la valeur convenable de μ est :

$$\boxed{\mu = 2.n_0} \qquad (5)$$

Encadrement de $\hat{u}'$ et $\hat{v}'$:

Comme M est une matrice de Lehmer bornée par β^{n_0}, à partir des égalités (1) et (2), et en utilisant la valeur de μ (5), on obtient :

$$\beta^{n-n_0}.\hat{u}' \le u' < \bar{u}'.\beta^{n-2.n_0}+\beta^{n-n_0}$$

D'où : $\hat{u}' < \bar{u}'.\beta^{-n_0}+1 \Rightarrow \hat{u}' \le \lceil \bar{u}'.\beta^{-n_0} \rceil$ ($\hat{u}'$ est entier)

En reprenant (1) et (2), on a aussi :

$$\bar{u}'.\beta^{n-2.n_0}-\beta^{n-n_0} \le u' < (\hat{u}' + 1).\beta^{n-n_0}$$

D'où : $\hat{u}' > \bar{u}'.\beta^{-n_0}-2 \Rightarrow \hat{u}' \ge \lfloor \bar{u}'.\beta^{-n_0} \rfloor -1$ ($\hat{u}'$ est entier)

Le calcul étant similaire pour encadrer $\hat{v}'$, on obtient finalement :

$$\begin{cases} \lfloor \bar{u}'.\beta^{-n_0} \rfloor -1 \le \hat{u}' \le \lceil \bar{u}'.\beta^{-n_0} \rceil \\ \lfloor \bar{v}'.\beta^{-n_0} \rfloor -1 \le \hat{v}' \le \lceil \bar{v}'.\beta^{-n_0} \rceil \end{cases}$$

On peut alors déterminer quatre entiers (u'_L,u'_H) et (v'_L,v'_H) pour encadrer u'/v', mais sans calculer u' et v':

Posons : $$\begin{bmatrix} u'_L & u'_H \\ v'_H & v'_L \end{bmatrix} = \begin{bmatrix} \lfloor \bar{u}'.\beta^{-n_0} \rfloor -1 & \lfloor \bar{u}'.\beta^{-n_0} \rfloor +1 \\ \lfloor \bar{v}'.\beta^{-n_0} \rfloor +1 & \lfloor \bar{v}'.\beta^{-n_0} \rfloor -1 \end{bmatrix}$$

On a alors : $$\boxed{\frac{u'_L}{v'_H} \le \frac{u'}{v'} \le \frac{u'_H}{v'_L}}$$

III.2. Algorithme

Lemme :

Soit $u_H \ge u_L \ge v_H \ge v_L \ge 1$ quatre entiers, et soit $\mathcal{M}_{Lehmer}(u_H,v_L,u_L,v_H)$ l'algorithme suivant :

$\mathcal{M}$**Lehmer**(u_H,v_L,u_L,v_H) : $\mathcal{M}_{2,2}(\mathbf{Z})$ ==

  M $\leftarrow I_{2,2}$;

  Tantque ($(|v_L|+|v_H| \neq 0)$ **et** $(\lfloor u_H/v_L \rfloor = \lfloor u_L/v_H \rfloor)$) **faire**

    $M_0 \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor u_H/v_L \rfloor \end{bmatrix}$;

    $M \leftarrow M_0.M$; $\quad \begin{bmatrix} u_L & u_H \\ v_H & v_L \end{bmatrix} \leftarrow M_0.\begin{bmatrix} u_H & u_L \\ v_L & v_H \end{bmatrix}$

  retourner M;

Alors $\mathcal{M}_{Lehmer}(u_H,v_L,u_L,v_H)$ calcule la matrice de Lehmer associée aux quatre entiers u_H,bv_L, u_L, v_H.

Preuve : directe d'après la définition de la matrice de Lehmer (définition 2).

Théorème
Soient $u \geq v \geq 1$ deux entiers positifs et soit (LG) l'algorithme suivant :

(LG)

$\mathcal{B} \leftarrow I_{2,2}$;
Tantque ($v \neq 0$) **faire**
  $n \leftarrow \lfloor \mathrm{Log}_\beta\, u \rfloor$;
  $M \leftarrow I_{2,2}$; $n_0 \leftarrow 0$;
  Tantque $\left(n_0 < \frac{n}{2}\right)$ **faire**
    $\mu \leftarrow 2.n_0$
    $\begin{bmatrix} \bar{u}' \\ \bar{v}' \end{bmatrix} \leftarrow M.\begin{bmatrix} \lfloor u.\beta^{-n+\mu} \rfloor \\ \lfloor v.\beta^{-n+\mu} \rfloor \end{bmatrix}$;
    $\begin{bmatrix} u_L \\ v_L \end{bmatrix} \leftarrow \begin{bmatrix} \lfloor \bar{u}'.\beta^{-n_0} \rfloor - 1 \\ \lfloor \bar{v}'.\beta^{-n_0} \rfloor - 1 \end{bmatrix}$; $\begin{bmatrix} u_H \\ v_H \end{bmatrix} \leftarrow \begin{bmatrix} u_L+2 \\ v_L+2 \end{bmatrix}$;
    $M_0 \leftarrow \mathcal{M}_{\mathrm{Lehmer}}(u_H, v_L, u_L, v_H)$;
    si ($M_0 = I_{2,2}$) **alors**
      $n_0 \leftarrow n_0 + 1$;
    sinon
      $M \leftarrow M_0.M$;
      $n_0 \leftarrow \lceil \mathrm{Log}_\beta (\mathrm{Max} \{ |M_{1,1}|, |M_{1,2}|, |M_{2,1}|, |M_{2,2}| \}) \rceil$;
  si ($M = I_{2,2}$) **alors** $M \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor u/v \rfloor \end{bmatrix}$;
  $\begin{bmatrix} u \\ v \end{bmatrix} \leftarrow M.\begin{bmatrix} u \\ v \end{bmatrix}$;
  $\mathcal{B} \leftarrow M.\mathcal{B}$;
$\delta \leftarrow u$;

Alors l'algorithme (LG) affecte à δ le pgcd des valeurs de u et v, et la matrice $\mathcal{B}$ est la matrice des coefficients de Bezout associés par l'algorithme d'Euclide.

Preuve : elle découle directement des propositions 1, 2, 3 et des résultats sur l'encadrement de u'/v'.

III.3. Implantation et expérimentation

De nombreuses remarques faites pour l'algorithme de Lehmer sont encore valides pour l'algorithme de Lehmer généralisé.
Lors du calcul de M_0 avec l'algorithme $\mathcal{M}_{Lehmer}$, u_H a souvent plusieurs ß-chiffres. En fait, ce calcul de M_0 doit être le plus efficace possible. Pour cela, il est préférable de ne sélectionner que les (γ-1) premiers bits de u_H (en y ajoutant 1 comme pour l'algorithme de Lehmer) et de considérer les bits correspondants de u_L, v_H et v_L.
Grâce au théorème 5 (§5.1), on déduit que les coefficients de M_0 obtenus sont inférieurs à ß/2. Il s'ensuit de nombreuses améliorations dans le calcul du produit matriciel $M.M_0$, similaires à celles décrites pour l'algorithme de Lehmer classique (§II.3 - (2.5.1)).

Une autre remarque importante vient de la complexité du calcul de $\overline{u}'$ et $\overline{v}'$ par le produit matrice-vecteur en début de deuxième boucle. La complexité de ce calcul est en effet $2.M(n_0,2.n_0)$. Pour être sûr d'avoir un gain effectif par rapport à l'algorithme de Lehmer classique, il faut que se coût soit inférieur à 2.n. Si la multiplication utilisée est standard (opérandes de moins de 300 chiffres), on en déduit que l'algorithme sera de toute façon meilleur si $n_0 \leq \sqrt{\frac{n}{2}}$. On a donc pour n_0 l'encadrement :

$$\boxed{\sqrt{\frac{n}{2}} \leq n_0 \leq \frac{n}{2}}$$

Il est donc essentiel d'étudier expérimentalement le choix de n_0.

Par ailleurs, l'implantation de l'algorithme de Lehmer classique étant particulièrement adapté à une implantation efficace, il faut prévoir, comme pour l'algorithme de Karatsuba, une valeur N critique en dessous de laquelle il sera plus rapide que celui présenté ci-dessus.

III.4. Complexité

C'est encore un mystère pour moi... Le pire cas est lorsqu'à chaque étape de l'algorithme d'Euclide le quotient ne peut être deviné par examen des poids forts de u et v, ce qui est de toute façon très rare (cf §II.4.1).
Une chose est sûre : cette complexité théorique est comprise entre celle de l'algorithme d'Euclide et celle de l'algorithme de Schönhage !

IV. L'ALGORITHME DE SCHÖNHAGE

IV.1. Présentation

L'algorithme d'Euclide, nous l'avons vu, ne fait pas apparaître de grands quotients à chaque étape. Il est donc impossible de tirer parti d'algorithmes de multiplication ou de division *rapides* puisqu'ils ne font apparaître de gain convainquant que lorsque le plus petit des deux opérandes est de grande taille.

Le problème est donc d'essayer de regrouper un certain nombre d'étapes de l'algorithme d'Euclide, mais un nombre qui varie avec la taille des opérandes - à la différence de la méthode de Lehmer classique, mais c'est là un point commun important avec l'algorithme de Lehmer généralisé - . En 1970, D.E. Knuth [KNU70] donne une première idée de méthode qui permet de ramener le coût du pgcd à $O(n.(\log n)^5.(\log \log n))$. En 1971, A. Schönhage [SCH71] propose une version plus fine de cet algorithme, en $O(n.(\log n)^2.(\log \log n))$. Il montre donc que le pgcd est au plus d'un facteur log plus complexe que la multiplication. La généralisation de cet algorithme à un domaine euclidien quelconque a été réalisée par Mœnck [MŒ73].

L'idée de base est de considérer non pas le calcul du pgcd final, mais le calcul des différents quotients obtenus successivement dans l'algorithme d'Euclide, autrement dit le calcul du développement en fractions continues d'un rationnel.
La présentation qui est faite ci-dessous est, dans un premier temps, volontairement qualitative : son but est d'expliciter la construction de l'algorithme.

Soient $u \geq v > 1$ deux entiers dont on désire calculer le pgcd.
Au lieu de considérer la fraction continue du rationnel $\frac{v}{u}$, on s'intéresse à l'intervalle maximal $[x_1,y_1]$ des nombres réels dont les premiers termes du développement en fractions continues coïncident avec celui de $\frac{v}{u}$.

Soit $\frac{v}{u} = <0,f_1,\ldots,f_p>$ le développement de $\frac{v}{u}$. Alors, (x_1,y_1) vérifient :

$$\forall x \in [x_1,y_1] : \quad x = <0,f_1,\ldots,f_p,\ldots> \qquad (1)$$

Proposition 1 [SCH71]: $\forall x \in \left[\frac{v}{u}, \frac{v}{u}+\frac{1}{u^2}\right] : \quad x = <0,f_1,\ldots,f_p,\ldots>$

Dans la suite, on considère quatre entiers u_L, v_L, u_H et v_H tels que :

$$1 \leq \frac{u_L}{v_L} \leq \frac{u}{v} \leq \frac{u_H}{v_H} \qquad \text{avec} \qquad \begin{cases} u_L \geq v_L > 0 \\ u_H \geq v_H > 0 \end{cases}$$

D'après la proposition 1, le choix de u_L, v_L, u_H et v_H est direct.

On pose alors : $x_1 = \frac{u_L}{v_L}$ et $y_1 = \frac{u_H}{v_H}$ et $x_0 = \frac{1}{x_1}$ et $y_0 = \frac{1}{y_1}$

On appelle amplitude de l'intervalle $[x_1,y_1]$ la quantité $\delta = x_0 - y_0$.
Si δ est trop grand, peu de termes des développements en fractions continues de x_1 et y_1 coïncideront. Par exemple, si $\delta = 1/2$, alors aucun terme ne sera commun aux deux développements.

La construction des développements en fractions continues de x_1 et y_1 est définie par l'itération (cf §I.4) :

$$(2) \qquad \begin{cases} \alpha_k = \lfloor x_k \rfloor & x_k = \alpha_k + \frac{1}{x_{k+1}} \\ \beta_k = \lfloor y_k \rfloor & y_k = \beta_k + \frac{1}{y_{k+1}} \end{cases} \qquad k=1,2,\ldots$$

Définitions
On appelle *indice des fractions continues de $[x_1,y_1]$* le plus petit entier positif t tel que : $\alpha_{t+1} \neq \beta_{t+1}$

On appelle *matrice des fractions continues d'ordre k de $[x_1,y_1]$* la matrice $M^{(k)}_{[x_1,y_1]}$ définie par : $M^{(k)}_{[x_1,y_1]} = \begin{bmatrix} 0 & 1 \\ 1 & -\alpha_k \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & -\alpha_{k-1} \end{bmatrix} \cdots \begin{bmatrix} 0 & 1 \\ 1 & -\alpha_1 \end{bmatrix}$

L'itération (2) menée pour $1 \leq k \leq t$ permet d'obtenir la fraction continue "commune" aux réels de l'intevalle $[x_1,y_1]$.
L'indice t dépend bien sûr de l'éloignement δ, ces deux quantités variant en sens contraire.
Soient ξ_k et ψ_k les tronqués des développements en fractions continues de x_1 et y_1 :

$$\begin{cases} \xi_k = <0,\alpha_1,\ldots,\alpha_k> \\ \psi_k = <0,\beta_1,\ldots,\beta_k> \end{cases}$$

Par propiété du développement d'un réel (cf fig.1 ci-dessous), les suites ξ_k (resp. ψ_k) encadrent x_0 (resp. y_0).
Les termes α_k et β_k coincident alors à coup sûr tant que les termes successifs ξ_{k-1} et ξ_k encadrent à la fois x_0 et y_0, c'est à dire :

$$(x_0, y_0) \in [\text{Min}(\xi_{k-1},\xi_k), \text{Max}(\xi_{k-1},\xi_k)]^2$$

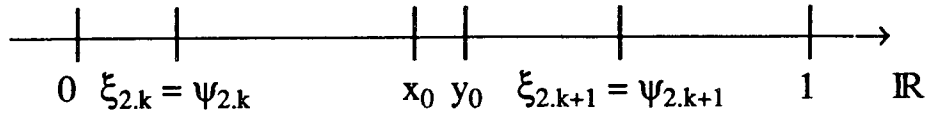


Fig.1 Encadrement de x_0 et y_0 par les fractions continues tronquées

Il s'agit alors d'obtenir une partie du développement commun aux réels de l'intervalle $[x_1,y_1]$ à partir de deux autres rationnels x_1' et y_1' plus faciles à manipuler, et vérifiant : $[x_1',y_1'] \supset [x_1,y_1]$

L'idée de base est donc similaire à celle de D.H. Lehmer, qui choisit pour x_1' (resp. y_1') le chiffre de poids fort de x_1 (resp. y_1).

