Thèse

présentée par **Gilles VILLARD**

pour obtenir le titre de **DOCTEUR**

de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 5 juillet 1984)

Spécialité : **Mathématiques Appliquées**

Calcul Formel et Parallélisme

RESOLUTION DE SYSTEMES LINEAIRES

Thèse soutenue le 23 décembre 1988

Composition du jury,

Président : J.P. VERJUS

Examinateurs : M. COSNARD
J. DELLA DORA
P. LASCAUX
J. MORGENSTERN

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

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

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

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. PAYAN Jean Jacques

Année Universitaire 1987 - 1988

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

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
BONNIER Jane Marie	Chimie Générale
BOUCHEZ Robert	Physique Nucléaire ISN
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
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
FONTAINE Jean-Marc	Mathématiques Pures
GAGNAIRE Didier	Chimie Physique
GERMAIN Jean-Pierre	Mécanique,
GIRAUD Pierre	Géologie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures
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.
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
PIERRE Jean-Louis	Chimie Organique
RENARD Michel	Thermodynamique
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
VALENTIN Jacques	Physique Nucléaire I.S.N.
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ADIBA Michel	Mathématiques Pures
ANTOINE Pierre	Géologie
ARMAND Gilbert	Géographie
BARET Paul	Chimie
BLANCHI J.Pierre	STAPS
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
COURT Jean	Chimie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
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
GUITTON Jacques	Chimie

HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
JARDON Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKHOVE Claude	Géologie
LONGEQUEUE Nicole	Sciences Nucléaires I.S.N.
LUCAS Robert	Physique
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
NEMOZ Alain	Thermodymamique CNRS - CRTBT
OUDET Bruno	Mathématiques Appliquées
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Jean-Claude	Physique du Solide
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
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
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 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

BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHEHIKIAN Alain	EEA. IUT 1
CHENAVAS Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PEFFEN René	Métallurgie IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
VINCENDON Marc	Chimie 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.

- Beve aco picho Toino ! Qu'aco te fara de ben

Bois ça ! Cela te fera du bien

- Acabarié lou troun de pas diou emmé soun trin !

Il avalerait le tonerre de Dieu et sa suite !

- Be vaï ! Faou l'abitua ! Lou boun diou fague, s'es par viouré, qu'aguesse jamaï ren de maï marri à beourré qué dé jus dé limaço !

Sûr ! Il faut l'habituer ! Le bon Dieu fasse, s'il est pour vivre, qu'il n'ait jamais rien de plus mauvais à boire que du jus d'escargot !

Pierre Magnan

Vous savez, dans les écritures, il est écrit :
<< Au commencement était le verbe. >>
Non ! Au commencement était l'émotion.

Louis-Ferdinand Céline.

A mes parents

Je voudrais exprimer ma profonde reconnaissance à Monsieur J. DELLA DORA, Professeur à l'ENSIMAG, qui m'a guidé pendant ces deux années. Ses conseils, l'enthousiasme qu'il m'a communiqué, et toute la confiance qu'il a su me témoigner, m'ont permis de mener à bien ce travail.

J'adresse toute ma gratitude à Monsieur M. COSNARD, Professeur à l'ENS de Lyon, dont j'ai pu apprécier les compétences scientifiques. L'intérêt avec lequel il a suivi ce travail, et qu'il a, ce faisant, porté au Calcul Formel, a énormément contribué à l'accomplissement de cette thèse.

Je tiens à remercier Monsieur J.P. VERJUS, Professeur à l'INP Grenoble, pour tout l'honneur qu'il me fait en présidant le jury de cette thèse.

Je remercie également Monsieur J. MORGENSTERN, Professeur à l'Université de Nice, d'avoir accepté d'être rapporteur et de m'avoir conseillé.

Mes remerciements vont également à Monsieur P. LASCAUX, Professeur au CNAM, pour l'honneur qu'il me fait en participant à ce jury.

Je ne peux oublier Yves Robert et Bernard Tourancheau, pour le travail que nous avons accompli ensemble, pour leur gentillesse et leur humour sans égal. Merci à Claire Dicrescenzo pour ses précieux et patients conseils. A Jean–Michel Muller pour ses relectures attentionnées, ses omelettes et son amitié. A Jean–Louis Roch sans le travail de qui, partie du mien n'aurait pu être réalisée.

Ah ! Ces aventuriers. Combien m'a été douce la présence de Pascale Sénéchaud, et réconfortant son rire ! (*Rire : manifestation d'un sentiment de gaieté par un mouvement des lèvres et de la bouche, et accompagné de sons rapidement égrénés. Larousse 87*). Combien il m'a été agréable de passer ces jours de labeur en compagnie de Françoise Siebert et de Jean-Louis Roch.

Vos accents du Sud, Abdel Barkatou et Afonso Ferreira, tintent encore à mes oreilles en évoquant ces rivages baignés de soleil.

Merci aussi à tous les membres de l'équipe d'Algorithmique Parallèle et de Calcul Formel, pour leur bonne humeur et leurs innombrables gâteaux.

Je remercie enfin, le *PRC Mathématiques et Informatique*, le *Greco Calcul Formel du CNRS* et la *Société Floating Point Systems* pour leur soutien financier. Ainsi que le personnel du service de reprographie de l'IMAG pour la réalisation matérielle de ce travail.

Table des matières

• **DEUXIEME PARTIE**

SYSTEMES A COEFFICIENTS ENTIERS

INTRODUCTION

La résolution exacte de systèmes linéaires Ax=b à coefficients entiers, est un problème déjà ancien. Mais si de nombreux chercheurs se sont consacrés et se consacrent toujours à accélérer son traitement par le Calcul Numérique, ceux qui utilisent les outils du Calcul Formel doivent encore parfois, faire la preuve de l'utilité de leur travail : combien pourtant serait privilégié leur rôle, si l'obtention de la solution exacte était aussi peu coûteuse que celle d'une solution approchée !
Ne soyons pas trop optimistes (pour ne pas dire utopistes), la réalité est tout autre. Le calcul exact n'a par exemple rien à jalouser aux problèmes d'instabilité des calculs effectués en virgule flottante, il connait le phénomène "d'explosion des coefficients". Cette expression, d'emploi très courant en Calcul Formel, n'est–t'elle pas justifiée si l'on pense qu'il faut 250 fois plus de mémoire (cf partie 2, chap.IV) pour stocker le vecteur solution d'un système de dimension 700, sous sa forme rationnelle qu'au format flottant ? Mais on ne pourra occulter l'intérêt premier de la forme rationnelle : elle est exacte. Et essayons d'oublier, ne serait–ce que le temps de l'introduction, qu'il faut également 2000 fois plus de temps pour l'obtenir.

De tels chiffres suggèrent que l'utilisation d'une arithmétique en précision infinie, est inhérente aux méthodes exactes. Les techniques n'utilisant pas les congruences, telles que les éliminations sans fraction sur le modèle de Gauss, décrites par E.Bodevig en 1959 [Bod] et considérablement améliorées par Bareiss [Bar], entrent dans ce schéma, c'est là justement que résident leurs inconvénients quand la matrice A est dense (cf partie 2, chap.I et chap.IV).
Une approche toute différente a commencée d'être considérée dès 1961 par Takahasi et Ishibashi [TI], et permet, en effectuant la plupart des calculs dans des corps finis $\mathbb{Z}/p\mathbb{Z}$ (p est un nombre premier), de limiter l'usage de l'arithmétique sur les entiers, aux derniers stades de la résolution. Cette approche peut consister plus exactement, à appliquer le théorème des restes chinois [Knu] aux résidus modulo différents nombres premiers du vecteur x solution (cf partie 2, chap.III). Ou à effectuer suffisamment d'itérations de type Newton–Schultz (le processus généralise la méthode du calcul de l'inverse d'un nombre réel par Newton) à partir de l'inverse de A modulo un unique nombre premier pour pouvoir [Dix, GK] retrouver x (cf partie 2, chap.II).

Faisons remarquer avant de poursuivre, que l'obtention de la solution à l'aide du théorème des restes chinois est subordonnée à un grand nombre de résolutions dans les corps finis. C'est donc à ce processus de base que commencerons à nous consacrer dans ce mémoire : le problème à peine plus général du calcul du noyau de A, est en effet l'objet de la première partie. Les possibilités du calcul parallèle nous conduisent ensuite, dans la deuxième partie, à reprendre sous de nouveaux aspects, les méthodes énoncées plus haut.

Ces études théoriques ont été menées avec le constant souci de pouvoir les appliquer, nous les corroborons par divers résultats expérimentaux.

Pour reprendre l'exemple d'un système de dimension 700, pensons qu'il faut effectuer quelques 100 milliards d'opérations arithmétiques (cf partie 2, fig. IV.1) avant de connaître le vecteur x. Le besoin d'utiliser un ordinateur puissant est donc impératif si l'on ne tient pas à passer 200.000 ans (donnons nous 1 minute pour chaque opération) à chercher x sous notre lampe de chevet...

C'est le parallélisme qui actuellement offre les plus importantes puissances de calcul, quoi de plus naturel donc, que de tourner le Calcul Formel vers ces nouvelles possibilités. L'idée bien sûr n'est pas originale. Diverses études théoriques ont déjà été menées sur le sujet qui nous intéresse, on pourra citer en particulier un résultat de L.Csanky datant de 1976 [Csa], qui montrait que l'on peut calculer le déterminant d'une matrice de dimension n en $O(\log^2 n)$ étapes avec un nombre polynômial de processeurs. Et citer le papier dans lequel en 1984, S.J.Berkowitz [Ber] donnait une méthode n'utilisant aucune division et permettant de résoudre le même problème, toujours en $O(\log^2 n)$ étapes, le nombre de processeurs à utiliser étant en $O(n^{3.496})$.

Retenons encore l'article plus récent de B.Gregory et E.Kaltofen [GKa], démontrant en 1987, que les algorithmes de triangularisation asymptotiquement les meilleurs [BH], qui requièrent (de même que la méthode de Berkowitz) un algorithme rapide de multiplication de matrices [CW], peuvent conduire à des accroissements prohibitifs des tailles des entiers. Les auteurs concluent en préconisant de faire appel au théorème des restes chinois.

Par delà les aspects théoriques, il faut se tourner vers la mise en pratique du Calcul Formel parallèle. Des implantations du langage Lisp ont été réalisées sur différents multiprocesseurs, notamment : *Multilisp* [Hal] sur *Concert* [HAOS], *Qlisp* [GMc] sur *Alliant* [All] et *Concurrent Common Lisp* [CCL] sur *l'hypercube Intel iPSC* [Int]. Mais très peu d'expérimentations sont, pour l'instant, reportées dans la littérature. C.G.Ponder [Pon], par exemple, s'est intéressé après leur étude théorique, à l'exécution d'algorithmes sur l'hypercube d'Intel et sur l'Alliant sans beaucoup de succès : ceci étant dû aux langages eux–mêmes. Et certaines applications utilisent l'arithmétique en précision infinie de type Lisp d'un *Cray* [Neu].
Enfin, S.M. Watt a conçu un *Système Mapple Distribué* [Wat], véritable prototype de ce que pourront être les *Systèmes Distribués de Calcul Formel* dans un futur sans doute proche.

PAC : Parallel Algebraic Computations

C'est un hypercube de la série T de Floating Point Systems [FPS] à 16 processeurs qui nous a permis de réaliser nos expériences. Elles s'inscrivent dans un projet plus important, PAC, qui est celui de la réalisation d'une bibliothèque d'algorithmes de base en arithmétique exacte, sur une machine de type MIMD [RSSV]. Terminons donc cette introduction en précisant le modèle d'architecture que nous utilisons pour donner nos algorithmes, ainsi que pour évaluer leurs coûts.

• L'hypercube

L'hypercube est un ordinateur parallèle dont les processeurs, et les connections de voisins à voisins reliant ces derniers, forment un d–cube binaire. C.Seitz [Sei], est le premier a avoir utilisé cette solution efficace pour le *Cosmic Cube*, en 1985. Solution qui permet d'utiliser un grand nombre de processeurs tout en gardant un réseau d'interconnection simplifié.

Définition
Un d-cube binaire est un graphe non orienté composé de $P=2^d$ sommets numérotés de 0 à 2^d-1, tel que deux sommets sont reliés entre eux si et seulement si, en écriture binaire, leurs numéros diffèrent exactement d'un bit.

Nous donnons ci–dessous certaines propriétés de l'hypercube dont nous nous servirons tout au long de ce travail (le T20 de FPS sera plus précisément décrit dans le troisième paragraphe de la première partie). Le lecteur pourra en trouver le détail et les démonstrations, ainsi que d'autres résultats dans le papier de Y.Saad et M.H.Schultz [SS]. On note désormais P, le nombre de processeurs de l'hypercube considéré.

Il faut d'abord retenir que le d–cube est un graphe de diamètre $d=\log_2 P$ (rappelons que le diamètre est la distance maximale séparant deux sommets du graphe). Et que chacun des P sommets possède exactement $\log_2 P$ voisins.

Une des caractéristiques les plus importantes du d-cube, est que l'on peut lui inscrire de nombreuses autres topologies. Nous aurons par exemple souvent l'occasion d'utiliser la structure d'anneau : un cycle de longueur P qui traverse chaque sommet une et une seule fois. De tel cycles sont facilement construits à partir des codes de Gray [SS]. Ou la structure d'arbre, notamment pour assurer la diffusion de données dans l'hypercube (part.1, §I.1.4).

• Le modèle d'architecture

Notre étude théorique repose donc sur un modèle d'architecture, consistant en un réseau de P processeurs et de connections de voisins à voisins suivant la topologie hypercube. Chaque processeur est associé à une mémoire privée suffisamment importante, et communique par envois de messages [HB]. Le comportement global est du type MIMD (la classification a été donnée par M.J.Flynn en 1966 [Fly]).
Le protocole de communication est basé sur les rendez–vous entre processeurs. Aucun méchanisme global de synchronisation n'est disponible. On supposera que les calculs ne s'effectuent pas en parallèle au communications. Les canaux de communication sont bidirectionnels (envois et receptions simultanés). Ils peuvent travailler simultanément en émission et en reception de données indépendantes.

• **Modèle de complexité**

Le temps C_N, nécessaire à communiquer N données de taille unité (taille qui correspondra pour chacun des algorithmes à celle utilisée pour évaluer le coûts arithmétique) consécutives en mémoire a été donné par Y.Saad [Saa 1] : il consiste en un temps d'initialisation du canal plus un temps proportionnel à N,

$$\boxed{C_N = \beta + N\,\tau}\ .$$

Nous verrons au chap.III part.1, les valeurs mesurées pour le FPS T20, et comparerons le modèle théorique aux résultats expérimentaux. De manière générale, pour juger des performances des algorithmes, il sera nécessaire de tenir compte du rapport β/τ qui peut varier considérablement d'une machine à une autre.

Les coûts arithmétiques sont donnés en comptant les nombres d'opérations effectuées, sur des mots de taille élémentaire fixée à l'avance. Pour les algorithmes étudiés dans la première partie, cette taille correspond à celle du nombre premier utilisé. Et l'on peut se ramener à cette situation quand les opérandes sont des entiers de tailles quelconques, en considérant leurs décompositions dans une base bien choisie (cf §I.1.3 part.2).

• **Ecriture des algorithmes**

Afin de simplifier leur compréhension, nous écrivons les algorithmes dans un *pseudo langage parallèle*. Le programme, souvent identique pour tous les processeurs, est seulement paramétré par un numéro : *proc*, auquel on peut donner des valeurs allant de 0 à P–1. Chaque processeur connait son identité.

Le parallélisme est en général intrinsèque : le programme donné s'exécute simultanément sur les P processeurs. Sinon, nous le spécifions à l'aide d'un constructeur PAR analogue à celui du langage Occam [Inm] : deux ordres de communication, indentés sous un même constructeur PAR sont appelés à s'exécuter en parallèle. Enfin, les instructions pour les transferts sont *envoyer* et *recevoir*, suivies du nom de la variable à envoyer ou à recevoir (la longueur du message sera déduite du contexte).

PREMIERE PARTIE

NOYAU D'UNE MATRICE A COEFFICIENTS DANS UN CORPS FINI

INTRODUCTION

Nous nous consacrons dans cette première partie, à l'étude de divers algorithmes pour le calcul d'une base du **noyau** d'une matrice A (de l'application linéaire associée à A), carrée et à coefficients dans un corps $\mathbb{Z}/p\mathbb{Z}$ (p premier). Ceci peut facilement conduire au calcul d'une solution générale d'un système Ax=b. Rappelons qu'une solution générale est la donnée d'une base du noyau de la matrice (quand elle n'est pas inversible) et d'une solution particulière du système. Le rapprochement avec la **résolution de systèmes** par élimination de Gauss ou **l'inversion de matrices** par celle de Jordan, est comme nous le verrons immédiat.

Avant d'être étudiée sur les corps finis, l'élimination de Gauss en parallèle l'a été sous les aspects du calcul numérique réel. Le travail que nous présentons, s'appuie donc naturellement sur les travaux réalisés dans ce domaine. Une difficulté nouvelle provient du **pivotage**. En calcul numérique, cela consiste à trouver un plus petit coefficient localement (*pairwise pivoting* [Sor]) ou globalement parmis toutes les lignes de la matrice. Dans $\mathbb{Z}/p\mathbb{Z}$, le pivotage est la recherche globale d'un élément non nul. Si la matrice n'est pas inversible, cette recherche peut bien sûr ne pas aboutir. Le problème est de minimiser les coûts de communication associés à cette recherche.

L'application de l'élimination de Gauss, doit souvent être complétée par la résolution d'un **système triangulaire**. Diverses solutions ont été proposées, et peuvent être trouvées dans la littérature [GH, LC, Mol, OR 1]. Mais ce ne sera pas ici notre propos. Nous avons utilisés pour les résolutions correspondant aux algorithmes de la deuxième partie, des versions simplifiées mise au point à partir des résultats de Ortega et Romine [OR 1]. Les coûts se sont révélés faibles en pratique.

Dans les deux prochains chapitres, nous reprenons pour les adapter au cadre nouveau de $\mathbb{Z}/p\mathbb{Z}$. L'**algorithme de diffusion** étudié en premier lieu par G.A.Geist, M.T.Heath et Y.Saad [Gei, GH, Saa 2]. L'idée de base en est particulièrement simple et naturelle, elle donnera de bon résultats. L'**algorithme du pipeline**, introduit par Y.Saad [Saa 2] (le principe était connu depuis longtemps [Sam]). M.Cosnard et al. [CTV 2] ont ajusté les résultats donnés par Saad concernant son coût. Cette méthode garde tout son intérêt si seul un anneau de processeurs est disponible, mais sera occultée par la précédente, qui utilise au mieux les possibilités de l'hypercube. Les deux méthodes peuvent aussi être retrouvée dans [GN, OR 2], dont les auteurs parviennent aux mêmes conclusions. Des versions par blocs (dites aussi *à plusieurs pas*, Partie 2) ont été mises au point et discutées par Y.Robert et B.Tourancheau [RT], les mêmes raisonnements pourraient s'appliquer ici, mais ne changeraient pas nos conclusions. Nous nous consacrons finalement aux **algorithmes à pivots locaux** mis au point par M.Cosnard et al.

[CTV 2]. Si leur démarche nouvelle, ne donne pas entière satisfaction en calcul numérique (problèmes de stabilité) ou même en calcul formel (tailles des coefficients intermédiaires, cf Partie 2, chapitre I), nous verrons que ces algorithmes sont, dans la situation qui nous intéresse, et pour certaines classes de matrices, les meilleurs.

Des études de **complexité** de l'élimination de Gauss pauvent être trouvées dans [Gen, ISS, Saa 1], qui donnent des bornes minimales pour les coûts des différents algorithmes. Certaines seront reprises au début du premier chapitre. Et complétées par un premier résultat d'**optimalité** mis en évidence par Y.Robert & al. [RTV].

Le problème du calcul du noyau en parallèle, a lui–même été abordé en rapport avec la factorisation des grands entiers et plus particulièrement la méthode des fractions continues [WW]. La factorisation des matrices étant envisagée avec l'algorithme de D.Wiedemann [Wie] pour les **systèmes creux** ou à l'aide de l'élimination de Gauss [MB, PW].
Des performances obtenues sur une architecture systolique sont décrites dans [LQRV]. Avec le modèle d'architecture qui nous intéresse, on trouvera le travail de M.Cosnard & Y.Robert [CR], qui complètent leur étude théorique dans $\mathbb{Z}/2\mathbb{Z}$ par divers résultats expérimentaux.

Après ce bref *état de l'art*, nous pouvons aborder dans cette introduction l'aspect séquentiel du problème et présenter les algorithmes que nous nous proposons de paralléliser.

LES ALGORITHMES SEQUENTIELS

La dimension de la matrice A est notée n. Et r son rang. La base recherchée possède donc n-r vecteurs. Deux méthodes à priori similaires permettent de résoudre le problème : on peut ou non considérer la matrice identité bordant la matrice A au cours de l'algorithme. Si en séquentiel, ces deux méthodes donnent des résultats analogues, nous verrons qu'il en est autrement en calcul parallèle.

.1 Avec la matrice identité bordante

Définition 1.1

Une matrice E de rang r est sous forme **échelonnée en lignes** si l'on peut trouver r indices de colonnes $1 \leq k_1 \leq k_2 \ldots \leq k_r \leq n$ tels que $E(i,k_i) \neq 0$, $E(i,j)=0$ pour $1 \leq j < k_i$ et si les (n-r) dernières lignes de E sont nulles.

La matrice E est **échelonnée réduite** si elle est échelonnée et si le seul élément non nul de la colonne k_i est $E(i,k_i)$. Les éléments $E(i,k_i)$ sont appelés éléments diagonaux de E.

Toute matrice triangulaire est trivialement sous forme échelonnée en lignes, ainsi que toute matrice diagonale sous forme échelonée réduite. On a une définition équivalente pour la forme échelonnée en colonne.

L'algorithme utilisé par exemple par Morrison et Brillhart [MB], rend la matrice ${}^tA$ sous forme échelonnée en opérant sur ses lignes. Les mêmes combinaisons sont appliquées à la matrice identité bordante de taille n, dont les n–r dernières lignes forment, en fin d'algorithme, une base du noyau de A.

Soit L la matrice des transformations appliquées à ${}^tA$, L est obtenue à partir de l'identité. Et E la matrice échelonnée obtenue à partir de ${}^tA$. Les matrices L et E vérifient $L{}^tA=E$,

$$\begin{pmatrix} L_1 & L_3 \\ L_2 & L_4 \end{pmatrix} \begin{pmatrix} A_1^t \\ A_2^t \end{pmatrix} = \begin{pmatrix} E_1 \\ 0 \end{pmatrix} \qquad (i.1)$$

où les matrices L_2, L_4 et A^t_2 ont n-r lignes. Si ^tz est une des n-r dernières lignes de L (ces lignes sont indépendantes puisque le déterminant de L est non nul), on a alors effectivement,

$$^tz\ ^tA = 0, \text{ ce qui est équivalent à } A\ z = 0.$$

La méthode utilisée pour parvenir à la relation (i.1) est l'élimination de Gauss. A une étape k donnée, en notant $^tA^{(k-1)}$ la transformée de la matrice ^tA avant la k-ième étape et en appelant **indice pivot**, l'indice augmenté de 1, de la dernière ligne ayant servi à éliminer :

- si aucun pivot non nul n'est trouvé dans la colonne k à partir de l'indice pivot, le rang décroit de 1 et l'algorithme reprend sur la colonne k+1. L'indice pivot n'est pas modifié.
- si, quitte à effectuer une permutation de lignes, le coefficient pivot est non nul, on annule les coefficients en colonne k des lignes d'indices supérieurs à l'indice pivot en appliquant les transformations usuelles de l'élimination de Gauss. L'indice pivot est incrémenté de 1.

Notons que seules des permutations de lignes sont autorisées. L'algorithme peut s'écrire comme suit,

- **Algorithme SeqId** *Séquentiel avec Identité*

début
ip = 1 *l'indice pivot*
rang = n
pour k de 1 à n
on cherche un indice i supérieur à ip et tel que $^tA^{(k-1)}[i,k] \neq 0$: *le pivot*
si l'on trouve un tel indice i
si i≠ip on permute les lignes i et ip (cette permutation pourra être réalisée en mémoire, en général elle ne sera que virtuelle)
Normalisation de la ligne ip
pour j=k à 2n
$^tA^{(k)}[ip,j] = {}^tA^{(k-1)}[ip,j] / {}^tA^{(k-1)}[ip,k] \bmod p$
Eliminations
pour i > ip si le coefficient de tête de la ligne est non nul
pour j = k+1 à 2n
$^tA^{(k)}[i,j] = {}^tA^{(k-1)}[i,j] - ({}^tA^{(k-1)}[i,k]\ {}^tA^{(k)}[ip,j]) \bmod p$
$^tA^{(k)}[i,k] = 0$
ip = ip + 1
sinon rang = rang - 1
fin

Lors de la recherche du pivot non nul, le choix de l'indice i importe peu pour le déroulement de l'algorithme séquentiel. Nous verrons plus loin qu'il n'en est pas de même pour les algorithmes parallèles (minimisation des coûts des communications).

• Le coût de l'algorithme

Pour le calcul des coûts arithmétiques des algorithmes on se placera toujours dans le **plus mauvais cas** : quand aucun pivot nul n'est rencontré et quand tout les coefficients de tête des lignes à éliminer sont différents de zéro. Cette hypothèse ne sera en général pas vérifiée (surtout si le nombre premier est petit), le coût donné représentera donc, dans la plupart des cas, une borne supérieure.
Totalisons le nombre d'opérations à effectuer sur des mots de taille $\log_2 p$: les additions, les multiplications et les moduli. On suppose qu'une réduction modulo p n'est effectuée qu'une fois par étape k pour un coefficient donné (comme spécifié dans SeqId), et que son coût est unitaire (en réalité sur des arguments de tailles double de ceux de la multiplication).

Pour l'algorithme SeqId, une étape k demande de l'ordre de :

• $(n-k)^2$ additions, multiplications et moduli sur les coefficients de la matrice $A^{(k-1)}$, ainsi qu'une inversion modulo.

• (n-k)k additions, multiplications et moduli sur la matrice identité, *a priori*. Mais suite aux permutations de lignes effectuées, les coefficients dans la matrice identité ne seront pas localisés de manière à permettre une **vectorisation** efficace des calculs. Dans le cas de l'utilisation d'une unité de calcul vectoriel, on a donc le même nombre d'opérations si l'on dispose d'une arithmétique sur des vecteurs creux. Dans le cas contraire, et si le coût des opération sscalaires est prohibitif (les boucles de calcul de SeqId vont jusqu'à 2n) le coût reste borné par n(n–k).

En sommant sur le nombre d'étape, on obtient donc pour le coût total,

$$\boxed{\text{ASeqId} \leq (3/2 + \gamma)\, n^3 + O(n^2),\ 0 \leq \gamma \leq 1}. \qquad (\text{i.2})$$

.2 Sans matrice identité bordante

On peut aussi calculer une base de l'espace nul de A sans stocker la matrice identité : en appliquant une généralisation de l'algorithme de Jordan pour rendre A sous forme échelonnée réduite. C'est le procédé utilisé en calcul formel pour définir une forme canonique de la solution générale d'un système linéaire à coefficients dans un corps [BMcL] ou sur un anneau intègre [McC]. On appelle toujours indice pivot l'indice plus 1 de la dernière ligne ayant servi à éliminer. A une étape k donnée, de façon analogue à SeqId :

• si aucun pivot non nul n'est trouvé dans la colonne k à partir de l'indice pivot, le rang décroit de 1 et l'algorithme reprend sur la colonne suivante. L'indice pivot n'est pas modifié.

• si, quitte à effectuer une permutation de lignes, le coefficient pivot est non nul, on annule à l'aide de la ligne pivot ip tous les autres coefficients en colonne k. La ligne pivot est normalisée.

L'algorithme s'écrit de même simplement,

- **Algorithme SeqNId** *Séquentiel Non Identité*

début
ip = 1 *l'indice pivot*
rang = n
pour k de 1 à n
on cherche un indice i supérieur à ip et tel que $A^{(k-1)}[i,k] \neq 0$ *le pivot*
si l'on trouve un tel indice i
si i≠ip on permute les lignes i et ip (cette permutation pourra être réalisée en mémoire, en générale elle ne sera que virtuelle)
Normalisation de la ligne ip
pour j=k à n
$A^{(k)}[ip,j] = A^{(k-1)}[ip,j] / A^{(k-1)}[ip,k] \bmod p$
Eliminations
pour i ≠ ip si le coefficient de tête de la ligne est non nul
pour j = k+1 à n
$A^{(k)}[i,j] = A^{(k-1)}[i,j] - (A^{(k-1)}[i,k]\ A^{(k)}[ip,j]) \bmod p$
$A^{(k)}[i,k] = 0$
ip = ip + 1
sinon rang = rang - 1
fin

En notant L_r la matrice des transformations et E_r la matrice echelonnée réduite, on a en fin d'exécution et après d'éventuelles permutations de colonnes : $L_rA=E_r$,

$$L_r A = \begin{pmatrix} Id_r & Z \\ E_1 & \end{pmatrix} \qquad (i.3)$$

Où Id_r est la matrice identité de taille r. La matrice Z, à n lignes et n–r colonnes permet alors de construire les vecteurs recherchés. On considère pour cela deux ensembles d'indices, $J_A=(j_1,\ldots,j_r)$ qui est l'ensemble des indices des colonnes de A participant à la matrice Id_r, et $K_A=(k_1,\ldots,k_{n-r})$ son complémentaire par rapport à (1,…,n). On peut montrer [McC] que les vecteurs z_k, $1\leq k\leq n-r$, donnés par

$$z_k[j_i] = Z[i,k] \text{ et } z_k[k_i] = \begin{cases} -1 \text{ si } k=i, \\ 0 \text{ sinon.} \end{cases} \qquad (i.4)$$

sont indépendants et vérifient,

$$Az_k = 0.$$

• Le coût de l'algorithme

Nous n'avons pas ici les problèmes posés au paragraphe précédent, par les coefficients de la matrice identité. D'où simplement,

$$\boxed{ASeqNId \leq 3/2\ n^3 + O(n^2)}. \qquad (i.5)$$

Les calculs peuvent être vectorisés de manière efficace puisque toujours effectués sur des vecteurs denses.

• Eliminations de Gauss et de Jordan

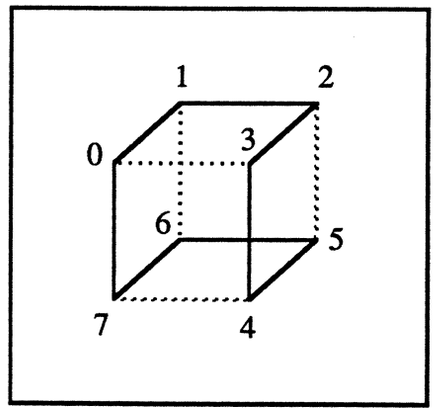
Il est clair que les algorithmes parallèles proposés pour les deux méthodes que nous venons de voir, fourniront des implantations possibles de la résolution de systèmes linéaires : l'obtention de la matrice triangulaire (SeqId sans la matrice identité) sera complétée par la résolution triangulaire correspondante. Et aussi du calcul de l'inverse d'une matrice (SeqNId avec la matrice identité bordante).

Nous utiliserons cette remarque dans la deuxième partie du mémoire, sans détailler le passage d'un problème à l'autre. Et présenterons des résultats expérimentaux pour chacuns des trois problèmes.

CHAPITRE I

ALGORITHMES DE DIFFUSION ET DU PIPELINE

Nous avons détaillé notre modèle de calcul dans l'introduction. P est le nombre de processeurs disponibles. Et la dimension n du problème, est un multiple de P. Rappelons que pour k fixé, $0 \leq k \leq \log_2 P$, il est toujours possible de trouver un cycle de longueur 2^k dans le d–cube, tel que les nœuds reliés par une arête du cycle soient voisins dans l'hypercube [SS] : et forment ainsi un anneau de processeurs. Ceci nous permet de supposer dans tous les cas, que les processeurs de l'hypercube sont numérotés en anneau de 0 à P–1 suivant les codes de Gray [SS].



Anneau de longueur 8 dans un hypercube de dimension 3.

Les implantations parallèles des méthodes SeqId et SeqNId sont donc étudiées pour la topologie hypercube. Mais les coûts donnés, en particulier ceux qui utilisent la diffusion de données (§I.2), pourraient facilement se généraliser à d'autres réseaux (complet, anneau, pyramidale,...).

I.1 PREMIERES PROPRIETES

Avant de discuter du problème de la répartition de la matrice entre les différents processeurs, et de son impact sur les coûts arithmétiques, complétons le modèle de calcul avec trois heuristiques. Elles ont été largement utilisées dans la littérature [GH, Saa1, Saa2, CTV]. Soulignons que l'une d'entre elles, le principe de complétude, sera quelque peu modifiée au chapitre II pour les éliminations à pivots locaux.

I.1.1 Un modèle pour les algorithmes

Il nous faut faire quelques hypothèses sur la classe des algorithmes parallèles auxquels nous nous intéressons, et nous placer dans un cadre plus précis, qui nous permettra plus loin, de travailler sur leur complexité [RTV, Saa 1, SS]. Les trois principes suivants ont été explicités par Gerasoulis et Nelken [GN], outre certaines conséquences spécifiques, ils simplifient considérablement l'étude et mèneront à des résultats d'optimalité :

- **Equidistribution des données**

(Eq1) : chaque processeur possède exactement $2n^2/P$ ou n^2/P coefficients de la matrice initiale, suivant que l'on utilise la matrice identité ou non. Aucun des coefficients n'étant stocké dans plus d'un processeur.

ou

(Eq2) : chaque processeur possède exactement n/P lignes (ou colonnes) de la matrice initiale. Aucune ligne (ou colonnes) n'étant stockée dans plus d'un processeur.

Cette première hypothèse permet essentiellement de minimiser la place mémoire requise par l'exécution des algorithmes. On peut remarquer aussi que l'équidistribution des données, n'est pas une condition nécessaire pour assurer que tous les processeurs effectuent le même volume de calculs (tel n'est d'ailleurs pas forcément le but recherché, cf fig.I.3) : le nombre d'opérations associées à un coefficient de la matrice dépend de son indice de ligne.

- **Localité des données**

Un processeur ne peut modifier que les données qu'il possède en mémoire.

Ce principe réduit le coût des communications [GN] : prenons l'exemple d'une élimination de Gauss par lignes en supposant (Eq2), la modification d'une ligne L peut se faire aussi bien dans le processeur P_L où elle est stockée, que dans le processeur qui possède la ligne pivot. Mais choisir cette deuxième possibilité, celle qui ne respecte pas la localité des données, augmente le coût des communications, puisque la ligne L, après son élimination, doit être renvoyée au processeur P_L.

- **Principe de complétude**

Nous étudions des implantations pour lesquelles le programme de chacun des processeurs a la forme [Saa 1],

début
  pour k = 1 à n
    (C1) : Echanges de données. A la fin de cette phase, chaque processeur a reçu, (ou en possédait certains) tous les coefficients de la ligne (ou colonne) pivot.
    (C2) : Calculs. Le processeur effectue les éliminations.
fin.

Et telles que, à une étape k donnée, la phase (C2) ne peut s'exécuter que si la phase (C1) est terminée, cette dernière ne pouvant elle–même débuter que si la phase (C2) de l'étape k–1 est finie. Ces conditions étant bien entendu locales à chaque processeur.

Remarquons que ce principe implique que calculs et communications ne s'effectueront pas simultanément (comme le modèle le présuppose). De même (C1) consiste éventuellement en une première étape de pivotage.
Il permet aussi de restreindre la place mémoire nécessaire au stockage des messages [GN].

Sous ces hypothèses, on peut parler pour chaque processeur, **de coût des communications** et **de coût arithmétique**. Ces coûts sont calculés en sommant ceux des phases (C1) et (C2) pour k allant de 1 à n, et incluent les temps de latence. Pour le coût d'un algorithme il suffit alors de prendre leur maximum sur les P processeurs, en supposant que ces derniers ont tous commencé à travailler à un même instant T_0.

Plus généralement, Y.Saad [Saa 1] a proposé de définir le temps des communications d'un algorithme, comme étant le temps nécessaire à son exécution si les opérations arithmétiques ont un coût total négligeable; et inversement pour l'arithmétique.

I.1.2 Répartition de la matrice dans le réseau

De nombreuses stratégies peuvent être envisagées pour répartir la matrice A (éventuellement bordée par l'identité) parmi les P processeurs. Nous supposerons dans tout ce qui suit que A est répartie par lignes, et appliquerons le principe Eq2; Eq1 sera utilisé pour un rappel de complexité dans le paragraphe suivant.

Avant de s'intéresser aux diverses répartitions possibles, définissons une fonction alloc, appelée **fonction d'allocation** des lignes de la matrice,

$$\text{alloc} : [1,\ldots,n] \longrightarrow [0,\ldots,P-1] \qquad (1.1)$$

définie telle que la ligne d'indice i de A se trouve initialement dans le processeur alloc(i). Et faisons, sans les détailler, quelques remarques concernant le choix éventuel d'une répartition de la matrice par colonnes.

Remarques 1.2.1

• Si l'on choisit une répartition par colonnes pour appliquer une version parallèle de l'algorithme SeqId. Et si les fonctions d'allocation des colonnes de A et de l'identités sont telles que pour tout j, $\text{alloc}_A(j)=\text{alloc}_{Id}(j)=\text{proc}$. Alors on peut se ramener à une version de SeqNId par lignes, les colonnes de l'identité étant traitées comme les lignes situées au dessus de la diagonale.

• De même si SeqNId est appliqué avec une répartition par colonnes, on se ramène à une version de SeqId par lignes.

• Dans les deux cas et pour l'algorithme de diffusion (cf §I.2) le pivotage est plus simple à effectuer, il n'y a pas d'élection : le processeur pivot peut seul décider d'une permutation (en fait pour SeqId par colonnes le problème se retrouve pour l'arithmétique sur l'identité). Pour l'algorithme du *pipeline* (cf §I.3) les différences sont sans conséquence sur nos conclusions. Les algorithmes à pivots locaux sont quant à eux bien plus difficiles à envisager par colonnes.

De nombreuses permutations de lignes pouvant être nécessaires en cours d'algorithme, elle seront dans la plupart des cas virtuelles de façon à ne pas demander de communications supplémentaires. On remarquer donc qu'en général, l'ordre final de stockage des lignes de la matrice n'est pas connu à l'avance.

On peut maintenant énumérer les fonctions d'allocations qui ont été données et étudiées par [Gei, GH, RTV, Saa 2],

• **Répartition "périodique"** (*wrap mapping*) : les lignes sont réparties une à une en tournant sur l'anneau, plus précisément,

$$\text{alloc}(i) = (i-1) \bmod P. \qquad (1.2)$$

• **Répartition "en miroir"** (*reflection mapping*) : comme ci–dessus, mais en changeant de sens à chaque tour, ($\lceil x \rceil$ est la partie entière supérieure de x)

$$\begin{aligned} \text{alloc}(i) &= (i-1) \bmod P \text{ si } \lceil i/P \rceil \text{ est impair,} \\ &= P-i \bmod P \text{ si } \lceil i/P \rceil \text{ est pair.} \end{aligned} \qquad (1.3)$$

• **Répartition par blocs de taille r** : r est un entier compris entre 1 et n/P qui divise n/P, cette répartition généralise la première en considérant à chaque fois r lignes consécutives,

$$\text{alloc}(i) = (\lceil i/r \rceil - 1) \bmod P. \qquad (1.4)$$

I.1.3 Répartition de la matrice et coût arithmétique

Le problème est alors de déterminer la fonction d'allocation qui minimise la somme du coût arithmétique et du coût des communications. On pourrait imaginer bien d'autres répartitions que celles que nous avons données, mais pour la plupart, les coûts correspondants ne peuvent être calculés.
Commençons par étudier dans ce paragraphe, comment le choix de la répartition influence le coût arithmétique.

• Pour des élimination du type SeqNId, et d'après une remarque de [RTV], la conclusion est immédiate. On sait que le coût arithmétique, indépendant de la répartition des lignes de la matrice, est asymptotiquement, le quotient du coût séquentiel par le nombre de processeurs. En effet, chaque processeur opère à chaque étape sur toutes les lignes qu'il a en mémoire, et la matrice est supposée équidistribuée (Eq2). On prendra,

$$\boxed{\text{AParNId} \le \frac{3}{2P} n^3 + O(n^2)}. \qquad (1.5)$$

Bien que les processeurs n'effectuent pas tous exactement le même nombre de calculs, puisque l'on n'opère pas sur la ligne pivot, les termes d'ordres inférieurs ne sont ici d'aucun intérêt. La différence des coûts arithmétiques de deux processeurs différents, est en $O(n^2)$ si $n>>P$ et en $O(n)$ si $P=\alpha n$.

• Si les éliminations sont du type SeqId, on sait [Gei, GN] que les allocations "périodique" et "en miroir" donnent des coûts du même ordre; la répartition des calculs étant mieux équilibrée dans le deuxième cas.
Pour les **répartitions par blocs** de taille r, on peut donner une formule générale en fonction de la taille de ces derniers. Calculons le coût sur le processeur P_{P-1}, le plus lent. A une étape $k=k_1Pr+k_2$, $0\le k_2<Pr$, P_{P-1} doit mettre à jour $(n/P-k_1r)-\max\{k_2-(P-1)r,0\}$ lignes (cette quantité a été mise en évidence dans [RTV]), chacune étant de longueur comprise entre n et n+n–k (suivant l'arithmétique utilisée). En sommant sur k, à l'aide de,

$$S_{min} = \sum_{k=1}^{n} (n/P - \lfloor k/Pr \rfloor r)\, n = \frac{n^3}{2P} - \frac{rn^2}{2} + nPr^2 + O(n) \quad \text{et,}$$

$$S_{max} = \sum_{k=1}^{n} (n/P - \lfloor k/Pr \rfloor r)\,(n-k) = \frac{n^3}{3P} + \frac{rn^2}{4} - \frac{r^2nP}{12} + O(n)\,, \qquad (1.6)$$

on obtient le coût arithmétique total,

$$\text{AParId}(r) \le 3S_{min} + \gamma\, 3S_{max}\,,\ 0 \le \gamma \le 1,$$

soit,

$$\boxed{\text{AParId}(r) \le \frac{n^3}{P}\left(\frac{3}{2}+\gamma\right) + rn^2\left(\frac{3\gamma}{4}-\frac{3}{2}\right) - nr^2\frac{\gamma P}{4} + 3nPr^2 + O(n),\ 0 \le \gamma \le 1}. \qquad (1.7)$$

Nous verrons que ces formules présentent un intérêt certain si la taille du problème et le nombre de processeurs sont du même ordre, en particulier si P s'écrit $P=\alpha n$. Dans ce cas, les coûts de l'arithmétique et des communications aussi seront du même ordre, et on pourra chercher à minimiser leur somme (cf I.3).

Si $n>>P$, les coûts des communications sont asymptotiquement négligeables. Le problème est donc simplement de minimiser le coût arithmétique. Il faut alors prendre $r=n/P$, et ne conserver qu'un terme en n^3 dans (1.7), pour avoir une quantité analogue à (1.5).

Enfin, pour des raisons évidentes la situation où $n<<P$ ne sera pas envisagée ! Et si $P=O(n^2)$ les communications prédomineront [Saa1].

I.1.4 Complexité des communications

Lors de l'étude des algorithmes, les termes en β des coûts des communications pourront ou non être donnés. Il est de toute façon facile de se convaincre, que les termes en τ sont dominants. Le modèle pourrait raisonablement se limiter à ces derniers [GN, RTV, Saa 1].

● Y.Saad a donné [Saa 1] différentes bornes minimales pour les coûts des communications de l'élimination de Gauss. En gardant les mêmes hypothèses mais avec le principe Eq1. On sait par exemple, que le terme en τ du coût de l'élimination sur un anneau de processeurs, est borné par,

$$T_{GA} \geq \frac{1}{2}\left(\frac{n^2}{16}\left(1-\frac{8}{P}\right) + \frac{Pn}{12}\left(1-\frac{3}{P}\right)\right)\tau .$$

Nous nous attachons ici à mettre en évidence des bornes analogues, quand la matrice est répartie par lignes. La situation est plus simple.

Proposition 1.4.1

Si les principes de localité, d'équidistribution et de complétude sont vérifiés, le terme en τ du coût des communications d'une version parallèle de SeqNID sur un hypercube de dimension $\log_2 P$, est tel que,

$$\boxed{T_{hyp} \geq \frac{n\,(n+1) - 2\log_2 P}{2\log_2 P}\,\tau} . \qquad (1.8)$$

Pour une version de SeqId, la borne est multipliée par deux.

Il suffit de remarquer que chaque processeur doit recevoir n–n/P lignes et en envoyer n/P. D'où un volume total de communications d'au moins,

$$v \geq \sum_{k=0}^{n-2} n-k.$$

Cette quantité divisée par $\log_2 P$, le nombre de canaux en entrée, donne le résultat. Si l'identité borde la matrice A, les lignes sont en moyenne deux fois plus longues.

Y.Robert et al. [RTV] ont montré que sur un anneau, la borne correspondante pour SeqNID,

$$\boxed{T_{ann} \geq n^2 + Pn + O(n)}$$

peut être atteinte. Pour l'hypercube, T_{hyp} ne sera obtenue qu'asymptotiquement en τ (d'après (I.11)).

• Diffusion de données

Avant de terminer ce paragraphe et commencer à décrire différents algorithmes, nous avons besoin de résultats sur le coût de l'opération de diffusion (*broadcast*) dans l'hypercube. Cette opération consiste en l'envoi d'un même message, d'un processeur vers tous les autres. Si le message est de taille unité, le coût est simplement donné par

$$\boxed{D_u = (\log_2 P)\,(\tau + \beta)}. \qquad (1.9)$$

Pour un message de longueur plus importante, une première technique [Saa 2] consiste à découper le message en paquets de tailles égales, puis à *pipeliner* ces derniers sur un arbre de hauteur $\log_2 P$ inscrit dans l'hypercube. De nombreux types d'arbres peuvent être utilisés. Pour une étude systématique, on se réfèrera à [SS, JH]. Le procédé peut se généraliser en utilisant $\log_2 P$ arbres aux arêtes disjointes. Si le message initial est de taille L, et qu'il est subdivisé en paquets de tailles T,

$$T = \sqrt{\frac{L\beta}{\tau \log_2^2 P}}, \qquad (1.10)$$

alors le coût de sa diffusion peut être réduit à [JH, SW],

$$\boxed{D_{min}(L) = \left(\sqrt{\frac{L\tau}{\log_2 P}} + \sqrt{\beta \log_2 P}\right)^2}. \qquad (1.11)$$

Une borne minimale pour le terme en τ étant clairement donnée par $L/\log_2 P$ (volume à transférer divisé par le nombre de canaux), cette borne peut être approchée par (1.11) si par exemple, $L\tau >> \log_2 P\beta$. D'un point de vue pratique, l'implantation demandera une partie contrôle coûteuse. Et si le temps d'initialisation β est important (cf §III.1), l'optimalité sera difficile à assurer.

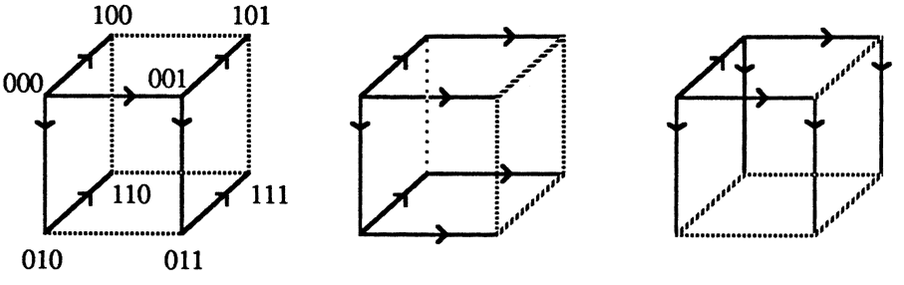


Figure I.1 : Trois arbres (arêtes non disjointes) obtenus par rotations, pour une diffusion dans un hypercube de dimension 3.

Nous complétons donc ce paragraphe en donnant le coût d'un algorithme plus simple à implanter. Il utilise toujours $\log_2 P$ arbres, maintenant aux arêtes non disjointes. Le message est découpé en $\log_2 P$ paquets, mais ces derniers ne sont plus *pipelinés* : chacun est transmis sur un des arbres en $\log_2 P$ étapes. Le coût correspondant [JH, SW] donné par,

$$\boxed{D(L) = L\,\tau + (\log_2 P)\,\beta}, \qquad (1.12)$$

est donc à un facteur $\log_2 P$ de la borne minimale. C'est celui que nous utiliserons par la suite. Remarquons pour terminer que si les processeurs sont synchronisés au début de la diffusion, le processus ne les désynchronise pas.

I.2 ALGORITHME DE DIFFUSION

Des algorithmes basés sur l'opération de diffusion ont été étudiés dans [Gei, Saa 2, GN]. L'idée en est simple :

- A une étape k donnée, si quitte à effectuer une permutation, le coefficient pivot est non nul, le processeur pivot la diffuse vers tous les autres processeurs.
- Si aucun coefficient pivot non nul ne peut être trouvé dans P_{proc}, il faut en rechercher un parmi les autres processeurs à l'aide d'un processus d'élection. Et effectuer la permutation correspondante. Si la recherche n'aboutit pas, le rang décroit.

• Processus d'élection

De nombreuses méthodes permettent de réaliser un tel processus et d'en minimiser le coût. On peut par exemple procéder comme suit,

- tous les processeurs commencent par chercher s'ils possèdent en mémoire une ligne de coefficient de tête non nul, et stockent éventuellement l'indice correspondant.
- Puis le processeur pivot ayant donné un ordre de pivotage (1° diffusion d'une variable), il reçoit ces indices (accompagnés des numéros des processeurs, 2° diffusion) par une remontée dans un arbre de l'hypercube.
- Il choisit un des indices, en informe le réseau (3° diffusion), et c'est le processeur correspondant, le nouveau processeur pivot, qui diffuse la ligne pivot pour permettre les éliminations.

Par défaut, il n'y a donc qu'une permutation virtuelle, un réindiçage des lignes. Ce qui permet de supprimer une diffusion de ligne, inutile. Le coût de l'opération (chaque processeur pouvant ne retransmettre qu'un seul des indices qu'il a reçu), si les informations peuvent être codées sur un mot, est borné par,

$$E \leq 3D_u = 3(\log_2 P)\,(\beta + \tau). \qquad (1.13)$$

On peut forcer l'ordre final des lignes à correspondre aussi à la fonction de répartition alloc, ce qui est par exemple nécessaire pour les applications du théorème des restes chinois (part.2 chap.III). Il suffit pour cela d'effectuer les permutations de mémoire à mémoire. Les deux processeurs concernés, qui peuvent être séparés d'une distance $\log_2 P$, doivent s'échanger une ligne. Le coût associé, si les deux lignes sont de longueur L, peut être borné d'après [SW], par :

$$E_{ord} \approx \frac{4L}{\log_2 P}\,\tau + (\log_2 P)\,\beta. \qquad (1.14)$$

Donnons maintenant une unique version parrallèle correspondant aux algorithmes SeqId et SeqNId. Si l'on ne détaille pas les éliminations et les longueurs des messages à communiquer, les deux méthodes sont effet équivalentes.

• Algorithme DParId *Diffusion Parallèle avec Identité*

Programme du processeur P_{proc}
début
ip = 1
rang = n
pour k de 1 à n *Pivotage*
si P_{proc} possède une ligne d'indice i ≥ ip et de coefficient de tête non nul
alors trouvé = i
sinon trouvé = faux
si alloc(ip) = proc *Le processeur pivot par défaut*
si trouvé = faux
diffusion de l'ordre de pivotage sur le réseau
reception des indices de lignes possibles
si un indice peut être choisi parmi les indices reçus
diffusion de l'indice choisi (réindiçage) et du numéro du nouveau processeur pivot
sinon diffusion de "pas de pivot"
sinon *Les autres processeurs*
si ordre de pivotage
envoi de trouvé au processeur $P_{alloc(ip)}$
reception de l'indice (réindiçage) choisi par $P_{alloc(ip)}$ et du numéro du nouveau processeur pivot ou de de "pas de pivot"

Diffusion de la ligne pivot et éliminations
si "pas de pivot" rang = rang - 1
sinon
diffusion de la ligne pivot par le processeur pivot par défaut ou par le nouveau processeur pivot.
élimination des lignes en mémoire à l'aide de la ligne reçue
ip = ip + 1
fin

D'après le protocole de communication, pour qu'il y ait diffusion, tous les processeurs doivent connaitre le numéro du processeur source : s'il y a élection il est donc bien nécessaire de diffuser le numéro du nouveau processeur pivot.

• Coût de l'algorithme

Les résultats donnés correspondent de même qu'en séquentiel aux situations les plus coûteuses à envisager. Il faut remarquer en particulier, que le plus mauvais cas pour le coût des communications qui voit s'effectuer une élection à chaque étape, est distinct du plus mauvais cas

pour l'arithmétique qui se présente quand aucun coefficient de tête nul n'est rencontré.

Remarque 2.1
On peut supposer que la répartition des lignes de la matrice n'influe pas sur les temps des communications. Pour s'en convaincre, il suffit de regarder les termes dominants en τ de (1.11) et (1.12), pour constater qu'ils ne diminuent pas avec la taille de l'hypercube. Ils resteront constants et minimisés si l'on utilise les P processeurs pour toutes les diffusions.

Nous avons vu que le choix de la fonction de répartition pour DParNId pouvait être quelconque. Pour DParID, on minimisera le coût arithmétique indépendemment de celui des communications (d'après la remarque) en prenant une répartition par blocs de tailles 1, soit (1.2) ou (1.3). Les coûts ont été donnés au §I.1.3.

Dans le pire cas des communications, à chacune des n étapes, en plus du pivotage il y a la diffusion d'une ligne de longueur n–k pour DParNId et $n+\gamma(n-k)$ pour DParId, où $0 \leq \gamma \leq 1$ de même que pour l'arithmétique. En sommant (1.12), (1.13) et (1.14) pour conserver l'ordre des lignes, les bornes sur le coût sont de l'ordre de,

$$\begin{cases} CDNId_{pire} = n^2 \left(\frac{2}{\log_2 P} + \frac{1}{2}\right) \tau + 5n(\log_2 P)\, \beta \\ CDId_{pire} = n^2 (2+\gamma) \left(\frac{2}{\log_2 P} + \frac{1}{2}\right) \tau + 5n(\log_2 P)\, \beta \end{cases} . \qquad (1.15)$$

Inversement, le pire cas pour l'arithmétique fournit les valeurs minimales du coût des transferts,

$$\begin{cases} CDNId_{mieux} = \frac{n^2}{2}\, \tau + n(\log_2 P)\, \beta \\ CDId_{mieux} = \frac{n^2}{2} (2+\gamma)\, \tau + n(\log_2 P)\, \beta \end{cases} . \qquad (1.16)$$

Il apparaît donc clairement que l'utilisation de la matrice identité pénalise les communications d'un facteur supérieur à 2.

Remarque 2.2
Nous avons donné (1.15) et (1.16) dans le cas où l'on conserve l'ordre des lignes. Ce qui peut en plus de ce que nous avons dit précédemment, assurer pour DParId une meilleure répartition des calculs en cours d'exécution : les processeurs sont pivots à tour de rôle, le nombre de lignes sur lesquelles on opère décroit uniformément.

Les opérations de base de cet algorithme : le processus d'élection et la diffusion qui doivent être envisagées globalement sur le réseau, sont complexes et demanderont une programmation très soignée pour en réduire le coût. Comme nous allons le voir avec les prochains algorithmes une telle vision globale peut être évitée.

I.3 ALGORITHME DU PIPELINE

L'algorithme du pipeline [Saa 2, CTV 2] présenté ici n'utilise que les liaisons de voisins à voisins d'un anneau. Précisons que les indices des processeurs, une fois obtenus par une opération, sont à prendre modulo P (ex: P_{proc+1} signifie $P_{(proc+1)modP}$). Les versions parallèles de SeqId (PParId) et SeqNId (PParNId), sont maintenant quelque peu différentes. En effet, pour l'algorithme PParId, si un processeur ne peut fournir de ligne pivot, aucune opération arithmétique ne sera à effectuer pour l'étape en cours dans ce même processeur (ce qui caractérise la triangularisation). Au contraire de PParNId : la ligne pivot devra de toutes façons rencontrer tous les processeurs pour éventuellement annuler les coefficients des lignes situées non seulement au–dessous mais aussi au–dessus de la diagonale (diagonalisation).

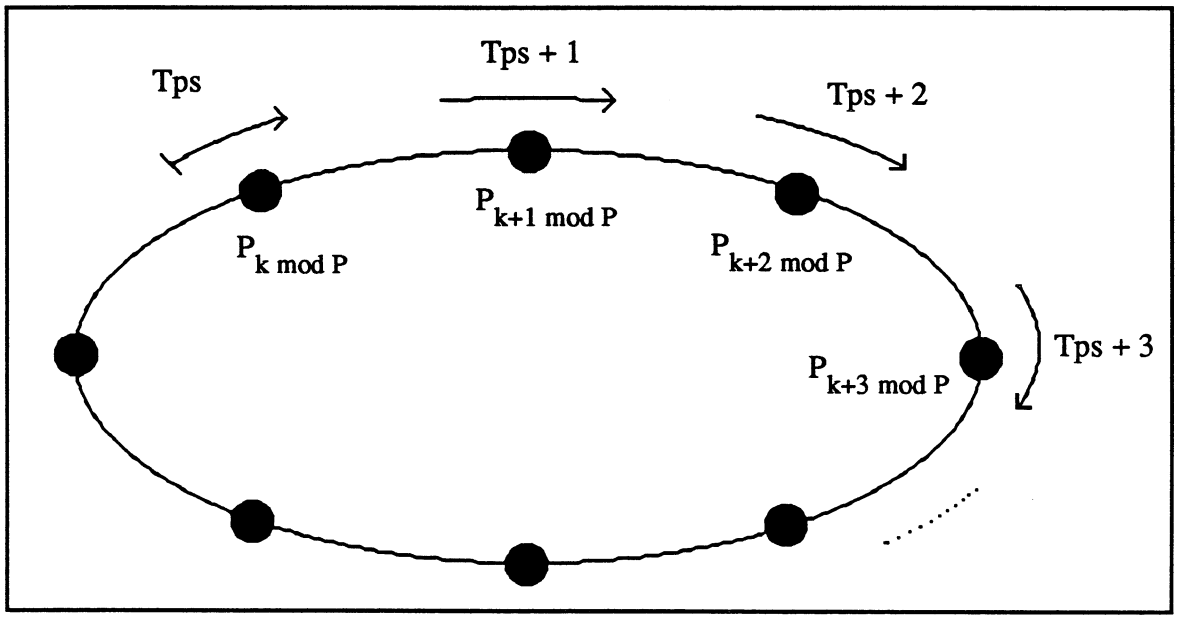


Figure I.2 : cheminement de la ligne pivot
à l'étape k de l'algorithme du pipeline.

• Avant de formaliser l'algorithme, décrivons l'étape k de la résolution. Chaque processeur reçoit la ligne pivot, la retransmet et effectue ses éliminations :

• Le processeur pivot P_{proc}, i.e. avec alloc(k)=proc,

1. envoie la ligne pivot sur l'anneau : à son successeur P_{proc+1}, et effectue ses éliminations.

2. S'il ne possède pas de ligne (n'ayant pas encore servi à éliminer) de coefficient de tête non nul, il transmet une ligne quelconque, une "mauvaise ligne" [CTV 2], puis se met en attente de P_{proc-1}. Une fois une ligne reçue, l'étape est terminée pour PParId. Pour PParNId le processeur doit retransmettre la ligne, et éliminer si cette dernière le permet, les lignes situées au–dessus de la diagonale.

• Les autres processeurs reçoivent une ligne de leur prédécesseur. Si la ligne est de coefficient de tête non nul, ils peuvent effectuer les éliminations. Sinon le processeur transmet :

1. une bonne ligne avec laquelle il peut effectuer ses éliminations et termine l'étape.
2. La mauvaise ligne reçue. Et se met en attente : la situation est alors la même que pour le processeur pivot.

• Permutations

Les permutations se font de mémoire à mémoire mais par simples échanges d'une ligne en mémoire avec une ligne reçue : aucun coût de communication supplémentaire n'est donc demandé [CTV 2]. Ceci assure que l'ordre des lignes de la matrice est conservé.

• Terminaison de l'algorithme

• A une étape k donnée de PParId seul le processeur $P_{alloc(k)}$, s'il reçoit une mauvaise ligne, est informé que le rang décroit (aucune ligne de coefficient de tête non nul n'a été trouvée sur l'anneau). Les variables ip et rang n'ont donc plus de sens. Nous notons donc iploc qui sera un indice local pour les lignes n'ayant pas encore servi à éliminer et rangloc qui décroit localement (et qu'il faudra sommer pour avoir le rang une fois l'exécution terminée). En particulier, le processeur pivot par défaut n'est plus déterminé par les indices de lignes (algorithme séquentiel et algorithme de diffusion) mais par k. Les lignes sont renumérotées au fur et à mesure : à chaque étape, le processeur pivot attribue un nouvel indice.

• Pour PParNId, quand après une première communication (figure I.5) les processeurs sont en attente pour éventuellement éliminer les lignes au dessus de la diagonale. Si $P_{alloc(k)}$ reçoit une mauvaise ligne, il peut ne pas retransmettre une ligne mais seulement *un jeton*, kchgt = vrai (changement d'étape) pour permettre aux autres processeurs de terminer. En particulier, tous les processeurs sont informés que le rang décroit : les variables ip et rang peuvent être conservées. Sinon la ligne pivot trouvée est envoyée après kchgt = faux.

• Dans tous les cas les messages transmis comporteront la ligne de la matrice mais aussi le numéro du processeur ayant envoyé le premier cette ligne sur le réseau : afin d'assurer la terminaison de chaque étape à moindre coût en situant un "avant dernier processeur" qui sait qu'il ne doit pas retransmettre.

PParId et PParNId sont donnés sur les deux pages suivantes. Précisons que les instructions "envoyer" et "recevoir" correspondent toujours à des communications sur l'anneau, d'un processeur vers son successeur, ou en provenance de son prédécesseur.

Et sans toutefois l'expliciter dans le programme des processeurs donné ci–dessous, puisque cela ne correspond pas à notre modèle, remarquons que la retransmission des lignes pourrait toujours se faire en parallèle aux calculs arithmétiques.

- **Algorithme PParId** *Pipeline Parallèle avec Identité*

début
iploc = 1
rangloc = 0
pour k de 1 à n

Programme du processeur $P_{alloc(k)}$
si le processeur possède une ligne d'indice local i ≥ iploc et de coefficient de tête non nul alors trouvé = vrai
sinon trouvé = faux
envoyer la ligne trouvée (ou une ligne quelconque) : (ligne, alloc(k))

si trouvé = vrai
élimination des lignes en mémoire
iploc = iploc + 1
sinon
recevoir : (ligne, n°) *Ligne pivot trouvée sur l'anneau*
si la ligne reçue est de coefficient de tête non nul
remplacer en mémoire la ligne qui a été envoyée par la ligne reçue
iploc = iploc + 1 *Il y a permutation*
élimination des lignes au dessus de la diagonale
sinon ranglog = rangloc - 1

Programme des autres processeurs : P_{proc}
recevoir : (ligne, n°)
si la ligne reçue est de coefficient de tête non nul
si (proc ≠ n° - 1) retransmission de (ligne, n°)
élimination des lignes en mémoire

sinon
recherche d'une ligne d'indice local i ≥ iploc et de coefficient de tête non nul
si trouvé = vrai *Il y a permutation*
la nouvelle ligne pivot est remplacée en mémoire par la ligne reçue
envoi de la ligne pivot : (ligne, proc)
élimination des lignes en mémoire
sinon
retransmission de la ligne qui a été reçue : (ligne n°)
fin

- **Algorithme PParNId** *Pipeline Parallèle Non Identité*

début

ip = 1
rang = n
pour k de 1 à n
Programme du processeur $P_{alloc(k)}$
si le processeur possède une ligne d'indice i ≥ ip et de coefficient de tête non nul
alors trouvé=vrai sinon trouvé=faux
envoyer la ligne trouvée (ou une ligne quelconque) : (ligne, alloc(k))
si trouvé=vrai
élimination des lignes en mémoire
ip = ip + 1
sinon
recevoir : (ligne, n°)
si la ligne reçue est de coefficient de tête non nul
remplacer en mémoire la ligne qui a été envoyée par la ligne reçue
si (alloc(k) ≠ n°-1)
envoyer kchgt = faux et (ligne, n°)
élimination des lignes d'indices inférieurs à ip
ip = ip + 1
sinon envoyer kchgt = vrai et rang = rang - 1
Programme des autres processeurs : P_{proc}
recevoir : (ligne, n°)
si la ligne reçue est de coefficient de tête non nul
si (proc ≠ n°-1) retransmission de (ligne, n°)
élimination des lignes en mémoire, ip = ip + 1
sinon
recherche d'une ligne d'indice i ≥ ip et de coefficient de tête non nul
si touvé = vrai *Il y a permutation*
la nouvelle ligne pivot est remplacée en mémoire par la ligne reçue
envoi de la ligne pivot : (ligne, proc)
élimination des lignes en mémoire, ip = ip + 1
sinon
retransmission de la ligne qui a été reçue : (ligne, n°)
réception de kchgt
si kchgt = faux, réception de (ligne, n°),
si (proc ≠ n°-1) envoi de kchgt = faux et retransmission de (ligne, n°)
éliminations au dessus de la diagonale, ip = ip + 1
sinon
rang = rang - 1

fin

• Les coûts des algorithmes

• Intéressons nous d'abord, pour le coût des communications de PParId et de PParNId, à la situation dans laquelle aucun pivot nul n'est rencontré. Le problème a été traité dans [RTV], d'après les conclusions des auteurs, il nous est possible d'énoncer les deux propositions suivantes.

Proposition 3.1

La borne minimale (relativement à l'anneau) pour le coût des communications de PParNId est atteinte, si la répartition des lignes est la répartition par blocs de tailles n/P (1.4). Cette borne est de l'ordre de,

$$\boxed{CPNId_{mieux} = (n^2 + P)\,\tau + 2n\,\beta}. \qquad (1.17)$$

Cette quantité correspond au coût de la transmission à chaque étape, d'une ligne de longueur n–k. Plus un temps identique de synchronisation. Le coût arithmétique correspondant à été donné en (1.5). Le coût total est minimisé.

On ne sait pas calculer le coût des communications de PParId pour une répartition quelconque des lignes. Mais il peut être donné sous les hypothèses qui ont permis d'obtenir le coût arithmétique en (1.7). C'est l'objet de la proposition suivante.

Proposition 3.2

Le coût des communications de l'algorithme PParId, si la répartition des lignes est par blocs de taille r, est de l'ordre de $(0 \le \gamma \le 1)$,

$$\boxed{CPId_{mieux} = [\, n^2\,(2+\gamma)(1+\frac{1}{2r}) + 2n\,(\gamma P - r) - \gamma\frac{P^2 r}{2} + O(n)]\,\tau + [2n - r + n/r]\,\beta}. \qquad (1.18)$$

Reprenons la démonstration du théorème 2 de [RTV] pour démontrer cette relation. Le coût se décompose en un temps *d'initialisation du pipeline* $T_1\tau+(P-2)\beta$, un temps de *receptions et d'envois* $T_2\tau+2(n-r+1)\beta$, et un temps $T_3\tau+(n/r-P+2)\beta$ de *latence* équivalent à la transmission d'une ligne toutes les r étapes. Plus précisément,

$$T_1 = (P\text{-}2)\,(n + \gamma\, n),\ T_2 = 2\sum_{k=1}^{n-r} n + \gamma\,(n\text{-}k) \quad \text{et, } T_3 = \sum_{k=0}^{n/r-P+1} n + \gamma\,(n\text{-}kr),$$

ce qui conduit au résultat.

Pour conclure sur le coût total de la méthode ($P=\alpha n$), il faut minimiser (1.7)+(1.18), en remarquant que (1.7) croit avec r pendant que (1.18) décroit, pour trouver la meilleure taille des blocs. En supprimant les termes en β, l'expression formelle de r correspondante, racine d'une équation de degré 3, dépend de n, P, τ et du coût τ_a d'une opération arithmétique. L'expliciter ne présente pas grand intérêt. Nous avons par contre tracé les valeurs des termes dominants de (1.7)+(1.18) quand r varie. En prenant n=1024, P=16, γ=0 et un rapport τ/τ_a=42 (rapport théorique pour le FPS T20, cf chap. III) nous avons représenté la courbe obtenue sur la figure I.3.

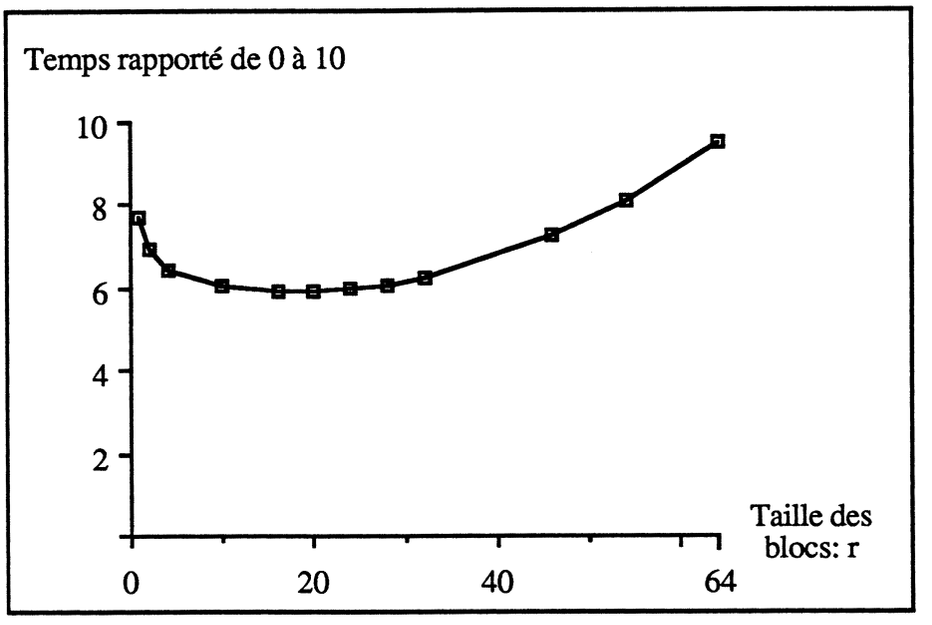


Figure I.3 : Temps d'exécution théoriques ($P=\alpha n$) de PParId pour différentes répartitions des lignes de la matrice.

Elle présente un minimum quand les lignes sont stockées par groupes de 20. En pratique on prendrait 16 puisque r doit diviser n/P. Soulignons que cette courbe supporte bien la comparaison avec les résultats expérimentaux de [RTV].

Rappelons que si n>>P, on minimise le coût arithmétique avant celui des communications. Pour r allant de 1 à n/P, le premier (γ=0) varie de $3n^3/2P\tau_a$ à $3n^3/P\tau_a$, et le second (γ=0) de $2n^2\tau$ à $3n^2\tau$. Il faut donc prendre r=1.

En conclusion, l'arithmétique et les communications de PParId n'atteignant jamais ensemble leur coût minimum, cet algorithme sera nettement moins performant que PParNId si la matrice est telle qu'aucun pivot est nul.

• Il nous faut maintenant mettre en évidence les plus mauvais cas. Si la valeur de γ est fixée. Le coût des communications de PParId ne dépend pas de la matrice considérée.

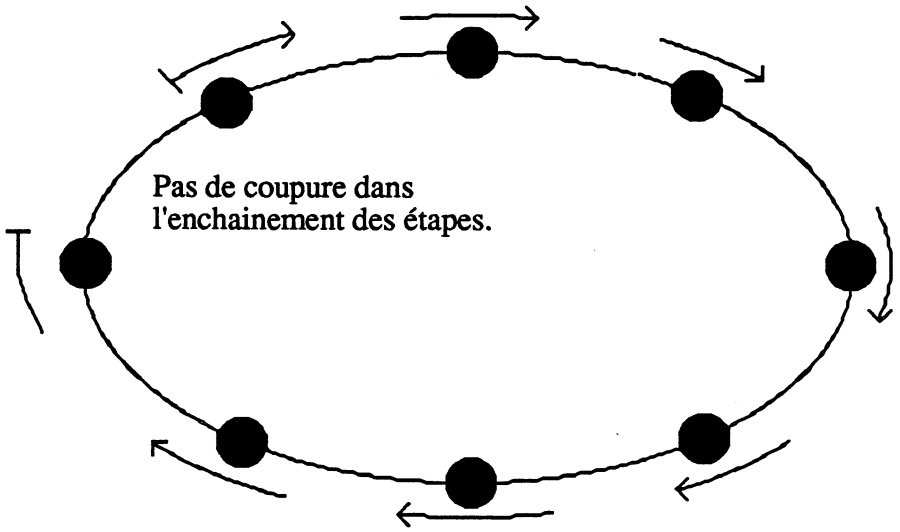


Figure I.4 : Algorithme du pipeline, PParId.
Cheminement de la ligne envoyée par le processeur $P_{alloc(k)}$.

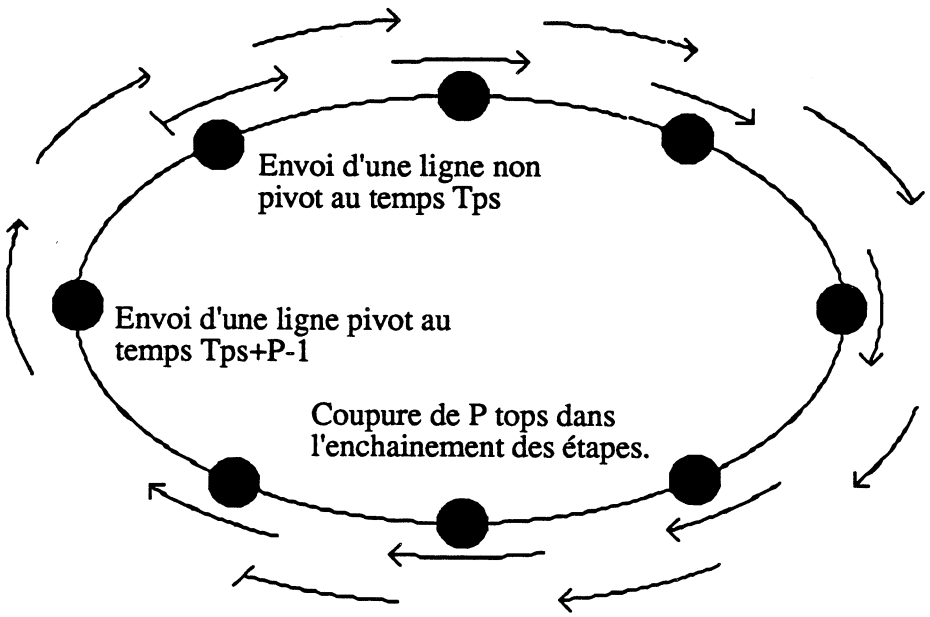


Figure I.5 : Algorithme du pipeline, PParNId.
La ligne pivot n'est trouvée que dans le processeur $P_{alloc(k)-1}$!

Et la quantité donnée par (1.18) n'est jamais dépassée : on a vu que si un processeur ne fournit pas de ligne pivot, il n'a pas non plus d'opération arithmétique à effectuer, il peut donc passer à l'étape suivante.

Contrairement à cela durant PParNId, un processeur ne fournissant pas de ligne pivot doit se mettre en attente pour éliminer les lignes au dessus de la diagonale. Un très mauvais cas se présente alors. Comme on peut le voir sur la figure I.5, si à chacune des étapes k la ligne pivot n'est trouvée que dans le processeur $P_{alloc(k)-1}$, l'enchainement des étapes est interrompu pendant l'équivalent de P réceptions. Le processeur pivot de l'étape suivante, doit attendre un tour complet de l'anneau. On obtient avec r=n/P,

$$\boxed{CPNId_{pire} = \frac{(P+2)\ n^2}{2}\tau + (2+P)n\ \beta}. \qquad (1.19)$$

Valeur qui est supérieure à $3n^2\tau$ (le pire coût si l'identité est stockée) dès que l'on utilise 8 processeurs.

Même si ce cas reste bien particulier, on peut penser que si le nombre premier est petit (beaucoup de coefficients de la matrice sont nuls) PParId sera meilleur que PParNId, du moins pour les communications : les contraintes liées à l'élimination des lignes situées au dessus de la diagonale jouent un rôle important.

Au contraire, si peu de coefficients nuls sont rencontrés, nous avons vu que ne pas stocker la matrice identité permet de minimiser, non seulement le coût arithmétique, mais aussi celui des communications.

En règle générale, la comparaison des différents algorithmes dépend donc étroitement de la structure des matrices : comment conclure pour les situations intermédiaires, si ce n'est expérimentalement ?

Quand conserver l'ordre n'est pas demandé, les coûts ci–dessus pourront parfois être réduits en ne transférant pas de ligne quelconque (de coefficient de tête nul) mais seulement un jeton l'indiquant. Cette technique est utilisée au chapitre suivant pour l'algorithme à pivots locaux (pour lequel il est beaucoup trop contraignant de conserver l'ordre). On pourra se reporter à l'étude de complexité correspondante pour les pires cas.

CHAPITRE II

ALGORITHMES A PIVOTS LOCAUX

On a vu à l'étude des algorithmes précédents, que les processeurs restent inoccupés à attendre une ligne pivot globale (i.e. la même pour tous). Une façon [CTV 2] de diminuer le coût en communications de la résolution (tout en restant sur un anneau) est d'anticiper sur les calculs. Dans chaque processeur, on peut choisir une ligne pour commencer les éliminations, la ligne pivot locale, tout en attendant une ligne d'un processeur voisin : le prédecesseur sur l'anneau, pour finalement éliminer cette ligne pivot locale.

Le programme de chacun des processeurs P_{proc}, pour le principe de complétude, a donc maintenant la structure,

début
pour k = 1 à n
(C1) : Echanges de donnée. A la fin de cette phase, chaque processeur a reçu tous les coefficients d'une ligne i.
(C2) : Calculs. Le processeur effectue les éliminations avec une de ses propres lignes ou avec la ligne reçue.
fin.

Remarquons que cela ne modifie en rien les résultats du chapitre précédent. Les étapes de calcul sont, bien qu'un peu plus complexes, tout à fait analogues aux étapes d'une élimination standard. De même, les matrices intermédiaires gardent la structure habituelle : commençons par supposer, qu'au début de chaque étape k, aucun coefficient de tête nul (i.e. dans la colonne k) n'est rencontré, et étudions le début de l'exécution. Chaque processeur entame les éliminations avec une ligne qu'il possède en mémoire, la matrice est donc mise sous la forme,

$$\begin{pmatrix} x & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \\ x & x & x & \dots & x & x & x & x \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & x & x & \dots & x & x & x & x \\ x & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \end{pmatrix}, \qquad (f1)$$

elle conserve exactement P coefficients non nuls dans la première colonne : ils correspondent au lignes pivots locales. Et sont donc situés dans des processeurs distincts. Une deuxième phase

consiste pour tous les processeurs sauf un, le processeur pivot (global), à procéder à l'élimination de ces coefficients, à l'aide de la ligne qu'ils ont reçue de leur prédecesseur sur l'anneau. On aboutit donc facilement à la matrice voulue,

$$\begin{pmatrix} x & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \\ 0 & x & x & \dots & x & x & x & x \end{pmatrix}. \qquad (f2)$$

Ce raisonnement peut être appliqué à toutes les étapes de la résolution, et montre, en particulier, que la matrice finale est triangulaire ou diagonale suivant la méthode choisie.
Plus généralement, quand un processeur P_{proc} ne trouve pas de ligne pivot locale (si tous les coefficients de tête sont nuls ou si, pour ses lignes, l'algorithme est terminé) :

• la première phase du calcul, à une étape donnée, ne pose aucun problème pour les autres processeurs : cette phase est locale. C'est l'obtention d'une matrice analogue à (f1).