Par analogie à x_0 et y_0, on pose : $x_0' = \frac{1}{x_1'}$ et $y_0' = \frac{1}{y_1'}$

On considère alors la quantité δ' : $\delta' = x_0' - y_0'$

***Proposition 2** [SCH71] :*
Soit t (resp. t') l'indice des fractions continues de $[x_1,y_1]$ (resp. $[x_1',y_1']$).
Et soient x_i et y_i les termes de la suite construits selon l'itération (2), appliquée à x_1 et y_1.
Alors, deux cas peuvent se produire :

- *Soit* $t \leq t'+2$
- *Soit* $t \geq t'+3$ *et* $\delta'' = |x_{t'+3} - y_{t'+3}| > \frac{\delta}{\delta'}$

Intérêt : une analyse qualitative de l'algorithme

On choisit x_1' et y_1' tels que : $\delta' \approx \sqrt{\delta}$

On peut alors calculer le développement en fractions continues sur l'intervalle $[x_1',y_1']$, ce qui est *"environ deux fois plus facile"* que le calcul de celui de l'intervalle $[x_1,y_1]$.

On applique la transformation de Lehmer (§III.1 - définition 2) ainsi calculée à partir de (x_1' et y_1') à (x_1 et y_1). Soient $x_{t'}$ et $y_{t'}$ les valeurs obtenues.

On effectue alors trois pas de l'itération (2) à partir de $x_{t'}$ et $y_{t'}$, qui permettent de déterminer les trois premiers termes du développement en fractions continues de $x_{t'}$ et $y_{t'}$.

Deux cas peuvent se produire :

♦ les termes de $x_{t'}$ diffèrent de ceux de $y_{t'}$. On connaît alors le développement en fractions continues de l'intervalle $[x_1,y_1]$: il a été obtenu grâce aux calculs ("deux fois plus simples") effectués pour $[x_1',y_1']$.

♦ ces termes coïncident. La proposition 2 montre que pour calculer les autres termes du développement de l'intervalle $[x_1,y_1]$, on est ramené à un problème de même taille que celui pour l'intervalle $[x_1',y_1']$.

Dans tous les cas, le problème initial a donc été ramené à au plus deux problèmes deux fois plus simples : *Diviser Pour Régner* oblige !!

IV.2. Algorithme

Lemme :
Soient $u_L \geq v_L > 0$ et $u_H \geq v_H > 0$ quatre entiers positifs et soit TroisPasEuclide l'algorithme suivant :

TroisPasEuclide(u_L, v_L, u_H, v_H) $\rightarrow \mathcal{M}_{2,2}(\mathbb{Q}) \times \mathbb{N}$==

$R \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$; $t \leftarrow 0$;

tantque $(t \leq 2)$ **et** $(v_L.v_H \neq 0)$ **et** $\left(\left\lfloor \frac{u_H}{v_H} \right\rfloor = \left\lfloor \frac{u_L}{v_L} \right\rfloor \right)$ **faire**

$R \leftarrow \begin{bmatrix} 0 & 1 \\ 1 - \left\lfloor \frac{u_L}{v_L} \right\rfloor & \end{bmatrix} . R$;

$\begin{bmatrix} u_L & u_H \\ v_L & v_H \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 - \left\lfloor \frac{u_L}{v_L} \right\rfloor & \end{bmatrix} . \begin{bmatrix} u_L & u_H \\ v_L & v_H \end{bmatrix}$;

$t \leftarrow t + 1$;

Retourner (R, t);

Alors l'algorithme TroisPasEuclide calcule l'entier t qui est :

- *soit l'indice des fractions continues de l'intervalle* $\left[\frac{u_L}{v_L}, \frac{u_H}{v_H} \right]$, *si cet indice est inférieur à trois.*
- *soit trois sinon.*

De plus, R est la matrice des fractions continues d'ordre t de $\left[\frac{u_L}{v_L}, \frac{u_H}{v_H} \right]$.

Preuve: immédiate, d'après les définitions de l'indice et de la matrice des fractions continues. L'algorithme effectue en effet les trois premiers pas de l'algorithme d'Euclide appliqué à (u_L, v_L) et (u_H, v_H) (si les quotients obtenus sont les mêmes).

Théorème 1

Soient $u_L \geq v_L > 0$ *et* $u_H \geq v_H > 0$ *quatre entiers vérifiant :* $\delta = \frac{v_H}{u_H} - \frac{v_L}{u_L} > 0$

Soit $\mathcal{U}(u_L, v_L, u_H, v_H)$ *l'algorithme suivant :*

$\mathcal{U}(u_L, v_L, u_H, v_H) \rightarrow \mathcal{M}_{2,2}(\mathbb{Q}) \times \mathbb{N} ==$

si $(v_L = 0)$ **ou** $(v_H = 0)$ **alors**

$(R, t) \leftarrow (I_{2,2}, 0)$

sinon

$\delta \leftarrow \frac{v_H}{u_H} - \frac{v_L}{u_L}$; $d \leftarrow \lceil \mathrm{Log}_2(\delta^{-1}) \rceil$; $m \leftarrow \left\lceil \frac{d+2}{2} \right\rceil$;

$v_L' \leftarrow \left\lceil \frac{2^m . v_L}{u_L} \right\rceil$; $v_H' \leftarrow \left\lceil \frac{2^m . v_H}{u_H} \right\rceil$; $u_H' \leftarrow 2^m$; $u_L' \leftarrow 2^m$;

$(R, t) \leftarrow \mathcal{U}(u_L', v_L', u_H', v_H')$;

$\begin{bmatrix} u_L'' & u_H'' \\ v_L'' & v_H'' \end{bmatrix} \leftarrow R . \begin{bmatrix} u_L & u_H \\ v_L & v_H \end{bmatrix}$;

$(R', t') \leftarrow$ *TroisPasEuclide* $(u_L'', v_L'', u_H'', v_H'')$;

si $(t' < 3)$ **alors**

$(R'', t'') \leftarrow (I_{2,2}, 0)$

sinon

$\begin{bmatrix} u_L''' & u_H''' \\ v_L''' & v_H''' \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^{t'+1} . R' . \begin{bmatrix} u_L'' & u_H'' \\ v_L'' & v_H'' \end{bmatrix}$;

$(R'', t'') \leftarrow \mathcal{U}(u_L''', v_L''', u_H''', v_H''')$;

$R \leftarrow R''.R'.R$; $t \leftarrow t''+t'+t$;

Retourner (R, t);

Alors l'algorithme $\mathcal{U}$ *calcule l'entier t et la matrice R tels que t est l'indice des fractions continues de l'intervalle* $\left[\frac{u_L}{v_L}, \frac{u_H}{v_H}\right]$ *et R la matrice des fractions continues d'ordre t du même intervalle.*

Preuve: elle découle directement des définitions du paragraphe précédent.
Par construction de u_L', v_L', u_H', v_H', on a : $\left[\frac{u_L'}{v_L'}, \frac{u_H'}{v_H'}\right] \supset \left[\frac{u_L}{v_L}, \frac{u_H}{v_H}\right]$.
Les termes du développement en fractions continues de $\left[\frac{u_L'}{v_L'}, \frac{u_H'}{v_H'}\right]$ sont donc identiques aux premiers termes du développement de $\left[\frac{u_L}{v_L}, \frac{u_H}{v_H}\right]$.
Les trois pas de l'algorithme d'Euclide effectués permettent de prouver que l'algorithme calcule bien tous les termes du développement en fractions continues de l'intervalle $\left[\frac{u_L}{v_L}, \frac{u_H}{v_H}\right]$, et par suite que la valeur t retournée est bien l'indice des fractions continues de cet intervalle.
L'algorithme s'arrête en un temps fini, puisque le développement en fractions continues d'un intervalle réel non réduit à un point est fini ($\mathbb{Q}$ est dense dans $\mathbb{R}$).

Corollaire 1
Soit $u \geq v > 0$ deux entiers, et soit $\mathcal{G}$ l'algorithme suivant :

$\mathcal{G}(u, v) \rightarrow \mathcal{M}_{2,2}(\mathbb{Q}) \times \mathbb{N} ==$
  $(M,t) \leftarrow \mathcal{U}(u, v, u^2, u.v + 1)$;
  $g \leftarrow [1\ 0] . M . \begin{bmatrix} u \\ v \end{bmatrix}$;
  Retourner (M, g)

L'algorithme $\mathcal{G}$ calcule la matrice de Bezout M délivrée par l'algorithme d'Euclide appliqué à u et v, et le pgcd g de u et v.

Preuve: elle se déduit directement de la proposition 1, et du théorème précédent.

Implantation :

L'algorithme décrit est relativement bien adapté à la programmation.
Différentes optimisations peuvent être effectuées :

- le dernier appel récursif dans $\mathcal{U}$ (récursivité à droite) peut être supprimé: il suffit ici de remplacer la condition d'arrêt portant sur la nullité éventuelle de v_L ou v_H par une boucle *tantque* sur la même condition.

- le calcul de m peut être effectué directement

IV.3. Complexité

Théorème 2

Soient $2^n > u_L \geq v_L > 0$ *et* $2^n > u_H \geq v_H > 0$ *quatre entiers avec :*

$$\delta = \frac{v_H}{u_H} - \frac{v_L}{u_L} > 0 \qquad et \quad \delta \geq 2^{-k}$$

Alors la complexité $C_{\mathcal{U}}(n,k)$ *du calcul de* $\mathcal{U}(u_L, v_L, u_H, v_H)$ *est :*

$$C_{\mathcal{U}}(n,k) = O(M(n) + M(k).Log(k))$$

où M(n) est la complexité du produit de deux entiers de n ß-chiffres.

Preuve:

Les coefficients de la matrice de Lehmer associée à u_L, v_L, u_H et v_H sont bornés par 2^n, d'après la proposition 3 (§III.1). Tous les produits matriciels (y compris les mises à jour de u_L, v_L, u_H et v_H) sont donc de complexité O(M(n)).

Il reste alors à borner les complexités des deux appels récursifs à $\mathcal{U}$, donc à borner les quantités u_L', v_L', u_H' et v_H' d'une part, et u_L'', v_L'', u_H'' et v_H'' d'autre part.

On pose : $d = \lceil \text{Log}_2(\delta^{-1}) \rceil$ et $m = \left\lceil \frac{d+2}{2} \right\rceil$

On se place ici dans le cas où $d \geq 4$. En effet, si $d \leq 3$, on applique l'algorithme d'Euclide directement à (u_L, v_L) et (u_H, v_H), et le coût est borné par O(M(n)).

Posons: $\delta' = \frac{v_H'}{u_H'} - \frac{v_L'}{u_L'}$ et $\delta'' = \frac{v_H''}{u_H''} - \frac{v_L''}{u_L''}$

On a l'encadrement important de δ' [SCH71]: $2^{-m} \leq \delta' < 2.2^{-m}$

N.B. : cette inégalité, qui repose sur des calculs de parties entières, est très fine, et, par suite, son obtention est très astucieuse.

En utilisant la proposition 2 du paragraphe précédent, on obtient alors la minoration de δ" : $\delta'' > \frac{\delta}{\delta'}$

D'où l'on tire : $\delta'' \geq 2^{-m}$

Finalement, δ' et δ" étant tous deux minorés par 2^{-m}, on en déduit que la complexité $C_{\mathcal{U}}(n,k)$ de l'algorithme $\mathcal{U}$ est bornée par :

si $k \geq 4$: $C_{\mathcal{U}}(n,k) \leq 2.C_{\mathcal{U}}(m+1,m) + O(M(n))$ avec $m = \left\lceil \frac{k+2}{2} \right\rceil$

si $k \leq 3$: $C_{\mathcal{U}}(n,k) = O(M(n))$

Après résolution de la récurrence, on obtient le résultat final :

$$C_{\mathcal{U}}(n,k) \leq O(M(n) + M(k).Log(k))$$

Corollaire 2

Soient $2^n > u \geq v > 0$ deux entiers .
Alors, la complexité $C_{\mathcal{G}}(n)$ du calcul de $\mathcal{G}(u, v)$ est :

$$C_{\mathcal{G}}(n) = O(M(n).Log(n))$$

où $M(n)$ est la complexité du produit de deux entiers de n ß-chiffres.