• Il faut ensuite assurer le passage de la forme (f1) à la forme (f2). Si le rang ne décroit pas, chaque processeur a dû recevoir une ligne pour éliminer sa ligne pivot locale : l'idée est de procéder sur une partie ou la totalité du réseau comme pour l'algorithme du pipeline. Le processeur doit attendre pour la retransmettre une ligne de P_{proc-1}, et permettre ainsi les éliminations dans P_{proc+1}. Les différences mises en évidence dans le précédent chapitre entre les versions parallèles de SeqId et SeqNId vont donc se retrouver ici pour PlParId et PlParNId. De même peuvent être reprises les techniques utilisées pour terminer les étapes quand le rang décroit.

• Avant de détailler les programmes des processeurs, montrons le fonctionnement de la résolution sans la matrice identité sur un anneau de taille 3, et pour une matrice A d'ordre 6 dans $\mathbb{Z}/5\mathbb{Z}$:

$$\begin{pmatrix} 2 & 3 & 0 & 1 & 1 & 0 \\ 1 & 2 & 0 & 2 & 3 & 1 \\ 3 & 2 & 3 & 2 & 1 & 1 \\ 1 & 2 & 3 & 0 & 0 & 2 \\ 2 & 1 & 4 & 3 & 1 & 4 \\ 1 & 3 & 4 & 4 & 3 & 1 \end{pmatrix},$$

la matrice obtenue en fin d'algorithme (sa forme a été donnée dans l'introduction (i.3)), étant :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

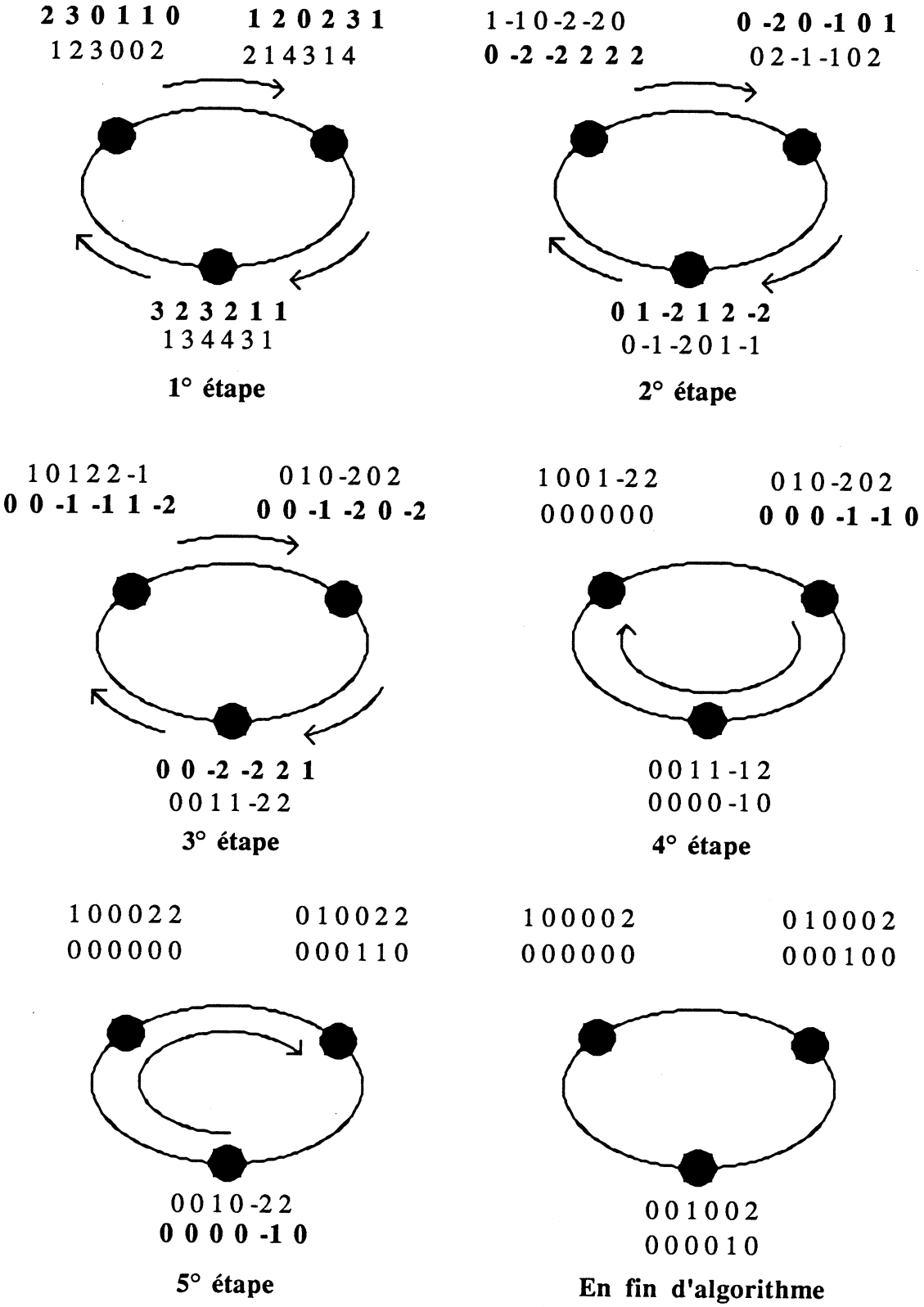


Figure II.1 : Les étapes de l'algorithme à pivots locaux NId sur un anneau de taille 3 et pour une matrice 6*6́, p=5.

Aucune permutation de colonnes n'étant nécessaire pour avoir la matrice identité de d'ordre 5 (i.e. le rang de A), l'ensemble J_A (des indices des colonnes qui participent à la matrice identité de dimension 5) est simplement (1,2,3,4,5). Il est alors immédiat (i.4) de construire un vecteur qui engendre le noyau, en égalant à -1 le coefficient de la dernière colonne et de la ligne dont tous les coefficients sont nuls,

$$z = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 0 \\ 0 \\ -1 \end{pmatrix}.$$

Les étapes de la résolution sont représentées sur la figure II.1. La sixième consistant seulement en un test d'égalité à zéro (en l'occurence ici sur le coefficient [5,5]), et en la normalisation de la 5° ligne. Les lignes pivots locales sont données en gras. Chacune permet d'éliminer non seulement les lignes du processeur où elle réside. Mais aussi, après son transfert (figuré par les flèches de longueur 1), la ligne pivot locale du processeur suivant. On remarquera des flèches de longueur 2 quand un processeur n'a pas de ligne de coefficient de tête non nul.

• A chaque étape k, une des lignes ne doit bien sûr pas être modifiée. Ce sera dans le processeur $P_{alloc(k)}$ ou dans le premier rencontré sur l'anneau qui possède une ligne de coefficient de tête non nul.
A la différence de l'algorithme du pipeline, les permutations ne sont pas effectuées en mémoire : la ligne pivot ne fera que rarement un tour complet de l'anneau. Un processeur n'ayant pas de ligne pivot n'enverra qu'un simple jeton le signifiant. En particulier, l'ordre des lignes de la matrice n'est pas conservé.

De façon très générale on a donc :

• quand le processeur P_{proc} possède en mémoire une ligne pivot locale L, il la transmet à son voisin de droite et commence les éliminations. S'il en reçoit une du processeur précédent, en provenance d'un processeur d'indice compris entre alloc(k) et proc–1, il peut alors éliminer la ligne L. Dans le cas contraire, la ligne L apparaitra dans la matrice finale.

• si aucune ligne pivot locale n'est trouvée, le processeur se contente de retransmettre les messages qu'il reçoit. Et effectue des éliminations (au dessus de la diagonale) le cas échéant.

Suivant PlParNId ou PlParId, le processeur $P_{alloc(k)-1}$ retransmet ou non les informations. De même, les processeurs restent ou non en attente (pour éliminer au dessus de la diagonale) s'ils n'ont pas de ligne pivot.

On note que la plupart des communications pourraient se faire parallèlement aux calculs. Les envois et réceptions pouvant eux–même s'effectuer simultanément (le constructeur PAR le spécifie dans les programmes donnés ci–dessous).

• Algorithme PlParId *Pivots locaux Parallèles avec Identité*

début

iploc = 1
rangloc = 0
pour k de 1 à n
si le processeur possède une ligne d'indice local supérieur à iploc et de coefficient de tête non nul alors trouvé = vrai
sinon trouvé = faux

Programme du processeur $P_{alloc(k)}$
si trouvé = vrai
envoyer trouvé puis la ligne
élimination des lignes en mémoire
iploc = iploc + 1
sinon envoyer trouvé

Programme des autres processeurs : P_{proc}
si trouvé = vrai
PAR
si (proc ≠ alloc(k)-1 mod P) *Ce n'est pas l'avant dernier processeur*
envoyer "vrai" puis la ligne
recevoir jeton et si jeton = vrai recevoir la ligne

élimination des lignes en mémoire à l'aide de la ligne iploc
si jeton = faux
iploc = iploc + 1 *Il y a permutation virtuelle*
sinon éliminer la ligne iploc
sinon
recevoir le jeton
si jeton = vrai
recevoir la ligne
si (proc ≠ alloc(k)-1 mod P) envoyer "vrai" puis la ligne
sinon
si (proc ≠ alloc(k)-1 mod P) envoyer "faux"
sinon rangloc = rangloc - 1 *Le rang décroit*

fin

- **Algorithme PlParNId** *Pivots locaux Parallèles Non Identité*

début

ip = 1, rang = n
pour k de 1 à n
compt = 0, perm = 0
si le processeur possède une ligne d'indice supérieur à ip et de coefficient de tête non nul alors trouvé = vrai
sinon trouvé = faux
si trouvé = faux
tant que ((jeton ≠ vrai) et (jeton < P-1))
PAR
recevoir le jeton
compt = compt + 1, envoyer compt
si jeton = vrai
PAR
recevoir la ligne
si (compt ≠ P-1) envoyer "vrai" et la ligne
ip = ip + 1 *Indice pivot global*
éliminations au dessus de la diagonale
sinon rang = rang - 1 *Le compteur est égal à P-1 dans tous les processeurs*

sinon

Programme du processeur $P_{alloc(k)}$
PAR
envoyer "vrai" puis la ligne
tant que ((jeton ≠ vrai) et (jeton < P-1))
recevoir le jeton
si jeton = vrai recevoir la ligne
éliminations des lignes en mémoire
ip = ip + 1
Programme des autres processeurs : P_{proc}
PAR
envoyer "vrai" puis la ligne
tant que ((jeton ≠ vrai) et (jeton < P-1))
recevoir le jeton
si jeton = (proc-alloc(k)) mod P
perm = 1 *Il y a permutation virtuelle*
si jeton = vrai recevoir la ligne
éliminations des lignes en mémoire à l'aide la ligne pivot locale, ip = ip + 1
si perm ≠ 1 éliminer la ligne pivot locale

fin

Quand le rang décroit

• **PlParId** : afin d'assurer la terminaison de l'étape k, le processeur $P_{alloc(k)}$ s'il n'a pas de ligne pivot envoie néanmoins un jeton (égale à *faux* par exemple). Le processeur $P_{alloc(k)-1}$ recevant ce jeton sera informé que le rang décroit.

• **PlParNId** : tout processeur n'ayant pas de ligne pivot envoie compt=1 à son successeur. De même s'il reçoit une quantité différente de "vrai" (pas de ligne pivot), il l'incrémente avant de la retransmettre. Recevant un jeton de valeur (P-1) tous les processeurs seront informés que le rang décroit.

Les remarques faites au paragraphe précédent au sujet des variables ip et iploc restent valables.

• Les coûts des algorithmes

• Si aucun pivot local nul n'est rencontré, c'est à dire qu'à chaque étape chaque processeur possède une ligne de coefficient de tête non nul (on négligera les P dernières étapes de l'algorithme), aucune ligne ne sera transmise à plus d'un processeur. Pendant la phase de communication, les processeurs sont synchronisés. Le coût des communications s'obtient directement d'après [CTV 2] et consiste donc essentiellement en n réceptions (et envois en parallèle). Le terme en τ de (1.17) est réduit d'un facteur 2,

$$\boxed{CPlNId_{mieux} = \frac{n^2}{2}\tau + n\,\beta}. \qquad (2.1)$$

Et le choix de la fonction d'allocation n'a pas d'influence si la matrice identité est utilisée,

$$\boxed{CPlId_{mieux} = (2+\gamma)\,\frac{n^2}{2}\tau + n\,\beta}. \qquad (2.2)$$

Il n'est donc pas utile de revenir sur les coûts arithmétiques correspondants à cette situation.

• D'autre part, il est immédiat que le coût des communications du plus mauvais cas d'un algorithme à pivots locaux restera inférieur au coût du plus mauvais cas de la forme correspondante de l'algorithme du pipeline.

Si à chacune des étapes k de l'exécution de PlParId, seul le processeur $P_{alloc(k)}$ trouve une ligne pivot locale : le coût est celui de PParId. Nous avons vu que ce dernier était indépendant de la matrice quand γ est fixé,

$$\boxed{CPlId_{pire} = CPId_{mieux}}. \qquad (2.3)$$

Nous ne reprenons pas la discussion sur le choix de la répartition des lignes, ses conclusions doivent être appliquées quand (2.2) ne peut être atteinte.

Concernant PlParNId, la situation que nous avions envisagée pour PParNId au chapitre précédent (figure I.5), ne peut plus l'être ici. Mais si, à chacune des étapes k, la seule ligne pivot est trouvée dans le processeur (n-k) mod P, on obtient une quantité du même ordre que celle donnée par (1.19),

$$\boxed{CPlNId_{pire} = \frac{P-1}{2} n^2 \tau + (P-1)n \beta}. \qquad (2.4)$$

Remarque 2.1

• Aussi bien pour un algorithme du type *pipeline* que pour un algorithme à pivots locaux, les coûts pourraient être réduits en considérant des communications dans les deux sens sur l'anneau à partir du processeur $P_{alloc(k)}$. Avec un gain d'au plus un facteur 2 pour les communications. Nous avons pour cela préféré rester dans le cas général.

COMPARAISON DES ALGORITHMES

Concluons ici d'un point de vue théorique. Nous ne reviendrons pas sur les coûts arithmétiques, ils sont tous du même ordre pour n>>P avec dans chacun des cas, la meilleure fonction d'allocation des lignes. Et si P=αn, une discussion est seulement nécessaire pour PParId.

Pour les coûts des communications, les termes dominants en $n^2/2\ \tau$ sont réunis dans le tableau ci–dessous. Les deux colonnes correspondent aux situations extrêmes que nous avons eu l'occasion d'exposer.

	Au pire	**Au mieux**
Diffusion		
DNId	$4/\log_2 P + 1$	1 ($1/\log_2 P$ si $n\tau >> \beta\log_2 P$)
DId (γ=0)	$2\ (4/\log_2 P + 1)$	2
Pipeline		
PNId	P	2
PId (γ=0)	entre 4 et 6	entre 4 et 6
Pivots locaux		
PlNId	P	1
PlId (γ=0)	entre 4 et 6	2.

Tableau II.1 : Récapitulatif des coûts des communications.

Si l'on ne rencontre aucun pivot nul au cours de l'exécution (ce qui sera facilement vérifié en pratique en prenant p suffisament grand), il est clair que le stockage de l'identité n'apporte rien : il augmente, au contraire, la durée des transferts dans un facteur supérieur à 2.
La méthode à utiliser est alors la diffusion quand la condition $n\tau >> \beta\log_2 P$ peut être approchée. Son coût est asymptotiquement optimal en τ. Le caractère local des communications de PlParNId donne sinon de meilleurs résultats : il ne demande aucun contrôle.

Dans le cas général, les coûts des communications de PParNId et PlParNId peuvent aller jusqu'à $Pn^2/2\ \tau$. Au contraire de ceux des autres algorithmes, dont les termes dominants restent indépendants du nombre de processeurs.

Enfin, comme nous l'avions fait remarquer, ces commentaires s'appliquent aussi à la triangularisation (coûts des méthodes Id divisés par 2) et à l'inversion par diagonalisation (coûts des méthodes NId avec la matrice identité bordante).

CHAPITRE III

IMPLANTATION DES ALGORITHMES

Afin de limiter la complexité du réseau d'interconnection d'une machine MIMD à mémoire distribuée, deux impératifs sont à prendre en compte. Le nombre de canaux de communication doit être faible par rapport au nombre de processeurs. Et les coûts des transferts doivent rester raisonnables. Une solution efficace consiste à connecter les processeurs suivant un réseau en hypercube.

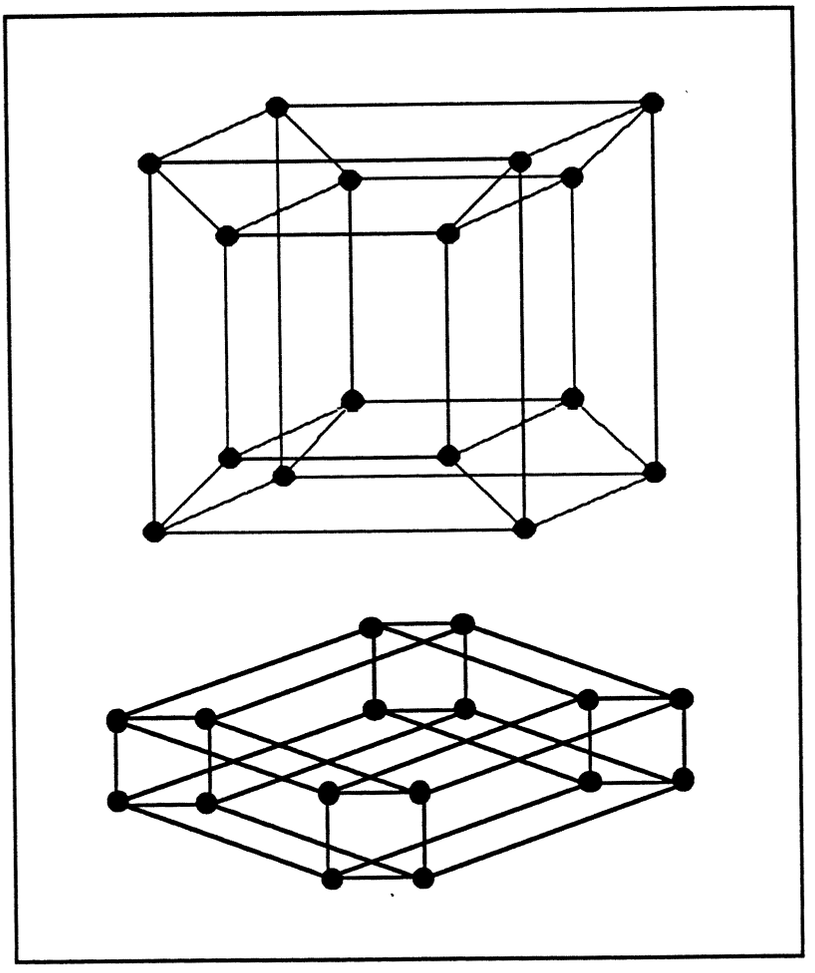


Figure I.1 : Hypercubes de dimension 4.

Cette solution, proposée par C.Seitz [Sei], a conduit à la commercialisation du Cosmic Cube, premier calculateur de ce type. Plusieurs machines sont maintenant disponibles sur le marché : Ametek, Intel iPSC, Mark III, NCUBE entre autres. Et le FPS T20, sur lequel nous avons testé

nos algorithmes. Après une brève description de son architecture (tout détail pourra être trouvé dans la documentation fournie par Floating Point Systems [FPS]), nous expliquons dans le premier paragraphe, comment nous avons utilisé sa puissance de calcul vectoriel sur des réels flottants, pour opérer modulo p. Et terminons en donnant des résultats expérimentaux qui nous obligent parfois, de revenir sur les conclusions théoriques.

III.1 ARITHMETIQUE VECTORIELLE DE $\mathbb{Z}/p\mathbb{Z}$

• Le T20 de Floating Point Systems [GHS] consiste essentiellement en 16 nœuds (unités opérationnelles) identiques, connectés pour former un hypercube de dimension 4, et communiquant par échanges de messages. Il n'existe pas méchanisme de synchronisation globale.

Un nœud de l'hypercube est lui–même formé de trois parties principales :

• Un *transputer* [WS] qui assure les communications, la fonction Unité Arithmétique et Logique, et le contrôle du programme.

• Une mémoire video, la VRAM de 1 MégaOctets à deux accès : aléatoire vers le transputer et série (par l'intermédiaire de registres à décalage) vers l'unité vectorielle.

• Une unité de calcul vectoriel VPU (Vector Processing Unit) assure des calculs en virgule flottante très rapides, pour des vecteurs de réels codés sur 64 bits. Elle est composée d'un additionneur et d'un multiplieur pipelinés, et permet d'effectuer une addition directement à partir des résultats d'un produit (i.e. sans repasser par les registres).

Performances théoriques

• Pour le calcul, chaque processeur peut effectuer 7.5 MIps et 12 MFlops simultanément, soit **120 MIps** et **192 MFlops** théoriques pour la machine complète.

• Et peut envoyer et recevoir en parallèle, 692 KOctets par canal et par seconde, soit 5 MOctets pour les quatre canaux en communication bidirectionnelle.

Communications

Les communications s'effectuent de voisin à voisin, suivant un protocole basé sur le "rendez–vous" des deux processeurs concernés. Différentes mesures nous permettent de considérer que le temps d'initialisation d'un canal est $9.5\mu s \leq \beta \leq 60\mu s$ suivant le nombre de liens utilisés simultanément (version B01 du logiciel FPS, [CTV 1]) ou $\beta \geq 710\mu s$ (version C00 [KT]), et que le temps élémentaire pour 64bits est $\tau = 11.5\mu s$. Un envoi étant le transfert de L données *consécutives en mémoire*.

L'unité de calcul vectoriel

Contrairement au transputer qui ne traite que des mots de 32 bits, l'unité de calcul vectoriel traite seulement des réels codés sur 64 bits au format virgule flottante IEEE (aucun traitement de vecteurs booléens ou entiers). Elle consiste en un additionneur et un multiplieur pipelinés (5 et 6 étages) accessibles, à partir de la VRAM, par l'intermédiaire de registres à décalage (1 Ko).
Une utilisation optimale en coût de l'unité de calcul, implique une importante gestion mémoire, en particulier,

- il faut découper les vecteurs de plus de 128 réels (la taille des registres), tout en minimisant le nombre de chargements et de déchargements de ces derniers vers la mémoire. Le traitement s'effectue alors en plusieurs étapes.
- La mémoire est séparée en deux bancs A et B. Pour une opération à deux arguments, il faut s'assurer du placement d'un des vecteurs dans le banc A et de l'autre dans le banc B.
- Un travail supplémentaire, consiste enfin, à préserver l'intégrité des données en mémoire. Le déchargement d'un registre dans la VRAM (qui ne peut s'effectuer que par blocs de 128 réels), ne doit pas, par exemple, écraser certaines valeurs.

L'implantation d'algorithmes utilisant le VPU, demande donc de trouver d'abord un stockage de données, permettant de minimiser la gestion de la mémoire au cours de l'exécution. Si toutes les conditions sont remplis on pourra, en pratique, disposer d'une puissance de 10MFlops quand l'additionneur et le multiplieur travaillent simultanément (pour un produit scalaire par exemple).

- Si les résolutions dans $\mathbb{Z}/p\mathbb{Z}$ peuvent être programmées dans un premier temps en utilisant l'arithmétique entière des transputers, un gain de performance considérable est obtenu à l'aide des unités de calcul vectoriel. Le travail préliminaire à toute implantation est donc la mise au point d'une arithmétique vectorielle des corps finis. Nous présentons la démarche suivie sur le T20. Elle pourrait être facilement étendue à toute unité de calcul flottant, puisque les fonctions de base utilisées sont standards.

Les nombres premiers

Les coefficients des matrices doivent être codés au format standard 64 bits IEEE. Format pour lequel la mantisse est de 52 bits et l'exposant de 11 bits, le réel est alors obtenu comme suit :

valeur = 1.<mantisse>*2<exposant>-1023.

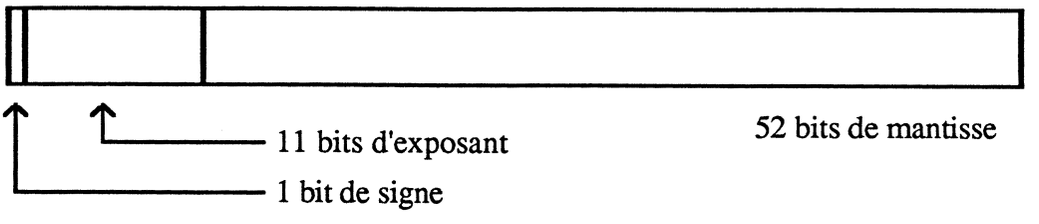


Figure III.2 : Format réel 64 bits IEEE.

On en déduit une borne supérieure pour les nombres entiers qui pourront être considérés en cours de calcul, c'est le plus grand entier μ tel que tout entier inférieur à μ peut être codé exactement. On a ici :

$$\mu = 2^{53}-1.$$

Les opérations modulo à effectuer étant des sommes et des produits, les nombres premiers possibles sont en fait donnés par la racine carrée de μ :

$$p \leq (2^{53}-1)^{1/2}$$

si l'on choisi de calculer le modulo après chaque opération.

Par souci d'efficacité, l'opération de base étant en fait un Saxpy (u-sv, u et v étant des vecteurs et s un scalaire), le modulo ne sera calculé qu'une fois terminés un produit et une somme, on a donc finalement la borne sur les nombres premiers que nous utiliserons :

$$\boxed{p \leq \sqrt{2^{52}-1} \approx 67108865} \quad . \qquad (3.1)$$

La partie entière

Si le codage d'un réel occupe 64 bits, les opérations sont en fait effectuées sur une longueur plus importante (des bits de sauvegarde dont le nombre varie d'une machine à une autre) sont toujours prévus. Une façon de calculer la partie entière d'un réel pourrait être, en connaissant le nombre de ces bits de sauvegarde, de jouer sur la normalisation qui est faite avant toutes les opérations (la partie décimale est écrasée par un décalage) :

$$n + x - x = \text{partie entière de } n.$$

La partie entière peut aussi être obtenue en utilisant les différents formats de codage. Le passage du format flottant au format entier du transputer à pour effet de tronquer le réel à l'entier supérieur ou inférieur (suivant la valeur du réel et l'implantation de la fonction de changement de format).

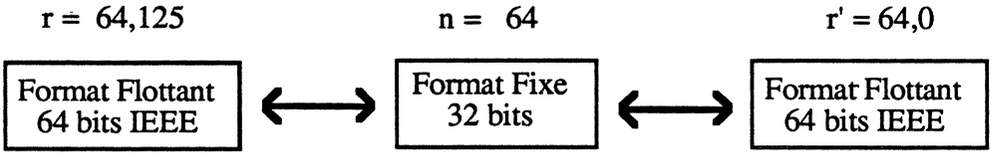


Figure III.3 : Calcul de la partie entière.

Sur le T20 les codes opératoires du passage flottant à fixe et inversement sont AF_XFIX et AF_XFLOAT (seul l'additionneur est utilisé). La partie entière peut donc se calculer en deux passages dans l'unité vectorielle. Remarquons aussi que les deux opérations peuvent être enchainées directement à partir des registres.

Le modulo

Une fois la partie entière disponible, le calcul de a mod p s'effectue en au plus **5 passages dans les opérateurs vectoriels**, l'inverse réel de p étant calculé par avance :

1. q_m = quotient réel de a par p = **produit** de a par l'inverse de p (1 passage),
2,3. e = **partie entière** de q_m (2 passages dans l'additionneur),
4. **produit** de p par e (1 passage),
5. le modulo est alors obtenu par un dernier passage pour la **soustraction** a-pe.

L'additionneur et le multiplieur pouvant travailler simultanément si l'on respecte certaines contraintes en agençant les vecteurs dans la mémoire [FPS], le coût réel en temps de cycle est le suivant pour le modulo d'un vecteur de longueur N :

1. et 2. enchainées soit $(e_+ + e_* + N-1)$ temps de cycle,
3. $(e_+ + N-1)$ temps de cycle,
4. et 5. enchainées soit $(e_+ + e_* + N-1)$ temps de cycle,

où l'additionneur et le multiplieur ont respectivement e_+ et e_* étages. Soit,

$$\boxed{(3e_+ + 2e_* + 3N-3) \quad \text{temps de cycle au total}} \; . \qquad (3.2)$$

La courbe ci–dessous nous montre les performances obtenues. L'implantation permet donc d'effectuer près de 1,8 millions de modulo par seconde (ce qui équivaut à 9 millions d'opérations flottantes par seconde) si les longueurs des vecteurs sont suffisamment importantes.

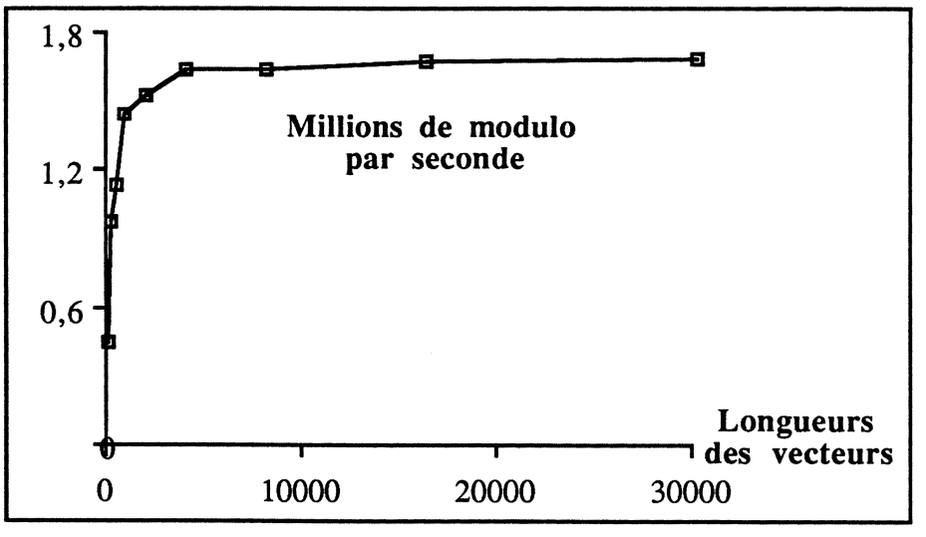


Figure III.4 : Implantation du modulo sur l'unité vectorielle.
Performances en millions de modulo par seconde.

Pour les dimensions de matrices que nous pourrons traiter, la longueur moyenne des vecteurs sera de 500, et la puissance du modulo limitée à 1,2 millions par seconde.

Représentation des éléments de $\mathbb{Z}/p\mathbb{Z}$

Le code opératoire AF_XFIX fourni par FPS, donne suivant la valeur du réel q_m, sa partie entière supérieure ou inférieure : e. On sait donc seulement que,

$$q_m \in [e, e+1] \text{ ou } q_m \in [e-1, e].$$

Si la précision du calcul de la division est 1/2r, on a alors,

$$a/p \in [e-1/2r, e+1+1/2r] \text{ ou } a/p \in [e-1-1/2r, e+1/2r],$$

et si $0<p\leq r$, a/p est un entier, ou est à distance au moins 1/r d'un entier, d'où,

$$a/p \in [e, e+1] \text{ ou } a/p \in [e-1, e].$$

• D'après ces relations [Mul] et l'inégalité (3.1), avec r donné par le format IEEE (fig.III.2), on sait donc que l'on ne commet pas d'erreur d'approximation pour e, et que l'on obtient la partie entière supérieure ou inférieure de q=a/p, si ce dernier n'est pas entier :

- si $e = \lceil a/p \rceil$, $0 < a - pe < p$, et
- si $e = \lfloor a/p \rfloor$, $-p < a - pe < 0$.

• Si q est entier, on peut avoir $q-1<q_m<q$ (resp. $q<q_m<q+1$) et obtenir q–1 (resp. q+1) après les deux passages dans l'additionneur. Donc,

- si $a \equiv 0 \bmod p$, $a - pe = 0$, p ou –p.

Afin de supprimer tout coût supplémentaire, lié à des tests et à de nouvelles opérations pour retrouver l'intervalle [0,p–1] ou [–p/2–1,p/2–1], nous pouvons alors choisir de coder les coefficients des corps finis de façon redondante : un nombre a non nul pourra être représenter par a ou par a–p,

$$\boxed{\mathbb{Z}/p\mathbb{Z} \approx [\,-p, p\,]}, \qquad (3.3)$$

et un test d'égalité (pour le pivotage ou pour savoir si la ligne est à éliminer) à zéro modulo p s'effectuera en testant si le codage du nombre est 0, p ou –p.

Les éléments des matrices sont donc codés au format IEEE pendant les résolutions. Mais la conversions des données du problème à ce format, et la conversion des résultats au format du transputer, pour être par exemple utilisés par l'arithmétique en précision infinie (cf partie 2, chap 4), peuvent se faire à moindre coût (de l'ordre de n^2 opérations) en utilisant à nouveau AF_XFIX et AF_XLOAT.

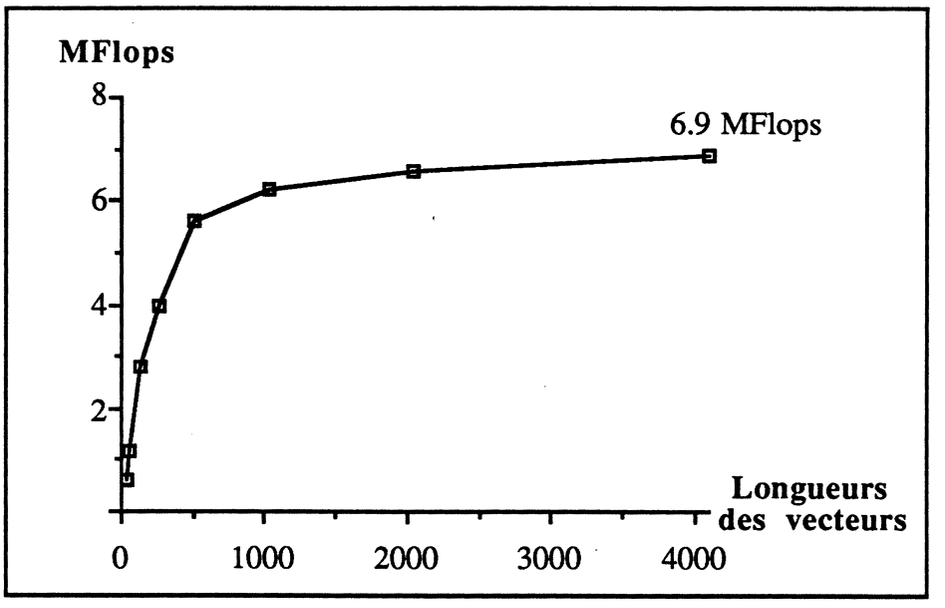


Figure III.5 : Implantation du Saxpy modulo p,
performances en millions d'opérations par seconde.

Opérations de base des algorithmes

Les deux opérations de base des algorithmes ne posent alors aucun problème. L'inverse modulo p d'un coefficient, est obtenu en intercalant entre les deux changements de formats utilisés pour la partie entière, le calcul de l'inverse sur le transputer. Le coût de cette opération scalaire serait prohibitif sur le VPU.
Enfin, la combinaison de deux lignes L_i et L de longueur N, un Saxpy modulo p,

$$L_i = L_i - C\,L \bmod p,$$

demande de l'ordre de 4N ou 5N temps de cycle, suivant que le produit et la soustraction sont enchainés ou non. Les performances de notre implantation sont données sur la figure III.5, le temps nécessaire à combiner deux lignes de longueur 512 est d'environ $6{,}4\ 10^{-4}$ secondes (5,6 MFlops).
Précisons pour terminer, que de façon analogue au coût des communications, le coût d'une opération vectorielle peut être modélisé à l'aide d'un temps d'initialisation β_a, qui en pratique est dû non seulement aux étages des pipelines mais aussi à la gestion de la mémoire. Et à l'aide d'un temps élémentaire de calcul τ_a. On a mesuré pour un Saxpy modulo p,

$$\boxed{\beta_a = 3.52\ 10^{-4}\ \text{s}\ \text{ et, }\ \tau_a = 9.7\ 10^{-7}\ \text{s}}\ . \qquad (3.4)$$

Remarquons que le rapport $\beta_a/\tau_a = 360$ est élevé. La longueur moyenne des vecteurs sur lesquels il opère influera donc considérablement le comportement d'un algorithme.

III.2 LES COMMUNICATIONS

Nous n'avons pas jusqu'à présent parlé des langages de programmation disponibles sur le T20 : ils n'ont en pratique aucune influence sur les performances de calcul de la machine. Il en est autrement pour les communications. Nous avons disposé pour nos algorithmes de deux versions du logiciel FPS :

- la première, la version B01, nous a permis de programmer en Occam [Inm] et de nous assurer de la validité du modèle pour le calcul des coûts des communications avec un rapport $\beta/\tau = 0,83$ "raisonnable".
- Sous la version suivante, C00, accompagnée d'un langage C standard augmenté de fonctions FPS pour les communications, le rapport des coûts de base est devenu $\beta/\tau = 61,73$. Et il s'est avéré difficile de gérer le parallélisme, sur un processeur donné, pour l'utilisation simultanée de plusieurs liens ainsi que pour les transferts bidirectionnels. Une des conséquence a été que nous n'avons pas programmé les algorithmes de diffusion mais utilisé les fonctions FPS.

- Il est intéressant de confronter les résultats des études théoriques menées pour les coûts des communications aux mesures que l'on a pu effectuer sur le FPS T20 (version B01).

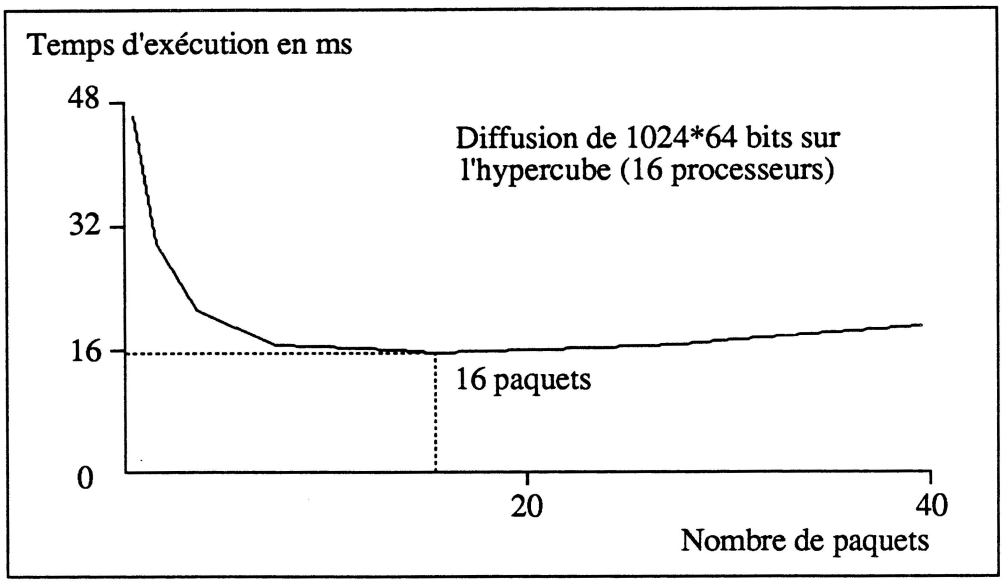


Figure III.6 : Diffusion et subdivision en paquets sur l'hypercube à 16 processeurs : le nombre optimal de paquets mesuré.

Cette première figure nous donne les coûts mesurés pour l'algorithme de diffusion (cf §I.1.4) donné par Y.Saad [Saa 2], en fonction du nombre des paquets pipelinés sur l'arbre. Le message initial consiste en L=1024 mots de 64bits.