Preuve:
D'après le théorème 5 (§I.5), les coefficients de la matrice M sont bornés par $2^{2.n}$. La complexité du produit matriciel est donc O(M(n)).
De plus, lors de l'appel à $\mathcal{U}$, on a (en reprenant la quantité δ définie pour $\mathcal{U}$):

$$\delta = \frac{u.v + 1}{u^2} - \frac{v}{u} = \frac{1}{u^2} \geq 2^{-2.n}$$

La complexité $C_{\mathcal{G}}(n)$ est donc bornée par : $C_{\mathcal{G}}(n) \leq O(M(n)) + C_{\mathcal{U}}(n,2.n)$
D'où l'on déduit : $C_{\mathcal{G}}(n) \leq O(M(n).Log(n))$

Conclusion

En utilisant la complexité asymptotique de la multiplication de deux entiers [SST71], on obtient alors :

$$\boxed{C_{\mathcal{G}}(n) \leq O(n . Log^2(n) . Log(Log(n)))}$$

V. UN META-ALGORITHME DE CALCUL DU PGCD

Les résultats expérimentaux montrent que chacun des trois algorithmes est le plus intéressant sur un intervalle donné. Le meilleur algorithme est alors un compromis entre l'algorithme de Lehmer classique, l'algorithme de Lehmer généralisé et l'algorithme de Schönhage.

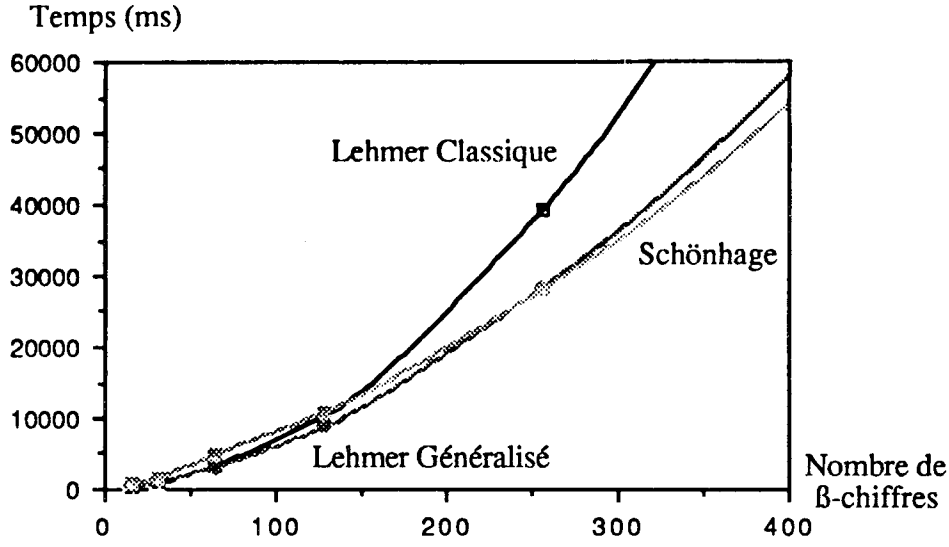


Fig. V Comparaison des trois algorithmes
(Implantation en cours d'amélioration pour Lehmer généralisé)

Nous avons vu (§ III) que l'algorithme de Lehmer classique pouvait être considéré comme un cas particulier de l'algorithme de Lehmer généralisé. Avec un bon choix des constantes de contrôle (cf III.3.), ce dernier devient toujours plus efficace que l'algorithme de Lehmer classique.

Seul, l'algorithme de Schönhage peut alors apporter un intérêt supplémentaire. L'expérimentation montre que le seuil entre les deux algorithmes se situe aux alentours de 700 chiffres décimaux. L'avantage apporté par l'algorithme de Lehmer généralisé est donc consistant.

Chapitre VII

ARITHMETIQUE DANS $\mathbb{Z}$

I. Représentation d'un entier signé

Un entier relatif est vu comme un entier naturel signé : il est donc représenté par sa valeur absolue en base ß, à laquelle on adjoint une information sur le signe (positif ou négatif) de l'entier.

Cette représentation est très naturelle et -*donc*...- classique : il est clair qu'avec ce choix, toutes les opérations arithmétiques sur $\mathbb{Z}$ pourront être réalisées à l'aide des opérations arithmétiques déjà définies sur $\mathbf{N}$.

D'autres représentations auraient pu être choisies. Citons, par exemple, la représentation en base négative, avec des chiffres positifs [KNU81e]. Le choix de la représentation en valeur absolue signée est motivé par :

1° L'encombrement mémoire est minimisé : le stockage dense sous format bloc est facile (§ Chapitre I). De plus, un entier positif tient la même place que son opposé. La remarque est vraie en base négative aussi.

2° L'arithmétique pourra être, à moindre coût, construite sur l'arithmétique dans $\mathbf{N}$.

3° Le calcul du signe est effectué en coût constant (coût linéaire pour la représentation en base négative à chiffres positifs).

Implantation

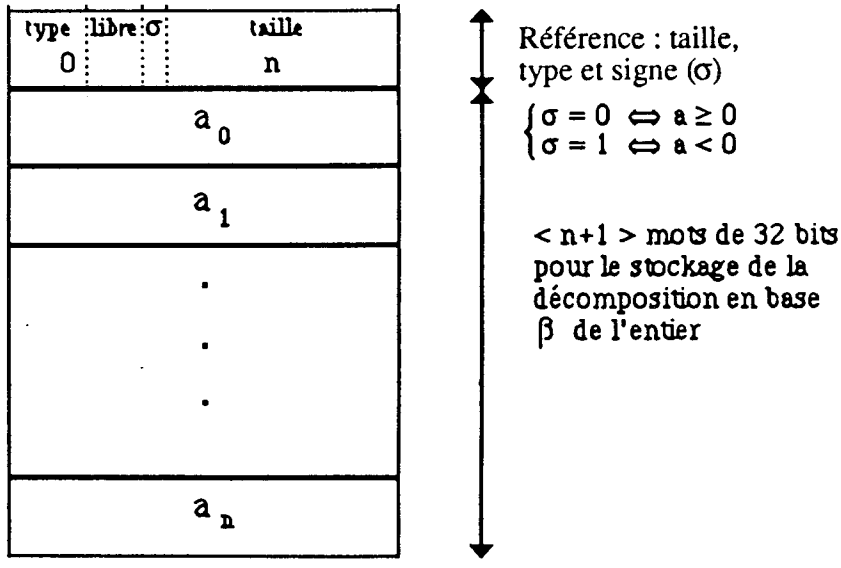


Fig. V.1 Représentation de l'entier $a = (-1)^s . \sum_{i=0}^{n} a_i . \beta^i$ avec $\begin{cases} 0 \le a_i < \beta & i=0 \ldots n \\ 0 < a_n < \beta & \text{si } n \neq 0 \end{cases}$

N.B. : On représente 0 de manière unique : $0 = (-1)^0.0.\beta^0$

II. Les opérateurs surchargés

Effectuer une opération arithmétique (+,-,*,/) sur des entiers signés se fait à l'aide de l'arithmétique non signée en trois étapes :

1° Calcul de l'opération correspondante dans **N** : opération et opérandes
2° Calcul effectif de l'opération dans **N**, et récupération du résultat
3° Calcul du résultat dans $\mathbb{Z}$ à partir du résultat dans **N**

Pour l'addition et la soustraction : on se ramène soit à l'addition des valeurs absolues, soit à la soustraction de la plus grande valeur absolue et de la plus petite.
La multiplication se ramène à la multiplication des valeurs absolues, le calcul du signe se ramenant à un ou-exclusif sur les signes des opérandes.

La division pose quelques problèmes. Il est essentiel de pouvoir disposer d'une opération semblable à la division euclidienne dans **N**, mais il existe plusieurs façon de la définir dans $\mathbb{Z}$. Elle intervient dans le calcul du pgcd de deux entiers, et est très utile pour calculer le résidu d'entiers modulo un entier déterminé (par exemple, pour borner la taille de coefficients intermédiaires lorsque c'est possible : une application peut être trouvée dans le calcul de la forme d'Hermite d'une matrice, où les calculs peuvent être réalisés modulo le déterminant de la matrice [SIE89]). Il reste à préciser les spécificités de cette opération.

Soient $(a,b) \in \mathbb{Z} \times \mathbb{Z}^*$. Cette opération doit permettre de calculer deux entiers relatifs q et r tels que :

$$a = b.q + r$$

A priori, différents choix sont possibles. Il est donc nécessaire de préciser les contraintes auxquelles sont soumis le quotient q et le reste r :

- q et r doivent être définis de manière unique
- q et r doivent correspondre au quotient et au reste de la division euclidienne lorsque les opérandes sont positifs (extension de la division dans **N** à $\mathbb{Z}$)
- Périodicité du modulo : soit m un entier; l'opération "modulo m" est de période m, i.e. : $\forall u \in \mathbb{Z} : (u + m) \bmod m = u \bmod m$

Cette propriéte est essentielle lorsque l'on effectue des calculs modulaires [SIE89].

Pour répondre à ces contraintes, la convention retenue est :

$$0 \leq r < |b|$$

Donc, si q et r sont respectivement le quotient et le reste de la division euclidienne de deux entiers positifs a et b, alors on a :

dividende	diviseur	quotient	reste
a	b	q	r
-a	b	-q si r=0 -(q+1) si r≠0	0 si r=0 (b-r) si r≠0
a	-b	-q	r
-a	-b	q si r=0 (q+1) si r≠0	0 si r=0 (b-r) si r≠0

Remarque

Dans la plupart des langages, la division est étendue aux entiers négatifs, en imposant comme contrainte sur r :

$$\begin{cases} 0 \le r < |b| \text{ si } a > 0 \\ -|b| < r \le 0 \text{ si } a < 0 \end{cases}$$

Cette opération est très facile à effectuer à partir de la division dans **N**. En effet, si q et r sont respectivement le quotient et le reste de la division euclidienne de deux entiers positifs a et b, on a alors:

dividende	diviseur	quotient	reste
a	b	q	r
-a	b	-q	-r
a	-b	-q	r
-a	-b	q	-r

Cette division présente un inconvénient majeur: elle ne conserve pas la propriété de périodicité du modulo.

Chapitre VIII

ARITHMETIQUE DANS $\mathbb{Q}$

I. Représentation d'un rationnel

I.1. Nécessité d'une pluri-représentation

En vue d'utiliser l'arithmétique signée, dejà définie, comme brique de base de l'arithmétique rationnelle, un rationnel sera considéré comme la juxtaposition de deux entiers : le numérateur et le dénominateur.

Définition :
On appelle forme réduite d'un rationnel r l'unique représentation de r pour laquelle le dénominateur est positif et premier avec le numérateur.

Le problème inhérent est alors que deux rationnels peuvent être égaux, sans avoir la même représentation, lorsque numérateur et dénominateur ne sont pas premiers entre eux. Il y a donc deux façons radicales pour traiter ce problème :

1° Réduire systématiquement tout rationnel via un calcul de pgcd

2° Ne jamais réduire un rationnel : les comparaisons (égalité, infériorité...) sont alors traitées en comparant la différence des deux opérandes avec zéro.

Chacune de ces méthodes a ses avantages et ses inconvénients : tout réside bien sûr dans le problème à traiter -et les calculs à effectuer-.
Pourtant, la façon d'effectuer les opérations arithmétiques dépend intrinsèquement de la représentation adoptée (cf §II.).
Il est donc essentiel de définir un format qui englobe les formes réduites et non réduites. Les opérations sur les rationnels dépendront bien sûr des formats des opérandes ou/et du résultat (c'est à dire réduits ou non).

Remarque
De façon à simplifier l'utilisation, le choix d'une forme par défaut pour tous les calculs rationnels peut être spécifiée.

I.2. Représentation interne

Un rationnel est considéré comme un bloc, formé de la *juxtaposition* de deux entiers : le numérateur et le dénominateur. Le numérateur a le signe du rationnel, le dénominateur est lui toujours strictement positif.

La représentation compacte sous forme de bloc est motivée par :

1° L'encombrement mémoire est minimisé. En outre, l'allocation et la libération sont simples; il est à noter que la représentation sous format de pointeurs vers deux entiers nécessite trois allocations ou libérations, donc un morcellement plus grand de la mémoire.

2° La communication d'un rationnel est très simple, et de coût minimum : c'est un bloc.

3° Le fait de retrouver le format d'entier pour le numérateur et le dénominateur permet de construire l'arithmétique à moindre coût, en utilisant l'arithmétique dans **Z**.

4° Le calcul du signe est effectué à coût constant.

5° Le fait qu'un rationnel soit réduit ou non est connu par consultation du libellé. Cela évite de réduire un rationnel déjà réduit, et nous avons vu que cela influe sur la façon d'effectuer les opérations arithmétiques. Le mélange des formes réduites et non réduites permet d'adapter le mieux possible l'arithmétique au problème considéré.

Implantation

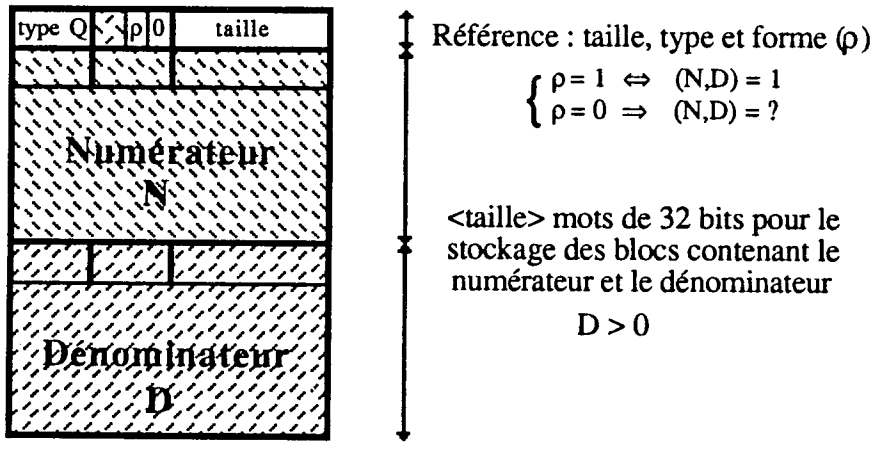


Fig. VI.1 Représentation du rationnel $\frac{N}{D}$

II. Arithmétique rationnelle surchargée

II.1. Généralités

Ce travail a été réalisé à partir de remarques données par Knuth [KNU81f]: elles sont à la base des algorithmes manipulant des rationnels réduits. Nous étudions les complexités de ces algorithmes et évaluons les complexités des opérations en format non réduit.