Le nombre de paquets à choisir pour minimiser le temps est **en théorie** (avec $\beta = 60\mu s$, puisque l'on utilise 4 liens en parallèle) [Saa 2],

$$Nb_{meilleur} = \sqrt{\frac{(\log_2 P - 1)L\tau}{\beta}} \approx 24 ,$$

il conduit à,

$$T_{min} = (\sqrt{L\tau} + \sqrt{(\log_2 P - 1)\beta})^2 \approx 14{,}9 \text{ ms.}$$

Les valeurs expérimentales sont donc satisfaisantes, mais T_{min} (resp. $NB_{meilleur}$) est un peu sous–estimé (resp. surestimé) : le coût du contrôle a pour effet d'augmenter la valeur de β, on a en pratique $\beta' \approx 135\mu s$.

Dans un deuxième temps nous avons mesuré des temps d'exécution de l'algorithme du pipeline pour trianguláriser la matrice identité. Aucun calcul n'est donc à effectuer et le temps d'exécution peut être considéré comme étant un temps de communication. Le coût théorique, a été donné dans [CTV 2], il est analogue à (PParId 1.18) avec r=1, γ=0 et sans la matrice identité bordante utilisée pour le noyau,

$$T_{th} = (3n^2/2 + nP - 5n/2 - P^2/2 + P/2 - 4)\tau + (3n+6)\beta.$$

Et nous a servi à calculer les temps théoriques du tableau de la figure III.7, en remplaçant directement β et τ par leurs valeurs. La fiabilité de leurs mesures de β et τ nous permet d'obtenir des résultats convainquants.

Taille de la matrice	Temps théoriques(ms)	Temps mesurés(ms)
16	6	13
32	23	34
64	82	98
128	310	339
256	1196	1279
512	4695	4994
1024	18592	19576

Figure III.7 : L'algorithme du *pipeline* (16 processeurs) appliqué à la matrice identité.

● Pour illustrer les programmes donnés dans les chapitres précédents et montrer qu'ils peuvent se programmer simplement, terminons ce paragraphe, en donnant la partie du programme C de l'algorithme PlParId qui intéresse les communications.

Les premières fonctions qui apparaissent permettent de définir la topologie anneau et d'initialiser les canaux de communication correspondants en précisant le nombre d'envois/réceptions autorisés en parallèle. Les fonctions **oto_x** sont les communications elles–mêmes : leurs 3–ièmes et

4–ièmes paramètres sont les numéros des processeurs concernés, les 5–ièmes et 6–ièmes étant les adresses des variables envoyées et reçues. L'instruction **wait_l** attend la complétion des instructions oto_x qui la précèdent. Si cette instruction n'est pas spécifiée, un ordre de communication ou de calcul vectoriel pourra être traité sans forcément attendre la complétion des ordres de communications préalablement instantiés (le traitement des instructions est alors exactement celui des processus d'un constructeur PAR en Occam [Inm]). On a donc là un moyen de programmer le parallélisme communications/communications et communications/arithmétique [FPS].

Le code est le même pour tous les processeurs. Pour se placer sur l'un ou sur l'autre, on a donc un certain nombre de conditionnels *if*.

Continuer à détailler les diverses fonction manquerait d'intérêt. Mais le lecteur peut se reporter au chapitre II, pour comparer les lignes ecrites en C ci–dessous, à l'algorithme lui–même.


```
main_PlParId(...)
    {
    config_torus_1d(P,&anneau);
    canaux = open_l(anneau,2,OTOX);
    pos = config_pos(anneau,0);
    posprec = (pos-1)%P;
    possuiv = (pos+1)%P;
    .....
    if (trouve == 0)        Pas de ligne pivot locale dans ce processeur
        {
        if (pos == k%P)
            {
            jeton = -1;
            oto_x(canaux, 0, pos, possuiv, &jeton, &jeton, 4);    Envoi sur l'anneau
            wait_l(canaux);
            }
        else
            {
            oto_x(canaux, 0, posprec, pos, &jeton, &jeton, 4);    Réception
            wait_l(canaux);
            if (jeton == 1)
                {
                oto_x(canaux, 0, posprec, pos, &lgnpiv, &lgnpiv, m+n-k+1);
                wait_l(canaux);
                if (pos != (k-1)%P)
                    {
                    oto_x(canaux, 0, pos,possuiv, &jeton, &jeton, 4);
```

```
                wait_l(canaux);
                oto_x(canaux, 0, pos, possuiv, &lgnpiv, &lgnpiv, m+n-k+1)
                wait_l(canaux);
                }
            }
        else
            {
            if (pos != (k-1)%P)
                oto_x(canaux, 0, pos,possuiv, &jeton, &jeton, 4), wait_l(canaux);
        else  rangloc = rangloc - 1;
            }
        }
    }

else        Une ligne pivot locale a été trouvée dans ce processeur
    {
    if (pos != (k-1)%P)
        jeton = 1, oto_x(canaux, 0, pos, possuiv, &jeton, &jeton,4);
    else if (pos != k%P)
        oto_x(canaux, 0, posprec, pos, &jeton, &jeton, 4);
    wait_l(canaux);
    if  (pos != (k-1)%P)    Envoi de la ligne pivot locale
        oto_x(canaux, 0, pos, possuiv, &lgnpiv, &lgnpiv, m+n-k+1);
    if (pos != k%P)
        {
        pluselim = 0;
        if (jeton == 1)    Réception d'une ligne pivot locale
            oto_x(canaux, 0, posprec, pos, &lgnpiv, &lgnpiv, 4);
        else pluselim = 1;  Il y a permutation virtuelle, la ligne iploc n'est pas à éliminer
        }
    wait_l(canaux);

    Elimination des lignes d'indices supérieurs à iploc
    ......
    if (pluselim == 0)
        {
        Elimination de la ligne d'indice iploc
        ......
        }
    }
.....
}
```

III.3 LES ALGORITHMES : RESULTATS EXPERIMENTAUX[1]

Nous pouvons maintenant nous consacrer à présenter les résultats de nos mesures. Les courbes ont été obtenues aussi bien à l'aide d'implantations d'une simple élimination de Gauss : triangularisation de la matrice. Qu'à l'aide d'implantations de l'inversion de la matrice par Jordan ou du calcul du noyau.

III.3.1 Quand la comparaison des méthodes est possible

Pour comparer les différentes méthodes de façon raisonnable, il faut bien entendu que les algorithmes soient appliqués à une même matrice, mais aussi **que tous effectuent, pour la matrice choisie, un nombre identique d'opérations.** Ce qui n'est pas a priori vérifié : à une étape k fixée, suivant la ou les lignes pivots qui sont utilisées, la matrice au début de l'étape k+1 n'est pas la même. Et puisque que l'on n'élimine pas une ligne de coefficient de tête déjà nul, le nombre d'opérations effectuées à l'étape k+1 peut lui aussi varier (suivant le nombre de zéros supplémentaires qui ont été introduits).

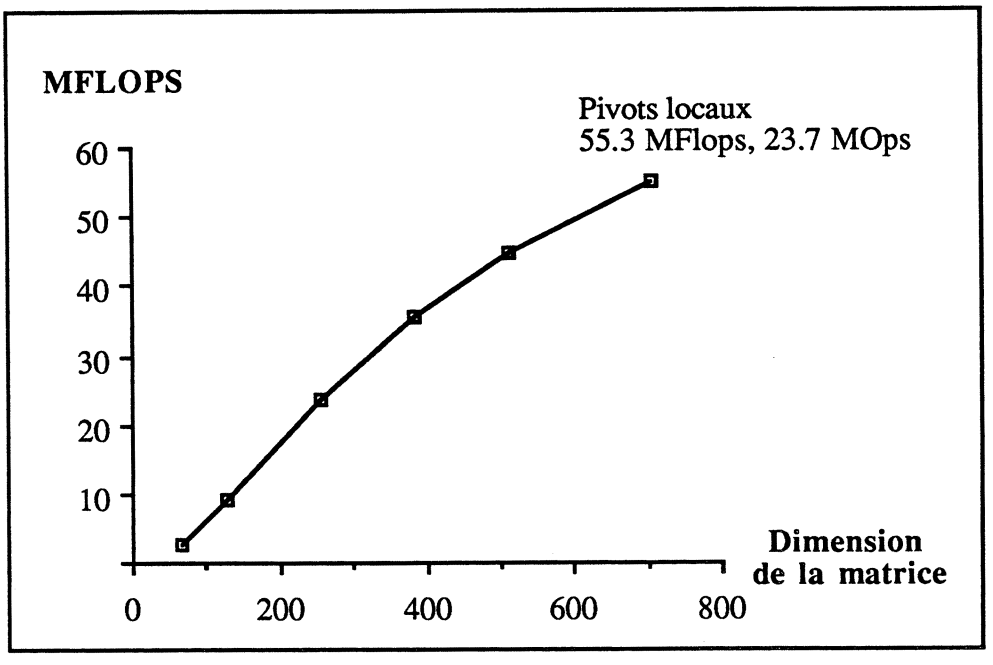


Figure III.8 : Performances de l'algorithme de Jordan appliqué à l'inversion d'une matrice A, $A \in \mathcal{H}$ (P = 16).

En pratique, l'hypothèse d'un nombre d'opérations effectuées constant est facilement vérifiée : si le nombre premier est assez grand et la matrice choisie aléatoirement, le nombre de coefficients de

[1] Toutes les mesures présentées dans ce paragraphe ont été réalisées sous la version C00 du logiciel FPS [FPS].

tête nuls rencontrés est réduit à quelques unités. Il n'en sera pas de même dans le cas général, en particulier si le nombre est premier est petit ou si la dimension de l'espace nul est importante, nous l'illustrerons par la suite.

Pour obtenir les courbes de ce paragraphe nous avons donc choisi un nombre premier proche du plus grand autorisé (3.1) : p=67000033. On appelle $\mathcal{H}$ la classe de matrice dont les coefficients sont pris au hasard entre 0 et p. A de rares exceptions près, ces matrices sont inversibles modulo p et le nombre d'opérations à effectuer pour chacune des résolutions est égal au maximum.

Les figures III.8 et III.9 nous montrent les performances des méthodes appliquées à l'inversion et à la triangularisation de la matrice. Sur les 16 MOctets disponibles au total, 8 seulement le sont pour le stockage de A, si l'on s'attache à respecter les contraintes associées à l'utilisation de l'unité vectorielle dans les meilleures conditions. On est donc limité à des matrices de dimensions 720 pour l'inversion et 1024 pour la triangularisation. Les chiffres, qui viennent confirmer les coûts théoriques, sont donnés

- en millions d'opérations flottantes effectuées par seconde : MFlops,
- ou en millions d'additions, de produits ou de réductions modulo p : MOps.

C'est la longueur des lignes et donc des vecteurs sur lesquels on opère, double en moyenne pendant l'exécution de Jordan, qui permet d'obtenir près de 11 MFlops de plus que pour l'élimination de Gauss.

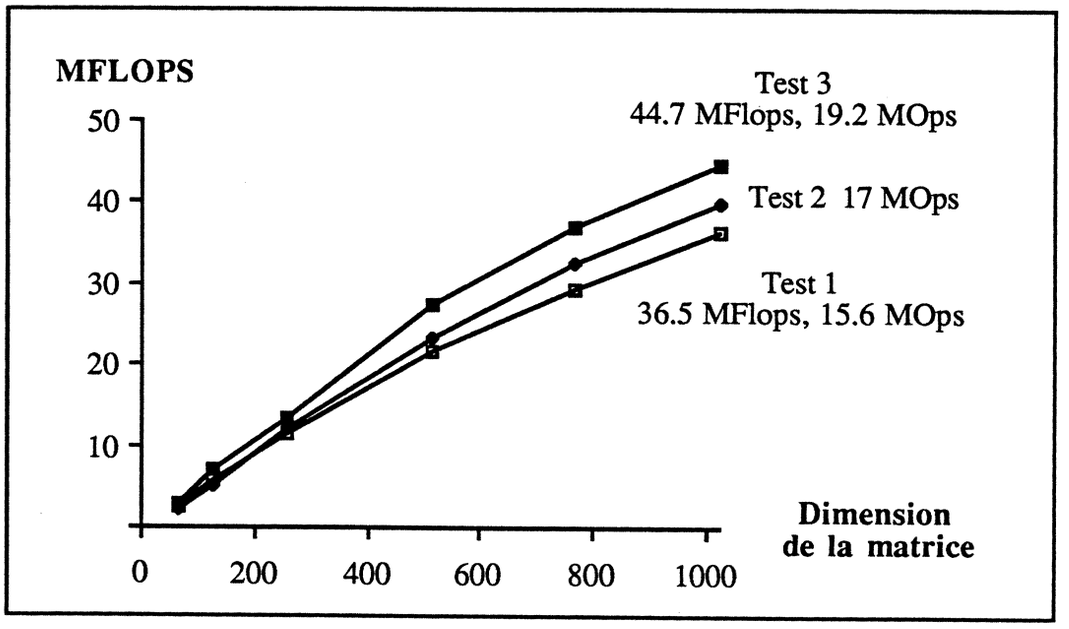


Figure III.9 : Performances de l'élimination de Gauss, A$\in\mathcal{H}$.
Test 1 : PPar, Test 2 : DPar, Test 3 : PlPar (P = 16).

● **Avec stockage de la matrice identité**

De la même façon, les figure III.10 et III.11 nous permettent de comparer les algorithmes, maintenant appliqués au calcul du noyau en utilisant la matrice identité, la dimension des matrices est toujours limitée à 720. Une nouvelle fois et comme l'étude théorique le laissait prévoir (les termes en $n^2\tau$ dans les complexités des communications font la différence asymptotiquement), l'algorithme à pivots locaux PlParId est meilleur que les algorithmes basés sur le transfert de lignes pivots globales (DParId et PParId).

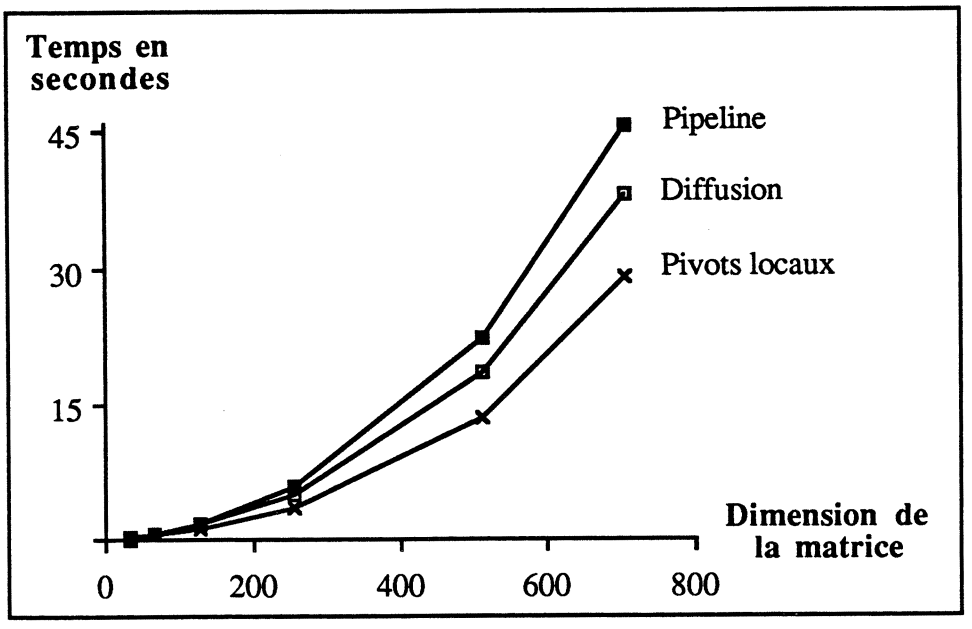


Figure III.10 : Comparaison des temps d'exécution des algorithmes avec stockage de la matrice identité, A∈$\mathcal{H}$ et P = 16.

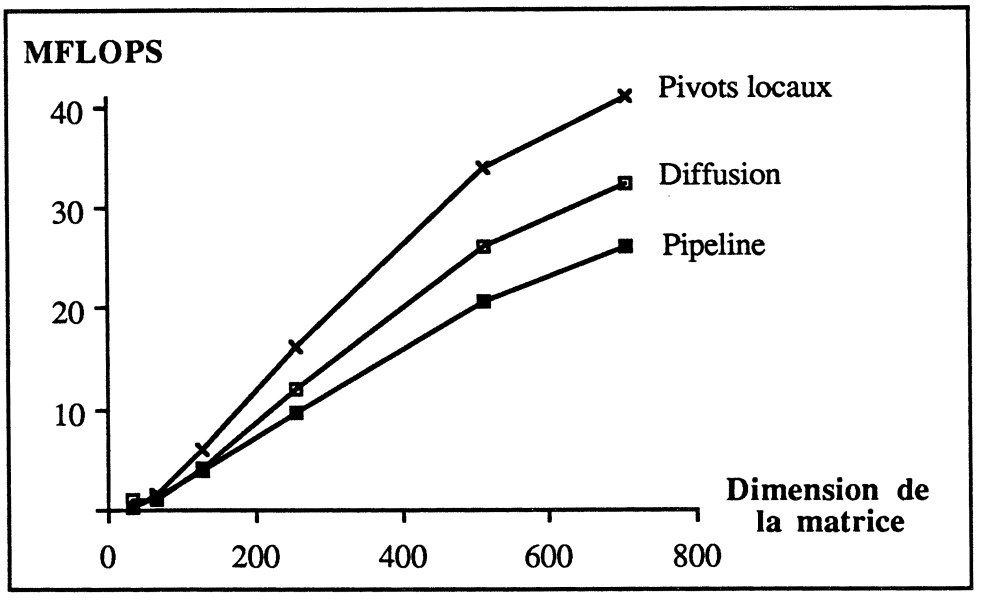


Figure III.11 : Performances des algorithmes avec stockage de la matrice identité, A∈$\mathcal{H}$ et P = 16.

- **Sans stockage de la matrice identité**

Supprimer le stockage de la matrice identité permet d'augmenter la dimension des matrices traitées. Nous avons vu, pour la classe des matrices que nous considérons pour l'instant, que le coût des communications est alors réduit dans un facteur au moins égal à 2.

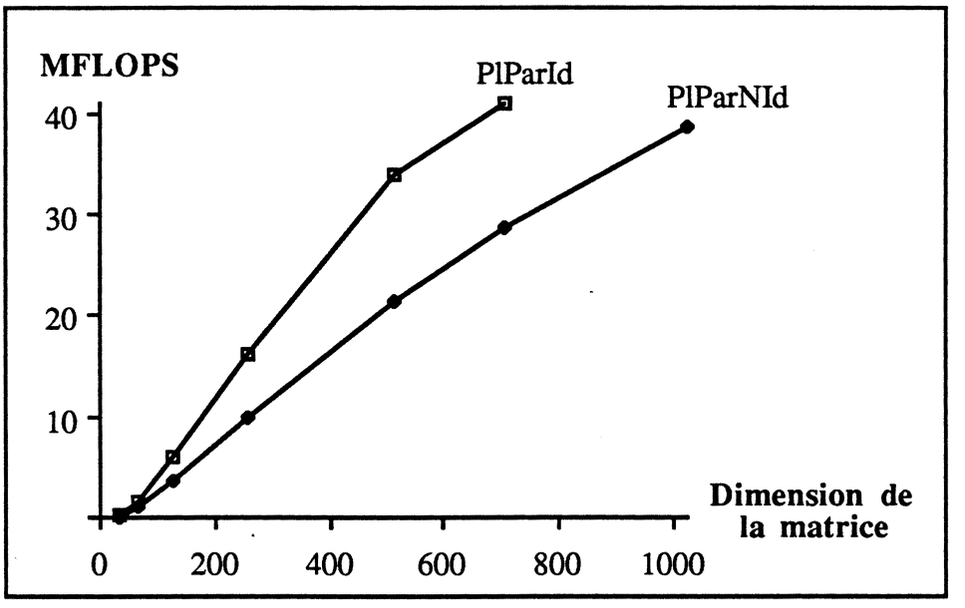


Figure III.12 : Performances de l'algorithme à pivots locaux, avec et sans stockage de la matrice identité, A∈$\mathcal{H}$ et P = 16.

Mais puisque l'unité vectorielle opère lignes par lignes, et que chaque ligne possède maintenant en moyenne moitié moins de coefficients (il est intéressant de situer les longueurs 512 et 1024 sur la fugure III.5), ses performances aussi sont diminuées. Cela apparait clairement sur la figure III.12. Il faut attendre une dimension de 1024 (soit près de 3.8 milliards d'opérations flottantes) avec PlParNId pour retrouver les performances de PlParId.

On remarque aussi que la courbe de PlParId va se stabiliser plus rapidemment. Donc si d'après la théorie, nous avions terminé sur la supériorité des méthodes n'utilisant pas la matrice identité. Nous concluerons ici en disant, qu'en pratique, le résultat reste vrai si la dimension du problème est assez grande : si l'unité vectorielle est utilisée au meilleur de ses performances.

III.3.2 Efficacité

L'efficacité d'un algorithme, c'est à dire le rapport du temps de l'exécution sur un processeur au produit du temps de l'exécution sur P processeurs par P, est la quantité habituellement mesurée pour juger d'une implantation parallèle.

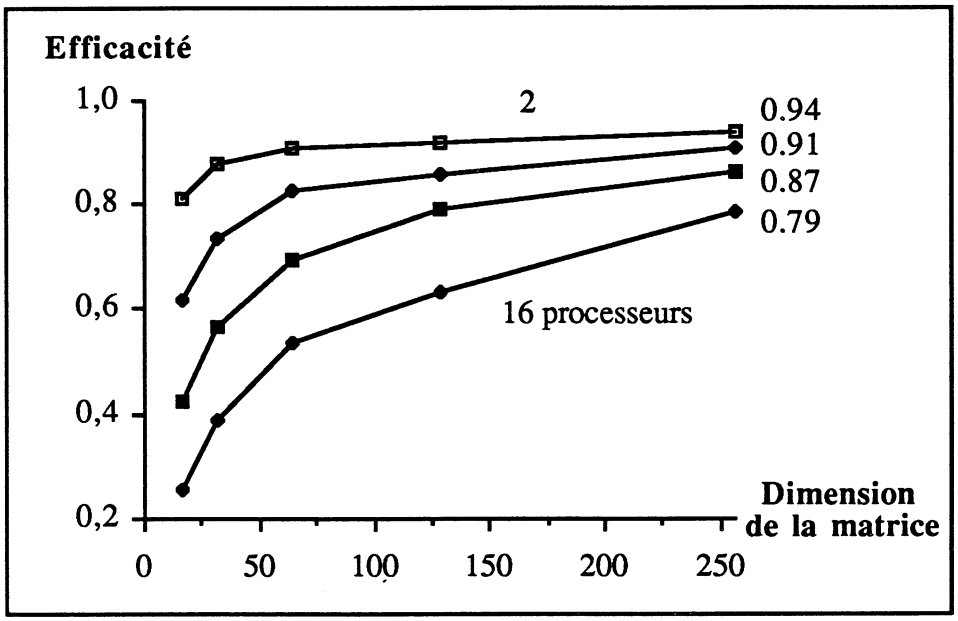


Figure III.13 : Efficacité de l'algorithme à pivots locaux pour la triangularisation de la matrice, $A \in \mathcal{H}$.

Cette définition sous–entend que le nombre total d'opérations effectuées reste le même pour les deux exécutions. Sous cette hypothèse ($A \in \mathcal{H}$), nous montrons sur les figures III.13 et III.14, les efficacité obtenues pour l'algorithme à pivots locaux et l'algorithme du pipeline. Ces quantités diminuent avec la taille de l'anneau utilisé : le coût des communications est asymptotiquement indépendant de P alors que celui de l'arithmétique décroit presque linéairement.

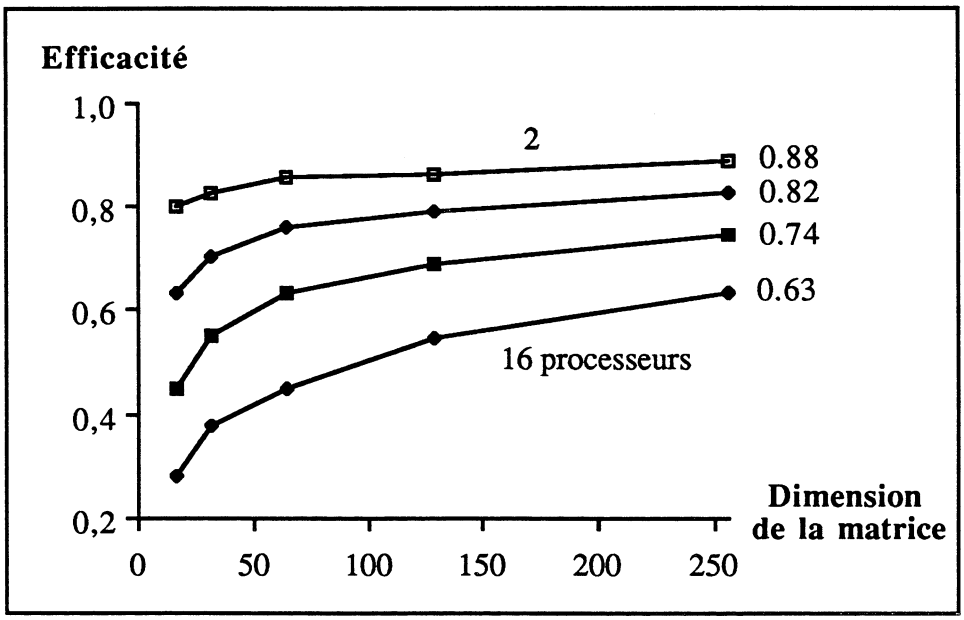


Figure III.14 : Efficacité de l'algorithme du pipeline pour la triangularisation de la matrice, $A \in \mathcal{H}$.

Mais nous avons expliqué au début du chapitre, que le nombre d'opérations effectuées au cours d'un algorithme, peut dépendre du choix des lignes pivots utilisées à chaque étape. Pour l'algorithme de diffusion ce choix lui–même dépend de l'élection.

Pour l'algorithme à pivots locaux, il dépend, pour une matrice A donnée, du nombre de processeurs utilisés. Considérons par exemple une matrice triangulaire dont tous les coefficients sont identiques :

- à la première étape d'une résolution sur 16 processeurs, les lignes sont pour la plupart éliminées avec des lignes pivots locales dont les premiers coefficients sont égaux aux leurs. Ce qui a pour effet d'introduire un grand nombre de zéros, non seulement dans la première colonne, mais dans toute la matrice. Le nombre d'opérations à effectuer à la deuxième étape est réduit d'autant, le phénomène se répète aux étapes suivantes.
- Contrairement à cela, une résolution séquentielle a le même coût effectif que celui associé aux matrice de $\mathcal{H}$.

Tout ceci est illustré sur la figure III.15, une résolution effectuée sur 16 processeurs est jusqu'à 24 fois plus rapide que sur un seul.

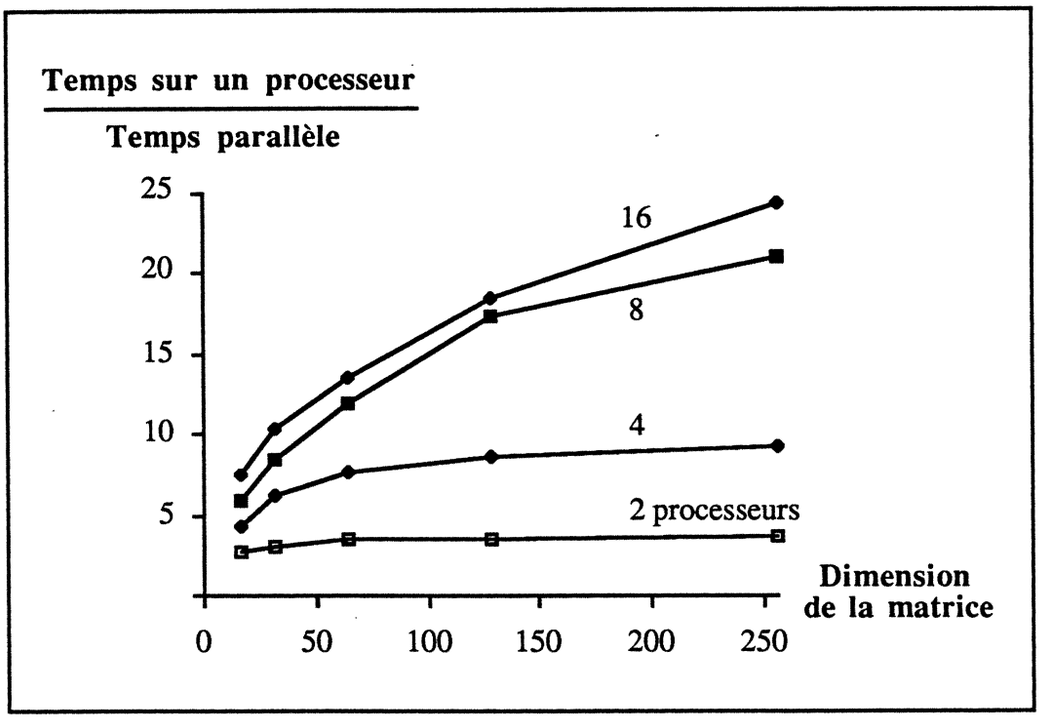


Figure III.15 : Accroissements de vitesse pour l'algorithme à pivots locaux appliqué à la matrice triangulaire inférieure dont tous les coefficients sont égaux à 4, p=7.

III.3.3 Quand la comparaison des méthodes n'est plus possible

• Les pires cas

Pour calculer les bornes sur les coûts en communication des différentes méthodes, nous avons caractérisé aux chapitres I et II les plus mauvaises situations respectives, dans lesquelles aucun calcul n'est à effectuer. Ces cas de figure sont repris sur la courbe ci–dessous.

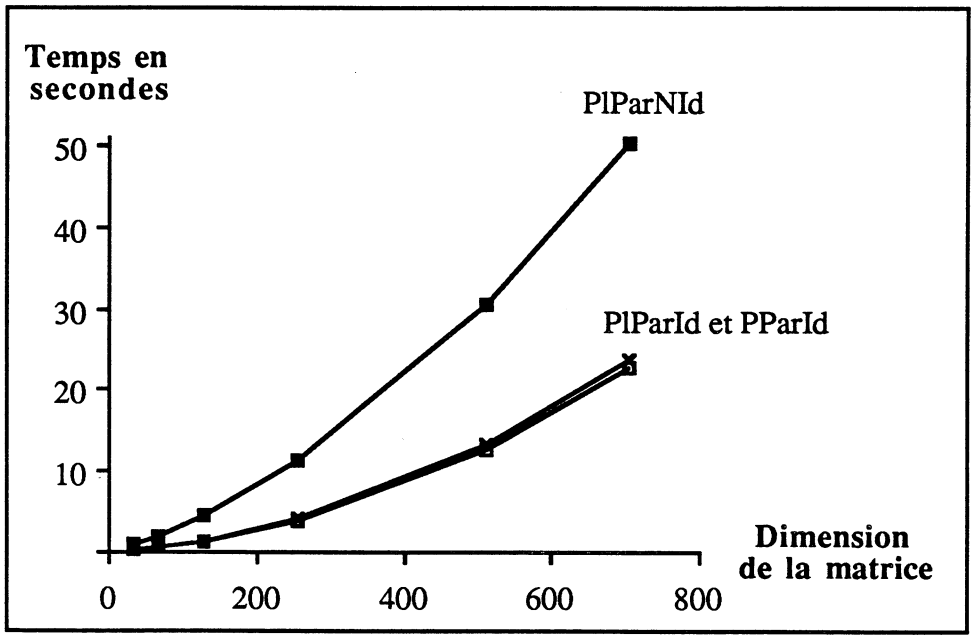


Figure III.16 : Temps d'exécution des pires cas
pour les coûts des communications, P = 16.

On voit clairement que l'algorithme à pivots locaux avec stockage de la matrice identité se comporte exactement comme un algorithme avec transfert d'une ligne pivot globale. Et que le coût de la diagonalisation reste toujours bien supérieur.

• Le cas d'une matrice triangulaire

Reprenons le cas d'une matrice triangulaire inférieure, le coût arithmétique des algorithmes à pivots locaux est, comme nous l'avons mis en évidence, bien inférieur à n^3 si l'on utilise plusieurs processeurs. Le coût en communications étant lui élevé. Cette situation est traduite sur les deux figures suivantes.

L'exécution de l'algorithme du pipeline est identique à celle que nous avions donnée sur la figure III.10. Pour PlParNId, la situation est plus mauvaise encore que le pire cas de la figure III.16 qui n'est obtenu qu'en regard des communications. Le nombre d'opérations effectuées est faible mais non nul : les performances sont données sur la figure III.18.

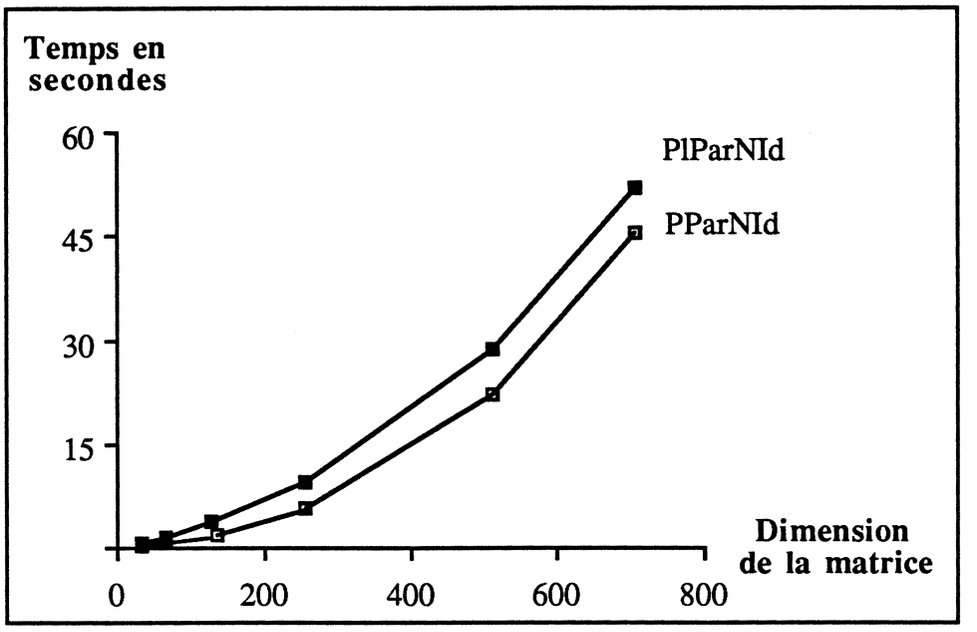


Figure III.17 : Temps d'exécution dans le cas d'une matrice triangulaire inférieure.
(A(i,j) = 4 si i≤j et 0 sinon, p=7, P = 16).

Comme nous l'avions précisé dans l'introduction (avec la quantité γ), résoudre le problème dans le cas général demande l'utilisation d'une **arithmétique vectorielle creuse**. N'en disposant pas d'une sur le FPS T20, et l'utilisation de routines scalaires (ou de déplacement en mémoire) ayant été trop prohibitive, nous ne pouvons présenter de mesures concernant les méthodes basées sur le stockage de la matrice identité.

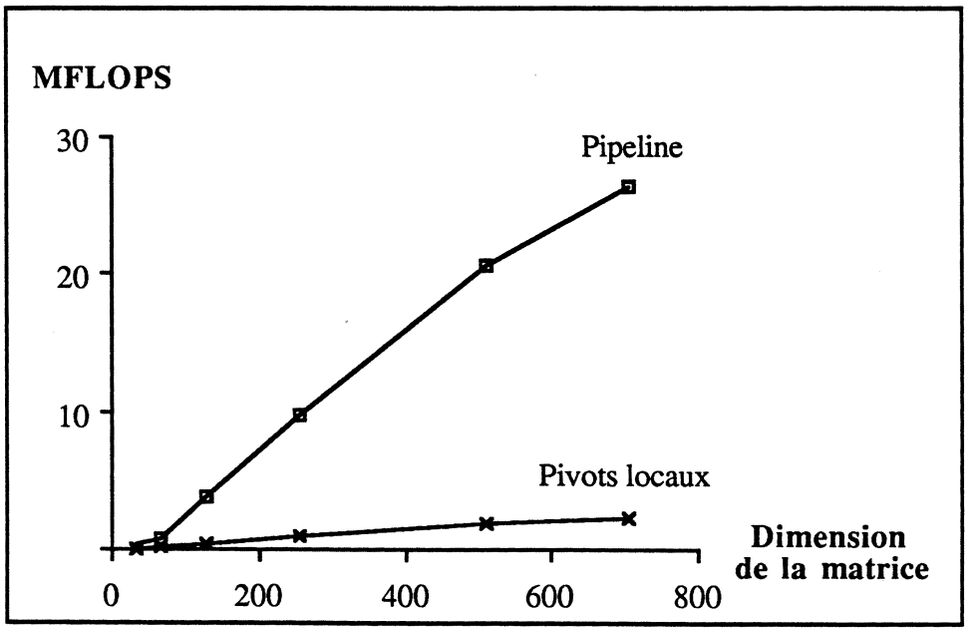


Figure III.18 : Performances correspondant à la figure III.17.

- **Quand la dimension de l'espace nul est importante**

Nous donnons sur cette dernière figure des temps d'exécutions dans des cas où l'espace nul de la matrice est de grande dimension. Il faut surtout noter que le temps associé à PlParNId est inférieur à celui de la figure III.16 : on ne se trouve pas dans le pire cas pour les communications, mais néanmoins, le coût arithmétique est faible (de l'ordre de $n^3/5$).

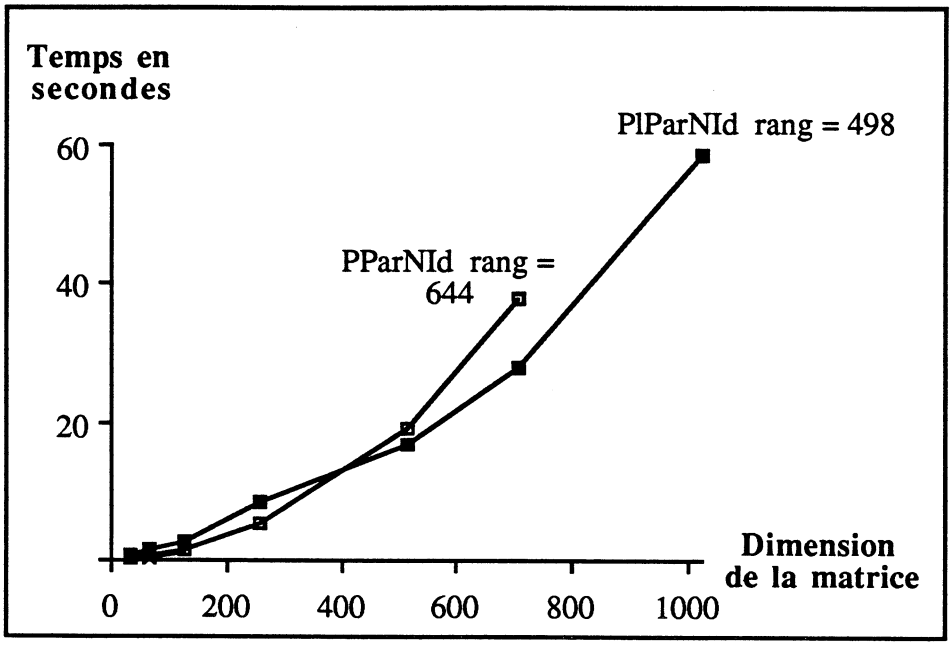


Figure III.19 : Temps d'exécution pour des matrices de faible rang.

Conclusion

Après avoir étudié d'un point de vue algorithmique l'implantation du calcul du noyau, nous venons de montrer combien les conclusions théoriques doivent être complétées ou même modifiées par les résultats expérimentaux.
La structure de la matrice A elle–même joue un tout premier rôle. Aussi, une étude complémentaire pourrait concerner la répartition (dynamique ?) de la charge de calcul entre les différents processeurs. Concernant l'étude des plus mauvais cas (situations dans lesquelles l'exécution de l'algorithme est la moins avantageuse) : si l'on connait d'une part la matrice réalisant le plus mauvais coût arithmétique, et d'autre part celle maximisant les communications, il est particulièrement difficile de mettre en évidence la matrice pour laquelle la résolution prendra, au total, le plus de temps. Nous ne savons pas, par exemple, si telle est la matrice triangulaire utilisée pour l'algorithme PlParNId sur la figure III.17.
Terminons en rappelant, que ces algorithmes constituent la partie essentielle de certaine méthodes de résolution de systèmes à coefficients entiers : le calcul vectoriel dans les corps finis, quand il pourra prendre le relai de l'arithmétique en précision infinie, deviendra un outil indispensable.

DEUXIEME PARTIE

SYSTEMES LINEAIRES A COEFFICIENTS ENTIERS

INTRODUCTION

Nous l'avons illustré dès les toutes premières pages de la thèse, la complexité des objets qu'il manipule, est une caratéristique essentielle du Calcul Formel. Aussi, l'*élimination de Gauss* appliquée à une matrice de dimension n à coefficients entiers, conduit–t'elle à effectuer de l'ordre de $O(n^5\log^2 n)$ opérations arithmétiques [Bar]. Et même s'il ne suffit pas, bien sûr, pour conclure sur la supériorité d'une méthode, d'arguer son coût théorique plus faible (un modèle de complexité aura toujours ses lacunes), il faut souligner avant de parler de parallélisme, les apports considérables des techniques de congruences. Le coût du calcul de la solution d'un système linéaire entier, en appliquant *le théorème des restes chinois*, est en $O(n^4\log n)$ [Bar]. Il se réduit à $O(n^3\log^2 n)$, ce qui est à un facteur $\log^2 n$ près le coût du calcul numérique, si les *développements p–adiques* sont utilisés [Dix].
L'intérêt des techniques de congruence est aussi établi, par exemple, pour la résolution de systèmes linéaires dans l'anneau des polynômes denses à plusieurs variables [McC].

D'autres méthodes permettent d'obtenir la solution d'un système linéaire, et utilisent par exemple *le théorème de Caley–Hamilton* (avec le polynôme caractéristique de la matrice). Divers résultats peuvent être trouvés dans la littérature, leur modèle de complexité parallèle étant différent du notre, concernant notamment le nombre de processeurs. L'algorithme de L.Csanky en demande $O(n^4)$ pour obtenir la solution en $O(\log^2 n)$ étapes [Csa]. Le nombre a été ramené à $O(n^{3.496})$ par S.J.Berkowitz [Ber] à l'aide, en particulier, d'un algorithme asymptotiquement rapide de multiplication de matrices [CW]. Remarquons que ces complexités, qui ne tiennent pas compte des communications, restent par là même très théoriques. Tel n'est pas notre propos, mais que donnerait, sous les mêmes hypothèses, la méthode p–adique dont le coût est a priori bien plus faible ?

L'objectif de cette deuxième partie est de montrer, en développant et en expérimentant sur un hypercube, différentes implantations des trois premières méthodes citées, que l'utilisation combinée du *parallélisme* et du *calcul vectoriel*, peut permettre de résoudre des problèmes aux dimensions jusque là difficiles à envisager en arithmétique exacte (bien que depuis longtemps dépassées en calcul numérique).

Précisons que seule la lecture du chapitre II, concernant la résolution p–adique, est indépendante de la première partie. A laquelle au contraire, on se réfèrera fréquemment pour présenter l'élimination de Gauss parallèle dans $\mathbb{N}$ (chap.I) et pour appliquer le théorème des restes chinois (chap.II).

CHAPITRE I

ELIMINATION DE GAUSS SANS FRACTION

Nous étudions dans ce chapitre la première technique qui vienne à l'esprit pour la résolution des systèmes linéaires à coefficients entiers : l'élimination de Gauss. Diverses méthodes ont été développées à partir de celle de E.Bodewig [Bod], qui visent à limiter le problème, crucial, de la croissance des données pendant le déroulement de l'élimination. Nous allons nous attacher à développer des versions parallèles des algorithmes "à un pas" et "à deux pas" sans fraction, pour la triangularisation de la matrice, tout en considérant le cas d'éliminations avec fractions qui s'en déduit facilement. Ces algorithmes ont été donnés par E.H.Bareiss [Bar], et améliorent nettement le coût de la méthode initiale. Le coût de la résolution d'un système triangulaire associée sera toujours supposé négligeable. Le cas général s'applique aux matrices carrées A inversibles, de dimension n à coefficients entiers, éventuellement bordées par m-n vecteurs de $\mathbb{Z}^n$:

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} & A_{1,n+1} & \dots & A_{1,m} \\ A_{21} & A_{22} & \dots & A_{2n} & A_{2,n+1} & \dots & A_{2,m} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nn} & A_{n,n+1} & \dots & A_{n,m} \end{pmatrix}.$$

Dans ce qui suit, on prendra m = n. Les résultats se généralisent directement en remplaçant la n–ième colonne de A, par chacun des vecteurs bordants.
La transformée de A après la k-ième étape d'une élimination sans fraction (tous les coefficients intermédiaires sont entiers) sera notée $A^{(k)}$ ($A^{(0)} = A$) et $A_{ij}^{(k)}$ ses éléments. Après la k-ième étape d'une élimination avec fractions (les coefficients intermédiaires sont rationnels) on aura de même $Q^{(k)}$ ($Q^{(0)} = A$) et $Q_{ij}^{(k)}$. Par convention les coefficients de $A^{(-1)}$ et $Q^{(-1)}$ sont tous égaux à 1.

I.1 ELIMINATION DE GAUSS SEQUENTIELLE

Avant de se placer dans le cadre du parallélisme, reprenons dans ce paragraphe l'apect séquentiel du problème et les conclusions apportées par Bareiss [Bar].

I.1.1 Eliminations dans $\mathbb{N}$

Les algorithmes "à un pas" et "à deux pas" permettent de mettre la matrice sous forme triangulaire

supérieure en n'effectuant que des divisions entières exactes. On suppose pour l'instant qu'aucun pivot nul n'est rencontré au cours de l'exécution. Et que le procédé d'élimination est appliqué à toutes les lignes d'indice supérieur à celui de la ligne pivot : même celles dont le coefficient de tête est déjà nul. Nous envisagerons les cas particuliers au §I.2.3.

On sait que les tailles des coefficients intermédiaires au cours de l'élimination de Gauss sont minimisées si l'on utilise pendant n–1 étapes k, $1 \le k < n$, le procédé d'**élimination à un pas** [Bar] suivant :

$$A_{ij}^{(k)} = \frac{1}{A_{k-1,k-1}^{(k-2)}} \begin{vmatrix} A_{kk}^{(k-1)} & A_{kj}^{(k-1)} \\ A_{ik}^{(k-1)} & A_{ij}^{(k-1)} \end{vmatrix}, \quad i \ge k+1, j \ge k. \qquad (1.1)$$

Après chaque élimination, les déterminants calculés sont des multiples du pivot de l'étape précédente. Nous donnons ci–dessous l'algorithme correspondant.

- **Algorithme Seq1p**

```
début
simp = 1
pour k = 1 à n-1
        pour i = k+1 à n
                pour j = k+1 à n
                  A_ij = (A_kk A_ij – A_ik A_kj) / simp
        simp = A_kk
fin
```

L'expression des coefficients des matrices $A^{(k)}$, $1 \le k < n$, est connue :

$$A_{ij}^{(k)} = \begin{vmatrix} A_{11} & \dots & A_{1k} & A_{1j} \\ \dots & \dots & \dots & \dots \\ A_{k1} & \dots & A_{kk} & A_{kj} \\ A_{i1} & \dots & A_{ik} & A_{ij} \end{vmatrix} \text{ si } i \le j \text{ et } 0 \text{ sinon.} \qquad (1.2)$$

La matrice triangulaire $A^{(n-1)}$ obtenue est donc,

$$\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ 0 & \Delta_{22} & \dots & \Delta_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \Delta_{nn} \end{pmatrix} \qquad (1.3)$$

où les Δ_{ij} sont les mineurs suivants de A :

$$\Delta_{ij} = \begin{vmatrix} A_{11} & \cdots & A_{1,i-1} & A_{1j} \\ \cdots & \cdots & \cdots & \cdots \\ A_{i-1,1} & \cdots & A_{i-1,i-1} & A_{i-1,j} \\ A_{i1} & \cdots & A_{i,i-1} & A_{ij} \end{vmatrix}. \qquad (1.4)$$

L'algorithme **à deux pas** [Bar] (nous verrons la généralisation au §I.2.2) consiste à calculer $A_{ij}^{(k)}$ directement à partir de $A_{ij}^{(k-2)}$. En appliquant (1.1) deux fois et pour $k \geq 2$, on a la relation,

$$A_{ij}^{(k)} = \frac{1}{\left(A_{k-2,k-2}^{(k-3)}\right)^2} \begin{vmatrix} A_{k-1,k-1}^{(k-2)} & A_{k-1,k}^{(k-2)} & A_{k-1,j}^{(k-2)} \\ A_{k,k-1}^{(k-2)} & A_{k,k}^{(k-2)} & A_{k,j}^{(k-2)} \\ A_{i,k-1}^{(k-2)} & A_{i,k}^{(k-2)} & A_{i,j}^{(k-2)} \end{vmatrix}, \; i \geq k+1, \; j \geq k-1. \qquad (1.5)$$

Quitte à effectuer d'abord une étape de Seq1p, on peut supposer que n–1 est pair. En décomposant l'égalité ci–dessus, l'algorithme correspondant s'écrit,

• **Algorithme Seq2p**

début
simp = 1
pour k = 1 à n–2 de 2 en 2
  pour j = k+1 à n *Mise à jour de la ligne k+1, un seul pas*
    $A_{k+1,j} = (A_{kk} A_{k+1,j} - A_{k+1,k} A_{kj}) / \text{simp}$
  $C_k = A_{kk} A_{k+1,k+1} - A_{k+1,k} A_{k,k+1}$

  pour i = k+2 à n *Mise à jour des autres lignes, deux pas*
    $C_1 = -(A_{kk} A_{i,k+1} - A_{i,k} A_{k,k+1})$
    $C_2 = A_{k,k+1} A_{i,k+1} - A_{i,k} A_{k+1,k+1}$
    pour j = k+2 à n
      $A_{ij} = (C_k A_{ij} + C_1 A_{k+1,j} + C_2 A_{kj}) / \text{simp}^2$
  simp = $A_{k+1,k+1}$
fin

I.1.2 Eliminations dans $\mathbb{Q}$

Une autre méthode, comme nous allons le voir analogue à la précédente, consiste à effectuer des éliminations en considérant que les coefficients de la matrice appartiennent à $\mathbb{Q}$. La ligne pivot est pour cela divisée par l'élément diagonal avant le calcul de la nouvelle matrice. On utilise donc au

lieu de (1.1) :

$$A_{kj}^{(k)} = A_{kj}^{(k-1)} / A_{kk}^{(k-1)},\ j \geq k \text{ et,}$$
$$A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k)},\ j \geq k,\ i \geq k+1. \qquad (1.6)$$

De façon équivalente à (1.2), si l'algorithme de Gauss est effectué en appliquant ces deux transformations, on connait à chaque étape k les dénominateurs des coefficients non nuls de la matrice $Q^{(k)}$ calculée (en posant $\Delta_{00}=1$) :

$$\begin{cases} Q_{ij}^{(k)} = \dfrac{A_{ij}^{(n-1)}}{\Delta_{i-1,i-1}} \text{ si } i \leq k+1, & \textit{Valeurs finales}. \\ Q_{ij}^{(k)} = \dfrac{A_{ij}^{(k)}}{\Delta_{kk}} \text{ si } i > k+1, & \textit{Valeurs intermediaires}. \end{cases} \qquad (1.7)$$

Ces égalités se démontrent par récurrence. C'est immédiat pour $Q^{(1)}$, en effet,

$$Q_{ij}^{(1)} = A_{ij} - \frac{A_{i1}A_{1j}}{A_{11}} = \frac{A_{ij}^{(1)}}{\Delta_{11}},\ i \geq 2.$$

Si $Q_{ij}^{(k-1)} = \dfrac{A_{ij}^{(k-1)}}{\Delta_{k-1,k-1}}$, alors,

$$Q_{ij}^{(k)} = Q_{ij}^{(k-1)} - \frac{Q_{ik}^{(k-1)} Q_{kj}^{(k-1)}}{Q_{kk}^{(k-1)}}$$

$$= \frac{A_{ij}^{(k-1)}}{\Delta_{k-1,k-1}} - \frac{\Delta_{k-1,k-1} A_{ik}^{(k-1)} A_{kj}^{(k-1)}}{A_{k,k}^{(k-1)} \Delta_{k-1,k-1}^{2}},$$

$$= \frac{A_{k,k}^{(k-1)} A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k-1)}}{\Delta_{k-1,k-1}\Delta_{kk}}, \text{ puisque } \Delta_{kk} = A_{k,k}^{(k-1)},$$

et d'après (1.1),

$$Q_{ij}^{(k)} = \frac{A_{ij}^{(k)}}{\Delta_{kk}}.$$

Si les fractions ne sont pas réduites au cours des calculs on vient donc de montrer que les tailles des coefficients intermédiaires sont les mêmes qu'au cours de l'élimination sans fraction. De ce point de vue, le procédé ne sera donc intéressant que si les fractions sont systématiquement simplifiées (avec un coût arithmétique important pour les calculs des pgcd).

I.1.3 Coûts des algorithmes

• Soit q un entier positif non nul. Le *coût arithmétique* d'un algorithme sera dans ce chapitre le nombre d'opérations effectuées sur des mots de taille $\log_2 q$. Cette taille pourra correspondre, bien sûr, à la taille du mot machine, mais aussi à la taille des nombres premiers utilisés par les algorithmes des prochains chapitres. On suppose alors que les coûts d'une addition, d'un produit et d'un quotient d'un entier de longueur u par un entier de longueur v (avec $u \geq v$) sont respectivement T_a et T_p et T_q donnés par,

$$T_a = u,\ T_p = uv \text{ et } T_q = (u-v)v. \qquad (1.8)$$

Nous faisons ces hypothèses pour obtenir des coûts en rapport avec l'arithmétique à précision infinie que nous avons utilisée [Ro], et donc en rapport avec les résultats expérimentaux que nous présenterons. Les égalités (1.8) ont été données par D.E.Knuth [Knu, §4.3.1]. Bareiss s'est placé dans un cadre plus général, nous le précisons plus loin.

• Algorithme à un pas

A chaque étape k, si B est une borne sur les valeurs absolues des coefficients initiaux de la matrice $A^{(0)}$, les coefficients de $A^{(k-1)}$ qui interviennent dans les calculs de la nouvelle matrice sont bornés en valeur absolue par l'inégalité d'Hadamard (d'après (1.2)),

$$| A_{ij}^{(k-1)} | \leq B^k k^{k/2}. \qquad (1.9)$$

Si l'on suppose qu'au plus $\log_2 b$ bits sont nécessaires pour coder un entier positif inférieur à b, on notera L_k leur longueur en mots de taille $\log_2 q$,

$$\boxed{L_k \leq k \log_q B + k/2 \log_q k}\ . \qquad (1.10)$$

En prenant m=n et en négligeant le coût des additions (d'après (1.8)), une étape k de Seq1p demande de l'ordre de ,

- $2(n-k)^2$ produits d'entiers de tailles L_k, soient $2(n-k)^2 L_k^2$ opérations, et,
- $(n-k)^2$ quotients d'entiers de tailles $2L_k$ par des entiers de tailles L_{k-1}, ce qui équivaut à $(n-k)^2 L_k^2$ opérations pour les termes du plus haut degré.

ASeq1p le coût total de l'algorithme à un pas est alors obtenu en sommant les valeurs ci–dessus pour k variant de 1 à n–1,

$$\boxed{\text{ASeq1p} \leq \frac{1}{10} n^5 \left(1/4 \log_q^2 n + \log_q n \log_q B + \log_q^2 B \right) + O(n^4 \log_q^2 n)}\ . \qquad (1.11)$$

• Algorithme à deux pas

Le coût des multiplications est asymptotiquement réduit d'un facteur 3/4 [Bar], et celui des divisions d'un facteur 1/2. D'après les contributions respectives donnés ci–dessus, on peut écrire,

$$\boxed{ASeq2p \leq \frac{1}{15} n^5 \left(1/4 \log_q^2 n + \log_q n \log_q B + \log_q^2 B \right) + O(n^4 \log_q^2 n)} . \qquad (1.12)$$

Bareiss a donné une a donné [Bar] une formule moins restrictive. Si le coût du produit de deux entiers de taille t est $t^{1+\varepsilon}$, alors le nombre de multiplications effectuées sur des mots de taille $\log_2 q$ au cours de la triangularisation est,

$$A(\varepsilon) \approx Cte(\varepsilon)\, 2n^{4+\varepsilon} (\log_q B + 1/2 \log_q n)^{1+\varepsilon} , \qquad (1.13)$$

où à ε fixé, $Cte(\varepsilon)$ est une constante. Les coûts précédents sont retrouvés en prenant $\varepsilon=1$ (1.8). Le lecteur pourra se reporter à [Bar, CL] pour les différentes valeurs de ε, et les conclusions correspondantes.

● De la même façon, nous évaluons ici le *coût mémoire* en nombre de mots de taille $\log_2 q$. Il consiste essentiellement en le stockage de la matrice triangulaire qui contient, sur chacune de ses lignes k, n–k coefficients de tailles au plus L_k. Le total est obtenu en sommant sur k,

$$\boxed{MSeq2p \leq \frac{1}{6} n^3 \left(\log_q B + 1/2 \log_q n \right) + O(n^2 \log_q n)} . \qquad (1.14)$$

Les résultats donnés dépendent bien sûr étroitement de l'inégalité d'Hadamard. Mais au contraire des algorithmes utilisant des techniques de congruence, qui nécessiteront des procédures particulières (tests d'arrêts) pour profiter du caractère en général pessimiste de la borne donnée par l'inégalité, la réduction des tailles se fait "d'elle même" au cours des résolutions directes.

I.2 ELIMINATION SEQUENTIELLE AVEC PIVOTS LOCAUX

Nous avons vu en étudiant dans la partie I diverses implantations parallèles pour la triangularisation dans $\mathbb{Z}/p\mathbb{Z}$, que la classe des algorithmes à pivots locaux permet d'obtenir de faibles coûts de communications : en supprimant la diffusion dans le réseau d'une ligne pivot globale. A l'étape k d'un algorithme d'élimination avec pivots locaux, un zéro est introduit en tête des lignes d'indice $i \geq k$, non plus forcément par combinaison avec la ligne k (ligne pivot globale), mais par combinaison avec une des lignes d'indice supérieur à k (ligne pivot locale). Le choix de cette dernière pouvant dépendre de l'étape et de l'algorithme considérés.

Dire d'un algorithme d'élimination qu'il est à pivots locaux, n'a pas seulement trait à la façon dont se font les communications. Le terme peut donc aussi s'appliquer à des algorithmes séquentiels, quand ils ne supposent pas, à chaque étape, l'existence d'une ligne pivot globale.

Pour tout k, $1 \leq k < n$ et tout i, $k < i \leq n$, on note $\varphi_k(i)$, l'indice de la ligne qui permet, à l'étape k,

d'introduire un zéro en tête de la ligne i. Nous avons besoin de supposer pour la suite, que la ligne choisie pour l'élimination soit située au dessus de la ligne i dans la matrice,

$$k \leq \varphi_k(i) < i.$$

Cette dernière hypothèse nous assure en particulier, qu'à chaque étape k, il n'existe pas c indices de lignes distincts $i_1,\ldots,i_c$ avec $2 \leq c \leq n-k$, tels que,

$$\varphi_k(i_1)=i_2, \ldots, \varphi_k(i_j)=i_{j+1}, \ldots, \varphi_k(i_c)=i_1.$$

Ce qui rendrait l'élimination impossible.

L'algorithme donné ci–dessous permet alors de triangulariser la matrice A de façon triviale.

- **Algorithme SeqTLoc**

pour k = 1 à n-1
  pour i = k+1 à n
    pour j = k+1 à n
      $A_{ij} = A_{\varphi_k(i),k}\ A_{ij} - A_{ik}\ A_{\varphi_k(i),j}$.

I.2.1 Comment choisir la stratégie d'élimination ?

L'algorithme SeqTLoc appliqué tel quel conduirait bien entendu à une croissance trop importante des tailles des coefficients : **une croissance exponentielle**. Nous nous attachons ici à obtenir des formules analogues à celles du précédent paragraphe, à mettre en évidence des simplifications, pour des suites φ_k particulières.

Définition 2.1.1

On appelera **stratégie d'élimination** toute suite de fonctions $\{\varphi_k\}_{1 \leq k \leq n-1}$,

$$\varphi_k : [k+1, k+2, \ldots, n] \dashrightarrow [k, k+1, \ldots, n-1],$$

telles que :

$$\varphi_k(i) < i, \text{ pour tout i dans } [k+1, k+2, \ldots, n].$$

Proposition 2.1.2

Quelque soit la stratégie d'élimination choisie, on pourra effectuer des divisions exactes (resp. des réductions de fractions) sur les coefficients calculés par SeqTLoc, afin d'obtenir la matrice triangulaire sous la forme (1.3) (resp. (1.7)).

démonstration 2.1.2

- pour une matrice de dimension n=1,2 le résultat est trivial.
- pour n=3 :

après les deux premières étapes de SeqTLoc, on a,

$$A^{(2)} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ 0 & \Delta_{22} & \Delta_{23} \\ 0 & 0 & A_{\varphi_1(3),1}\,\Delta_{33} \end{pmatrix}. \qquad (1.15)$$

Si $\varphi_1(3) = 1$ on simplifie par A_{11}, ce qui correspond à l'algorithme à un pas (1.1). Si la ligne 3 est éliminée deux fois à l'aide de la ligne 2, on peut simplifier par A_{21}. Dans les deux cas, la matrice est obtenue sous la forme (1.3). On peut tenir le même raisonnement dans $\mathbb{Q}$ et obtenir les égalités (1.7).

Montrons alors la proposition par récurrence sur n, supposons la vraie pour les matrices de dimension n-1. Soit A de dimension n,

- si l'on travaille avec des éliminations entières, on a après la première étape,

$$A^{(1)} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ 0 & A_{22}^{(1)} & \dots & A_{2n}^{(1)} \\ \dots & \dots & \dots & \dots \\ 0 & A_{n2}^{(1)} & \dots & A_{nn}^{(1)} \end{pmatrix}.$$

Soit la sous–matrice,

$$S = \begin{pmatrix} A_{22}^{(1)} & \dots & A_{2n}^{(1)} \\ \dots & \dots & \dots \\ A_{n2}^{(1)} & \dots & A_{nn}^{(1)} \end{pmatrix},$$

en lui appliquant l'hypothèse de récurrence, on sait que pour toute stratégie d'élimination on pourra effectuer des divisions exactes permettant d'obtenir son mineur

$$\Delta S_{ij} = \begin{vmatrix} A_{22}^{(1)} & \dots & A_{2j}^{(1)} \\ \dots & \dots & \dots \\ A_{i2}^{(1)} & \dots & A_{ij}^{(1)} \end{vmatrix},$$

en position (i,j), dès la (i–1)–ème étape. On prendra i et j supérieurs ou égaux à 3, les autres coefficients ne posent aucun problème. De la même façon on notera $\Delta A_{ij}^{(1)}$ le mineur correspondant de $A^{(1)}$. Les i premières lignes de $A^{(1)}$ sont calculées en appliquant une matrice de passage T_{ij} au i premières lignes de A, avec

$$\det(T_{ij}) = \prod_{k=2}^{i} A_{\varphi_1(k),1}\,.$$

Donc,

$$\Delta S_{ij} = \frac{\Delta A_{ij}^{(1)}}{A_{11}} = \frac{\Delta_{ij} * \prod_{k=2}^{i} A_{\varphi_1(k),1}}{A_{11}},$$

et puisque forcément $A_{\varphi_1(2),1} = A_{11}$,

$$\Delta S_{ij} = \Delta_{ij} * \prod_{k=3}^{i} A_{\varphi_1(k),1}.$$

En simplifiant par les $A_{\varphi_1(k),1}$, on obtiendra alors dans tous les cas Δ_{ij} en place $A_{ij}^{(n-1)}$, ce qui nous donne bien la forme (1.3).

• en travaillant dans $\mathbb{Q}$ (c'est à dire en utilisant des relations de type (1.6)) on sait par hypothèse de récurrence (sur les éléments des lignes d'indices inférieurs à n–1) que pour toutes les stratégies on trouvera un rationnel q tel que $A^{(n-1)}$ soit donnée par,

$$Q^{(n-1)} = \begin{pmatrix} A_{11} & A_{12} & \dots & \dots & A_{1n} \\ 0 & \frac{\Delta_{22}}{A_{11}} & \dots & \dots & \frac{\Delta_{2n}}{A_{11}} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \frac{\Delta_{n-1,n}}{\Delta_{n-2,n-2}} & \frac{\Delta_{n-1,n}}{\Delta_{n-2,n-2}} \\ 0 & 0 & \dots & 0 & q \end{pmatrix}.$$

Le déterminant ne variant pas au cours des éliminations (une élimination de type (1.6) est une multiplication par une matrice de déterminant 1), on peut écrire,

$$\det(Q^{(n-1)}) = \frac{A_{11}\Delta_{22}\dots\Delta_{n-1,n-1}}{A_{11}\Delta_{22}\dots\Delta_{n-2,n-2}} q = \det(A) = \Delta_{nn},$$

d'où finalement la valeur de q,

$$q = \frac{\Delta_{nn}}{\Delta_{n-1,n-1}},$$

$Q^{(n-1)}$ satisfait donc bien à (1.7).

Pendant l'algorithme à un pas , les divisions exactes sont effectuées à chaque étape. Ce qui n'est pas précisé dans la proposition 2.1.2 qui n'assure une taille raisonnable que pour les coefficients finaux. Sous certaines conditions, la proposition suivante nous permet de pouvoir réduire aussi les tailles des coefficients intermédiaires.

On supposera désormais que les éliminations sont effectuées dans $\mathbb{Z}$. Le cas rationnel (réductions des fractions au lieu des divisions exactes) est en tout point analogue.

Proposition 2.1.3

Si la stratégie d'élimination utilisée par l'algorithme SeqTLoc vérifie pour les étapes k–1 et k, $1<k\leq n-1$, et pour les lignes i_1, i_2 avec $1\leq i<i_1<i_2\leq n$,

- $\varphi_{k-1}(i_1) = \varphi_{k-1}(i_2) = i$ et $\varphi_k(i_2) = i_1$, alors les coefficients de la i_2-ième ligne de $A^{(k)}$ sont des multiples de $A_{i,k-1}^{(k-2)}$.
- si $\varphi_{k-1}(i_1) = \varphi_k(i_1) = i$ on peut de même diviser exactement la i_1-ème ligne de $A^{(k)}$ par $A_{i,k-1}^{(k-2)}$.

Remarquons que la stratégie d'élimination de l'algorithme à un pas se place pour tous les k dans la première situation.

démonstration 2.1.3

- pour montrer le premier point, il suffit d'écrire les coefficients de $A^{(k)}$ en fonction de ceux de $A^{(k-2)}$,

$$\begin{aligned} A_{i_2j}^{(k)} &= A_{i_1k}^{(k-1)} A_{i_2j}^{(k-1)} - A_{i_2k}^{(k-1)} A_{i_1j}^{(k-1)} \\ &= (A_{i,k-1}^{(k-2)} A_{i_1k}^{(k-2)} - A_{i_1,k-1}^{(k-2)} A_{ik}^{(k-2)}) * (A_{i,k-1}^{(k-2)} A_{i_2j}^{(k-2)} - A_{i_2,k-1}^{(k-2)} A_{ij}^{(k-2)}) \\ &\quad - (A_{i,k-1}^{(k-2)} A_{i_2k}^{(k-2)} - A_{i_2,k-1}^{(k-2)} A_{ik}^{(k-2)}) * (A_{i,k-1}^{(k-2)} A_{i_1j}^{(k-2)} - A_{i_1,k-1}^{(k-2)} A_{ij}^{(k-2)}). \end{aligned}$$

ou encore, à l'aide d'un déterminant,

$$A_{i_2j}^{(k)} = A_{i,k-1}^{(k-2)} \begin{vmatrix} A_{i,k-1}^{(k-2)} & A_{i,k}^{(k-2)} & A_{i,j}^{(k-2)} \\ A_{i_1,k-1}^{(k-2)} & A_{i_1,k}^{(k-2)} & A_{i_1,j}^{(k-2)} \\ A_{i_2,k-1}^{(k-2)} & A_{i_2,k}^{(k-2)} & A_{i_2,j}^{(k-2)} \end{vmatrix}. \qquad (1.16)$$

- en procédant de la même façon pour montrer la deuxième assertion, on obtient toujours à l'aide d'un déterminant,

$$A_{i_2j}^{(k)} = A_{i,k-1}^{(k-2)} \begin{vmatrix} A_{\varphi_{k-1}(i),k-1}^{(k-2)} & A_{\varphi_{k-1}(i),k}^{(k-2)} & A_{\varphi_{k-1}(i),j}^{(k-2)} \\ A_{i,k-1}^{(k-2)} & A_{i,k}^{(k-2)} & A_{i,j}^{(k-2)} \\ A_{i_1,k-1}^{(k-2)} & A_{i_1,k}^{(k-2)} & A_{i_1,j}^{(k-2)} \end{vmatrix}. \qquad (1.17)$$

Ces deux propositions nous permettent de donner un algorithme sans fraction avec pivots locaux, analogue à l'algorithme Seq1, qui calcule la matrice triangulaire sous la forme (1.3) et effectue les divisions exactes dès que cela est possible : on teste à chaque étape k et pour chaque ligne i, i>k,

- si l'une des deux conditions de la proposition précédente est vérifiée. Le cas échéant, on peut simplifier par $A_{\varphi_{k-1}(i),k-1}^{(k-2)}$.
- sinon, le coefficient $A_{\varphi_{k-1}(i),k-1}^{(k-2)}$ est gardé en mémoire, il divisera la ligne i après l'étape i.

On peut rapidement expliquer la fonction des variables utilisées. Pour chaque ligne i et à une étape k donnée, on a

- la variable φ(i) qui correspond à la valeur $\varphi_k(i)$ de la stratégie d'élimination,
- φ_prec(i) qui donne $\varphi_{k-1}(i)$,
- simp_prec(i) qui s'il y a lieu donne le coefficient par lequel on divise la ligne i à cette étape,
- prod_simp(i) qui accumule en les multipliant les simplifications ne pouvant être effectuées qu'en fin d'élimination de cette ligne (à la fin de l'étape i–1) .

• Algorithme SeqLoc (Séquentiel à pivots Locaux)

début
pour i de 1 à n simp_prec(i) = 1, φ_prec(i) = i
prod_simp = 1
pour k de 1 à n-1
  pour i de k+1 à n
    simp = 0
    φ(i) = "donné par la stratégie d'élimination"
    simp_cour(i) = $A_{\varphi(i),k}$
    si φ(i) = φ_prec(i) ou φ_prec(φ(i)) = φ_prec(i)
    alors *Un des deux points de la proposition 2.1.3 est vérifié*
      simp = 1
    sinon *On accumule les simplifications*
      prod_simp(i) = prod_simp(i) * simp_prec(i)
    pour j de k+1 à m *Eliminations*
      $A_{ij} = A_{\varphi(i),k} A_{ij} - A_{ik} A_{\varphi(i),j}$
      si simp = 1 alors $A_{ij} = A_{ij}$ / simp_prec(i) *Division exacte*
    simp_prec(i) = simp_cour(i)
    φ_prec(i) = φ(i)
  pour j de k+1 à m
    $A_{k+1,j} = A_{k+1,j}$ / prod_simp(k+1) *Avec les simplifications accumulées*
fin

Sur le modèle de la démonstration de la proposition 2.1.2, on peut montrer par récurrence que l'algorithme calcule la matrice triangulaire prévue. C'est immédiat pour $n \leq 3$. Par hypothèse de récurrence, l'algorithme appliqué à S nous donnerait pour la position $A_{ij}^{(k-2)}$ à la i–ème étape,

$$\Delta S_{ij} = \Delta_{ij} * \prod_{k=3}^{i} A_{\varphi_1(k),1} .$$

Mais puisqu'une première étape est effectuée sur A, l'algorithme modifie en fait la sous–matrice au fur et à mesure : les lignes d'indice $k \leq i$ sont divisées après la 2–ième (proposition 2.1.3) ou la (k–1)–ème étape par $A_{\varphi_1(k),1}$. Ce qui d'après la valeur de ΔS_{ij} produit bien Δ_{ij}.

I.2.2 Croissance des coefficients intermédiaires

Dans le cas de la méthode de Gauss habituelle où, à chaque étape k, on utilise une unique ligne pivot pour introduire de nouveaux zéros dans la matrice, on a vu que les coefficients intermédiaires s'écrivent comme des mineurs de A, ce qui permet de les borner raisonnablement (1.10).

● Pour les algorithmes à pivots locaux, on peut dans le cadre de stratégies particulières, déduites de la proposition 2.1.3, écrire les coefficients de la matrice à chaque étape en fonction des coefficients aux étapes précédentes :

• Si de l'étape k-p à l'étape k comprises, on effectue l'élimination sur le modèle de Gauss habituel, l'identité de Sylvester [Bar] pour les mineurs bordés nous donne :

$$A_{ij}^{(k)} = \frac{1}{\left(A_{k-p,k-p}^{(k-p-1)}\right)^{p}} \begin{vmatrix} A_{k-p+1,k-p+1}^{(k-p)} & \cdots & A_{k-p+1,k}^{(k-p)} & A_{k-p+1,j}^{(k-p)} \\ \cdots & \cdots & \cdots & \cdots \\ A_{k,k-p+1}^{(k-p)} & \cdots & A_{k,k}^{(k-p)} & A_{k,j}^{(k-p)} \\ A_{i,k-p+1}^{(k-p)} & \cdots & A_{i,k}^{(k-p)} & A_{i,j}^{(k-p)} \end{vmatrix} . \qquad (1.18)$$

• *Str 1*: plus généralement et d'après le premier point de la proposition 2.1.3, si de l'étape k-p à l'étape k, l'élimination est effectuée à *des permutations de lignes près* sur le modèle de Gauss habituel, c'est à dire si l'on peut trouver $i_1,\ldots,i_{p+1}$ vérifiant

$$i_{p+1} \leq i_p \leq \ldots \leq i_1 \leq i, \text{ avec } i_j \geq k-j+1 \text{ pour } j = 1,\ldots,p+1$$

et tels que,

$$\begin{cases} \varphi_{k-p}(i_p) = \varphi_{k-p}(i_{p-1}) = \ldots = \varphi_{k-p}(i) = i_{p+1}, \\ \varphi_{k-p+1}(i_{p-1}) = \ldots = \varphi_{k-p+1}(i) = i_p, \\ \ldots \\ \varphi_k(i) = i_1. \end{cases} \qquad (1.19)$$

Alors on a directement à partir de l'identité de Sylvester,

$$A_{ij}^{(k)} = \frac{1}{(A_{i_{p+1},k-p}^{(k-p-1)})^{p}} \begin{vmatrix} A_{i_p,k-p+1}^{(k-p)} & \cdots & A_{i_p,k}^{(k-p)} & A_{i_p,j}^{(k-p)} \\ \cdots & \cdots & \cdots & \cdots \\ A_{i_1,k-p+1}^{(k-p)} & \cdots & A_{i_1,k}^{(k-p)} & A_{i_1,j}^{(k-p)} \\ A_{i,k-p+1}^{(k-p)} & \cdots & A_{i,k}^{(k-p)} & A_{i,j}^{(k-p)} \end{vmatrix}. \qquad (1.20)$$

• *Str 2*: suivant le deuxième point de la proposition, si pour les mêmes étapes, la ligne d'indice i est toujours éliminée avec la ligne d'indice i_1, et si l'on peut trouver $i_2,\ldots,i_p$ vérifiant (une ligne i_j est toujours éliminée avec la ligne i_{j+1}),

$$i_{p+1} \le i_p \le \ldots \le i_1 \le i, \text{ avec } i_j \ge k\text{-}j+1 \text{ pour } j = 1,\ldots,p+1$$

et tels que,

$$\begin{cases} \varphi_j(i) = i_1 = \varphi(i) & \text{pour } j = k\text{-}p \text{ à } k, \\ \varphi_j(i_1) = i_2 = \varphi(i_1) & \text{pour } j = k\text{-}p \text{ à } k\text{-}1, \\ \cdots \\ \varphi_{k-p}(i_p) = i_{p+1}. \end{cases} \qquad (1.21)$$

On peut considérer que la sous–matrice,

$$S = \begin{pmatrix} (\text{ligne } i_p)^{(k-p)} \\ \cdots \\ (\text{ligne } i_1)^{(k-p)} \\ (\text{ligne } i)^{(k-p)} \end{pmatrix},$$

a été triangularisée. On sait donc que les coefficients de la ligne i en sont des mineurs. Avec les simplifications apportées aux coefficients de $A^{(k-p+1)}$, on a facilement,

$$A_{ij}^{(k)} = \frac{1}{(A_{i_{p+1},k-p}^{(k-p-1)}) \ldots (A_{i_3,k-p}^{(k-p-1)})(A_{i_2,k-p}^{(k-p-1)})} \begin{vmatrix} A_{i_p,k-p+1}^{(k-p)} & \cdots & A_{i_p,k}^{(k-p)} & A_{i_p,j}^{(k-p)} \\ \cdots & \cdots & \cdots & \cdots \\ A_{i_1,k-p+1}^{(k-p)} & \cdots & A_{i_1,k}^{(k-p)} & A_{i_1,j}^{(k-p)} \\ A_{i,k-p+1}^{(k-p)} & \cdots & A_{i,k}^{(k-p)} & A_{i,j}^{(k-p)} \end{vmatrix}. \qquad (1.22)$$

Le lecteur pourra de même s'assurer que (1.18), (1.20) et (1.22) vérifient plus généralement,

$$A_{ij}^{(k)} = \frac{1}{(A_{\varphi_{k-p}(i_p),k-p}^{(k-p-1)}) \dots (A_{\varphi_{k-p}(i_1),k-p}^{(k-p-1)})} \begin{vmatrix} A_{\psi_{k-p+1}(i),k-p+1}^{(k-p)} & \dots & A_{\psi_{k-p+1}(i),j}^{(k-p)} \\ \dots & \dots & \dots \\ A_{\psi_k(i),k-p+1}^{(k-p)} & \dots & A_{\psi_k(i),j}^{(k-p)} \\ A_{i,k-p+1}^{(k-p)} & \dots & A_{i,j}^{(k-p)} \end{vmatrix}, \quad (1.23)$$

où à k fixé les fonctions ψ_{k-j} sont des composées de j+1 stratégies :

$$\psi_k(i) = \varphi_k(i), \text{ et,}$$
$$\psi_{k-j}(i) = \varphi_{k-j}(\varphi_{k-j+1}(\dots\dots \varphi_k(i))\dots)) \text{ pour } 1 \le j \le p. \quad (1.24)$$

Si d'une étape (k_1) à une étape (k_2), on reste dans le cadre d'une de ces stratégies on pourra simplifier, et borner partiellement les coefficients intermédiaires : l'inégalité d'Hadamard peut s'appliquer à chacun des ces déterminants.

Remarques 2.2.1

• Si l'on utilise Str 1, Str 2 ou Str 1 et Str 2 alternativement de la première à la dernière étape, les tailles seront du même ordre. C'est la plus favorable des situations.

• Les méthodes à 2 ou p pas pourront être développées directement à partir de (1.23). On aura des conclusions analogues quant au choix des stratégies.

• Si le choix de la ligne pivot locale est laissé au hasard, on ne pourra pas forcément réduire les coefficients intermédiaires. Considérons la matrice

$$A = \begin{pmatrix} 7 & 1 & 2 & 3 \\ 3 & 4 & 2 & 1 \\ 5 & 2 & 3 & 3 \\ 2 & 1 & 2 & 4 \end{pmatrix}.$$

Et comparons trois stragégies d'élimination différentes dans $\mathbb{Q}$:

• l'élimination de Gauss standard,

• la stratégie Str 2,

• une stratégie arbitraire. Les lignes 2 et 3 sont respectivement éliminées avec les lignes 1 et 2. La quatrième ligne l'est une fois avec la ligne 1 et deux fois avec la ligne 3.