Nous avons vu que les algorithmes permettant d'effectuer les opérations arithmétiques dépendent de la forme - *réduite ou non* - des opérandes.

Remarque :
Il est à noter que lorsque une opération a pour opérandes un rationnel et un entier, l'algorithme utilisé est le même que si l'on considérait l'entier comme un rationnel réduit, dont le dénominateur est un. Seuls quelques calculs (pgcd par exemple) sont simplifiés, et c'est pourquoi ils ont été implantés avec quelques optimisations.

Dans la suite, nous considérons deux rationnels : $r_1 = \frac{p_1}{q_1}$ et $r_2 = \frac{p_2}{q_2}$.

On pose : $\begin{cases} L_{p_1} = \lfloor Log_\beta\ p_1 \rfloor + 1 \\ L_{q_1} = \lfloor Log_\beta\ q_1 \rfloor + 1 \end{cases}$ et $\begin{cases} L_{p_2} = \lfloor Log_\beta\ p_2 \rfloor + 1 \\ L_{q_2} = \lfloor Log_\beta\ q_2 \rfloor + 1 \end{cases}$

$A_Q^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2})$ (resp. $A_Q(L_{p_1},L_{q_1},L_{p_2},L_{q_2})$) désigne la complexité de l'addition (ou la soustraction) de r_1 et r_2 lorsqu'ils sont réduits (resp. non réduits).

$M_Q^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2})$ (resp. $M_Q(L_{p_1},L_{q_1},L_{p_2},L_{q_2})$) désigne la complexité de la multiplication de r_1 et r_2 lorsqu'ils sont réduits (resp. non réduits).

N.B. : On négligera dans la suite :

- le coût de l'addition de deux entiers devant celui de leur multiplication et de leur pgcd.
- le coût des réductions (divisions) de deux entiers par leur pgcd devant le coût du calcul de ce pgcd.

II.2. Addition et Soustraction

La soustraction étant un cas particulier de l'addition, nous nous intéressons au calcul d'une représentation de $r = r_1 + r_2$.

II.2.1. Forme réduite

r_1 et r_2 sont réduits. On a donc : $(p_1,q_1) = 1$ et $(p_2,q_2) = 1$

Deux algorithmes sont envisageables :

($\mathcal{A}_1$) Calcul puis réduction :

On calcule d'abord une forme R non réduite de r. Soient $P = (p_1.q_2 + p_2.q_1)$ et $Q = (q_1.q_2)$. $R = \frac{P}{Q}$ est une représentation de r.

On réduit R via un calcul de pgcd. Soit $\delta = (p_1.q_2 + p_2.q_1, q_1.q_2)$.

On obtient : $\frac{(P/\delta)}{(Q/\delta)}$ est une représentation réduite de r.

Le coût A_1^r de cet algorithme est :

$$A_1^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2}) = M_\beta(L_{p_1}, L_{q_2}) + M_\beta(L_{p_2}, L_{q_1}) + M_\beta(L_{q_1}, L_{q_2}) + Gcd_\beta(Sup(L_{p_1}+L_{q_2},L_{p_2}+L_{q_1}),L_{q_1}+L_{q_2})$$

$$A_1^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2}) = O\left(Gcd_\beta(Sup(L_{p_1}+L_{q_2},L_{p_2}+L_{q_1}),L_{q_1}+L_{q_2})\right)$$

($\mathcal{A}_2$) Pré-réduction puis calcul :

Comme $(p_1,q_1) = 1$ et $(p_2,q_2) = 1$, la réduction à effectuer sur R provient en grande partie du pgcd $\delta = (q_1,q_2)$.

Proposition

Soit $\delta = (q_1,q_2)$, *et soient* $\begin{cases} q_1' = q_1/\delta \\ q_2' = q_2/\delta \end{cases}$. *Soit* $d = (p_1.q_2' + p_2.q_1', \delta)$.

Alors : $\dfrac{(p_1.q_2'+p_2.q_1')/d}{q_1'.q_2'.(\delta/d)}$ *est une forme réduite de r.*

Preuve :

Lemme : *Si* $(q_1,q_2)=1$, *alors* : $\dfrac{p_1.q_2 + p_2.q_1}{q_1.q_2}$ *est une forme réduite de r.*

Preuve :
Soit $d = (p_1.q_2 + p_2.q_1, q_1.q_2)$. Si $d \neq 1$, alors il existe un facteur $c \neq 1$ de d qui divise q_1 ou q_2.
Supposons que c divise q_1. Comme $(p_1,q_1) = 1$, on en déduit que c ne divise pas p_1. Or c divisant d, c divise $(p_1.q_2 + p_2.q_1)$.
On en déduit que c divise q_2, ce qui est absurde puisque $(q_1,q_2) = 1$.
Donc $d = 1$, et le lemme est démontré.

Supposons maintenant que $\delta \neq 1$, et posons $q_1' = q_1/\delta$ et $q_2' = q_2/\delta$.
On a alors : $R = \dfrac{p_1.q_2' + p_2.q_1'}{q_1'.q_2'.\delta}$ est une forme non réduite de r.
D'après le lemme précédent, on a : $(p_1.q_2' + p_2.q_1', q_1'.q_2') = 1$.
Posons $d = (p_1.q_2' + p_2.q_1', \delta)$. On en déduit que :

$\dfrac{(p_1.q_2' + p_2.q_1')/d}{q_1'.q_2'.(\delta/d)}$ est une forme réduite de r. cqfd

Soit A_2^r le coût de l'algorithme défini à partir de la proposition.
Soit L_δ le nombre de β-chiffres de δ.

On a :

$$A^r_{2(L_{p_1},L_{q_1},L_{p_2},L_{q_2})} = Gcd_\beta(L_{q_1},L_{q_2}) + Gcd_\beta(Max(L_{p_2}+L_{q_1}-L_\delta,L_{p_1}+L_{q_2}-L_\delta),L_\delta) + M_\beta(L_{p_2},L_{q_1}-L_\delta) + M_\beta(L_{p_1},L_{q_2}-L_\delta) + M_\beta(L_{q_1}-L_\delta,L_{q_2}-L_\delta) + M_\beta(L_{q_1}+L_{q_2}-2.L_\delta,L_\delta)$$

On en déduit donc que l'on a toujours :

$$\boxed{A^r_{2(L_{p_1},L_{q_1},L_{p_2},L_{q_2})} \leq A^r_{1(L_{p_1},L_{q_1},L_{p_2},L_{q_2})}}$$

Conclusion :

Pour la forme réduite, ($\mathcal{A}_2$) a une complexité plus faible que ($\mathcal{A}_1$)

II.2.2. Forme non réduite

La multiplication étant de complexité négligeable devant le pgcd, on en déduit que l'algorithme le plus efficace pour l'addition de rationnels en forme non réduite est ($\mathcal{A}_1$), sans réduction finale. Nous désignerons par ($\mathcal{A}_3$) cet algorithme.

Autrement dit : $\frac{p_1.q_2 + p_2.q_1}{q_1.q_2}$ est la représentation la moins coûteuse de r.

Le coût de ce calcul est :

$$(A^r_{3(L_{p_1},L_{q_1},L_{p_2},L_{q_2})} = M_\beta(L_{p_1}, L_{q_2}) + M_\beta(L_{p_2}, L_{q_1}) + M_\beta(L_{q_1}, L_{q_2})$$

Conclusion :

Pour la forme non réduite, ($\mathcal{A}_3$) a une complexité plus faible que ($\mathcal{A}_2$)

II.3. Multiplication et Division

Comme la division est un cas particulier de la multiplication, nous nous intéressons au calcul d'une représentation de $r = r_1.r_2$.

II.3.1. Forme réduite

r_1 et r_2 sont réduits. On a donc : $(p_1,q_1) = 1$ et $(p_2,q_2) = 1$

On cherche deux entiers p et q premiers entre eux tels que : $\frac{p}{q} = \frac{p_1}{q_1}.\frac{p_2}{q_2}$

Deux algorithmes sont envisageables :

($\mathcal{A}_1$) Calcul puis réduction :

On calcule d'abord une forme R non réduite de r. Soient $P=p_1.p_2$ et $Q=q_1.q_2$.

$R = \frac{P}{Q}$ est une représentation de r. On réduit R via un calcul de pgcd.

Soit $\delta = (P,Q)$. $\frac{(P/\delta)}{(Q/\delta)}$ est alors une représentation réduite de r.

Le coût M_1^r de cet algorithme est :

$$M_1^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2}) = M_\beta(L_{p_1},L_{p_2})+M_\beta(L_{q_1},L_{q_2})+Gcd_\beta(L_{p_1}+L_{p_2},L_{q_1}+L_{q_2})$$

D'où : $\boxed{M_1^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2}) = O(\,Gcd_\beta(\,L_{p_1}+L_{p_2},\,L_{q_1}+L_{q_2}))}$

($\mathcal{A}_2$) Pré-réduction puis calcul :

Comme $(p_1,q_1) = 1$ et $(p_2,q_2) = 1$, la réduction à effectuer sur R provient uniquement des pgcd croisés : $\delta_1=(p_1,q_2)$ et $\delta_2=(p_2,q_1)$.
Un deuxième algorithme ($\mathcal{A}_2$) consiste donc à calculer d'abord ces pgcd.

On obtient alors : $\begin{cases} p = (p_1/\delta_1).(p_2/\delta_2) \\ q = (q_1/\delta_1).(q_2/\delta_2) \end{cases}$

Le coût M_2^r de cet algorithme est borné par :

$$M_2^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2}) \le M_\beta(L_{p_1},L_{p_2}) + M_\beta(L_{q_1},L_{q_2}) + Gcd_\beta(L_{p_1},L_{p_2}) + Gcd_\beta(L_{q_1},L_{q_2})$$

D'où : $\boxed{M_2^r(L_{p_1},L_{q_1},L_{p_2},L_{q_2}) = O(Gcd_\beta(L_{p_1},L_{p_2}) + Gcd_\beta(L_{q_1},L_{q_2}))}$

La fonction Gcd_β étant super-linéaire, on en déduit que :

Conclusion :

Pour la forme réduite, ($\mathcal{A}_2$) a une complexité plus faible que ($\mathcal{A}_1$)

II.3.2. Forme non réduite

On ne connaît pas (p_1,q_1) et (p_2,q_2).

On cherche deux entiers p et q tels que : $\frac{p}{q} = \frac{p_1}{q_1}.\frac{p_2}{q_2}$

Alors : $\begin{cases} p = p_1.p_2 \\ q = q_1.q_2 \end{cases}$ permet de calculer des valeurs pour p et q.

N.B. : ($\mathcal{A}_3$) correspond à l'algorithme ($\mathcal{A}_1$) avec un résultat non réduit.

Le coût de cet algorithme est :

$$\boxed{T_3 = M_\beta(\, L_{p_1}, L_{p_2}) + M_\beta(\, L_{q_1}, L_{q_2})}$$

M_β étant négligeable devant Gcd_β ($M_\beta = o(Gcd_\beta)$), on en déduit que :

Conclusion :

Pour la forme non réduite, ($\mathcal{A}_3$) a un coût plus faible que ($\mathcal{A}_2$)

II.3.3. Forme mixte

De manière générale, aussi bien pour l'addition que pour la multiplication, deux cas sont à distinguer :

1° On désire un résultat réduit : l'étude précédente montre que le meilleur algorithme consiste à effectuer une pré-réduction sur les opérandes avant de calculer la forme finale du résultat - type ($\mathcal{A}_2$)-.
Il est donc intéressant de réduire de toute façon les opérandes pour calculer leur somme ou leur produit, plutôt que de réduire leur résultat. Dans ce cas, on gardera la forme réduite pour les opérandes.

2° On désire un résultat non réduit : le meilleur algorithme consiste à effectuer un calcul direct du résultat, sans pré-réduction des opérandes - type ($\mathcal{A}_3$)-.

Conclusion générale

L'algorithme de calcul ne dépend que de la forme cherchée pour le résultat :

- *si le résultat doit être réduit, le meilleur algorithme est de type ($\mathcal{A}_2$) (Pré-réduction, puis calcul)*
- *si le résultat ne doit pas être réduit, le meilleur algorithme est de type ($\mathcal{A}_3$) (Calcul direct, sans pré-réduction)*

Remarque importante concernant l'implantation des rationnels :

Deux types d'algorithmes pour les opérations sont implantés, selon qu'un résultat réduit ou non est demandé. Les opérations à résultat non réduit sont effectuées directement.
Pour les opérations à résultat réduit, les opérandes sont d'abord réduits : un rationnel non-réduit occupant plus de place que sa représentation réduite, cette opération est faite en place : la nouvelle représentation réduite remplace donc l'ancienne représentation non réduite. Cette technique est très efficace puisqu'elle évite de réduire plusieurs fois un même rationnel, lors de différentes opérations.

III. Extension au calcul dans $\mathbb{Q} \cup \{ -\infty, +\infty \}$

Il est possible, à moindres frais, d'étendre le calcul dans $\mathbb{Q}$ au calcul dans $\mathbb{Q} \cup \{ -\infty, +\infty \}$.

Il suffit pour cela d'adopter la convention de représentation :

$$\begin{cases} \frac{1}{0} \text{ représente } +\infty \\ \frac{-1}{0} \text{ représente } -\infty \\ \frac{0}{0} \text{ est indéfini} \end{cases}$$

Les opérations arithmétiques précédentes permettent alors d'obtenir les tables des valeurs suivantes :

*	0	$\in Q^-$	$\in Q^+$	$-\infty$	$+\infty$
0	0	0	0	indéfini	indéfini
$\in Q^-$	0	$\in Q^+$	$\in Q^-$	$+\infty$	$-\infty$
$\in Q^+$	0	$\in Q^-$	$\in Q^+$	$-\infty$	$+\infty$
$-\infty$	indéfini	$+\infty$	$-\infty$	$+\infty$	$-\infty$
$+\infty$	indéfini	$-\infty$	$+\infty$	$-\infty$	$+\infty$

+	0	$\in Q^-$	$\in Q^+$	$-\infty$	$+\infty$
0	0	$\in Q^-$	$\in Q^+$	$-\infty$	$+\infty$
$\in Q^-$	$\in Q^-$	$\in Q^-$	$\in Q$	$-\infty$	$+\infty$
$\in Q^+$	$\in Q^+$	$\in Q$	$\in Q^+$	$-\infty$	$+\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	indéfini
$+\infty$	$+\infty$	$+\infty$	$+\infty$	indéfini	$+\infty$

Fig. VI.3 Règles de calcul dans $\mathbb{Q} \cup \{ -\infty, +\infty \}$

Chapitre IX

VECTORISATION DE L'ARITHMETIQUE ENTIERE

I. Définition du modèle vectoriel

Dans toute cette partie, on appelle *unité vectorielle* une unité composée de :

- une unité de contrôle (capable de décoder une instruction par cycle)

- une unité arithmétique, comprenant un additionneur et un multiplieur virgule flottante. Ces deux unités sont pipelinées et ont respectivement e^+ et e^* étages.

- un ensemble de registres vectoriels, permettant un accès direct sur chacun des pipelines

- deux registres scalaires, associés à l'additionneur et au multiplieur. Ils sont directement accessibles depuis la mémoire.

- une mémoire, partagée en deux bancs A et B, chacun des bancs ayant un accès direct dans les registres.

On suppose que l'unité vectorielle fonctionne selon le schéma :

1° Chargement de deux registres opérandes, chargement des registres scalaires, chargement des codes opérations
2° Calcul, sur une taille de vecteur donnée
3° Récupération du résultat dans l'un des registres

Nous ne tenons pas compte dans la suite des temps de chargement/déchargement des registres dans la mémoire, mais uniquement du coût arithmétique des opérations étudiées. Ce temps pouvant être non négligeable selon son rapport avec le temps de cycle de l'unité vectorielle, nous étudierons son influence dans les expériences pratiques.

N.B. : Ce modèle correspond à l'unité vectorielle WEITEK sur laquelle ont été effectués les tests [TOU89] [VIL89] [FPS88].
La taille des registres opérandes est supposée limitée à T_r. Pour l'unité vectorielle WEITEK, $T_r = 128$.
Le format flottant de l'unité est le format IEEE-64 bits, que nous noterons IEEE64.
Le nombre d'étages de l'additionneur et du multiplieur sont : $e^+ = 5$ et $e^* = 6$.

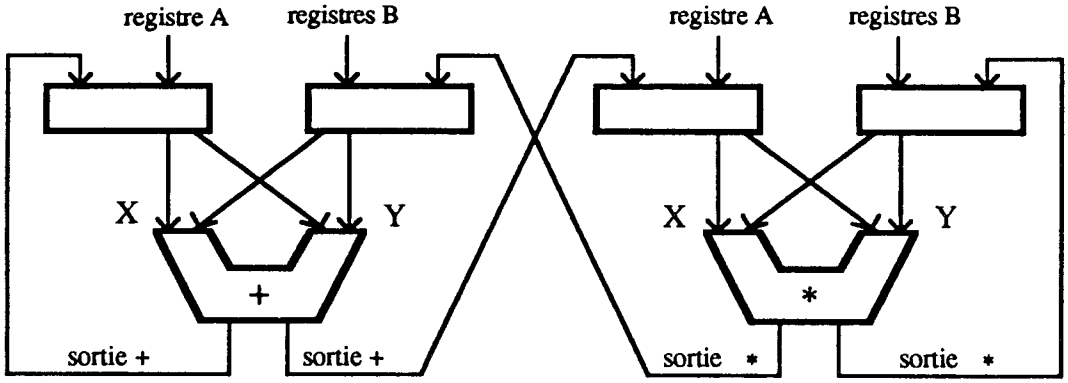


Fig VII.1 Unité vectorielle modèle

II. Représentation à retenue conservée

II.1. Nécessité d'une représentation adaptée

L'unité vectorielle admettant comme opérandes des flottants, l'arithmétique entière dans U n'est pas accessible directement. Il faut donc éviter le calcul des retenues.

Par ailleurs, la propagation des retenues dans les opérations d'addition et de multiplication pose un problème de dépendance arrière,difficile à éliminer sans trop alourdir le calcul.

Prenons pour exemple l'addition de deux entiers.

Soient $A = \sum_{i=0}^{n} a_i.\beta^i$ et $B = \sum_{i=0}^{n} b_i.\beta^i$ deux entiers de même taille.

Leur somme $R = A+B = \sum_{i=0}^{n+1} r_i.\beta^i$ est définie par la récurrence :

$$\begin{cases} t_i = a_i + b_i \\ c_{-1} = 0 \\ c_i = I\{(t_i + c_{i-1}) > \beta\} \quad i=0\ldots n \\ r_i = t_i + c_{i-1} - c_i.\beta \\ r_{n+1} = c_n \end{cases}$$

Le calcul de c_i pose donc un problème de dépendance arrière. Cette dépendance étant relativement simple, il est possible de l'éliminer en la déroulant sur un certain nombre de termes k, et en choisissant k en fonction du nombre d'étages de l'unité vectorielle : il suffit d'écrire c_i non plus en fonction de c_{i-1}, mais en fonction de c_{i-k} [KOG81]. Cette méthode a cependant l'inconvénient de compliquer le calcul -élémentaire- de c_i.

En pratique, la probabilité qu'une retenue entrante entraîne une retenue sortante est excessivement faible lorsque β est grand. La technique utilisée est donc de boucler tant que le vecteur de retenues n'est pas nul [MEL88].

Le problème de la propagation de retenue est lié au choix de la représentation. Il est alors essentiel de choisir une représentation qui autorise le dépassement de base, sans propagation de retenues obligatoire. Pour cela, nous nous inspirons d'une technique utilisée en arithmétique machine [MUL89] pour effectuer des opérations en temps constant, et que nous désignerons par le terme *notation à retenue conservée*. Cette notation permet ici une vectorisation efficace de l'addition.

Définition

Soit Δ et M deux entiers. La séquence $\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M}$, avec $-M \leq u_i \leq M$, représente en notation à retenue conservée en base Δ l'entier :

$$U = \sum_{i=0}^{n} u_i.\Delta^i$$

Notation : nous appellerons Δ_M-entier un entier dans cette représentation

Nous désignerons par *recalage* l'opération qui consiste à passer de la représentation à retenue conservée en base Δ à la représentation standard en base Δ :

$$\text{recalage} : U=\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M} \mapsto V=[v_p, v_{p-1},\ldots,v_0]_\Delta \,/\, U=V$$

Cette opération est très importante : s'effectuant en temps linéaire, elle permet un passage facile d'une représentation à l'autre.

Remarque :

Le passage d'un Δ-entier u au Δ_M-entier U qui lui correspond est réalisé de la façon suivante :

Si u est positif : $u= [u_n, u_{n-1},\ldots,v_0]_\Delta$ est converti en $U=\{u_n,u_{n1},\ldots,u_0\}_{\Delta,M}$

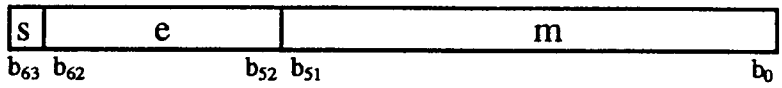
Si u est négatif : $u= -[u_n, u_{n-1},\ldots,v_0]_\Delta$ est converti en $U=\{-u_n,-u_{n1},\ldots,-u_0\}_{\Delta,M}$

Les coefficients de la représentation pouvant être négatifs, l'addition de nombres de signes opposés est une vraie addition. La comparaison, nécessaire préalable pour la décision de l'opération en représentation standard, est ainsi évitée.

Il est cependant clair que la comparaison entre deux Δ_M-entiers nécessite - dans la plupart des cas, et sauf choix spécifique de $M \leq \Delta-1$ - un recalage.

II.2. Le format flottant IEEE 64 bits

Soit x l'entier représenté en format flottant IEEE-64 bits ($b_0 \ldots b_{63}$) par [IEE82] :



s : bit de signe (s=1 $\Leftrightarrow$ nombre négatif)
e : exposant, sur 11 bits, en représentation signée
m : mantisse sur 52 bits, en représentation non signée

On a alors : $x = (-1)^s.(2^{52} + M).2^{e-1075}$

avec :
$$\begin{cases} s = b_{63} \\ e = \sum_{i=0}^{10} b_{52+i}.2^i \\ M = \sum_{i=0}^{51} b_i.2^i \end{cases}$$

II.3. Choix de Δ et de M

De façon à tirer au maximun parti de la représentation flottante -supposée être en format binaire-, M est choisi comme étant *le plus grand entier impair* codé exactement en format flottant.
Dans le format IEEE64, on choisit donc : $M = 2^{53}-1$

Le choix de Δ est lié aux contraintes suivantes :

1°/ pour pouvoir effectuer des multiplications exactes, sans propagation de retenue au premier ordre, il est nécessaire d'avoir:

$$(\Delta-1)^2 \leq M$$

2°/ la multiplication entraîne le cumul d'un grand nombre d'additions.
T_r étant la taille limite des vecteurs opérandes, il est nécessaire de pouvoir effectuer, sans dépassement de M, T_r -1 additions de produits d'entiers bornés par Δ.

On a donc : $(T_r-1).(\Delta-1)^2 \leq M$, d'où : $\boxed{\Delta \leq 1 + \sqrt{\frac{M}{T_r - 1}}}$

Dans le cadre du modèle, on a donc : $\Delta \leq 8\,421\,570$
On en déduit que toute valeur de Δ vérifiant $\Delta < 2^{23}$ convient.

3°/ il est essentiel de pouvoir effectuer des recalages faciles : β étant une puissance de 2, si Δ est une puissance de 2, le passage d'un β-entier à un Δ_M-entier peut être effectué à l'aide de décalages très simples.
Pour le modèle considéré, nous avons $\beta = 2^{32}$: $\Delta = 2^{16}$ permet donc des conversions peu coûteuses.

Finalement, on a : $\boxed{\begin{cases}\Delta = 2^{16} \\ M = 2^{53}-1\end{cases}}$

II.4. Dépassement et sécurité : validité de la représentation

Le problème de cette représentation est de déterminer l'opération pour laquelle l'un des coefficients du résultat risque de devenir, en valeur absolue, supérieur à la borne M. Cette détermination doit être peu coûteuse, pour être négligée devant le coût d'opérations aussi élémentaires que l'addition, mais assez fine pour éviter des recalages trop fréquents, ...et surtout assez sûre pour ne pas risquer d'erreurs de dépassement !

Pour garantir que tous les coefficients restent bornés par M, on associe à tout Δ_M-entier $U=\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M}$ un indice de validité : $\|U\|_{\Delta,M}$, défini par :

1° Lors du passage de la représentation du β-entier $[v_{\lfloor \frac{n}{2} \rfloor},\ldots,v_0]_\beta$ à U, on définit :

$$\|U\|_{\Delta,M} = \operatorname*{Max}_{i=0\ldots n} |u_i|$$

Remarque : le calcul de $\|U\|_{\Delta,M}$ peut être simplifié en lui associant la valeur initiale Δ-1 : les bornes successives sont alors moins bonnes (donc les recalages plus fréquents), mais le coût de l'indice de validité est constant quelle que soit la taille de l'entier, donc nul.

2° Lors de l'addition U+V de deux Δ_M-entiers U et V, on a :

$$\|U+V\|_{\Delta,M} = \|U\|_{\Delta,M} + \|V\|_{\Delta,M}$$

3° Lors du calcul du produit U.V de $U=\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M}$ par $V=\{v_p, v_{p-1},\ldots,v_0\}_{\Delta,M}$, on a :

$$\|U.V\|_{\Delta,M} = \text{Inf}\,(n,p).\|U\|_{\Delta,M}.\|V\|_{\Delta,M}$$

Lors du passage de la représentation standard à la représentation à retenue conservée, le calcul de $\| \|_{\Delta,M}$ coûte -au plus- le calcul du maximum des coefficients : il peut donc être effectué en un passage dans l'additionneur.
Lors des additions et des multiplications ultérieures, le calcul de $\| \|_{\Delta,M}$ est effectué en temps constant, avant l'opération. Si la valeur de ce calcul est supérieure à M, alors il est nécessaire d'effectuer un recalage avant l'opération, pour garantir la validité des calculs.

$\| \|_{\Delta,M}$ permet donc de valider, *à moindre coût*, les calculs.