Les étapes de la résolution nous donnent les matrices suivantes,

	Gauss	Str2	Stratégie arbitraire
k=1	$\begin{pmatrix} 7 & 1 & 2 & 3 \\ 0 & \frac{25}{7} & \frac{8}{7} & \frac{-2}{7} \\ 0 & \frac{9}{7} & \frac{11}{7} & \frac{6}{7} \\ 0 & \frac{5}{7} & \frac{10}{7} & \frac{22}{7} \end{pmatrix}$	$\begin{pmatrix} 7 & 1 & 2 & 3 \\ 0 & \frac{25}{7} & \frac{8}{7} & \frac{-2}{7} \\ 0 & \frac{-14}{3} & \frac{-1}{3} & \frac{4}{3} \\ 0 & \frac{1}{5} & \frac{4}{5} & \frac{14}{5} \end{pmatrix}$	$\begin{pmatrix} 7 & 1 & 2 & 3 \\ 0 & \frac{25}{7} & \frac{8}{7} & \frac{-2}{7} \\ 0 & \frac{-14}{3} & \frac{-1}{3} & \frac{4}{3} \\ 0 & \frac{5}{7} & \frac{10}{7} & \frac{22}{7} \end{pmatrix}$
k=2	$\begin{pmatrix} 7 & 1 & 2 & 3 \\ 0 & \frac{25}{7} & \frac{8}{7} & \frac{-2}{7} \\ 0 & 0 & \frac{29}{25} & \frac{24}{25} \\ 0 & 0 & \frac{30}{25} & \frac{80}{25} \end{pmatrix}$	$\begin{pmatrix} 7 & 1 & 2 & 3 \\ 0 & \frac{25}{7} & \frac{8}{7} & \frac{-2}{7} \\ 0 & 0 & \frac{29}{25} & \frac{24}{25} \\ 0 & 0 & \frac{11}{14} & \frac{40}{14} \end{pmatrix}$	$\begin{pmatrix} 7 & 1 & 2 & 3 \\ 0 & \frac{25}{7} & \frac{8}{7} & \frac{-2}{7} \\ 0 & 0 & \frac{29}{25} & \frac{24}{25} \\ 0 & 0 & \frac{-135}{98} & \frac{-328}{98} \end{pmatrix}$

et la matrice triangulaire résultat,

k=3 $\begin{pmatrix} 7 & 1 & 2 & 3 \\ 0 & \frac{25}{7} & \frac{8}{7} & \frac{-2}{7} \\ 0 & 0 & \frac{29}{25} & \frac{24}{25} \\ 0 & 0 & 0 & \frac{64}{29} \end{pmatrix}$.

Certaines fractions n'ont pas été complètement réduites, nous n'avons effectué que les simplifications prévues par l'algorithme. Il faut noter que l'utilisation de la troisième stratégie conduit au dénominateur 98=7*14 : la quatrième ligne ne peut être simplifiée après la deuxième étape. En particulier, pgcd(135,98)=pgcd(135,328)=1.

En pratique, le problème du choix de la stratégie ne se pose bien sûr pas pour une implantation séquentielle ! Nous verrons au §I.4 comment utiliser un nombre maximale de fois les stratégies Str 1 et Str 2 comme stratégies partielles des différentes méthodes parallèles.

I.3 PIVOTAGE

Il nous faut revenir ici sur deux hypothèses faites au début du chapitre.

On appellera *coefficient pivot* à l'étape k, le coefficient $A_{kk}^{(k-1)}$ pour un algorithme à pivots globaux (Gauss standard). Ou tout coefficient $A_{jk}^{(k-1)}$ tel qu'il existe i avec $\varphi_k(i)=j$, pour un algorithme à pivots locaux. Il faut pivoter dès lors qu'un coefficient pivot est nul (le test porte sur p éléments quand l'algorithme est à p pas), en effectuant des permutations de colonnes ou de lignes.

● Permutations de colonnes :
Remarquons d'abord que si à l'étape k, la colonne k est permutée avec une colonne d'indice $l \geq k$, les égalités (1.16) et (1.17) de la proposition 2.1.3 peuvent être adaptées en changeant les indices k en indices l. Inclure le pivotage dans un algorithme à pivots globaux pourra donc se faire simplement en ajoutant une procédure chargée de rechercher un coefficient non nul (sinon le rang décroit) et de permuter.
Si l'algorithme est à pivots locaux, on n'est pas assuré de trouver un indice l tel que tous les coefficients pivots soient non nuls. On s'en persuadera en considérant la matrice (inversible)

$$\begin{pmatrix} 0&1&1&1 \\ 1&1&0&1 \\ 1&0&1&1 \\ 1&1&1&1 \end{pmatrix},$$

et une stratégie qui vérifie $\varphi_1(i)=i-1$ pour tout i. Il faut alors envisager d'effectuer des permutations de lignes.

● Permutations de lignes :
De même que ci–dessus, une simple recherche d'un élément non nul suffit pour modifier l'algorithme s'il est à pivots globaux.
Sinon, pour continuer à réduire les tailles des coefficients, il faut donner un caractère local au pivotage. C'est à dire effectuer au besoin et dans la mesure du possible plusieurs permutations de lignes de manière à rester dans le cadre de la proposition. Supposons qu'à l'étape k, le ou les coefficients pivots soient nuls,

- si une ligne i_1 a été, à l'étape k–1, simplifiée selon Str 2. Ne pouvant continuer les éliminations avec la même ligne i, on cherchera à se ramener au modèle de Str 1. En prenant comme ligne pivot locale de la ligne i_1 une ligne ayant aussi, à l'étape k–1, été éliminée par la ligne i.
- De même, si la simplification à l'étape k–1 a été du type Str 1, on pourra continuer les calculs avec la même ligne pivot, donc selon Str 2. Ou avec une nouvelle qui permette de poursuivre Str 1.

On ne détaillera pas, ni ici, ni en parallèle, l'implantation de ce processus. Il faut seulement noter

qu'une ligne devrait être repérée par un indice absolu, i.e. indépendamment des permutations, pour les valeurs des variables φ(i) et φ_prec(i) utilisées par SeqLoc. On prendrait par exemple son indice dans la matrice initiale.

- Quand le coefficient de tête d'une ligne est déjà nul :

En reprenant les notations du premier point de la proposition 2.1.3, si $A_{i_1,k-1}^{(k-2)}$ est nul, il n'est pas nécessaire d'appliquer de transformation à la ligne i_1 pendant l'étape k–1. Après les deux étapes sur la ligne i_2, (1.16) se réécrit,

$$A_{i_2 j}^{(k)} = \begin{vmatrix} A_{i,k-1}^{(k-2)} & A_{i,k}^{(k-2)} & A_{i,j}^{(k-2)} \\ 0 & A_{i_1,k}^{(k-2)} & A_{i_1,j}^{(k-2)} \\ A_{i_2,k-1}^{(k-2)} & A_{i_2,k}^{(k-2)} & A_{i_2,j}^{(k-2)} \end{vmatrix},$$

et aucune simplification n'est à effectuer. On associera donc à chaque ligne une variable indiquant si elle a été ou non modifiée à l'étape précédente.

Avant de discuter des aspects parallèles de ce que nous venons de voir, concluons sur ces trois premiers paragraphes. Si l'intérêt d'une élimination à pivots locaux est, nous l'avons vu, indéniable dans les corps finis, maintenant que les coefficients sont entiers la situation est tout autre. Si l'on veut réduire au mieux les tailles des coefficients, le libre arbitre n'est plus laissé pour le choix des lignes pivots locales. Et le pivotage devient une opération complexe.
De plus, le problème ne réside pas dans le choix d'un algorithme avec ou sans fraction : il faudrait de la même façon recourir à des calculs de pgcd dans les cas où les simplifications ne peuvent s'appliquer.

I.4 ELIMINATIONS PARALLELES

Après l'étude menée dans la première partie sur l'implantation parallèle de l'élimination de Gauss dans les corps finis, le problème de la répartition des lignes de la matrice dans le réseau est ici considéré résolu : les conclusions de Robert & al. [RTV] reposent essentiellement sur l'hypothèse faite sur le nombre de processeurs du réseau, $P=\alpha n$. Nous avons vu qu'il était nécessaire dans ce cas de trouver la taille optimale pour les blocs de la fonction d'allocation (cf partie 1, §I.3). C'est à dire le meilleur compromis temps de calcul / temps de communication. L'hypothèse étant elle–même justifiée par un coût arithmétique en $O(n^3/P)$ et un coût de communication en $O(n^2)$.

Quand les coefficients sont des entiers de tailles quelconques, le coût arithmétique devient au mieux, sous les hypothèses (1.8), $O(n^5\log^2 n/P)$ (le temps séquentiel divisé par le nombre de processeurs). Pour un coût de communication (calculé plus loin) en $O(n^3\log n)$, ce qui est négligeable si P est en $o(n^2\log n)$: il faut dans ce cas minimiser l'arithmétique, et choisir, comme nous l'avons vu dans la première partie, une fonction d'allocation par blocs consécutifs de taille 1. C'est l'objet de l'hypothèse faite ci–dessous. Une discussion sur les tailles des blocs demanderait de supposer $P=\alpha n^2\log n$ (les coûts de l'arithmétique et des communications seraient alors du même ordre), ce qui ne nous semble pas réaliste.

Hypothèse
On suppose par la suite que les lignes de la matrice sont initialement réparties entre les P processeurs de l'hypercube (numérotés de 0 à P–1) suivant la répartition "périodique" ou "en miroir" par blocs consécutifs de taille 1 (cf part.1 §I.1.2).

Remarque 4.1
Si l'on disposait d'une multiplication rapide permettant de réduire le coût arithmétique, en utilisant l'égalité (1.13), à $O(n^4\log n/P)$ [Bar, CL]. L'hypothèse $P=\alpha n$ pourrait être reprise pour mettre au point une discussion analogue à celle que l'on peut trouver dans [RTV].

Il nous reste donc à étudier le comportement des différentes méthodes de transfert de la ligne pivot (diffusion, pipeline ou pivots locaux, cf Partie 1) du point de vue de la taille des coefficients intermédiaires. Et à donner les coûts correspondants.

I.4.1 Réduction des coefficients

● Pour les algorithmes de **diffusion** ou du **pipeline** la stratégie d'élimination peut bien évidemment être celle de la méthode de Gauss habituelle : avec pivots globaux. Les simplifications s'effectuent alors sans problème (cf §I.3). Nous ne reprenons pas ici les problèmes de pivotage décrits dans la partie 1.

Les mêmes techniques s'appliquent à l'algorithme à deux pas, en prenant une fonction d'allocation par blocs consécutifs de taille 2. Et en transférant deux lignes au lieu d'une.

Pour illustrer le premier point de la remarque 2.2.1. Changeons un instant le modèle d'architecture pour supposer qu'il permette le calcul en parallèle aux communications. Afin de masquer les communications, on peut garder le même algorithme de transfert de la ligne pivot mais commencer les calculs avec une ligne pivot local (la ligne de plus petit indice dans le processeur) tout en attendant la ligne pivot globale. La stratégie d'élimination peut alors se représenter comme suit avec P=4, multiple de 4 et la répartition "périodique",

n° ligne	k=1	k=2	.....						
1	.	.							
2	1	.							
3	1	2							
4	1	2	3						
5	1	2	3	4					
6	2	2	3	4	5				
7	3	3	3	4	5	6			
8	4	4	4	4	5	6	7		
9	1	5	5	5	5	6	7	8	
10	2	2	6	6	6	6	7	8	9
.....									
n-1	3	3	3	3	...	(n-5)	(n-4)	(n-3)	(n-2)
n	4	4	4	4	...	(n-4)	(n-4)	(n-3)	(n-2) (n-1).

Figure I.1 : Diffusion ou pipeline,
les communications sont masquées par les calculs.

Où j est en position (i,k) si la ligne i a été éliminée avec la ligne j à la k-ième étape. Les chiffres soulignés correspondent aux étapes à la fin desquelles la simplification s'est faite sur le modèle de Str 2. En particulier, il est clair que toutes les simplifications pourront être effectuées à chaque étape.

● Pour les algorithmes à **transfert local de la ligne pivot**, de nombreuses stratégies ne permettront pas de réduire les tailles des coefficients au mieux. Et de la même façon rendront difficile le pivotage. Au cours de l'algorithme LPR décrit dans [CTV 2], toutes les lignes sont éliminées à l'aide d'une ligne reçue du processeur précédent sur l'anneau (celle de plus petit indice), la stratégie est représentée sur la figure I.2 avec P=4 et en gardant la même répartition des lignes. On remarquera facilement que pour une ligne donnée et toutes les P étapes (soulignées sur le diagramme), on ne se trouvera ni dans le cadre de Str1 ni dans celui de Str2 (de même que pour l'exemple du §I.2.2).

La taille des coefficients intermédiaires ne pourra être minimisée pendant le déroulement de l'algorithme, mais seulement à la fin de son exécution.

n° ligne	k=1	k=2	.....						
1	.	.							
2	1	.							
3	2	2							
4	3	3	3						
5	4	4	4	4					
6	1	5	5	5	5				
7	2	2	6	6	6	6			
8	3	3	3	7	7	7	7		
9	4	4	4	4	8	8	8	8	
10	1	5	5	5	5	9	9	9	9
.....									
n-1	2	2	6	6	6	...	(n-2)	(n-2)	(n-2) (n-2)
n	3	3	3	7	7	...	(n-5)	(n-1)	(n-1) (n-1) (n-1).

Figure I.2 : Transfert local de la ligne pivot, LPR.
Les réductions des coefficients intermédiaires sont incomplètes.

Au cours de l'algorithme LPM [CTV 2] les lignes sont éliminées avec une ligne pivot locale au processeur, cette dernière étant elle même éliminée à l'aide d'une ligne reçue du processeur précédent. La stratégie est donnée sur la figure I.3.

Pour une ligne i donnée, l'étape i-P+1 ne permet pas d'effectuer de simplification. De plus le manque de simplification dès la ligne 5 (à la deuxième étape) va se répercuter de lignes en lignes jusqu'à l'étape n-P+1. On pourrait montrer que le coût correspondant n'est pas négligeable, même si après une étape k donnée, seules les lignes k+2 et k+3 sont concernées par cette croissance.

n° ligne	k=1	k=2		.....					
1	.	.							
2	1	.							
3	2	2							
4	3	3	3						
5	1	4	4	4					
6	2	2	5	5	5				
7	3	3	3	6	6	6			
8	4	4	4	4	7	7	7		
9	1	5	5	5	5	8	8	8	
10	2	2	6	6	6	6	9	9	9
.....									
n-1	3	3	3	7	7	...	(n-5)	(n-2)	(n-2) (n-2)
n	4	4	4	4	8	...	(n-4)	(n-4)	(n-1) (n-1) (n-1).

Figure I.3 : Transfert local de la ligne pivot, LPM.
Les réductions restent incomplètes.

• Pour arriver à une situation aussi favorable que dans le cas des pivots globaux, il nous faut utiliser la répartition en miroir. Le processeur P_{proc} (les processeurs sont numérotés suivant l'anneau) possède donc les lignes,

$$\begin{cases} 2\,(k\text{-}1)\,P + proc + 1 \text{ et,} \\ 2\,kP - proc \qquad \text{avec } 1 \le k \le n/2P. \end{cases}$$

La stratégie d'élimination est donnée sur la figure I.4 quand n=16 et P=4. De manière plus générale, considérons la ligne d'indice i dans le processeur P_{proc}, on écrira avec $1 \le j \le P$,

$$i = 2lP + j \quad (D1)$$

$$\text{ou, } i = 2lP + P + j \quad (D2).$$

Et explicitons le déroulement des éliminations :

• Dans le cas particulier où i≤P, la ligne i est éliminée i–1 fois à l'aide de la ligne i–1 qui se trouve dans le processeur proc–1.

• 1. Sinon la ligne i est éliminée avec la ligne de plus grand indice inférieur à i, aussi stockée dans P_{proc}. Et ce, le plus longtemps possible, en effectuant donc des simplifications de type Str 2. On a,

$$\varphi_k\,(i) = i - 2j + 1 \text{ pour } 1 \le k \le i\text{-}2j+1.$$

Si j=1, l'élimination est terminée.

• 2. On utilise ensuite une fois la ligne i–2j+2 qui se trouve dans P_{proc-1} ou P_{proc+1} selon

que i est sous la forme (D1) ou (D2),

$$\varphi_{i-2j+2}(i) = i - 2j + 2.$$

On peut effectuer une simplification de type Str 1, puisque forcément,

$$\varphi_{i-2j+1}(i) = \varphi_{i-2j+1}(i-2j+2) = i - 2j + 1.$$

• 3. On termine les éliminations avec la ligne (i–1) qui de même se trouve dans P_{proc-1} ou P_{proc+1} selon la valeur de i,

$$\varphi_k(i) = i - 1 \text{ pour } i-2j+3 \leq k \leq i-1,$$

en effectuant une simplification du type Str 1 à l'étape i–2j+3 puis du type Str 2 jusqu'à la fin.

n° ligne	k=1	k=2	.....												
1															
2	1														
3	2	2													
4	3	3	3												
5	4	4	4	4											
6	3	3	3	4	5										
7	2	2	3	6	6	6									
8	1	2	7	7	7	7	7								
9	8	8	8	8	8	8	8	8							
10	7	7	7	7	7	7	7	8	9						
11	6	6	6	6	6	6	7	10	10	10					
12	5	5	5	5	5	6	11	11	11	11	11				
13	12	12	12	12	12	12	12	12	12	12	12	12			
14	11	11	11	11	11	11	11	11	11	11	11	12	13		
15	10	10	10	10	10	10	10	10	10	10	11	14	14	14	
16	9	9	9	9	9	9	9	9	9	10	15	15	15	15	15

Figure I.4 : Transfert local de la ligne pivot.
Les réductions des coefficients sont complètes.

L'algorithme de transfert des lignes pivots locales (figure I.5, les indices soulignés correspondent aux lignes reçues d'un processeur voisin) n'est pas aussi uniforme que pour LPR ou LPM, mais il reste local. Soit k une étape de l'élimination, on écrira de même avec $1 \leq j \leq P$,

$$k = 2lP + j \quad (D1)$$

$$\text{ou, } k = 2lP + P + j \quad (D2).$$

A l'étape k, un processeur "expéditeur" envoie toujours la ligne de plus petit indice supérieur à k qu'il possède,

• si k est décomposé suivant (D1), les processeurs P_{proc} avec $j-1 \leq proc < P-2$ envoient vers P_{proc+1} et ceux qui vérifient $1 \leq proc \leq j-1$ vers P_{proc-1}.

• si k est sous la forme (D2), les processeurs P_{proc} avec $P-j \leq proc < P-2$ envoient vers P_{proc+1} et ceux qui vérifient $1 \leq proc \leq P-j$ vers P_{proc-1}.

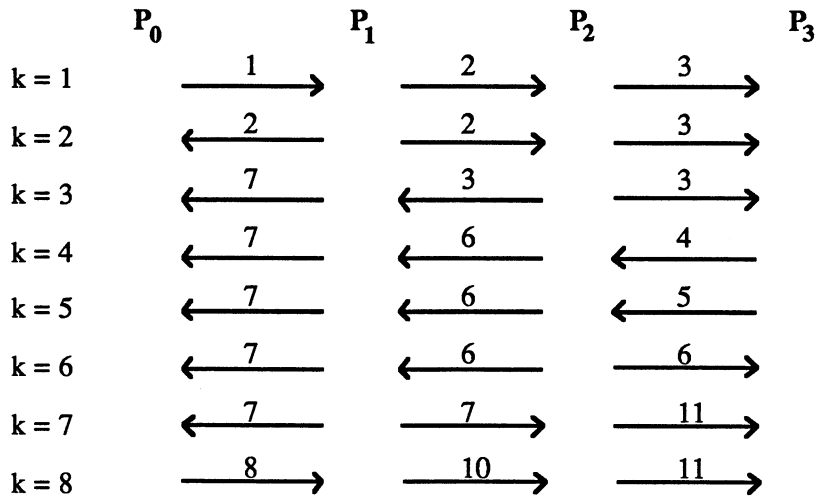


Figure I.5 : Transferts des lignes pivots locales
pour une réduction complète des coefficients intermédiaires.

D'un point de vue théorique, ce procédé nous permet de montrer qu'il est possible de mettre au point l'équivalent parallèle de l'algorithme à un pas en utilisant des lignes pivots locales. En pratique, il sera d'un intérêt bien limité car trop complexe à mettre en œuvre (pivotage). Aussi, ne détaillerons nous pas l'implantation d'une stratégie à deux pas.

I.4.2 Coûts des algorithmes

Les coûts s'obtiennent directement d'après les résultats de la partie 1. Pour le *coût arithmétique* et le *coût mémoire*, on a asymptotiquement le coût séquentiel divisé par P, soit pour l'algorithme à deux pas,

$$\boxed{APar2p \leq \frac{1}{15\,P}\, n^5 \left(1/4 \log_q^2 n + \log_q n \log_q B + \log_q^2 B \right) + O(n^4 \log_q^2 n)}. \qquad (1.25)$$

Et,

$$\boxed{MPar2p \leq \frac{1}{6P}\, n^3 \left(\log_q B + 1/2 \log_q n \right) + O(n^2 \log_q n)}. \qquad (1.26)$$

Pour le *coût des communications*, le coût unitaire de même que pour l'arithmétique correspond à un mot de taille $\log_2 q$. Chaque étape k, demande maintenant le transfert d'une ligne de longueur (n–k), les coefficients étant eux–mêmes des entiers de longueur L_k (cf (1.10)).

• Modèle de complexité

Les quantités (τ, β) du modèle de complexité ne sont plus suffisantes : il serait illusoire de conserver l'hypothèse selon laquelle les n–k entiers sont stockés de manière contiguë en mémoire. On peut par contre supposer qu'un entier est gardé sous la forme de L_k mots consécutifs de taille $\log_2 q$ [Ro]. Et qu'une réception consiste en l'allocation de la place mémoire nécessaire à ces L_k mots, quantité connue après une première communication portant sur la taille de l'entier. Puis en leur réception effective (ce que nous reprendrons au point "communications" du chapitre IV). Si $A(L_k)$ est le coût de l'allocation, celui d'un transfert est donc,

$$C_e = L_k \tau + 2 \beta + A(L_k) \qquad (1.27)$$

Pour une ligne entière, deux possibilités se présentent, on peut répéter n–k fois le processus précédent,

$$C_l = (n-k) (L_k \tau + 2 \beta + A(L_k)) \qquad (1.28)$$

ou, dans un premier temps, stocker les entiers dans un vecteur de $(n-k)L_k$ positions contiguës,

$$C'_l = (n-k) (L_k \tau + D(L_k)) + A((n-k)L_k) + 2 \beta, \qquad (1.29)$$

où $D(L_k)$ donne le coût des déplacements en mémoire. En supposant que $A((n-k)L_k) \leq (n-k)A(L_k)$ (ce qui est vérifié en langage C standard sous le système d'exploitation Unix [Ult]), C'_l est plus faible que C_l si le coût $D(L_k)$ du déplacement d'un entier de taille L_k est inférieur à 2β. Ce que nous avons aisément pu vérifier dans le cadre du FPS T20 puisque,

$$D(L) \approx L \ \mu s, \ 2\beta \geq 1400 \ \mu s,$$

et que par exemple,

$$L \leq 100 \text{ si } n \leq 256 \text{ et } B=100.$$

Il a donc été primordial pour nous de minimiser la quantité de β.

Remarque 4.2.1
Ces communications ne peuvent a priori se faire simultanément sur des liens différents, que si les messages envoyés ou reçus sont identiques : les coûts mémoire sont séquentiels. On pourrait aussi decomposer μ en un coût mémoire (séquentiel) et un coût effectif de communication (parallèle).

• Algorithme du pipeline

Le facteur 3 que l'on retrouve dans chacune des égalités suivantes, concernant le coût de l'algorithme du *pipeline* (part.1), est donné par la démonstration de la proposition 3.2 (part.1 §I.3), et correspond à un temps de réceptions, un temps d'envois et un temps de latences.

L'algorithme à un pas, d'après (1.28) et les relations données ci–dessus pour A() et D(), a donc son coût borné par,

$$CPar1p \leq 3 \left(\sum_{k=1}^{n-1} (n-k)\, L_k \right) \mu + 6\, n\, \beta,$$

soit au total,

$$\boxed{CPar1p \leq \frac{n^3}{4} \left(\log_q n + 2\log_q B \right) \mu + 6n\, \beta} \,, \qquad (1.30)$$

le nouveau temps de transfert élémentaire réel étant majoré par μ,

$$\boxed{\mu = \tau + D(1) + A(1)} \,. \qquad (1.31)$$

De la même façon,

$$\boxed{CPar2p \leq \frac{n^3}{4} \left(\log_q n + 2\log_q B \right) \mu + 3n\, \beta} \,, \qquad (1.32)$$

puisque pour l'algorithme à deux pas le nombre total d'entiers transférés reste le même, seul le nombre d'envois de vecteurs est divisé par deux (on envoie deux lignes au lieu d'une). Concernant les communications, les deux méthodes se différencient donc peu.

Remarques 4.2.2

- On peut obtenir des inégalités analogues en utilisant la relation (1.27).
- Il est particulièrement difficile de modéliser les coûts des allocations des emplacements mémoire, aussi avons nous préférer majorer les coûts totaux (plutôt que de conserver des sommations en $A((n-k)L_k)$), afin surtout de faciliter leur lecture. En outre, la gestion de la mémoire sera mise en évidence expérimentalement dans le dernier chapitre (fig.IV.8 et (4.4)).

De même que dans les corps finis, les algorithmes de diffusion et à pivots locaux réduiraient les quantités que nous avons données, dans une proportion au mieux égale à 3.

Mais le coût des communications reste à un facteur $(n^2 \log n)/P$ de celui de l'arithmétique. Ce facteur est bien plus élevé qu'il ne l'était pour les résolutions modulo p : n/P. C'est d'ailleurs pourquoi dans la première partie, les différences étaient nettement marquées entre les temps d'exécution des différentes méthodes (part.1 fig.III.10). Il faut donc maintenant s'attacher à réduire le coût arithmétique.

Notons pour terminer, que nos conclusions pourrait se généraliser au traitement des matrices non inversibles, une nouvelle fois, selon la démarche suivie pour les résultats de la première partie.

CHAPITRE II

RESOLUTION P-ADIQUE

Nous avons vu au chapitre précedent, combien il peut être difficile de trouver un bon compromis entre le coût des communications et les tailles des coefficients (mise au point de l'algorithme à pivots locaux). Mais aussi que l'arithmétique en précision infinie induit un coût de gestion de la mémoire ("ramasse–miettes",...) dont il faudra tenir compte au moment de la lecture des résultats expérimentaux. Une alternative à ces problèmes réside en l'utilisation de *techniques de congruences*. Des algorithmes basés sur le fameux *théorème des restes Chinois* seront étudiés au chapitre III. Nous nous intéressons ici à l'utilisation des **développements p–adiques.**

A est une matrice carrée et b un vecteur de dimensions n à coefficients entiers bornés en valeur absolue par B. Soit r le rang de A et p un nombre premier.

II.1 L'ALGORITHME SEQUENTIEL

L'algorithme séquentiel que nous reprenons ci–dessous a été donné par Dixon [Dix]. Il peut aussi être trouvé sous une version un peu différente dans [GK]. Ces méthodes déterminent la solution d'un système Ax=b quand la matrice A est inversible. On montre en fait, qu'elles se généralisent au calcul d'une base du noyau et d'une solution particulière quand le déterminant de A est nul et si le système admet au moins une solution.

II.1.1 La matrice A est inversible modulo p

Plaçons nous d'abord dans la situation où A est inversible, et où le nombre premier p choisi ne divise pas le déterminant de A. Autrement dit, A est inversible dans $\mathbb{Z}/p\mathbb{Z}$.

• La première phase de la résolution [Dix] consiste en le calcul de l'inverse de A modulo p. Il suffit pour cela d'utiliser l'algorithme de Jordan qui diagonalise la matrice, en appliquant aussi les transformations sur la matrice identité, qui nous donnera la matrice $C=A^{-1}$ mod p recherchée. Cette étape (comme nous le verrons négligeable en coût) sera calculée sur le modèle des algorithmes de la partie 1, nous n'entrerons donc pas dans les détails.

• C nous permet alors d'effectuer des **itérations de type Newton–Schultz** pour calculer deux suites de vecteurs $\{x_i\}$ et $\{b_i\}$ à l'aide de la relation,

$$\boxed{\begin{cases} b_0 = b \\ x_i = Cb_i \bmod p, \\ b_{i+1} = p^{-1}\,(b_i - Ax_i\,),\ i \geq 0 \end{cases}}\ . \qquad (2.1)$$

Remarquons que les b_i ont des composantes entières puisque,

$$b_i - Ax_i = A\,(Cb_i - x_i\,) = 0 \bmod p. \qquad (2.2)$$

Les itérations se terminent quand on connait les valeurs de x_{m-1} et b_m avec m assez grand donné par [Dix],

$$m = 2\log_p(\overline{B}\,\lambda^{-1}), \qquad (2.3)$$

les numérateurs et dénominateurs des coefficients de la solution x étant bornés en valeur absolue par $\overline{B}$. Et λ est la solution positive de l'équation $\lambda^2+\lambda-1=0$ (λ=0,618...). En appliquant l'inégalité d'Hadamard, on obtient,

$$m \leq 2(n\log_p B + n/2\,\log_p n - \log_p \lambda).$$

Pour simplifier les calculs de complexité, on pourra utiliser,

$$\boxed{m = 2h = 2n\log_p nB}\ . \qquad (2.4)$$

A partir des vecteurs x_i on calcule alors l'**approximation p–adique d'ordre m de la solution.** C'est le vecteur,

$$\bar{x} = \sum_{i=0}^{m-1} x_i\,p^i, \qquad (2.5)$$

qui vérifie,

$$A\bar{x} = \sum_{i=0}^{m-1} p^i\,Ax_i = \sum_{i=0}^{m-1} p^i\,(b_i - p\,b_{i+1}\,) = b_0 - p^m\,b_m\,,$$

ou encore,

$$A\bar{x} \equiv b \bmod p^m. \qquad (2.6)$$

• Un **processus de remontée** donne finalement le moyen de retrouver le vecteur x à l'aide de son approximation p–adique. Nous n'écrirons pas la démonstration qui utilise certaines propriétés des fractions continues.

Notons $x=(x_1,\ldots,x_n)$ le vecteur recherché et $\bar{x}=(\bar{x}_1,\ldots,\bar{x}_n)$. Pour i fixé, si $x_i=p_i/q_i$ on a d'après (2.3) et (2.6),

$$p_i \equiv q_i\,\bar{x}_i \bmod p^m \text{ et } |p_i|,\ |q_i| \leq \lambda p^{m/2}.$$

Ce qui nous assure de pouvoir calculer x_i en utilisant l'algorithme d'Euclide généralisé : soient les suites $\{u_i\}$, $\{q_i\}$ et $\{v_i\}$ définies par,

$$\begin{cases} u_{-1} = p^m, \ v_{-1} = 0 \\ u_0 = \bar{x}_i, \ v_0 = 1 \\ q_i = u_{i-1} / u_i \\ u_{i+1} = u_{i-1} - q_i u_i, \quad v_{i+1} = v_{i-1} - q_i v_i , \end{cases} \qquad (2.7)$$

alors si k est le premier indice tel que $u_k < p^{m/2}$, on sait que,

$$x_i = \frac{u_k}{v_k} .$$

• Coût de la méthode

Dixon a souligné que les entiers rencontrés au cours des itérations (2.1), ont des tailles bien inférieures à celles du chapitre précédent. On peut en effet montrer par récurrence que,

$$\boxed{|b_m| \le nB \frac{1-(1/p)^m}{1-1/p}} . \qquad (2.8)$$

2nB sera la borne utilisée par la suite sur les valeurs absolues des b_i. Quant aux x_i ils sont calculés modulo p. Le *coût arithmétique* en nombre d'opérations sur des mots de taille $\log_2 p$ s'obtient alors facilement, pour les deux produits matrice–vecteur (2.1) on a,

$$A_{prod} = O(n^2 \log_p nB),$$

soit pour les m itérations,

$$A_{iter} = O(h\, n^2 \log_p nB) = O(n^3 \log_p{}^2 nB).$$

Le coût de la remontée (y compris le calcul de $\bar{x}$) des n composantes du vecteur x étant donné par,

$$A_{rem} = O(n\, h^2) = O(n^3 \log_p{}^2 nB).$$

on a au total pour l'algorithme en incluant l'inversion de la matrice,

$$ASeqPA = O(h\, n^2 \log_p nB + n\, h^2 + n^3),$$

$$\boxed{ASeqPA = O(n^3 \log_p^2 nB)} . \qquad (2.9)$$

On a gardé le coût en $O(n^3)$ de Jordan dans la première relation, il est ensuite asymptotiquement négligeable.

La *place mémoire* nécessaire à l'exécution, en mots de taille $\log_2 p$, est déterminée par le stockage de la matrice A, de son inverse modulo p et du vecteur $\bar{x}$ (les emplacements de $\bar{x}$ sont utilisés au

fur et à mesure pour stocker les coefficients de x), ce qui donne,

$$MSeqPA \leq n^2 + n^2 \log_p B + 2nh + O(h),$$

$$\boxed{MSeqPA \leq n^2 (1 + \log_p B + 2\log_p nB) + O(n\log_p nB)} \qquad (2.10)$$

- **Test d'arrêt**

Afin de tenir compte du caractère pessimiste de l'inégalité d'Hadamard, il est possible [GK] d'arrêter prématurément les itérations. Si pour deux entiers m' et m"≤m on applique le processus de remontée aux approximations p–adiques d'ordre m' et m" et si les deux vecteurs calculés sont égaux, alors la solution est connue.
Le coût de cette opération est en $O(n(m'^2+m"^2))$. Pendant que le nombre d'opérations évitées si le test se révèle concluant, est en $O((m–m")n^2\log_p nB)$ en ce qui concerne les itérations (m">m'), et en $O(n(m^2–m'^2–m"^2))$ concernant les processus de remontées. Cette dernière quantité pourra être négative (si le test est inavantageux). Il faudra donc en pratique, trouver un compromis entre les différents coûts et surtout, ne pas augmenter la valeur initiale (2.9).

II.1.2 La matrice A mod p n'est pas inversible

Même si nous n'en discuterons pas l'aspect parallèle, il est intéressant de détailler ici le cas où le rang décroit, qui est moins immédiat que pour la méthode directe, ou comme nous le verrons pour les restes chinois.

Le début de la résolution consiste toujours à appliquer l'algorithme de Jordan. Si un pivot nul est rencontré, seules les permutations de lignes sont autorisées. Et si à une étape k, une colonne entière de zéros est obtenue (pour les éléments d'indices supérieurs à k), le rang décroit et l'algorithme reprend sur une sous–matrice.
Notons J la forme échelonnée réduite aux coefficients diagonaux égaux à 1 (les lignes pivots sont normalisées), calculée à partir de A. On suppose que les lignes nulles ont les plus grands indices. Soit C la matrice des transformations : CA=J. Et r_p le rang de A modulo p.

Si le système initial admet au moins une solution, on montre ci–dessous qu'il est possible de calculer, par un procédé analogue à celui du paragraphe précédent, une solution particulière et une base du noyau de A. Pourvu que r_p soit égal à r. Les cas contraires correspondent aux "mauvais" nombres premiers, ils sont en nombre fini [McC] (diviseurs du déterminant ou de certains mineurs).

La démarche à suivre est donc,

- appliquer la méthode détaillée ci–dessous en supposant que $r_p=r$,
- si les vecteurs calculés sont effectivement tels que Ax=b ou Ax=0, la solution générale est obtenue,
- sinon il faut réappliquer Jordan avec un autre nombre premier p' (pouvant cette fois vérifier det(A) mod p' ≠ 0), et effectuer à nouveau le processus de remontée.

• Remontée d'une solution particulière

On suppose donc que $r_p=p$ et que le système initial Ax=b admet au moins une solution. En utilisant les notations de Mc Clellan [McC], soit $J_A=(j_1,\ldots,j_r)$ les indices, en ordre croissant, des colonnes de J qui participent à sa diagonale. Précisons que J_A ne dépend pas de la méthode utilisée pour Jordan, puisque seules sont effectuées des permutations de lignes. En complétant J_A par rapport à (1,...,n), notons $K_A=(k_1,\ldots,k_{n-r})$. En correspondance avec J_A, l'application de Jordan définit de même un ensemble d'indices croissants de lignes, $I_A=(i_1,\ldots,i_r)$, tel que la sous–matrice carrée $\overline{A}$ de dimension r, construite avec les lignes de I_A et les colonnes de J_A :

$$\overline{A} = A\begin{pmatrix} i_1,\ldots,i_r \\ j_1,\ldots,j_r \end{pmatrix}, \qquad (2.11)$$

soit de rang r.
De la même façon, en complétant I_A on note $H_A=(h_1,\ldots,h_{n-r})$. Les indices h_i peuvent être vus directement dans C, ce sont les indices des colonnes n'ayant pas été modifiées. Ce sont aussi les indices des lignes qui ont été annulées (écrites comme combinaisons linéaires des autres). On peut remarquer que I_A depend de l'algorithme d'élimination, et en particulier de la stratégie suivie pour les permutations.

Considérons maintenant la sous–matrice $\overline{C}$ de C,

$$\overline{C} = C\begin{pmatrix} 1,\ldots,r \\ i_1,\ldots,i_r \end{pmatrix}, \qquad (2.12)$$

par construction on a,

$$\overline{C} \equiv \overline{A}^{-1} \bmod p. \qquad (2.13)$$

Cette égalité nous place dans la situation canonique du paragraphe précédent puisque l'on a det($\overline{A}$) mod p ≠ 0, et va nous permettre de calculer les approximations p–adiques recherchées.

En effet, soit x_0 une solution particulière du système modulo p (donnée par exemple par le vecteur Cb [McC]), on sait qu'il est possible de la choisir telle que,

$$x_{0,k_1} = x_{0,k_2} = \ldots = x_{0,k_{n-r}} = 0. \qquad (2.14)$$

Pour utiliser les matrices $\overline{C}$ et $\overline{A}$, ainsi que la relation (2.13), il nous faut limiter x_0 à ses éléments pouvant être non nul, et considérer le vecteur ξ_0 :

$$\xi_0 = (x_{0,j_1}, x_{0,j_2}, \ldots, x_{0,j_r}).$$

Et de la même façon que l'on obtient $\overline{A}$ à partir de A, limiter b aux lignes I_A, soit β :

$$\beta = (b_{i_1}, b_{i_2}, \ldots, b_{i_r}).$$

La définition de ξ et β nous permet d'écrire,

$$\overline{A}\ \xi_0 \equiv \beta \ \textbf{mod p ou encore,}\ \ \xi_0 \equiv \overline{C}\ \beta \ \textbf{mod p.} \qquad (2.15)$$

En appliquant l'algorithme dans le cas canonique, on calcule l'approximation p–adique ξ d'ordre m d'une solution particulière du système (2.15), ξ vérifie donc,

$$\overline{A}\ \xi \equiv \beta \bmod p^m. \qquad (2.16)$$

Il est alors facile de construire l'approximation p–adique d'ordre m d'une solution particulière du système initial. Soit le vecteur $\overline{x}$ de dimension n dont les coefficients d'indices dans J_A sont donnés par ξ, et complété par des zéros. Et soit $\overline{A}'$ la matrice r*n obtenue en complétant $\overline{A}$ avec les colonnes partielles de A correspondantes. Il est clair que (2.16) est conservée,

$$\overline{A}'\ \overline{x} \equiv \beta \bmod p^m.$$

A étant de rang r, les lignes d'indices $h_1,\ldots,h_{n-r}$ s'écrivent comme combinaisons linéaires des lignes $i_1,\ldots,i_r$. Et puisque que l'on a supposé que Ax=b admet au moins une solution, les mêmes combinaisons linéaires se retrouvent sur les composantes de b. L'égalité précédente peut donc être modifier pour devenir,

$$A\ \overline{x} \equiv b \bmod p^m, \qquad (2.17)$$

$\overline{x}$ n'est autre que l'approximation p–adique recherchée.