II.5. Représentation des entiers sous format vectoriel

Compte-tenu des remarques précédentes sur le choix de Δ et M, la représentation d'un entier sous format vectoriel est la suivante :

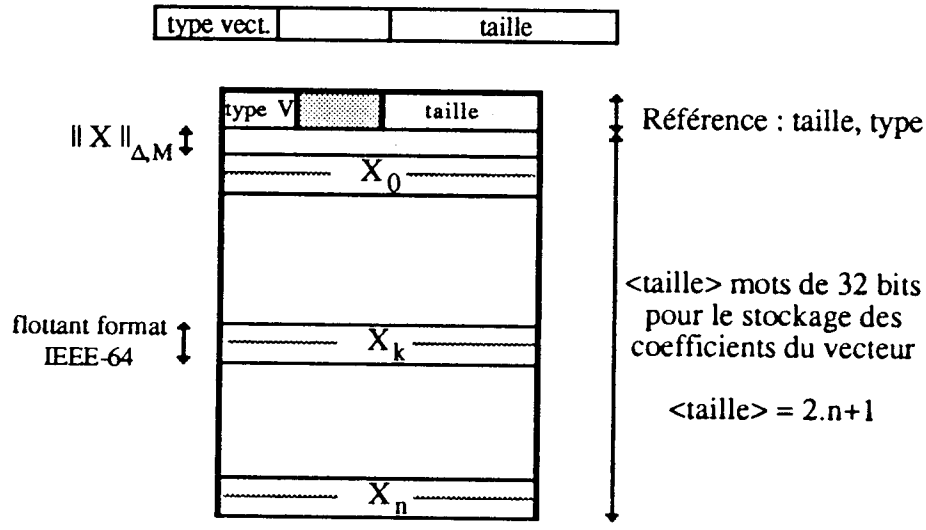


Fig. VII.2.5. Représentation vectorielle de $X = \sum_{i=0}^{n} x_i.\Delta^i$ avec $-M \le x_i.\Delta^i \le M$ i=0,...,n

III. Vectorisation des algorithmes standards

III.1. Implantation des recalages

III.1.1. Passage du format β-entier au format Δ_M-entier

Soit n un entier inférieur à 2^{16}, et soit x un flottant.
L'affectation à x de la valeur de n peut être effectuée en trois temps, à l'aide d'une seule opération arithmétique flottante :

1° mise à zéro de x
2° recopie de n dans les seize bits de poids faible de x (mantisse)

3° affectation de la constante hexadécimale 433_{16} dans les douze bits de poids fort de x (exposant qui indique que le bit sous-entendu du format correspond à 2^{53}). x vaut alors $n+2^{53}$.

4° soustraction de la constante flottante 2^{53} au flottant obtenu, pour recaler la mantisse

Le passage d'un β-entier à un Δ_M-entier peut facilement être réalisé vectoriellement à l'aide de la primitive ci-dessus.
Sans tenir compte des temps de déplacements mémoire - certes coûteux, mais inévitables ne serait-ce que pour amener les vecteurs opérandes dans des bancs différents -, la conversion d'un β-entier de n β-chiffres en Δ_M-entier de 2.n coefficients peut donc s'effectuer vectoriellement, en :

$$\boxed{(e^+ + 2.n - 1) \text{ temps de cycle}}$$

N.B. : e^+ est le nombre d'étages de l'additionneur pipeliné.

III.1.2. Passage du format Δ_M-entier au format β-entier

Le passage d'un entier inférieur à 2^{16} du format flottant au format entier peut être réalisé en utilisant l'algorithme inverse de celui présenté ci-dessus: la seule modification à apporter étant de soustraire vectoriellement -au lieu d'additionner- la constante flottante 2^{53}.

Le problème est donc de passer de la représentation en retenue conservée à la notation -signée- standard flottante.

En fait, pour rendre une opération en retenue conservée valide lorsqu'il y a un dépassement possible de M, il est suffisant de ramener les coefficients des opérandes dans $\{-\Delta+1,\ldots,\Delta-1\}$, sans qu'il soit nécessaire de les obtenir tous de même signe. Deux types de recalage sont donc distingués :

1° recalage : retenue conservée → retenue conservée :

Soit $U=\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M}$ tel que : $|u_i| < M$ i=0...n. On cherche V en représentation à retenue conservée $V=\{v_p, v_{p-1},\ldots,v_0\}_{\Delta,M}$ tel que :

$$\begin{cases} 0 \le v_i < \Delta \quad i=0\ldots p-1 \\ (0<v_p<\Delta) \text{ ou } (-\Delta<v_p<0) \\ \sum_{i=0}^{n} u_i.\Delta^i = \sum_{i=0}^{p} v_i.\Delta^i \end{cases}$$

L'algorithme de recalage (noté : recalage V→V) consiste en un parcours linéaire des coefficients de U, poids faible d'abord. Pour chaque coefficient, on additionne la retenue entrante. Le quotient euclidien par Δ du résultat obtenu est propagé comme nouvelle retenue. Le reste modulo Δ donne le nouveau coefficient de v.

p est donc borné par : $p \leq n + \lfloor \mathrm{Log}\,(\frac{M}{\Delta}) \rfloor$

2° recalage : retenue conservée → représentation standard flottante :

Soit $U=\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M}$ tel que : $|u_i| < M$ i=0...n. On cherche V en représentation à retenue conservée $W=\{w_p, w_{p-1},\ldots,w_0\}_{\Delta,M}$ tel que :

$$\begin{cases} (0 \leq w_i < \Delta \quad i=0\ldots p) \text{ ou } (-\Delta < w_i \leq 0 \quad i=0\ldots p) \\ \sum_{i=0}^{n} u_i.\Delta^i = \sum_{i=0}^{p} w_i.\Delta^i \end{cases}$$

L'algorithme de recalage (noté : recalage V→Z) s'effectue en deux étapes. Après un recalage de type V→V (1° ci-dessus), U est transformé en $V=\{v_p, v_{p-1},\ldots,v_0\}_{\Delta,M}$. A partir de V, le signe de U est connu : c'est le signe de v_p. Si v_p est positif, alors la représentation W obtenue est la représentation cherchée.
Si v_p est négatif, soit k le plus petit indice de v tel que $v_k \neq 0$. Si k=p, alors la représentation cherchée pour W est V. Sinon, la représentation de W est obtenue par soustraction vectorielle sur les (p-k+1) chiffres de poids fort de V : $W = \{v_p+1, v_{p-1}-(\Delta-1), v_{p-2}-(\Delta-1),\ldots,v_{k+1}-(\Delta-1), v_k-\Delta, \underbrace{0,\ldots\ldots,0}_{\text{(k-1) chiffres à 0}}\}$

III.2. Addition et Soustraction

Soient $U=\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M}$ et $V=\{v_p, v_{p-1},\ldots,v_0\}_{\Delta,M}$ deux Δ_M-entiers, dont on connaît les indices de validité $\|U\|_{\Delta,M}$ et $\|V\|_{\Delta,M}$.

Si $//U//_{\Delta,M} + //V//_{\Delta,M} \leq M$: alors l'addition de U et V peut être effectuée vectoriellement en :

$$\overrightarrow{T_{Add}} = (e^+ + Inf(n,p) + 1) \text{ temps de cycle}$$

Le résultat $W=\{w_{Sup(n,p)}, w_{Sup(n,p)-1},\ldots,w_0\}_{\Delta,M}$ est donné par :

$$\begin{cases} w_i = u_i + v_i \quad i=0\ldots Inf(n,p) \\ w_i = u_i \quad (i=p+1\ldots n) \quad \text{si } n>p \\ w_i = v_i \quad (i=n+1\ldots p) \quad \text{si } n<p \\ \|W\|_{\Delta,M} = \|U\|_{\Delta,M} + \|V\|_{\Delta,M} \end{cases}$$

Si $\|U\|_{\Delta,M} + \|V\|_{\Delta,M} > M$: alors il est nécessaire de recaler le plus grand de U et V : pour l'implantation, il vaut mieux éviter de recaler les deux vecteurs, sauf dans le cas où l'indice de validité du plus petit est de l'ordre de Δ.

La soustraction suit le même principe. Cependant, lorsque le vecteur opérande droit a une taille plus grande que le vecteur opérande gauche (p>n), le changement de signe sur les (p-n) chiffres de poids fort peut être très efficacement effectué par l'unité arithmétique scalaire (ou exclusif sur un bit). Il peut bien sûr être également effectué par l'unité vectorielle, lorsque cela est plus avantageux.

III.3. Multiplication

Soient $U=\{u_n, u_{n-1},\ldots,u_0\}_{\Delta,M}$ et $V=\{v_p, v_{p-1},\ldots,v_0\}_{\Delta,M}$ deux Δ_M-entiers, dont on connaît les indices de validité $\|U\|_{\Delta,M}$ et $\|V\|_{\Delta,M}$. On suppose ici -sans restrictions- que n>p.

Si $p.\|U\|_{\Delta,M}.\|V\|_{\Delta,M} \leq M$: alors la multiplication de U et V peut être effectuée vectoriellement en :

$$\overrightarrow{T_{Mul}} = n.(p+1) + (p+1).e^* + p.e^+ \text{ temps de cycle}$$

L'algorithme standard est en effet -avec la représentation à retenue conservée- très vectoriel. Nous notons ici R le registre résultat, et R_f le réel de poids le plus faible de R.

Algorithme standard vectorisé:

1° Pré-allocation du vecteur $W=\{w_{n+p}, w_{n+p-1},\ldots,w_0\}_{\Delta,M}$, et mise à zéro

2° Chargement de v_0 dans le registre scalaire du multiplieur,
Calcul de $R = v_0.U$
Enregistrement de R_f en w_0
Shift de R d'une position (un réel -ou 64 bits-)

3° Pour i =1...p :
Chargement de v_i dans le registre scalaire du multiplieur,
Calcul de $R = v_i.U + R$ (R registre)
Enregistrement de R_f en w_i
Shift de R d'une position (un réel -ou 64 bits-)

4° Déchargement de R dans $w_{n+p}, w_{n+p-1},\ldots,w_{p+1}$

Remarques :
Il est possible d'adapter facilement cet algorithme au calcul de U.V+T avec T de taille inférieure à n.
Pour éviter une explosion des coefficients dans le vecteur résultat, On recale le vecteur résultat immédiatement après le calcul de la multiplication, dans un vecteur $\{w_{n+p+1}, w_{n+p},\ldots,w_0\}_{\Delta,M}$: il suffit de pré-allouer un vecteur résultat de (n+p+1) chiffres dans l'algorithme précédent, pour ne pas entraîner une gestion mémoire supplémentaire.
Cet algorithme étant basé sur un Saxpy (opération du type U-s.V, où U et V sont des vecteurs et s un scalaire), il tire le meilleur parti de l'unité vectorielle.
Il est à noter cependant que l'algorithme vectorisé est l'algorithme standard: il ne sera donc pas efficace pour des entiers de grande taille. L'algorithme de Karatsuba mixte se prête cependant très mal à la vectorisation. L'algorithme de Schönhage-Strassen, basé sur la FFT, pourrait être un bon candidat...

III.4. Performances

Nous présentons ici les résultats obtenus pour les algorithmes précédents.

III.4.1. Recalages

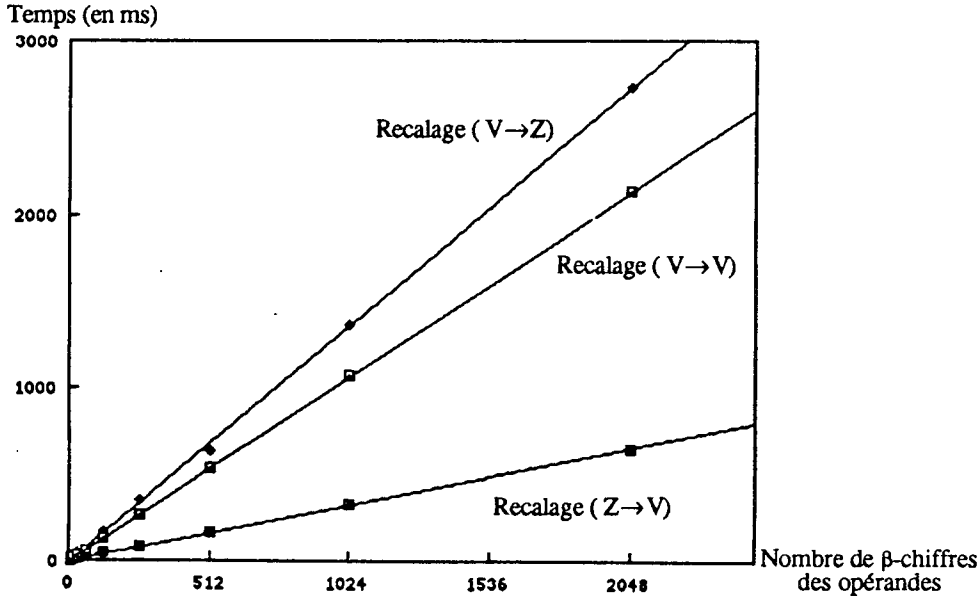


Fig 1.1 Recalages *(non vectorisés)*

RECALAGES $[\,]_\beta \rightarrow \{\,\}_{\Delta,M}$		
Opération	Coût expérimental	Qualité
$[\,]_\beta \rightarrow \{\,\}_{\Delta,M}$	0,31.n + 4,8	1,00
$\{\,\}_{\Delta,M} \rightarrow \{\,\}_{\Delta,M}$	1,04.n + 4,3	1,00
$\{\,\}_{\Delta,M} \rightarrow [\ldots]_\beta$	1,3.n - 2,1	1,00

Fig 1.2 Vérification du coût linéaire des recalages

Remarque :

Les recalages n'ont pas été vectorisés : cela explique en grande partie leur coût expérimental important, notamment pour le passage de la notation retenue conservée à la notation standard.
Néanmoins, étant linéaires comme le prouve expérimentalement le tableau ci-dessus, ils pourront être négligés lors d'opérations plus complexes comprenant des multiplications.

III.4.2. Addition

L'addition de nombres de petites tailles (moins de 400 chiffres décimaux) est étudiée séparément de celle de nombres plus grands (jusqu'à 20000 chiffres).

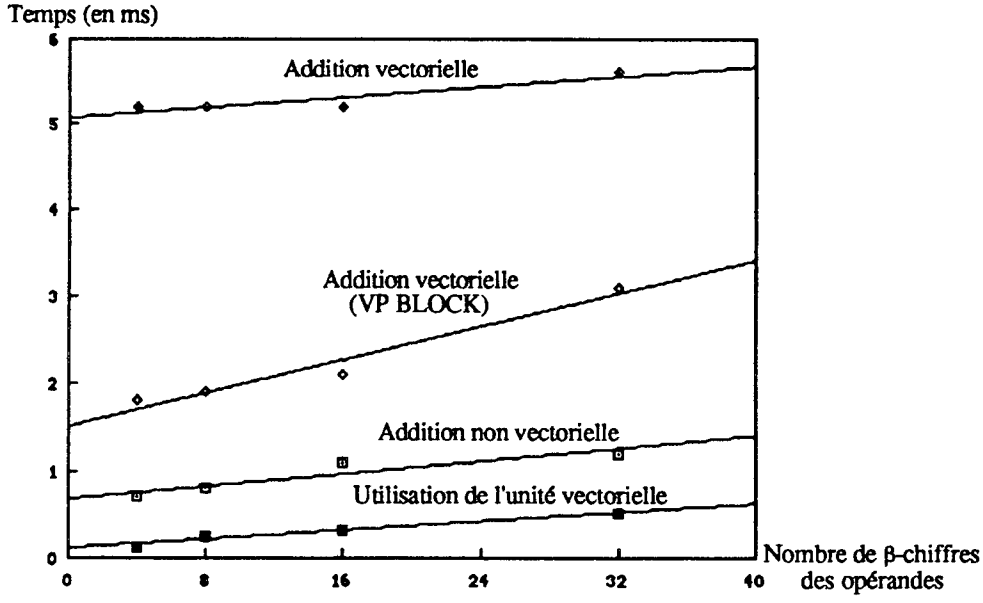


Fig 2.1 Addition d'entiers de moins de 400 chiffres (en base 10)

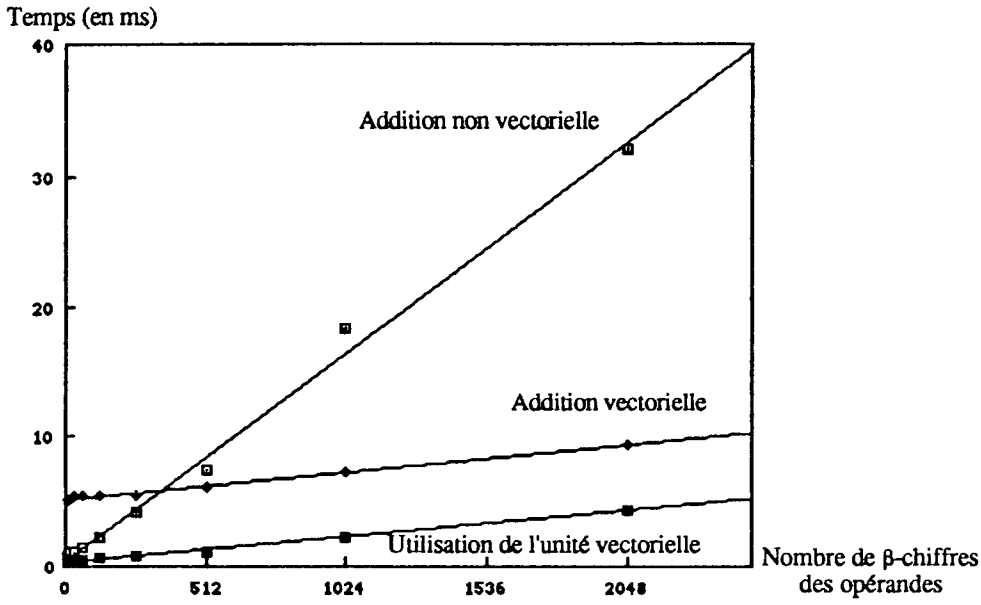


Fig 2.2 Addition d'entiers de moins de 20000 chiffres (en base 10)

ADDITION		
Opération	Coût expérimental	Qualité
Non vectorielle	0,016.n + 0,55	1,00
vectorielle	0,002.n + 5,3	0,99
Utilisation VPU	0,002.n + 0,31	1,00

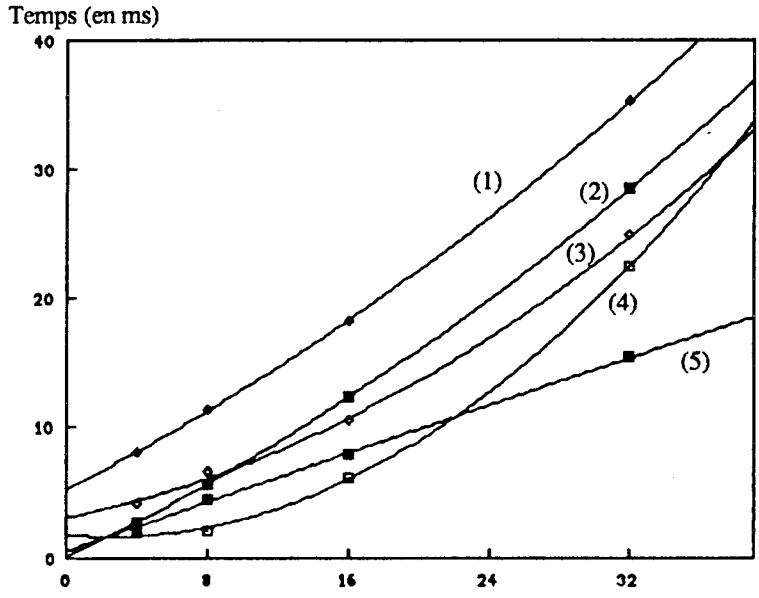
Fig 2.3 Coût expérimental de l'addition vectorielle

Remarques:

Le chargement des opérandes dans les registres est donc très important, par rapport au coût de l'opération. En outre, le stockage d'un nombre dans l'unité vectorielle est 4 fois plus coûteux, et deux fois plus d'opérations sont nécessaires ($\Delta = \sqrt{\beta}$). Il s'ensuit que pour les petites tailles, l'addition non vectorielle est beaucoup plus rapide, même si l'on programme finement l'unité vectorielle (utilisation de parameter block [FPS88]). Cependant, asymptotiquement, l'opération vectorielle est 8 fois plus rapide que l'opération standard : par conséquent, pour des entiers suffisament grands (plus de 4000 chiffres décimaux), la vectorisation semble très avantageuse. Il ne faut cependant pas oublier que c'est le mécanisme de propagation de retenues qui coûte le plus cher dans l'algorithme standard : ce problème est évité grâce à la représentation vectorielle adoptée, mais il est reporté dans les recalages.

III.4.3. Multiplication

Comme l'addition, la multiplication est étudiée dans deux cas : opérandes de petites tailles (moins de 400 chiffres décimaux), et opérandes plus grands (jusqu'à 20000 chiffres).



Légende : (1) Vectorielle (3) Utilisation VPU (4) Non vectorielle
(2) Vectorielle optimisée (vp block) (5) Utilisation VPU (vp block)

Fig 3.1 Multiplication d'entiers de moins de 400 chiffres (en base 10)

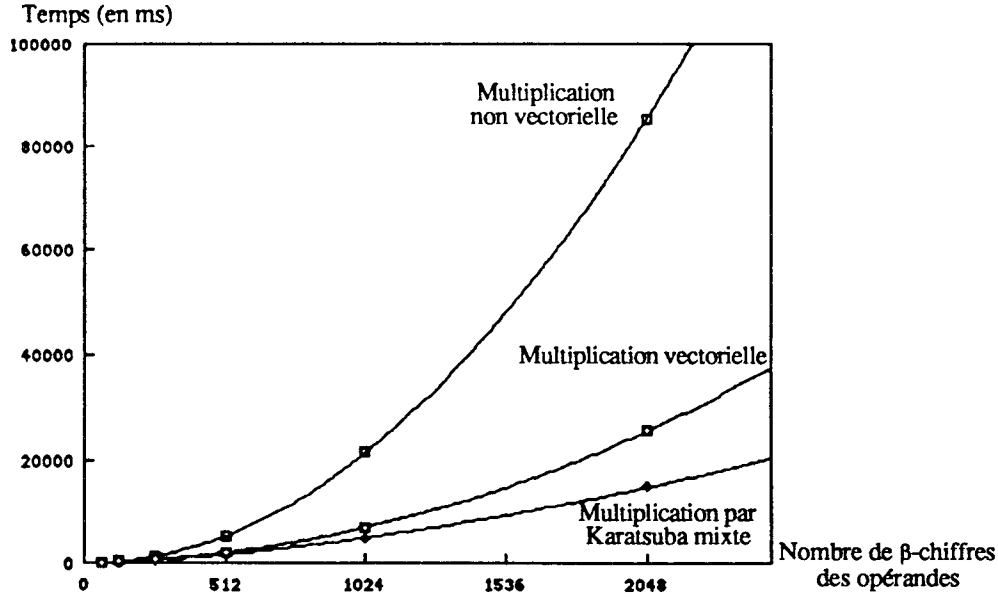


Fig 3.2 Multiplication d'entiers de moins de 20000 chiffres (en base 10)

MULTIPLICATION		
Opération	Coût expérimental	Qualité
Non vectorielle	$0{,}020.n^2 + 0{,}037.n + 0{,}55$	1,00
vectorielle	$0{,}0055.n^2 + 1{,}2.n + 5{,}2$	1,00
Karatsuba mixte	$0{,}090.n^{1{,}58}$	1,00

Fig 3.3 Coût expérimental de la multiplication vectorielle

Les coûts expérimentaux montrent que la vectorisation permet de diminuer d'un coefficient 4 le terme en n^2, pour des entiers suffisamment grands. Pourtant, le temps de démarrage de l'algorithme (déplacements des données, chargement des registres) est très important. C'est pourquoi la vectorisation ne peut être intéressante que pour des entiers grands; or, pour de tels entiers, l'algorithme de Karatsuba mixte (d'un ordre inférieur) est beaucoup plus rapide.

IV. Conclusion

Les résultats expérimentaux montrent clairement que la puissance de calcul fournie par l'unité vectorielle ne peut être exploitée de manière intéressante, de par le temps perdu lors du chargement des registres (environ 1,5 Mflops seulement pour la multiplication, contre 12 théoriques). Ce temps de gestion des registres est pourtant inévitable, puisque les opérandes d'une opération arithmétique peuvent être situés n'importe où en mémoire, éventuellement dans un même banc. La nécessité d'effectuer des opérations sur des entiers de taille quelconque ajoute un problème de plus dans les chargements et déchargements des registres (notamment pour la multiplication).

Quelle unité de calcul serait adaptée à la précision infinie ?

L'intérêt des primitives LONGADD et LONGMUL pour les algorithmes standards est fondamental. Deux types d'optimisation sont alors possibles :

1° La fonction LONGMUL peut recevoir en entrée deux vecteurs a[i] et b[i] : la retenue sortante est alors directement injectée en retenue entrante, et les valeurs a[i+1] et b[i+1] sont mises en entrée du multiplieur pendant le calcul de a[i].b[i]. Cette optimisation permet alors d'enchaîner cette opération, et diminue donc d'un facteur important les temps des algorithmes standards.

2° LONGMUL et LONGADD sont pipelinées et assemblées pour former une unité vectorielle "entière". Il est alors essentiel que les retenues générées aussi bien dans l'additionneur que dans le multiplieur avec les $i^{èmes}$ composantes de deux vecteurs soient réinjectées de manière interne pour le calcul sur les $(i+1)^{èmes}$ composantes.

Une autre méthode consiste à choisir une représentation interne adaptée qui permette de faire du pipeline au niveau des chiffres d'un nombre. C'est le cadre des opérateurs On-Line : le projet OCAPI [GUY89] s'inscrit dans ce contexte.

CONCLUSION

La parallélisation des problèmes tests choisis montre que les machines massivement parallèles constituent un outil essentiel dans le traitement des problèmes conséquents du Calcul Formel.
Encore faut-il que les structures manipulées soient adaptées, et qu'il soit possible d'utiliser ou de développer des algorithmes complexes, à un haut niveau d'expression du parallélisme. PAC s'inscrit dans ce contexte : fournir un outil autorisant l'utilisation de manière transparente de la puissance d'une machine parallèle depuis un système séquentiel classique de Calcul Formel, mais aussi, mettre à disposition un environnement suffisamment riche et performant nodalement, de façon à permettre le développement d'algorithmes parallèles d'ordre supérieur.

L'implantation de PAC sur d'autres modèles physiques est essentielle. D'une part cela permettra de l'évaluer sur d'autres architectures, et inversement d'évaluer les capacités d'une machine sur des problèmes caractéristiques du Calcul Formel. D'autre part, les outils de base -communication, extensions à d'autres domaines- pourront être affinés.

BIBLIOGRAPHIE

"Mon livre n'a rien de tout cela, parce que je n'ai que noter à la marge, ni que commenter à la fin, et moins encore sais quels auteurs j'ai suivis en icelui, afin de les mettre au commencement, comme ils font tous selon l'ordre de l'A,B,C, commençant par Aristote et achevant par Xénophon, Zoïle ou Zeuxis, encore que l'un fût un médisant et l'autre un peintre."