- **Remontée des vecteurs du noyau**

Ramenons nous à la situation canonique pour appliquer l'algorithme à la matrice $\overline{A}$, et montrer que n–r vecteurs $z_1,\ldots,z_{n-r}$ formant une base du noyau de A peuvent être calculés. Soit $z_{01},\ldots,z_{0n-r}$ une base du noyau de A modulo p (obtenue dans la matrice J, [McC]). On sait que l'on peut choisir chaque vecteur z_{0i} tel que ses composantes d'indices $k_1,\ldots k_{i-1},k_{i+1},\ldots,k_{n-r}$ soient nulles et tel que la composante k_i soit –1 (on l'a vu Partie 1, §1.2).
Pour introduire les matrices $\overline{A}$ et $\overline{C}$ notons, pour un i fixé à l'aide de J_A,

$$\zeta_0 = (z_{0i_{j1}}, \ldots, z_{0i_{j1}}).$$

Si A_{k_i} est la k_i–ième colonne de A, en nous limitant aux lignes $i_1,\ldots,i_r$ notons $\overline{A}_{k_i}$:

$$\overline{A}_{k_i} = A_{k_i}(i_1, \ldots, i_r). \qquad (2.18)$$

On a par construction une relation analogue à (2.15),

$$\overline{A}\,\zeta_0 \equiv \overline{A}_{k_i} \bmod p \quad \text{et,} \quad \zeta_0 \equiv \overline{C}\,\overline{A}_{k_i} \bmod p, \qquad (2.19)$$

et comme pour l'obtention de (2.16), l'algorithme appliqué avec $\overline{A}$ et $\overline{C}$ nous permet de calculer pour les n–r valeur de i, le vecteur ζ qui vérifie,

$$\overline{A}\,\zeta \equiv \overline{A}_{k_i} \bmod p^m. \qquad (2.20)$$

En complétant $\overline{A}$ avec les colonnes k_i et $k_1,\ldots k_{i-1},k_{i+1},\ldots,k_{n-r}$ de A, et le vecteur ζ en plaçant –1 en position k_i et 0 sinon pour former $\overline{z}$, on peut écrire,

$$\overline{A}'\,\overline{z} \equiv 0 \bmod p^m,$$

qui en reprenant l'argument utilisé pour (2.17) nous donne,

$$\overline{A}\,\overline{z} \equiv 0 \bmod p^m. \qquad (2.21)$$

Il reste finalement à appliquer le processus de remontée pour calculer la solution générale sous sa forme rationnelle. En pratique, les calculs de (2.16) et de (2.20) pour les diférentes valeurs de i seront regroupés, et les itérations (2.1) écrites sous une forme matricielle.

II.2 ASPECTS PARALLELES

Pour Jordan, on pourra se référer à la première partie. On suppose donc qu'à la fin de cette phase initiale, les lignes de A et de sa matrice inverse modulo p sont réparties entre les différents processeurs suivant la fonction d'allocation initiale (ordre des lignes conservé) ou au hasard (pivot local). Il en est de même pour les vecteurs b_i et x_i. Plaçons nous, sans perte de généralité, dans le cas où A est inversible.

• Produits matrice–vecteur

La principale opération demandée à chacune des itérations (2.1) est le calcul de deux produits matrice–vecteur. Chaque processeur va calculer les composantes de b_i et x_i correspondant aux lignes de A et C qu'il possède en mémoire, les indices pouvant ne pas être les mêmes pour les deux matrices. Soit proc, $0 \le proc < P$, fixé. On note maintenant $alloc_A$ et $alloc_C$ les fonctions d'allocation initiales et finales. Au processeur P_{proc} sont associés les indices j, de A et de b_i, tels que $alloc_A(j)=proc$. Et les indices j, de C et de x_i, tels que $alloc_C(j)=proc$.

Pour sa contribution à un produit, P_{proc} doit donc recevoir n/P coefficients de chacun des autres

processeurs. Et lui–même diffuser le même nombre de données. Cette opération dite **"tous vers tous"** (*all to all*) a été étudiée sur l'hypercube par [JH] et [SW], nous ne la détaillons pas dans l'algorithme présenté ci–dessous et utilisons pour les calculs de coûts les résultats des auteurs cités.

Remarque 2.1
Il est essentiel de souligner que **seules des données modulo p doivent être transmises.** Pour les coefficients de x_i c'est leur définition qui le permet. Pour ceux de b_i il suffit de remarquer que, dans les processeurs qui les reçoivent, ils ne servent qu'au calcul de Cb_i mod p; ou encore, seul P_{proc} a besoin de la valeur en multi–précision des $b_i[j]$ avec $alloc_A(j)=proc$, pour calculer les $b_{i+1}[j]$. On prendra donc le modulo avant de les transmettre.

• Le processus de remontée

Cette dernière étape ne présente aucune difficulté : une fois les itérations terminées, les composantes du vecteur $\bar{x}$ sont, de même que la matrice C, distribuées sur le réseau selon la fonction $alloc_C$. Et chaque processeur remonte les coefficients qui lui sont associés, sans aucune communication.

Nous pouvons maintenant donner l'algorithme.

• Algorithme ParPA

Programme du processeur P_{proc}
début
on note $h_1,\ldots,h_{n/P}$ les indices des lignes de A telles que $alloc_A(h_k)=proc$, $1\le k\le n/P$
Jordan (cf partie 1)
calcul des lignes $j_1,\ldots,j_{n/P}$ de C telles que $alloc_C(j_k)=proc$, $1\le k\le n/P$
Les itérations
pour i = 1 à m–1
premier produit
PAR *tous vers tous*
envoyer les $b[h_k]$ mod p, $1\le k\le proc$, à tous les processeurs
recevoir n/P coefficients de chacun des autres processeurs
pour k = 1 à n/P $x[j_k] = \Sigma_l\, C[j_k, l]\, b[l]$ mod p
deuxième produit
PAR *tous vers tous*
envoyer les $x[j_k]$, $1\le k\le proc$, à tous les processeurs
recevoir n/P coefficients de chacun des autres processeurs
pour k = 1 à n/P $b[h_k] = (b[h_k] - \Sigma_l\, A[h_k, l]\, x[l]) / p$
pour k = 1 à n/P remontée du coefficient j_k de la solution *Remontée*
fin

• Coût de l'algorithme

Dans la mesure où l'opération tous vers tous est symétrique (les arbres de diffusion s'obtiennent par des rotations ou translations sur les bits des codages des numéros des processeurs, [JH]), l'hypercube est synchronisé avant chaque produit. Le coût arithmétique est donc asymptotiquement, le coût séquentiel divisé par le nombre de processeurs utilisés, soit,

$$AParPA = O(\frac{h}{P} n^2 \log_p nB + n h^2 + n^3),$$

$$\boxed{AParPA = O(\frac{n^3}{P} \log_p^2 nB)}. \qquad (2.22)$$

La borne minimale pour l'opération de tous vers tous a été donnée dans [JH]. Si les messages au départ de tous les processeurs sont de taille L, et si les messages à l'arrivée (de taille (P–1)L) sont découpés en paquets de taille T (on n'utilise pas la technique du *pipeline* comme pour une diffusion, mais un paquet est attribué à un canal), alors le coût de l'opération est au moins de,

$$\boxed{\frac{(P-1)\,L}{\log_2 P}\tau + (\log_2 P)\,\beta}. \qquad (2.23)$$

cet optimum étant atteint dès que les paquets sont de tailles suffisament importantes,

$$T \geq \sqrt{\frac{2}{\pi}} \frac{P\,L}{\log_2^{3/2} P}.$$

Le coût des communications au cours de ParPA, est donc obtenu en prenant L=n/P dans (2.23), ce qui donne le temps d'une itération (avec un facteur 2 pour les deux produits),

$$C_i = \frac{2\,(P-1)\,n}{P \log_2 P}\tau + 2\,(\log_2 P)\,\beta, \qquad (2.24)$$

puis pour les 2m produits (en simplifiant P-1 et P),

$$C_{iter} = 4\,h\,(\frac{n}{\log_2 P}\tau + \log_2 P\,\beta). \qquad (2.25)$$

Puisque pour Jordan, on a (part.1, algorithme de diffusion),

$$C_J = \gamma n^2 (\frac{4}{\log_2 P} + 1)\,\tau + 5n\log_2 P\,\beta, \quad 1 \leq \gamma \leq 3/2,$$

le coût total de l'algorithme est borné par,

$$CParPA \leq (\frac{4hn}{\log_2 P} + \frac{3}{2}(\frac{4}{\log_2 P} + 1)n^2)\,\tau + ((4h+5n)\log_2 P)\,\beta.$$

L'exécution de Jordan peut donc être asymtotiquement négligée,

$$\boxed{CParPA \leq \frac{4n^2 \log_p nB}{\log_2 P}\ \tau\ + 4n \log_p nB \log_2 P\ \beta}\ . \qquad (2.26)$$

Cette quantité est multipliée par n–r, le nombre des vecteurs d'une base du noyau, si A n'est pas inversible.
Quant au coût mémoire, on peut prendre le coût séquentiel divisé par P,

$$\boxed{MParPA \leq \frac{n^2}{P}(1+ \log_p B + 2\log_p nB) + O(n\log_p nB)}\ . \qquad (2.27)$$

• Test d'arrêt

Soit une itération m" donnée, on peut facilement interrompre les itérations en fin de la boucle principale en i de ParPA, si bien sûr des rationnels ont été déjà remontés à une étape m' antérieure (si (2.7) a aboutit). On a donc dans l'algorithme ci–dessous une variable "remontprec" l'indiquant, elle aura été mise à "faux" au tout début de la résolution. Les rationnels éventuellement remontés à l'étape m' sont stockés dans "ratprec" (ayant au début, des valeurs non rationnelles).

Algorithme TA

Programme du processeur P_{proc}
début
obtenu=vrai
pour k = 1 à n/P et tant que obtenu=vrai
si (2.7) n'aboutit pas à partir de l'approximation p–adique d'ordre m"
obtenu = faux
sinon affecter le rationnel à rat[j_k]
tous_vers_tous(obtenu) en effectuant le produit des variables reçues
remont = obtenu

si remont = remontprec = vrai et tant que obtenu=vrai
pour k = 1 à n/P et tant que obtenu=vrai
si rat[j_k] ≠ ratprec[j_k] obtenu=faux
tous_vers_tous(obtenu) en effectuant le produit des variables reçues
si obtenu=vrai retourner(vrai) *Interruption possible*
sinon
remontprec = remont
si remont = vrai
pour k = 1 à n/P ratprec[j_k] = rat[j_k]
fin

Pour les deux phases du test : calcul de (2.7) puis test d'égalité, les processeurs ont une conclusion locale qu'ils communiquent ensuite par deux *tous vers tous* dans l'hypercube. Il faut donc exécuter ce processus une fois les calculs de l'étape en cours terminés, s'il retourne la valeur "vrai" les rationnels recherchés sont obtenus. La résolution peut prendre fin.

La méthode utilisant les développements p–adiques est, nous venons de le voir, particulièrement simple à implanter sur l'hypercube : les produits matrices–vecteurs nécessitent simplement l'opération *tous vers tous* sur des entiers modulo p. Et ne demandent, en particulier, aucune gestion mémoire pour le transfert d'entiers longs. Les remontées se font ensuite séquentiellement sur chaque processeur. Ceci permet d'obtenir un coût de communication en $O(n^2 \log n)$ alors qu'il était, pour la méthode directe, en $O(n^3 \log n)$. Aussi, si l'on remarque que le rapport des coûts mémoire est le même que celui des communications, et qu'il est en $O(n^2)$ pour l'arithmétique, on peut s'attendre, en pratique, dès les petites valeurs de n, à une nette supériorité de la version parallèle de l'algorithme de Dixon.

CHAPITRE III

EN APPLIQUANT LE THEOREME DES RESTES CHINOIS

Après la présentation des méthodes directes et p–adique (utilisant un unique nombre premier), nous nous attachons dans ce chapitre à développer des algorithmes parallèles à partir du théorème des restes Chinois. On sait que les méthodes utilisant ce principe ont un très bon comportement en séquentiel, aussi bien pour des coefficients entiers [CL], que pour des polynômes denses [McC].

De même que dans les chapitres précédents, n est la dimension de la matrice A, b est le vecteur bordant. B est une borne sur les valeurs absolues de leurs coefficients.

III.1 LE THEOREME DES RESTES CHINOIS

Le théorème des restes chinois [Knu, chap. 4.3.2] peut être directement appliqué pour calculer le vecteur solution d'un système linéaire. Nous utiliserons une de ses conséquences et sa démonstration sous la forme suivante.

- ***Une conséquence du Théorème des Restes Chinois***

Soient $p_1, p_2,\ldots, p_h$, h nombres premiers. Et soient $x_1, x_2,\ldots, x_h$ entiers. Alors si l'on note $P_t=p_1p_2\ldots p_h$, il existe un unique entier x qui vérifie

$$-P_t/2 < x < P_t/2, \text{ et } x \equiv x_i \bmod p_i \text{ pour } 1\leq i \leq h.$$

Une preuve constructive du théorème permet d'obtenir x en h étapes à partir de ses résidus :

$$x = y_1 + y_2p_1 + y_3p_1p_2 + \ldots + y_hp_1p_2\ldots p_{h-1}, \qquad (3.1)$$

où, en travaillant avec la représentation équilibrée des corps $\mathbb{Z}/p_i\mathbb{Z}$ i.e. $-p_i/2 < y_i < p_i/2$ ($p_i>2$),

$$\begin{aligned} y_1 &= x_1 \bmod p_1, \\ y_2 &= (x_2 - y_1)c_{12} \bmod p_2, \\ y_3 &= ((x_3 - y_1)c_{13} - y_2)c_{23} \bmod p_3, \\ &\ldots \\ y_h &= (\ldots((x_h - y_1)c_{1h} - y_2)c_{2h} - \ldots - y_{h-1})c_{h-1,h} \bmod p_h. \end{aligned} \qquad (3.2)$$

Les constantes c_{ij} étant données par :

$$c_{ij}p_i \equiv 1 \bmod p_j \quad \text{pour} \quad 1 \leq i < j \leq h. \qquad (3.3)$$

On suppose par la suite que la matrice A est inversible. Dans le cas contraire, la démarche présentée ci–dessous pourrait être suivie pour une solution particulière et pour les vecteurs d'une base du noyau [McC].

Si d est le déterminant de A, les coefficients du vecteur solution sont de la forme x/d, x et d étant bornés en valeur absolue par l'inégalité d'Hadamard :

$$|x|, |d| \leq B^n n^{n/2}.$$

On peut donc appliquer le théorème au déterminant et aux entiers x (les résidus sont obtenus en résolvant les systèmes triangulaires dont les vecteurs bordants ont été multipliés par le déterminant) et en considérant h nombres premiers tels que leur produit P_t vérifie,

$$P_t \geq 2B^n n^{n/2}.$$

Si tous les nombres premiers utilisés sont supérieurs à p, il suffit donc de prendre :

$$h \geq n\log_p B + n/2 \log_p n + \log_p 2.$$

Pour simplifier, on utilisera

$$\boxed{h = n\log_p nB} \qquad (3.4)$$

dans les calculs de complexité (par analogie avec la quantité (2.4) du paragraphe précedent).

• Coût de la méthode

Le *coût arithmétique* est obtenu ici en termes d'opérations sur des nombres de taille $\log_2 q$, q étant une borne supérieure pour les nombres premiers utilisés.Les h résolutions modulo demandent au plus de l'ordre de hn^3 additions, multiplications et modulo (un modulo est effectué après chaque somme et produit). Pour la remontée d'un coefficient ou du déterminant, l'étape consistant en des calculs modulo p_i (3.2) et l'étape de *reconstruction des entiers* une fois les produits des nombres premiers calculés (3.1) requièrent respectivement $3h^2/2$ et h^2 opérations. A ces coûts viennent s'ajouter ceux des calculs des constantes c_{ij} et des produits des nombres premiers qui sont de $h^2/2$ inversions modulo et $h^2/2$ multiplications.

D'où ASeqCr le coût arithmétique total,

$$\text{ASeqCr} \leq n^3 h + 5/2\, nh^2 + O(h^2),$$

$$\boxed{\text{ASeqCr} \leq n^4 \log_p nB + 5/2\, n^3 \log_p^2 nB + O(n^2 \log_p^2 nB)}. \qquad (3.5)$$

La place mémoire (en mots de taille $\log_2 q$) nécessaire au stockage de la matrice A et des matrices réduites modulo p_i avant chaque résolution modulo est bornée par

$$n^2 + n^2 \log_q B + O(n).$$

En ajoutant nh + O(h), le coût du stockage des n coefficients du vecteur solution on a le coût pour l'algorithme :

$$\boxed{MSeqCr \leq n^2 (1+ \log_q B + \log_p nB) + O(n\log_p nB)} . \qquad (3.6)$$

Comme le souligne D.E.Knuth, un avantage de la méthode est que son coût provient essentiellement de l'arithmétique modulo p_i. Cet avantage sera confirmé par la suite au vu des résultats expérimentaux.

Remarques 1.1
Nous ne tenons pas compte ici du fait qu'il existe un nombre fini de *mauvais nombres premiers*, ce sont ceux qui divisent le determinant (supérieurs à p il y en a au plus h) : ils ne permettent pas de conserver le résultat de la résolution modulo correspondante. Soulignons aussi que le théorème des restes chinois est plus général que le résultat que nous utilisons : il peut s'appliquer pour des p_i seulement premiers entre eux, mais il faut dans ce cas vérifier, si le vecteur x remonté est tel que Ax=b.

- **Test d'arrêt récursif**

L'inégalité d'Hadamard peut-être particulièrement pessimiste pour certaines classes de matrices. Un **test d'arrêt récursif** (les valeurs testées dépendent les unes des autres) donné par E.H.Bareiss et S.Cabay [Bar, Cab] permet alors de réduire le coût de la remontée de la solution. Si les nombres premiers sont rangés en ordre croissant, soit max le plus petit entier tel que,

$$\| A,b \|_\infty < 2p_1 \cdots p_{max} , \qquad (3.7)$$

alors si max zeros consecutifs sont rencontrés dans les représentations à bases multiples (3.1) du déterminants et des coefficients du vecteur solution, ce dernier est obtenu et la remontée peut être interrompue.
Dans certains cas d'utilisation de ce test, le déterminant de la matrice peut n'être obtenu qu'à un facteur près.

Nous présentons dans les paragraphes suivant diverses implantations possibles de la méthode en parallèle. nous verrons que les coûts arithmétiques seront tous identiques. Les coûts de communication et les coûts mémoire seront donc déterminants au moment de conclure.

Les algorithmes sont d'abord donnés en supposant que l'on effectue en deux phases distinctes : les h résolutions modulo et les remontées des n coefficients. Nous expliquons ensuite pour chacun des cas les modifications qui permettent d'inclure le test d'arrêt récursif.

Revenons brièvement sur les algorithmes employés pour les résolutions modulo. Nous les avons présentés dans la première partie, sans les détailler nous les utiliserons ici comme processus de

base. On rappelle qu'ils supposent la matrice et donc le vecteur solution, $x=(x_1,\ldots,x_n)$, répartis par lignes sur l'hypercube : la ligne i au processeur alloc(i). Une fois la résolution terminée, le déterminant d mod p_i est connu sur tous les processeurs ayant participé à la résolution modulo p_i. Précisons alors qu'il faut utiliser un algorithme qui *conserve l'ordre des lignes de la matrice* (donc pas un algorithme à pivots locaux). L'étape de remontée deviendrait sinon trop complexe en pratique : situer puis réunir les résidus de chacun des coefficients serait coûteux. H et n sont divisibles par P, le nombre de processeurs de l'hypercube. Enfin, pour aucun des algorithmes, ne sera explicité le cas où le nombre premier choisi divise le déterminant puisqu'il suffit d'effectuer une résolution supplémentaire.

III.2 LES REMONTEES SONT SEQUENTIELLES

La méthode la plus simple (une fois l'étude de la première partie menée à bien) pour implanter la résolution en utilisant le théorème des restes chinois est d'utiliser la totalité du réseau pour effectuer chacune des h résolutions modulo. Après cette première phase chaque processeur P_{proc} possède les résidus de n/P des coefficients du vecteur solution : des x_k tels que alloc(k)=proc. Et tous peuvent donc travailler indépendamment les uns des autres pendant l'étape finale afin de remonter les x_k qui leurs sont associés.
On pourrait supposer qu'un seul des processeurs est chargé du calcul du déterminant (le coût est négligeable devant celui associé à n/P coefficients). La remarque 3.1.2 faite au paragraphe suivant, permettra néanmoins de le distribuer.

Nous donnons ci–dessous une version de l'algorithme de remontée séquentielle,

- **Algorithme RemSeq** (Remontées Séquentielles)

Programme du processeur P_{proc}
début
pour i = 1 à h *Algorithmes du pipeline ou de diffusion (cf partie 1)*
calcul de x_k mod p_i pour les k tels que alloc(k) = proc
pour i = 1 à h *Local à chaque processeur : sans communication*
pour les k tels que alloc(k) = proc
i-ème étape de la remontée de x_k suivant (3.2)
pour i = 1 à h *Distribué ou non sur le réseau*
i-ème étape de la remontée du déterminant suivant (3.2)

pour les k tels que alloc(k) = proc
reconstruction de x_k suivant (3.1)
reconstruction du déterminant suivant (3.1)
fin

NB : la première étape (i=1) de la remontée ne demande aucun calcul.

Au paragraphe concernant l'implantation des algorithmes nous verrons comment la deuxième boucle peut être *vectorisée.*

• Coût de l'algorithme

En se plaçant dans le cas où n >> P (pour ne pas reprendre ici la discussion correspondant au cas P=αn, cf partie 1. chap I), le coût arithmétique, en nombre d'opérations sur des mots de taille $\log_2 q$, d'une résolution modulo peut être borné par $n^3/P + O(n^2)$. En totalisant les h résolutions et les n/P remontées par processeur, on obtient AParCr le coût arithmétique total :

$$AParCr \leq n^3h/P + 5/2P\ nh^2 + O(h^2),$$

$$\boxed{AParCr \leq \frac{1}{P}(n^4\log_p nB + 5/2\ n^3\log_p^2 nB) + O(n^2\log_p^2 nB)}. \qquad (3.8)$$

Si l'on utilise l'algorithme de diffusion (et sa meilleure fonction d'allocation) pour la première phase, rappelons de même que le coût en communications de chacune des résolutions modulo est au pire de l'ordre de $n^2\tau + 5n\log_2 P\beta$ (β et τ pour des mots de taille $\log_2 q$). D'où au total,

$$CRemSeq \leq hn^2\tau + 5hn\log_2 P\beta + O(nh),$$

$$\boxed{CRemSeq \leq n^3\log_p nB\ \tau + 5n^2\log_2 P\log_p nB\ \beta + O(n^2\log_p nB)}. \qquad (3.9)$$

Cette quantité, de même que le sont les coûts des résolutions modulo, est indépendante de la dimension du noyau (si la matrice n'est plus supposée inversible).

Enfin, la place mémoire nécessaire *par processeur* est simplement le quotient du coût séquentiel par le nombre de processeurs,

$$\boxed{MRemSeq \leq \frac{n^2}{P}(1+\log_q B + \log_p nB) + O(n\log_p nB)}. \qquad (3.10)$$

• Test d'arrêt récursif

Quand les remontées sont séquentielles, le test récursif est aussi simple à implanter que dans le chapitre précédent : chaque processeur effectue le test localement puis communique son résultat par une opération du type *tous vers tous* (§II.2) aux autres processeurs. La décision d'interrompre ou non la remontée peut alors être prise.

Algorithme TAR

Programme du processeur P_{proc}
début
obtenu = vrai
pour les k tels que alloc(k) = proc
s'il n'y a pas max zéros dans la représentation à bases multiples (3.1) de x_k
obtenu = faux
opération tous vers tous
PAR
envoyer (obtenu) à tous les processeurs
pour j de 0 à P-1 et j≠proc
recevoir (obt[j]) du processeur P_j

pour j de 0 à P-1 et j≠proc
obtenu = obtenu * obt[j]
obtenu = obtenu * (résultat du *test global* pour le déterminant)
retourner (obtenu)
fin

Pour le déterminant (si son calcul est distribué), un algorithme plus global sera utilisé (§ III.3.1).

L'algorithme RemSeq peut être facilement modifié pour permettre le test récursif. Il suffit de regrouper les trois premières boucles en une seule, d'appliquer TAR à la fin de chaque étape i pour déterminer si la solution est obtenue, et le cas échéant de passer à l'étape de reconstruction des entiers.

III.3 LES RESOLUTIONS MODULO SONT SEQUENTIELLES

Nous verrons plus loin que les intérêts du premier algorithme sont l'équidistribution des données et le coût négligeable des communications pendant la deuxième phase, celle de la remontée. A l'opposé, l'algorithme présenté dans ce paragraphe assure que c'est la première phase qui s'effectuera sans transfert de donnée. Chacun des P processeurs va calculer h/P résolutions modulo en séquentiel. Les résidus des x_k seront alors répartis sur tout le réseau : il sera nécessaire de communiquer pour la remontée de chacun des coefficients.

Si A un algorithme permettant de calculer le vecteur solution x. Et si le coefficient x_k ou le déterminant sont obtenus par A sous la forme (3.1) :

$$x_k = y_{1k} + y_{2k}\,p_1 + y_{3k}\,p_1p_2 + \ldots + y_{hk}\,p_1p_2\cdots p_{h-1},$$

ou,

$$d = d_1 + d_2 p_1 + d_3 p_1p_2 + \ldots + d_h p_1p_2\cdots p_{h-1}. \qquad (3.11)$$

Alors le h–uplet de nombres premiers $(p_1,p_2,\ldots,p_h)$ est appelé **h–uplet de bases de x_k ou du déterminant pour l'algorithme A.**
Remarquons que les x_k et le déterminant ont tous le même h–uplet de bases pour l'algorithme RemSeq (de même que pour les algorithmes séquentiels).

Nous sommes amenés ici à considérer deux cas de figure pour les algorithmes du type RmS (Résolutions modulo Séquentielles),

- les x_k ont tous le même h–uplet de bases (global) : algorithme RmS-G (§ III.3.1).
- Le h–uplet de bases varie d'un x_k à l'autre (non global) : algorithme RmS-NG (§ III.3.2).

III.3.1 Le h-uplet de bases est global

Nous ne revenons pas sur les résolutions modulo, chaque processeur en effectue h/P en séquentiel et possède donc à la fin de cette première étape, h/P résidus de chacun des x_k et du déterminant.

• Les h étapes de la remontée

On peut maintenant utiliser la remarque faite par Knuth [Knu, chap. 4.3.2] sur le parallélisme inhérent à la construction (3.2) : à une étape i de la remontée, on peut simultanément affecter (on supposera que les $y_{.,k}$ ont été initialisés aux valeurs des résidus des x_k) :

$$y_{jk} = (y_{jk} - y_{ik})c_{ij} \bmod p_j \quad \text{pour } i<j\leq h \text{ et pour } 1\leq k\leq n+1, \qquad (3.12)$$
$$\text{où } y_{j,n+1} = d_j.$$

Notons maintemant allocp(i) le numéro du processeur sur lequel s'est effectuée la résolution modulo p_i. Le **rapprochement avec l'élimination de Gauss** s'impose : à chaque étape i, le processeur $P_{allocp(i)}$ doit transmettre le (n+1)–uplet $(y_{i1},\ldots,y_{in},y_{i,n+1})$ aux autres processeurs pour qu'il puissent effectuer (3.12). Nous avons regrouper ci–dessous les correspondances qu'il est possible de faire entre les deux méthodes.

Elimination de Gauss	*Algorithme RmS-G*
• n étapes i de résolution,	• h étapes i de résolution,
• ligne L_i pivot,	• (n+1)–uplet $(y_{i1},\ldots,y_{in},y_{i,n+1})$,
• allocation des lignes de la matrice,	• allocation des nombres premiers,
• $(L_j$ - pivot * $L_i)$ mod p pour $i<j\leq h$.	• $y_{jk}=(y_{jk} - y_{ik})c_{ij}$ mod p_j pour $i<j\leq h$, $1\leq k\leq n$.

Figure III.1 : correspondances entre les algorithmes de Gauss et RmS-G.

Sans revenir sur les différentes fonctions d'allocation possibles et en se plaçant toujours dans le cas où n >> P, on pourra donc développer la technique de diffusion (sans pivotage) ou du pipeline. L'algorithme correspondant est donné plus loin : les communications seront spécifiées à l'aide des deux instructions "envoyer" et "recevoir" (par un algorithme de diffusion ou dans un sens sur l'anneau).

Remarques 3.1.1

• L'ordre des calculs doit être ici rigoureusement respecté (hypothèse d'unicité du h–uplet de bases), on n'a donc pas l'analogue de l'algorithme à pivots locaux.

• Les coûts donnés ci–dessous pourraient bien entendu être réduits par l'utilisation de méthodes à plusieurs pas.

Coût

Il suffit de reprendre les résultats de la partie 1 pour obtenir les coûts CD et CP et AR (diffusion, pipeline et arithmétique) :

$$CD \leq hn\,\tau + h\log_2 P\,\beta,\quad CP \leq 3hn\,\tau + 3h\,\beta + O(h),$$
$$AR \leq 3nh^2/2P + O(h^2),$$

soit,

$$CD \leq n^2\log_p nB\,\tau + n\,(\log_2 P)\,\log_p nB\,\beta,$$
$$CP \leq 3n^2\log_p nB\,\tau + 3n\,\log_p nB\,\beta, \qquad (3.13)$$

et,

$$AR \leq 3n^3\,\log^2_p nB\,/2P\; + O(n^2\log^2_p nB). \qquad (3.14)$$

• L'étape de reconstruction des entiers

Pour éviter toute communication durant cette dernière étape, il faut qu'au cours de la remontée

• (c1) tous les $y_{i,k}$ $1\leq i\leq h$, k fixé, aient été stockés dans un même processeur P_{proc}.

Sans changer les coûts précédents (obtenus avec une fonction d'allocaton par blocs de taille 1), on peut faire l'hypothèse suivante,

• (h2) tous les processeurs ont reçu tous les (n+1)–uplets.

Pour vérifier la condition (c1), il suffit alors qu'à chaque étape i de la remontée chacun des processeurs P_{proc} stocke $y_{i,k}$ pour n/P valeurs fixées (fonction de proc) de k, par exemple celles vérifiant alloc(k)=proc. Précisons ici que cela représente un coût mémoire de l'ordre de n (ce qui sera négligeable) puisque toutes les P étapes e telles que allocp(e)=proc ces valeurs pourront remplacer les n emplacements libérés par les $(y_{e,1},\ldots,y_{e,n})$.

Sous ces hypothèses on se retrouve alors, pour l'étape de reconstruction, dans la situation du paragraphe III.2.

Remarques 3.1.2

• La reconstruction du déterminant, négligeable en coût, pourrait être affectée à un seul des processeurs. Il est néanmoins intéressant de donner ici une méthode pour la distribuer sur le réseau. Pour cela les d_i restent stockés (au cours de la remontée) sur le processeur qui les a calculés. Chacun peut ensuite commencer à effectuer les sommes partielles correspondant à (3.11). Le déterminant est finalement obtenu en sommant ces dernières deux à deux sur l'hypercube en $\log_2 P$ étapes (processus en arbre). Pour les coûts associés on obtient facilement $h^2/P + O(h)$ en opérations arithmétiques et $\log_2 P(h\tau + \beta)$ en communications.

• Si la condition (c1) ne peut être vérifiée (par exemple si la fonction d'allocation est par blocs de taille h/P) on pourra conserver la méthode naturelle de stockage et reformer les entiers comme expliqué ci–dessus mais avec un coût élevé de $\log_2 P(nh\tau + \beta)$.

Coût

La reconstruction se fait donc sans communication; le coût arithmétique est simplement donné par,

$$\begin{aligned} ARF &\leq nh^2/P + O(h^2), \\ ARF &\leq n^3 \log^2_p nB /P + O(n^2 \log^2_p nB). \end{aligned} \qquad (3.15)$$

On peut maintenant donner une version complète de l'algorithme RmS–G,

- **Algorithme RmS-G**

Programme du processeur P_{proc}
début
pour i = 1 à h et allocp(i) = proc *Résolutions modulo : sans communication*
pour k = 1 à n
$y_{ik} = x_k \bmod p_i$
$y_{i,n+1} = d \bmod p_i$
pour i = 1 à h *Remontée des coefficients : distribué sur le réseau*
si proc = allocp(i)
envoyer le n–uplet $(y_{i1},\ldots,y_{in},y_{i,n+1})$
stockage de $d_i = y_{i,n+1}$
sinon
recevoir $(y_{i1},\ldots,y_{in},d_i)$
(B1) pour j de i+1 à h et tel que allocp(j) = proc
pour k de 1 à n+1
$y_{jk}=(y_{jk} - y_{ik})c_{ij} \bmod p_j$
stockage de y_{ik} pour les k tels que alloc(k) = proc
pour les k tels que alloc(k) = proc *Reconstruction sans communication*
reconstruction des x_k
reconstruction du déterminant *Distribué ou non*
fin

• Coût de l'algorithme

En totalisant AR et ARF donnés en (3.14) et (3.15) avec le coût des résolutions modulo, on obtient un coût arithmétique du même ordre que celui de RemSeq (3.8),

$$AParCr \leq n^3h/P + 5/2P\, nh^2 + O(h^2),$$

$$\boxed{AParCr \leq \frac{1}{P}(n^4\log_p nB + 5/2\, n^3 \log_p^2 nB) + O(n^2\log_p^2 nB)}.$$

Pour les communications, le total est donné par exemple par le coût CD, donné en (3.13), de la méthode de diffusion

$$CRmS\text{-}G \leq hn\,\tau + h\log_2 P\,\beta,$$

$$\boxed{CRmS\text{-}G \leq n^2\log_p nB\,\tau + n\,\log_p nB\log_2 P\,\beta}. \qquad (3.16)$$

Cette quantité sera clairement multipliée par n–r (le nombre de vecteurs à calculer) si le noyau de A n'est pas nul.

En ce qui concerne la place mémoire on supposera, sans perte de généralités, que la matrice initiale est stockée une seule fois pour tout le réseau. En prenant en compte le stockage des matrices pour les résolutions modulo et celui des coefficients du vecteur solution, on a alors,

$$MRmS\text{-}G \leq n^2 + n^2\log_q B\ /P + nh/P + O(h).$$

$$\boxed{MRmS\text{-}G \leq \frac{n^2}{P}(P + \log_q B + \log_p nB) + O(n\log_p nB)}. \qquad (3.17)$$

• Test d'arrêt récursif

L'algorithme TAR peut être directement utilisé si, de même que précédemment, l'hypothèse (h2) est faite et la condition (c1) est vérifiée. Si la remontée du déterminant est distribuée ou s'il est impossible de satisfaire à (c1) le test sera effectué en totalisant partiellement les nombres de zeros dans les représentations à bases multiples sur chaque processeur puis sur l'hypercube en $\log_2 P$ étapes. Mais il faut remarquer qu'en fait, la boucle (B1) *anticipe sur les étapes* de RemSeq : à l'étape i de RmS–G plus de calculs auront été effectués qu'à l'étape i de la construction (3.2).

Soit G le plus petit nombre d'étapes permettant l'obtention de la solution (en comptant les étapes du test). Quand RemSeq aura effectué pour la remontée de l'ordre de $3nG^2/2$ opérations avant que celle–ci ne soit interrompue, RmS-G en aura effectué $3n(G^2/2 + (h–G)G)$: en tenant compte de la classe des matrices traitées,

• si G est proche de h, on pourra modifier Rms–G en regroupant les deux premières boucles et en appliquant TAR après le transfert du n–uplet. Une résolution modulo sera calculée sur chaque processeur toutes les P étapes : le coût superflu sera borné par (h–G)G plus le coût d'une de ces résolutions sur un processeur.

• si la borne h est trop pessimiste, il faut différer les calculs : chaque processeur peut garder en mémoire tous les n–uplets qu'il reçoit, anticiper toutes les P étapes sur les calculs des P prochains y_{ik} et n'effectuer sinon à chaque étape i que les calculs nécessaires au nouveaux y_{ik} . Le coût de communication et le coût arithmétique (pour son terme dominant) restent inchangés, mais la place mémoire nécessaire est, asymptotiquement en n, multipliée par P,

$$\text{MRmS-G'} \leq n^2 + n^2 \log_q B / P + nh + O(h).$$

$$\boxed{\text{MRmS-G'} \leq \frac{n^2}{P}(P + \log_q B + P\log_p nB) + O(n\log_p nB)} \quad . \qquad (3.18)$$

Le nombre d'opérations effectuées en trop est alors borné par PG plus le coût d'une résolution modulo sur un processeur, le plus mauvais cas étant obtenu si G=lP+1.

Appliquer le test récursif dans le cas général se révèle ici coûteux en place mémoire. Nous nous attachons dans le paragraphe suivant à limiter les anticipations **en supprimant la diffusion globale des n–uplets** (ce que l'on peut rapprocher avec le raisonnement tenu pour les *algorithmes avec pivots locaux*). Pour permettre de supprimer cette diffusion globale, il ne faut pas travailler sur tout le réseau avec un même p_i à l'étape i : *le h–uplet de bases va varier* d'un x_k à l'autre.

III.3.2 Les x_k n'ont pas tous le même h-uplet de bases

Nous n'utiliserons plus dans ce paragraphe qu'un *anneau* inscrit dans l'hypercube, en particulier les communications se borneront à des envois (resp. receptions) d'un processeur P_{proc} à son successeur P_{proc+1} (resp. en provenance de son prédécesseur P_{proc-1}). Une nouvelle fois, à la fin des résolutions modulo les processeurs possèdent h/P résidus de chacun des x_k et du déterminant.

L'idée de l'algorithme que nous allons présenter est la suivante :

1. chaque processeur P_{proc} commence la remontée des n/P coefficients x_k, avec alloc(k)=proc, à l'aide des h/P nombres premiers p_i tels que allocp(i)=proc.

E. Il transmet ses résultats (et ceux reçus à l'étape précédente) à P_{proc+1}. Et effectue la remontée des coefficients correspondants aux résultats reçus de P_{proc-1} en continuant à utiliser les mêmes p_i (il possède les résidus nécessaires). Le processus est recommencé P–1 fois.

F. Un fois la remontée terminée, la reconstruction des entiers peut s'effectuer sans communications.

• Les h étapes de la remontée

Nous donnerons plus loin une version détaillée de l'algorithme, mais on peut expliciter ici les représentations à bases multiples des x_k en fin du processus de remontée. Il faut pour cela considérer une nouvelle numérotation des nombres premiers : on note $p_{1,proc},\ldots,p_{h/P,proc}$ les p_i tels que allocp(i)=proc.

Pour k tel que alloc(k)=proc *le h–uplet de bases* H_k de x_k sera (3.19) :

$$H_k=(p_{1,\mathbf{proc}},\ldots,p_{h/P,proc},p_{1,\mathbf{proc+1}},\ldots,p_{h/P,proc+1},\ldots\ldots\ldots,p_{1,\mathbf{proc+P-1}},\ldots,p_{h/P,proc+P-1}),$$

où les numéros des processeurs sont pris modulo P. La représentation à bases multiples de x_k sera alors,

$$\begin{aligned}
&x_k = y_{1k} + y_{2k}\, p_{1,proc} + \ldots + y_{h/P,k}\, p_{1,proc}\cdots p_{h/P-1,proc} && \textit{Calculé par } P_{proc}\\
&\textit{Calculé par } P_{proc+1}\\
&\quad + y_{h/P+1,k}\, p_{1,proc}\cdots p_{h/P,proc}\\
&\quad +\ldots+ y_{2h/P,k}\, p_{1,proc}\cdots p_{h/P,proc}p_{1,proc+1}\cdots p_{h/P-1,proc+1} && (3.20)\\
&\textit{Calculé par } P_{proc+i}\\
&\quad +\ldots\ldots\ldots +\\
&\textit{Calculé par } P_{proc-1}\\
&\quad + y_{(P-1)h/P+1,k}\, p_{1,proc}\cdots p_{h/P,proc},\ldots\ldots,p_{1,proc+P-2}\cdots p_{h/P,proc+P-2}\\
&\quad +\ldots+ y_{h,k}p_{1,proc}\cdots p_{h/P,proc},\ldots\ldots,p_{1,proc+P-2}\cdots p_{h/P,proc+P-2},p_{1,proc+P-1}\cdots p_{h/P-1,proc+P-1}.
\end{aligned}$$

Afin de faciliter la suite de la lecture, nous avons essayé d'illustrer sur la figure III.2, trois des différentes façons de répartir les calculs que nous avons étudiées pour l'étape de remontée :

• pour l'*algorithme séquentiel*, il est naturel de représenter le volume de calcul par un volume à face triangulaire de hauteur h pour la construction (3.2), et d'épaisseur n pour la dimension du vecteur solution.