Cervantes, "L'ingénieux Hidalgo Don Quichotte de la Manche", Prologue

[AHU74a] The Design and Analysis of Computer Algorithms(p.252-276)
A.V.Aho, J.E. Hopcroft, J.D. Ullman
Addison-Wesley (1974)

[AHU74b] The Design and Analysis of Computer Algorithms(p.277-316)
A.V.Aho, J.E. Hopcroft, J.D. Ullman
Addison-Wesley (1974)

[AHU83] Data Structure & Algorithm (p. 378-407)
A.V.Aho, J.E. Hopcroft, J.D. Ullman
Addison-Wesley (1983)

[ALA87] A fast, low-space Algorithm for Multiplying dense multivariate polynomial
V.S. Alagar, D.K Probst
ACM Toms 13/1 (p. 35-37) (1987)

[ASS85] Structure & Interpretation of Computer Programs
H. Abelson, G. Jay Sussman, J. Sussman
Mc Graw-Hill Book Company pp 491-503 (1985)

[AVI61] Signed-digit number representation for fast parallel arithmetic
A. Avizienis
IRE Trans. Electronic Computers,10,pp.389-400 (1961)

[BAL89] L'arithmétique Polynômiale de PAC
J. Ballasi, D. Berthet, J.L. Roch
Rapport de recherche Imag-Cmap (à paraître) (1989)

[BKU83] Systolic VLSI Arrays for Linear-Time Gcd Computation
R.P. Brent, H.T. Kung
VLSI'83, Anceau&Aas Ed., Elsevier Science Publishers B.V.
North Holland, IFIP (1983)

[BOR82] Fast Parallel Matrix and Gcd Computations
A. Borodin, J. Von Zur Gathen, J.E. Hopcroft
Inf. Control, 52, pp. 241-256 (1982)

[CHP87] Le Langage Lustre et sa Sémantique Opérationnelle
P. Caspi, N. Halbwachs, D. Pillaud, J. Plaice
IIème colloque C^3 - Angoulème (1987)

[CMR88] Parallel Gaussian Elimination on an MIMD Computer
M. Cosnard, M. Marrakchi, Y. Robert, D. Trystram
Parallel Computing, North-Holland n°6 pp.275-296 (1988)

[COO66] On the Minimum Computation Time of Functions
S.A. Cook
Ph. D Thesis, Harvard Univ. (1966)

[COO69] On the Minimum Complexity of Functions
S.A. Cook, S.O. Aanderaa
Trans. Amer. Math. Soc., 142, pp 291-314 (1969)

[COO85] A Taxonomy of Problems with Fast Parallel Algorithms
S.A. Cook
Inf. Control, 64, pp 2-22 (1985)

[COL74] The average running-time of Euclid's algorithm
G.E. Collins
SIAM Journal of Computing, vol.3 pp 1-10 (1974)

[COL80] Aldes and SAC-2 Now Available
G.E. Collins
Sigsam Bulletin , 10,2 pp 14-15 (1980)

[COM77] Analysis of the Pope-Stein Division Algorithm
G.E. Collins, D.R. Musser
Information Processing Letters, vol.6 n°5, pp 151-155 (1977)

[DDD85] About a new method for computing in algebraic number fields
J. Della-Dora, C. Dicrescenzo, D. Duval
Eurocal 1985 - Linz (1985)

[DEL88] Machines Auto-Reproductibles : de l'intérêt du parallélisme dans la fabrication du café...
J. Della-Dora
Journal Libération - 16 Avril 88 (1988)

[DON86] Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine
J.J. Dongarra, F.G. Gustavson, A. Karp
SIAM review 26, pp. 91-112 (1984)

[DON87] The Linpack Benchmark : An Explanation
J.J Dongarra
Speedup, vol 1, n°1 - Nov. 87 (1987)

[DST87] Calcul Formel : Systèmes et Algorithmes de Manipulations Algébriques
J. Davenport, Y. Siret, E. Tournier
Masson- collection eri (1986)

[DUP46] Sur le nombre de divisions à effectuer pour obtenir le plus grand commun diviseur entre deux nombres entiers
A. Dupré
Journal de Mathématiques, Tome XI - Fév. 1846 (1846)

[DUP88] Laios : un Réseau Multi-Processeur Orienté Intelligence Artificielle
J. Duprat
Thèse INP Grenoble (1988)

[DUV88] Algebraic Extensions and Algebraic Closure in Scratchpad II
D. Duval
Lect. Notes in Comp. Sci.-Spring. Ver. (p. 440-446) (1988)

[EST87] Estelle : un Langage ISO pour les Algorithmes Distribués et les Protocoles
J.P. Courtiat, P. Dembisky, R. Groz, C. Jard
TSI vol.6 n° 2 - Avril 87 (1987)

[FEK89] Enriching Simultaion by Computer Algebra : Illustrative Examples and Parallelization Aspects
A.Fekken,D.Hilhorst,E.J.HKerchkoff,L.Dekken,H.Koppelaan
Third European Simulation Congress (1989)

[FIT89] Can Reduce Be Run in Parallel ?
J. Fitch
Proc. Issac 89-Portland (à paraître dans Lect.Notes) (1989)

[FLY66] Very High-speed Computing Systems
M.J. Flynn
Proc. IEEE, 54, 1901-1909 (1966)

[FPS88] Programming the FPS - T Series, Release C
Floating Point System
Portland Oregon 97223 (1988)

[FRA89] Contribution à l'Etude des Méthodes de Contrôle Automatique de l'Erreur d'Arrondi : la méthodologie SCALP
P. François
Thèse INP Grenoble (1989)

[GUY89] OCAPI : A Parallel Arithmetic Operator for Very Large Numbers
A. Guyot, Y. Kusumaputri, J.M. Muller
IFIP Workshop on Parallel Architecture on Silicon (1989)

[GUY90] Intégration de l'algorithme de Lehmer sur silicium
A. Guyot, R. Bouraoui, J.L. Roch
Rapp. Rech. Imag - Grenoble (à paraître) (1990)

[HAL88] Parallel Computing Using Multilisp
Robert H. Halstead, Jr
Parallel Computation and Computers for Artificial Intelligence
Kluwer Academic Publishers - S. Kowalik, pp 21-51 (1988)

[HER89] Sur l'implantation et le cablage de Schönhage-Strassen
Y. Herreros
IEEE Congrès 1989 - Los Angeles (1989)

[IEE85] IEEE Standard for Binary Floating-Point Arithmetic
ANSI / IEEE Std 754 - 1985
IEEE, New-York (1985)

[INT] 286/7 ou 386/7 : Manuels de référence
Intel - Silicone Valley

[JAU84] Le microprocesseur 68000 et sa programmation
Patrick Jaulent
Eyrolles (1984)

[JOH74] Sparse Polynomial Arithmetic
S.C. Johnson
Bell Telephone Labs (1974)

[KAL89a] Parallel Algebraic Computing Design
Erich Kaltofen
ISSAC 89-Lectures Notes for a Tutorial - Portland (1989)

[KAL89b] Communication personnelle - Juillet 89 (1989)

[KAO62] Multiplication of multidigit numbers on automata
A. Karatsuba, Y. Ofman
Dokl. Akad. Nauk. SSSR vol.145 (p. 293-294) (1962)

[KNU70] The Analysis of Algorithms
D.E. Knuth
Congrès Int. de Math. Nice,Tome 3, p.269-274 (1970)

[KNU81a] Seminumerical Algorithms
D.E. Knuth
Addison-Wesley (pp. 250-515) (1981)

[KNU81b] Seminumerical Algorithms
D.E. Knuth
Addison-Wesley (pp. 256-257) (1981)

[KNU81c] Seminumerical Algorithms
D.E. Knuth
Addison-Wesley (pp. 268-277) (1981)

[KNU81d] Seminumerical Algorithms
D.E. Knuth
Addison-Wesley (pp. 278-300) (1981)

[KNU81e] Seminumerical Algorithms
D.E. Knuth
Addison-Wesley (pp. 178-197) (1981)

[KNU81f] Seminumerical Algorithms : Analysis of Euclid's Algorithm
D.E. Knuth
Addison-Wesley (pp. 316-364) (1981)

[KOG81] The Architecture of Pipelined Computers
P.M. Kogge
Hemisphere Publishing Corporation (1981)

[LAM44] Notes sur la limite du nombre de divisions dans la recherche du plus grand diviseur entre deux nombres entiers
G. Lamé
Cptes. Rendus Acc. Sci. Tome XIX (pp. 867-870) (1844)

[LEH38] Euclid's Algorithm for Large Number
D.H. Lehmer
AMMM 45 , Avril1938, pp 227, 233 (1938)

[MAC78] History of Lisp
J. Mac Carthy
SIGPLAN Notices, 13 (8), pp 217-223 (1978)

[MEL88] Reduce installation guide for Cray 1/X-MP
Systems Running COS version 3.3
H. Melenk, W. Neun
CAP Proc., J.Della-Dora, J.Fitch ed., North-Holland (1988)

[MEU89] Résultats du Benchmark CEA sur le Cray Y-MP/832
G. Meurant
Rapport de recherche - CEA-N-2594 - Mars 1989 (1989)

[MIG89] Mathématiques pour le Calcul Formel
M. Mignotte
Presses Universitaires de France - Mathématiques (1989)

[MŒ70] Is a Linked List the Best Storage for an Algebra System
R.T. Mœnck
Dept. of Computer Science - Univ. Waterloo, Canada (1970)

[MŒ72] Practical Fast Polynomial Multiplication
R.T. Mœnck
Dept. of Computer Science - Univ. Waterloo, Canada (1972)

[MŒ73] Fast Computation of Gcds
R.T. Mœnck
Proc. 5th ACM Symp. Theory Comp. pp142-151 (1973)

[MUL89] Arithmétique des Ordinateurs
J.M. Muller
Masson - collection eri (1989)

[MUL88] VLSI Manipulation of Polynomials
J.M. Muller, F. Siebert-Roch
CAP Proc., J.Della-Dora, J.Fitch ed., North-Holland (1988)

[OCC88] Occam 2 Reference Manual
Inmos Limited
Prentice Hall / C.A.R. Hoare Series editor (1988)

[OZE87] Calcul Exact des Formes de Jordan et Frobenius d'une Matrice
P. Ozello
Thèse INP Grenoble (1987)

[PER89] Algorithme de Buchberger dans $\mathbb{Q}[X_1,...X_n]$
L. Perret, N. Revol
Rapport de Recherche Imag-Cmap (à paraître) (1989)

[PON88] Evaluation of Performance Enhancements in Algebraic Manipulation Systems
C.G. Ponder
Ph. D thesis, Berkeley (1988)

[POP60] Multiple Precision Arithmetic
D.A. Pope, M.L. Stein
Comm. of the ACM, vol. 3, pp. 652-654 (1960)

[REG88] Multiprecision Integer Division Examples Using Arbitrary Radix
E. Regener
ACM-TMS Vol.10 N°3, p. 325-328 (1984)

[ROC88] Towards A Computer Algebra System Based on Parallelism
J.L. Roch
CAP Proc., J.Della-Dora, J.Fitch ed., North-Holland (1988)

[ROC89] PAC sur FPS-T40 : Manuel d'utilisation
J.L. Roch
Rapport technique Imag-Cmap (à paraître) (1989)

[RSS86] PAC : Parallel Algebraic Computing
J.L. Roch, P. Sénéchaud, F. Siebert-Roch, G. Villard
Rap. Rech. Imag Grenoble, RR-686 I (1987)

[RSS88] Computer Algebra on MIMD Machine
J.L. Roch, P. Sénéchaud, F. Siebert-Roch, G. Villard
SIGSAM Bull., vol. 23, n° 1, Janv. 89 (1989)

[SCH71] Schnelle Berechnung von Kettenbruchentwicklungen
A. Schönhage
Acta Informatica vol.1, p. 139-144 (1971)

[SEN90] Calcul Formel et Parallélisme : Bases de Gröbner booléennes, Preuve de circuit et Parallélisme
P. Sénéchaud
Thèse INP Grenoble (à paraître) (1990)

[SIE89] Parallel Algorithms for Hermite Normal Form of Matrices
F. Siebert-Roch
Lectures Notes Springer-Verlag (1989)

[SIE90] Calcul Formel et Parallélisme : Formes Normales d'Hermite et Parallélisme
F. Siebert-Roch
Thèse INP Grenoble (à paraître) (1990)

[SIR87] Lisp Machine : technique cdr-coding
Y. Siret - Communication personnelle (1987)

[SIR88] Y. Siret, Communication personnelle (1988)

[SST71] Schnelle Multiplikation Grosser Zahlen
A. Schönhage, V. Strassen
Computing vol.7, p. 281-292 (1971)

[SSV87] Scratchpad II : Un nouveau langage de Calcul Formel
P. Sénéchaud, F.Siebert-Roch, G. Villard
Rapp. Rech Imag RR640-M (1987)

[STE67] Computing gcd without divisions
J. Stein
J. Comp. Phys. vol.1 , pp 397-405 (1967)

[TOU89] Algorithmique Parallèle pour Machines à Mémoire Distribuée
B. Tourancheau
Thèse INP Grenoble (1989)

[TRY89] D. Trystram (1989)
Communication personnelle à la première pause café du matin

[TRA85] T414 Transputer Reference Manual and Product Data
Inmos Limited
(Sep. 85) (1985)

[VIL89] Calcul Formel et Parallélisme : Résolution de Systèmes Linéaires
G. Villard
Thèse INP Grenoble (1988)

[WAT86] Bounded Parallelism in Computer Algebra
S.M. Watt
Ph.D Thesis, Waterloo - Ontario (1986)

[ZIM89] Multiplication Rapide en Le_Lisp
P. Zimmermann
Rapport de Recherche INRIA -Oct 89 (1989)

département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- Monsieur LAZARD Daniel
- Monsieur MULLER Jean-Michel

Monsieur ROCH Jean-Louis

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 "Mathématiques Appliquées"

Fait à Grenoble, le 16 Novembre 1989

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