• Représenter la remontée RemSeq consiste alors simplement à découper verticalement le volume précédent en P morceaux égaux.

• Nous n'avons pas représenté la remontée RmS-G. Le volume serait découpé en P morceaux (égaux ou non suivant la fonction d'allocation des nombres premiers *allocp*), mais cette fois–ci, horizontalement.

• Pour la répartition qui nous intéresse dans ce paragraphe (le dernier dessin), chaque numéro indique le processeur chargé du volume en question (dont deux ont été volontairement ommis). Rappelons que chaque processeur calcule toujours modulo les mêmes nombres premiers. On peut remarquer quatre types de faces verticales (une face correspond à un coefficient donné), ce qui illustre quatre décompositions à bases multiples différentes.

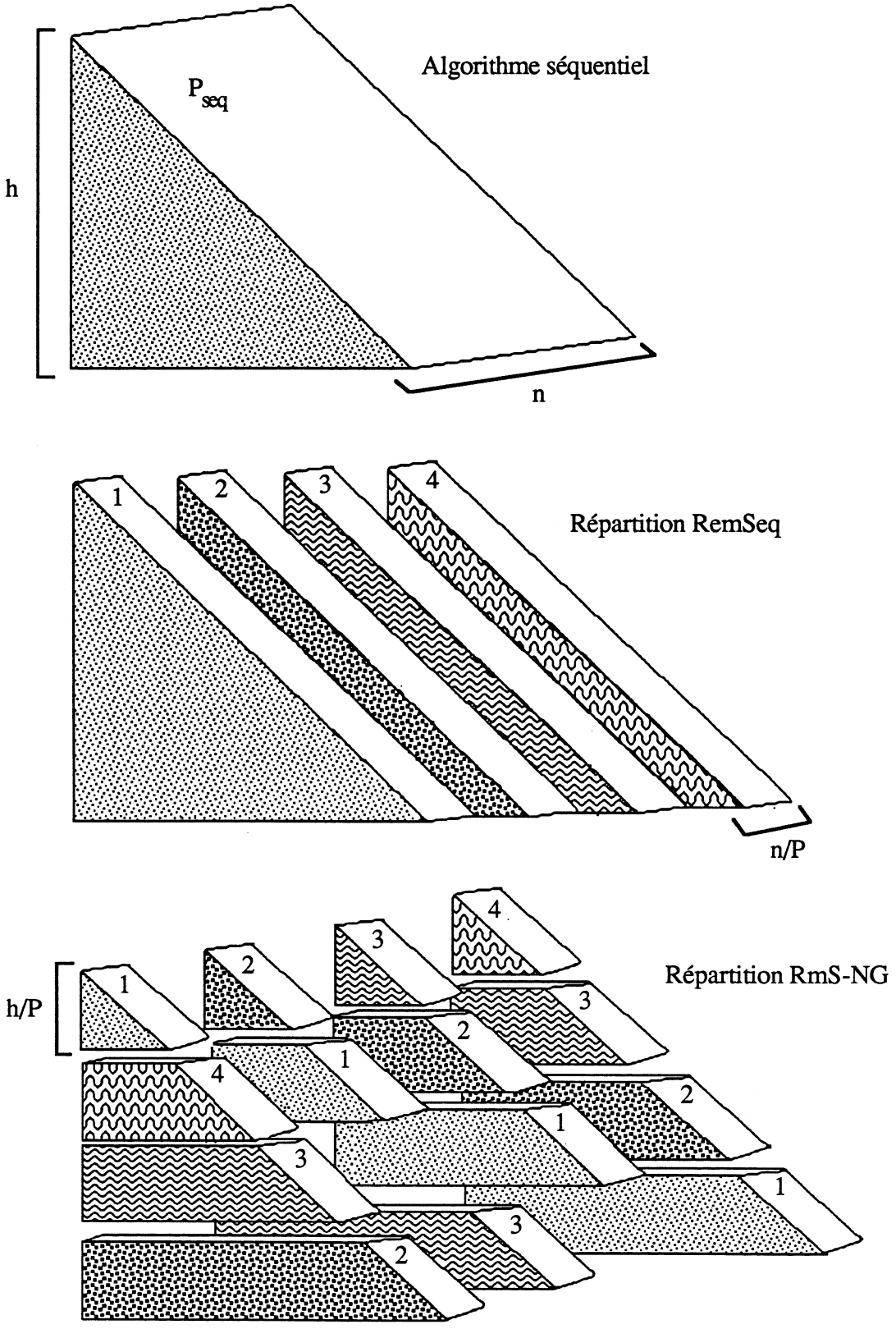


Figure III.2 : Trois répartitions différentes des calculs

La figure III.3 nous détaille alors les P étapes de la remontée dans le cas où P=h=4. Chaque flèche représentant un transfert de données. Au temps T_i le processeur P_{proc+i} va :

- calculer modulo $p_{proc+i+1}$ la valeur de $y_{i+1,k}$ (alloc(k)=proc), il aura pour cela reçu (i>0) le i–uplet $(y_{1,k},\ldots,y_{i,k})$ de son prédécesseur.
- envoyer le (i+1)–uplet $(y_{1,k},\ldots,y_{i+1,k})$ à son successeur pour le calcul de $y_{i+2,k}$ au temps T_{i+1}.
- Après la quatrième étape et dernière étape de la remontée, il peut reformer $x_{k'}$ avec k' tel que alloc(k') = proc+i+1.

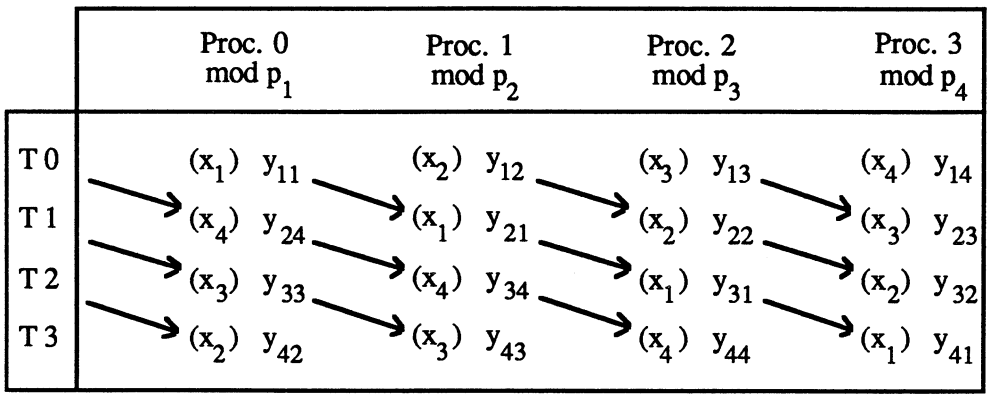


Figure III.3 : Déroulement de la remontée RmS-NG, P = h = 4.

Une version de l'algorithme RmS–NG est donnée ci–après (on ne fait plus aparaitre le calcul du déterminant qui peut être distribué ou effectué sur un seul processeur) :

• Algorithme RmS-NG

Programme du processeur P_{proc}
début
pour i = 1 à h et allocp(i) = proc *Résolutions modulo : sans communication*
pour k = 1 à n
$y_{ik} = x_k \bmod p_i$
$y_{i,n+1} = d \bmod p_i$
pour i = 0 à P-1
PAR si i > 0
recevoir $(y_{1,k},\ldots,y_{hi/P,k})$ pour les k tels que alloc(k) = (proc-i) mod P
envoyer $(y_{1,k},\ldots,y_{hi/P,k})$ pour les k tels que alloc(k) = (proc-i+1) mod P
(B2) pour j de 1 à h/P
pour k de 1 à n+1 tels que alloc(k) = (proc-i) mod P
calcul de $y_{hi/P+j,k}$ modulo $p_{j,proc}$ d'après (3.2)
pour les k tels que alloc(k) = (proc + 1) mod P *Reconstruction sans communication*
reconstruction des x_k
fin

• Coût de l'algorithme

Il est clair que le coût arithmétique est ici le même que celui des algorithmes précédents,

$$AParCr \leq n^3h/P + 5/2P\ nh^2 + O(h^2),$$

$$\boxed{AParCr \leq \frac{1}{P}(n^4\log_p nB + 5/2\ n^3\log_p^2 nB) + O(n^2\log_p^2 nB)}.$$

Pour les communications : une étape i, $0<i<P$, demande le transfert de hin/P^2 entiers modulo, on obtient donc un total de (envois et réceptions se font en parallèle),

$$CRmS\text{-}NG \leq hn(1 - 1/P)/2\ \tau + P\ \beta,$$

$$\boxed{CRmS\text{-}NG \leq n^2\log_p nB\ (1 - 1/P)/2\ \tau + P\ \beta}. \qquad (3.21)$$

Quant au coût mémoire, c'est exactement celui de l'algorithme RmS–G,

$$MRmS\text{-}NG \leq n^2 + n^2\log_q B\ /P + nh/P + O(h).$$

$$\boxed{MRmS\text{-}NG \leq n^2\ (P + \log_q B + \log_p nB) + O(n\log_p nB)}. \qquad (3.22)$$

• Test d'arrêt récursif

Une nouvelle fois TAR peut directement être appliqué : à la fin de chaque étape j (boucle B2) par exemple, puisque chaque processeur possède toute la décomposition de chacun des x_k qui lui sont associés. Les résolutions modulo sont de même calculées à chaque étape j si i=0.

Supposons maintenant que G=h/P (le plus petit nombre d'étapes permettant l'obtention de la solution). Après l'interruption de la remontée par le test, chaque processeur aura calculé la décomposition de n/P coefficients mais en ayant effectué h/P résolutions modulo. Le calcul de ces dernières n'aura donc pas été distribué.

Ce problème peut être résolu en augmentant le volume des communications. Soit L un entier, $1 \leq L \leq h/P$, un algorithme au fonctionnement analogue à celui vu précédemment est obtenu en réduisant la taille de la boucle (B2) à h/PL, et en augmentant celle de la boucle principale en i à LP. Chaque processeur P_{proc} effectuant h/PL nouvelles résolutions modulo après chaque tour sur l'anneau, c'est à dire quand il retrouve la décomposition de x_k avec alloc(k)=proc. TAR peut encore être placé à la fin de chaque étape j.

Le coût arithmétique sur h étapes et le coût mémoire sont inchangés. Quant aux communications : une étape i, $0<i<LP$, demande le transfert de hin/LP^2 entiers modulo, on obtient donc un total de,

$$CRmS\text{-}NG' \leq hn(L - 1/P)/2\,\tau + PL\,\beta,$$

$$\boxed{CRmS\text{-}NG' \leq n^2 \log_p nB\,(L - 1/P)/2\,\tau + PL\,\beta}\ . \qquad (3.23)$$

Remarque 3.2.1

• Pour L fixé, on pourra toujours trouver G tel que si la remontée est interrompue après G étapes, h/LP résolutions modulo n'auront pas été distribuées, et correspondront donc à un coût séquentiel. Par exemple si,

$$G=(l-1)h/L + m,\ 1 \leq l \leq L,\ 0 < m \leq h/L,$$

et en prenant m=h/LP; sur un total de G résolutions modulo, G–h/LP auront été effectuées en parallèle.

Toute conclusion doit donc encore tenir compte de la classe des matrices traitées. Dans le cas général, quitte à effectuer plus de communications mais sans accroître le coût mémoire, on peut prendre L=h/P et ainsi borner le coût des calculs superflus par le coût d'une résolution modulo sur un processeur.

III.4 RESOLUTIONS ET REMONTEES SONT PARALLELES

Nous reprendrons et comparerons à la fin de ce chapitre les différents coûts que nous avons donnés. Mais on peut déjà noter que si les algorithmes RmS requièrent peu de communications, ils demandent par contre plus de place mémoire que les algorithmes Rem–Seq. Il est donc naturel de chercher ici un compromis entre les deux techniques : les résolutions modulo et les remontées vont être distribuées.

Rappelons qu'un hypercube H_d est un ensemble de $P=2^d$ processeurs numérotés de 0 à 2^d-1 et de connections de voisins à voisins : deux processeurs étant voisins si et seulement si, en écriture binaire, leurs numéros diffèrent exactement d'un bit.

lemme 4.1

Soient e et c deux entiers positifs tels que c+e=d. Un hypercube H_d contient 2^e hypercubes H_c disjoints. Avec les connections non utilisées, les processeurs de ces hypercubes H_c peuvent alors être numérotés de 0 à 2^c-1 de façon à ce que, pour $0 \leq i < 2^c$, les ensembles formés par les processeurs i donnent 2^c hypercubes H_e disjoints.

La démonstration est immédiate. Si les écritures binaires des numéros des processeurs de H_d sont

notées $a_1 \ldots a_e a_1 \ldots a_c$, chaque hypercube H_c (resp. H_e) est obtenu en fixant les bits $a_1 \ldots a_e$ (resp. $a_1 \ldots a_c$) et la numérotation correspondante est donnée par les c (resp. e) premiers bits.

Le lemme nous donne un double indiçage des processeurs de H_d :

$$\boxed{P_{proc} = P_{a_1 \ldots a_e,\, a_1 \ldots a_c}}.$$

• Algorithme RRPar

Soit K un diviseur de P de la forme $K=2^e$,

• on effectue chacune des H résolutions modulo sur un des 2^{d-e} hypercubes H_e ("carrés verticaux" de la figure III.4) donnés par le lemme, P/K résolutions seront donc calculées simultanément. La résolution modulo p_i est distribuée sur les processeurs $P_{j,proc}$, avec $0 \leq j < K$ et allocp(i)=proc, *allocp* étant une fonction d'allocation des nombres premiers sur P/K processeurs. Une même fonction d'allocation des lignes sur K processeurs, *alloc*, est utilisée pour toutes les résolutions.

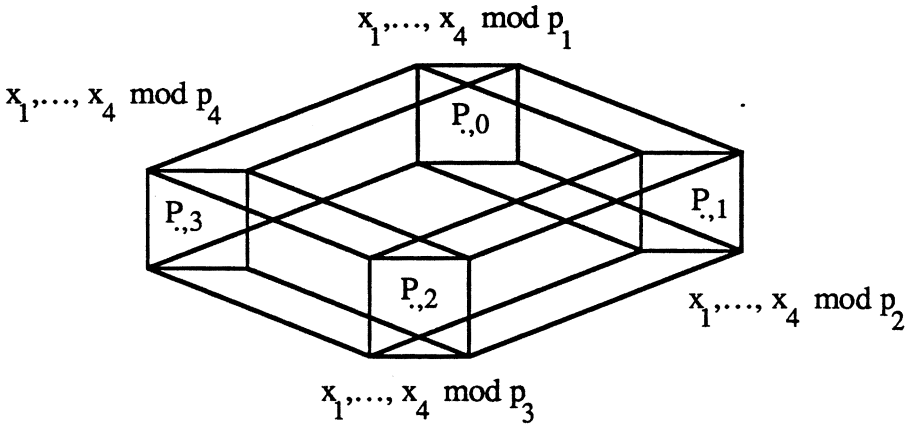


Figure III.4 : Résolutions et remontées en parallèle, P = 16, K = 4.

• Une fois terminée la phase des résolutions modulo, pour k tels que alloc(k)=pos, les résidus de x_k sont répartis sur les processeurs $P_{pos,j}$, avec $0 \leq j < P/K$.

• On peut alors appliquer les algorithmes vus pour chacune des remontées RmS sur un des 2^e hypercubes H_{d-e} ("carrés horizontaux" reliant les sommets des "carrés verticaux" de la figure III.4). K processus de remontée sont effectués simultanément.

• Coût de l'algorithme

Le coût arithmétique est toujours le même (3.8).

• Puisque P/K résolutions modulo peuvent s'effectuer simultanément (chacune nécessite K processeurs) le coût total des communications de cette phase est le coût de hK/P résolutions. Ce qui nous donne en utilisant l'algorithme de *diffusion* (3.9) :

$$CD \leq Khn^2/P\ \tau + 5hK/P\ n\log_2 K\ \beta + O(nh),$$
$$CD \leq Kn^3\log_p nB/P\ \tau\ + 5Kn^2/P\ \log_2 K\log^2_p nB\ \beta\ + O(n^2\log_p nB). \qquad (3.24)$$

De la même façon K processus de remontée vont s'effectuer simultanément. Le coût est donc celui de la remontée de n/K coefficients sur P/K processeurs (nous n'écrirons pas le cas particulier K=P); il est directement obtenu en remplaçant n par n/K et P par P/K dans (3.16) et (3.21) pour les algorithmes à h–uplet de bases global (CG) ou non (CNG),

$$CG \leq hn/K\ \tau + h\log_2(P/K)\ \beta,$$
$$CG \leq n^2/K\ \log_p nB\ \tau + n\ \log_p nB\log_2(P/K)\ \beta$$

$$CNG \leq hn(1 - K/P)/2K\ \tau + P/K\ \beta,$$
$$CNG \leq n^2\log_p nB\ (1 - K/P)/2K\ \tau + P/K\ \beta$$

Ou encore en prenant L=h/P dans (3.23) pour appliquer le *test d'arrêt récursif* dans de bonnes conditions,

$$CNG' \leq h^2n/2KP\ \tau + h/K\ \beta + O(nh),$$
$$CNG' \leq n^3\log^2_p nB/2KP\ \tau + h/K\ \beta + O(n^2\log_p nB). \qquad (3.25)$$

En totalisant on obtient CRRPar,

$$CRRPar \leq\ (Khn^2/P\ + \varphi\ h^2n/2KP)\ \tau +\ 5Khn\log_2 K/P\ \beta + O(nh),$$

$$\boxed{CRRPar \leq \frac{n^3\log_p nB}{P}\ (K + \varphi/2K\ \log_p nB)\ \tau\ +\ \frac{5Kn^2\log_p nB\log_2 K}{P}\ \beta}, \qquad (3.26)$$

où φ vaut 1 s'il y a test récursif et h–uplet de bases non global et 0 sinon.

• Les matrices des résolutions modulo sont maintenant stockées sur K processeurs et le vecteur solution l'est encore sur tout le réseau. Mais on pourra avoir un coût mémoire supplémentaire dû à l'application du test d'arrêt dans le cas d'un h–uplet global (3.18). Avec la matrice initiale (stockée une seule fois) on a alors,

$$MRRPar \leq n^2/K + n^2\log_q B\ /P + nh/P + \varphi\ nh/K + O(h).$$

$$\boxed{MRRPar \leq \frac{n^2}{P}\ (\frac{P}{K} + \log_q B + \log_p nB + \varphi(P/K - 1)\log_p nB) + O(n\log_p nB)} \qquad (3.27)$$

où φ vaut 1 s'il y a test récursif et h–uplet de bases global et 0 sinon.

III.5 COMPARAISON DES METHODES

Il est nécessaire pour conclure ce chapitre de récapituler les résultats. Nous le faisons dans le tableau ci–dessous. Nous ne reprenons pas les coûts arithmétiques puisque nous avons vu qu'ils étaient tous identiques. Pour les communications on peut se borner à comparer les termes en τ : ils sont toujours dominants en n et les termes en β seraient ordonnés de la même façon.

	RemSeq (K = P)	RmS–G (K = 1)	RmS–NG (K = 1)	RRPar (1 < K ≤ P)
Communications : termes en τ				
h étapes	n^2h	nh	$\mathbf{nh/2}$	n^2hK/P
avec TAR	n^2h	$\mathbf{nh}$	$nh^2/2P$	n^2hK/P (G) $n^2hK/P + nh^2/2KP$ (NG)
Place mémoire : termes en n^2/P				
h étapes	$\mathbf{1+\log_p nB}$	$P+\log_p nB$	$P+\log_p nB$	$P/K+\log_p nB$
avec TAR	$\mathbf{1+\log_p nB}$	$P+P\log_p nB$	$P+\log_p nB$	$P/K+\log_p nB$ (NG) $P/K+P/K\log_p nB$ (G)

Figure III.5 : Tableau récapitulatif des temps de communications et des coûts mémoire.

Quant aux coûts mémoire, ils ont été reportés dans le tableau en supprimant le terme en $\log_q B$ et en factorisant n^2/P qui correspond au coût d'un stockage équidistribué de la matrice modulo q.

Pour chacun des algorithmes les coûts sont donnés d'une part dans le cas où le maximum des coefficients résultats et du déterminant est connu exactement ou quand les éventuels calculs superflus on un coût négligeable (h étapes). Et d'autre part quand on effectue le test récursif.

- Si l'on dispose sur chaque processeur d'une mémoire de taille illimitée, c'est à dire si en pratique la limitation provient des temps d'exécution, il est clair qu'il faut utiliser un algorithme effectuant les résolutions modulo en séquentiel. Le h–uplet de bases sera global ou non selon que la borne h est pessimiste ou non.
- Il faut sinon chercher à utiliser le moins possible de processeurs pour chacune des résolutions modulo. Et effectuer des remontées à h–uplet de bases global tant que l'espace mémoire le permet.

Quand la matrice n'est pas inversible

Le cas où la matrice est de rang r, r<n, a été détaillé dans [McC]. Nous ne reprendons pas les résultats, les différents algorithmes de ce chapitre se généralisent facilement. On peut tout de même préciser que toutes les méthodes auront, encore une fois, les même coûts arithmétiques. Mais s'il est préférable, quand la matrice est inversible, d'effectuer les résolutions modulo en séquentiel, la situation est maintenant inversée. Le rang qui décroit n'augmente pas le coût de ces dernières. Alors que chacun des K processus de remontée simultanés va s'appliquer à (n–r)/K vecteurs. Leur coût est augmenté d'autant.

COMPARAISON DES ALGORITHMES

Pour terminer sur les présentations des différentes méthodes, nous les comparons ici en restant sur un plan théorique. Nous pourrons plus loin, après avoir exposé leurs inconvénients ou avantages respectifs en relation avec la machine que nous avons utilisée, revenir sur cette discussion, en s'appuyant cette fois sur des résultats expérimentaux.

Précisons tout de suite, que le parallélisme offre des puissances de calcul bien supérieures à celles jusqu'à présent utilisées par les Systèmes de Calcul Formel. En particulier, les dimensions des matrices que nous traiterons seront supérieures à 500 : les termes dominants des coûts théoriques, seront de même, dominants en pratique.

Quand Bareiss terminait son papier en disant,

> *"The conclusion is that the 2–step method with slow multiprecision arithmetic is faster than the congruence method for problems of order*
>
> $$n \leq 30\omega/\alpha.$$ " [Bar, 1972]
>
> (ω étant la taille du mot machine et α une borne sur le nombre de chiffres des données),

il comparait la méthode directe à l'application du théorème des restes chinois. L'algorithme utilisant les développements p–adiques n'étant à l'époque pas connu.

Mais nous ne voulons pas revenir sur la comparaison pour les petites valeurs de n (≤ 40), les temps que nous avons mesurés (chap.IV) ne sont d'ailleurs pas représentatifs (certains temps d'initialisation prédominent).

Envisageons la résolution de grands systèmes ($n \geq 40$ et $n >> 40$). Il est maintenant justifié de considérer que n tend vers l'infini.

Si la matrice A est pleine, la méthode P–Adique semble plus prometteuse, tant du point de vue arithmétique que du point des communications. Viennent ensuite les algorithmes basés sur le théorème des Restes Chinois, puis la méthode Directe.

On a les inégalités suivantes pour les calculs,

$$\boxed{PA.a = O(n^3 \log^2 nB) < RC.a = O(n^4 \log nB) < D.a = O(n^5 \log^2 nB)},$$

et,

$$\boxed{PA.c = RC.c_{min} = O(n^2 \log nB) < RC.c_{max} = D.c = O(n^3 \log nB)},$$

pour les communications. Sans trop anticiper sur le prochain chapitre, disons que ces relations pourront être modifiées quand les calculs seront vectorisés. De même que le seront les relations sur les coûts des stockages des données,

$$\boxed{PA.m = RC.m = O(n^2 \log nB) < D.m = O(n^3 \log nB)},$$

la gestion des emplacements mémoire étant en pratique, aussi importante que leur nombre.

CHAPITRE IV

RESULTATS EXPERIMENTAUX

Nous avons vu, en présentant les implantations du calcul de l'espace nul dans $\mathbb{Z}/p\mathbb{Z}$, qu'il était primordial, pour conclure sur la supériorité de l'une ou l'autre des méthodes, de s'attacher à une classe particulière de matrices. Nous sommes bien entendu ici dans la même situation, et avons choisi de nous intéresser plus particulièrement, à la résolution de systèmes linéaires denses pour lesquels les coefficients de la matrice sont pris au hasard.

Tous nos algorithmes ont donc été transcrits en langage C dans cette optique, ce qui nous a permis de profiter au mieux des connaissances que nous avions du FPS T20 et d'augmenter les dimensions des problèmes traités jusqu'à 700. Il nous faut préciser que le T20 est une machine mono–utilisateur, un programme qui s'exécute dispose donc du maximum de sa puissance de calcul. De plus, nous n'avons pas été au cours de nos expérimentations, limités par les temps de calculs, mais comme nous le verrons par la place mémoire disponible (16 MOctets).

• Arithmétique en précision infinie

Si le coût majeur des méthodes utilisant le théorème du reste chinois, provient de calculs dans $\mathbb{Z}/p\mathbb{Z}$, les méthodes directes et p–adique nécessitent au contraire une arithmétique en précision infinie. Nous avons utilisé, pour ce faire, le module **Pac** écrit en C et en assembleur du transputer Inmos T414 par J.L.Roch [Ro] (quitte à modifier les modules de communication, les programmes pourrait donc facilement être portés sur toute machine à base de transputer), qui nous a fournit, sur chaque nœud, les opérations dont nous avons eu besoin sur les entiers.

La structure de données choisie par l'auteur de Pac, a été motivée par une contrainte propre au parallélisme : il faut chercher à minimiser le coût des transferts des longs entiers. La solution s'impose d'elle même (nous avons vu en effet que le coût de l'envoi de N données consécutives en mémoire est de la forme $\beta+N\tau$), et consiste à stocker un entier x (décomposé en base 2^{32}) dans un **bloc**, c'est à dire un tableau dont la première position est affectée à sa taille, $\lceil \log_2 x/32 \rceil+1$. La gestion de la mémoire est assurée par les deux primitives C, *malloc* et *free*, permettant de s'allouer (pour un nouvel entier) et de libérer (pour un entier devenu inutile) un tableau de la taille voulue.

Concernant les performances de Pac. Pour des entiers inférieurs par exemple à 10^{500}, le coût d'une addition est inférieur à une milliseconde [Ro], le coût d'une multiplication pouvant être au pire 60 fois plus élevé. Une fonction particulière, qui utilise la méthode de Lehmer [Knu §4.5.2],

nous permettant de réaliser le processus de remontée de la méthode p–adique (chapitre II).

• Communications

Nous avons indiqué, au moment de calculer le coût des communications de la méthode directe, que l'on peut considérer pour modéliser l'envoi et la réception d'un ensemble d'entiers, un nouveau temps d'initialisation $\beta'=2\beta$ et un nouveau temps élémentaire μ. L'envoi d'un entier consiste alors en l'envoi de sa taille (première position du tableau) puis de sa décomposition (restant du tableau). Et sa réception en la réception de la taille, l'allocation de la place nécessaire (un appel de la fonction *malloc* avec la taille comme paramètre), puis la réception de l'entier lui–même.

• Implantation de la méthode directe

Une fois mis en place le module arithmétique et mis au point les fonctions de base pour les communications, les méthodes que nous avons développés dans le premier chapitre peuvent alors être écrites. Les résultats que nous présentons plus loin ont été obtenus en mesurant les temps d'exécution de l'algorithme du pipeline pour la méthode à deux pas. En pratique, nous l'avons déjà souligné, les méthodes à pivots locaux sont difficiles à mettre au point, et après maints problèmes, nous avons abandonné cette voie. De plus, si les communications marquent clairement les différences entre les différentes implantations utilisant le théorème du reste chinois, tel n'est pas le cas pour les méthodes directes pour lesquelles il faut s'attacher à réduire en priorité le coût arithmétique et le coût de la gestion mémoire.

• Implantation de la méthode p-adique

La méthode p–adique ne conduit à aucun problème majeur, et l'algorithme ParPA peut être transcrit tel qu'il a été présenté au chapitre II. Nous verrons que son coût dominant provient des itérations et non du processus de remontée (le rapport se serait peut–être inversé si nous n'avions pas été limités dans la dimension du problème).
On sait, d'après la relation (2.8) que les entiers rencontrés au cours des itérations restent inférieurs à $2nB$. Cela ne permet donc pas d'éviter l'utilisation de l'arithmétique en précision infinie pour les produits matrice–vecteur. Mais si la quantité $2nB$ était suffisamment petite, la première phase pourrait bien entendu être vectorisée sur le modèle des corps finis. On pourrait aussi tirer profit d'une arithmétique à précision limitée (moins coûteuse que la précision infinie) en travaillant modulo un nombre premier voisin de $2nB$.

Confrontée à la précédente, nous allons voir que cette méthode donne en pratique de très bons résultats : ce qui au vu de l'étude théorique, n'est pas une surprise.

• En utilisant le théorème des restes chinois

Modéliser le coût d'une opération sur des entiers de longueurs quelconques est facile, si aucun regard n'est porté sur le coût de la gestion de la mémoire : il suffit en détaillant son implantation, de "compter" le nombre de calculs effectués sur les mots de la taille voulue. Pour nous, utilisateurs du module Pac, c'est ce qui a conduit aux égalités (1.8), et par là même aux coûts donnés tout au long des trois premiers chapitres. En pratique, le problème est autrement plus compliqué, l'opération doit en effet être replacée dans le contexte de l'algorithme. En particulier, la mémoire disponible n'est pas dédiée au seul entier que l'on cherche à calculer : stocker le résultat de l'opération demande par exemple, de s'allouer la place nécessaire, et induit un coût supplémentaire (nous le mettrons en valeur en mesurant les *efficacités* des implantations parallèles).

L'intérêt des méthodes utilisant le théorème des reste chinois, dont le coût dominant est celui des résolutions modulo, est justement qu'elles permettent d'éviter de tenir compte de ces considérations : le coût se limite effectivement au coût arithmétique. De plus, nous avons vu dans la première partie, comment il est simple, et combien il est rentable, de vectoriser les calculs modulo.

Il est aussi possible de vectoriser le début de la remontée des coefficients, et la construction (3.2) en l'occurence. Avec la relation (3.12) on peut en effet réaliser les calculs en affectant pour les h valeurs de j avec i tel que $i<j\leq h$,

$$y_{jk} = (y_{jk} - y_{ik})c_{ij} \bmod p_j,\ 1\leq k\leq n. \qquad (4.1)$$

En suivant une démarche analogue à celle présentée au chapitre III de la première partie, (4.1) se calcule en sept passages dans l'additionneur ou dans le multiplieur. Pour les algorithmes RemSeq et RmS–NG, les vecteurs sont de longueur constante égale à n/P puisque chaque processeur se consacre à la remontée de n/P composantes du vecteur solution. Pour RmS–G, utiliser l'unité vectorielle est plus intéressant encore, puisque chaque processeur calcule tous les y_{jk} pour n/P valeurs de j, les vecteurs sont donc de longueur n.

Précisons ici un détail qui a son importance : la représentation des corps $\mathbb{Z}/p\mathbb{Z}$ est comme nous l'avons vu bien particulière au cours des résolutions modulo. Il faut donc retrouver à un moment ou à un autre la représentation équilibrée. Nous avons choisi de le faire au moment le moins contraignant, c'est à dire, pour chacun des y_{jk}, une fois son calcul terminé (il suffit de tester sa valeur et de lui ajouter ou de lui soustraire p_j), juste avant qu'il ne soit utilisé pour obtenir les autres coefficients de la décomposition.

Terminons, avant d'exposer les mesures, en disant qu'elles ont été obtenues à l'aide d'exécutions des algorithmes RemSeq, Rms–NG et RRPar. Pour ce dernier les valeurs de K pouvant être 2, 4 ou 8. En particulier, le test d'arrêt récursif a été implanté en augmentant le volume des communications d'après (3.23). Les résolutions modulo utilisent l'algorithme du pipeline.

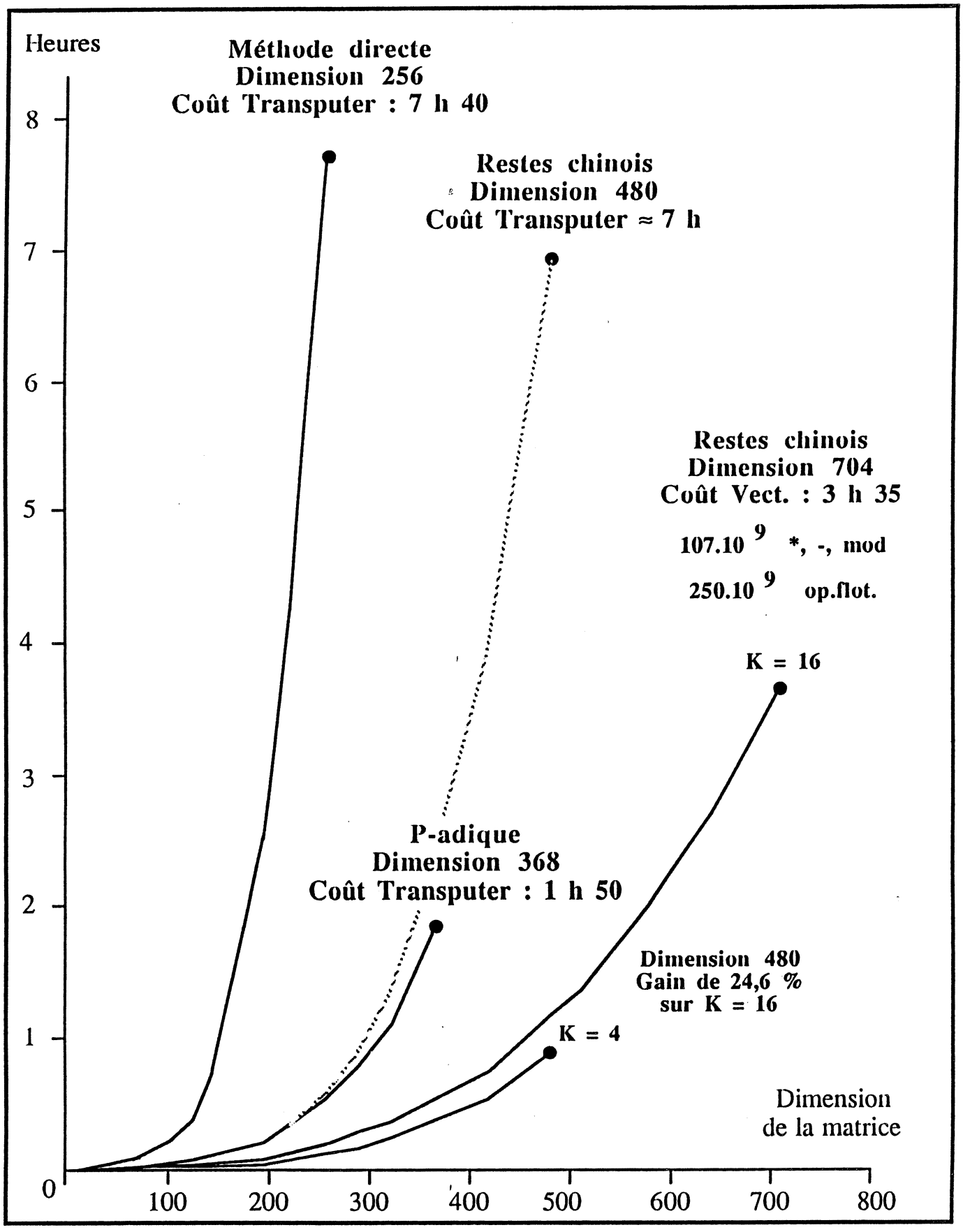


Figure IV.1 : Temps d'exécution des différentes méthodes appliquées à des matrices dont les coefficients, inférieurs à 100, sont choisis au hasard.

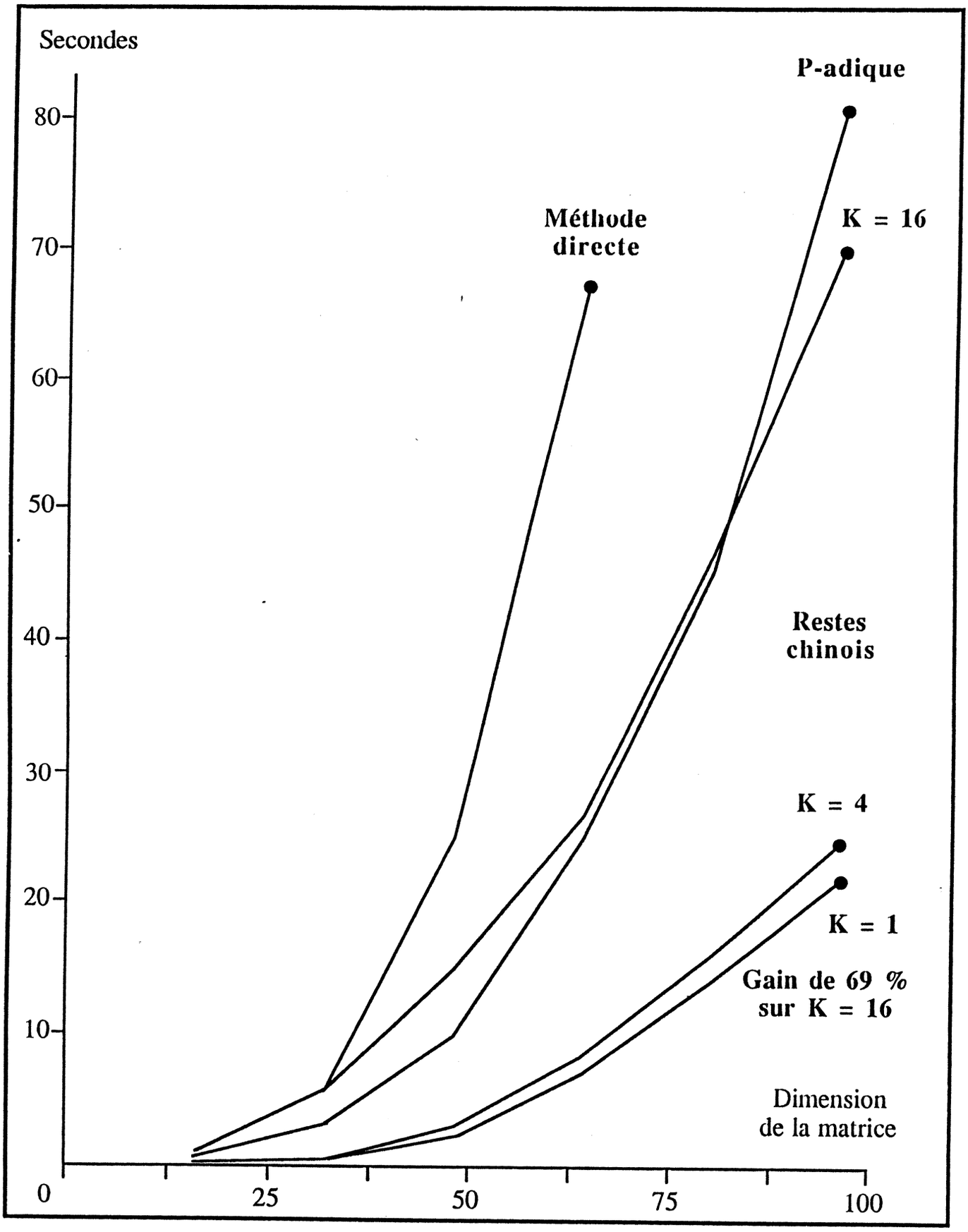


Figure IV.1.b : Temps d'exécution des différentes méthodes appliquées à des matrices dont les coefficients, inférieurs à 100, sont choisis au hasard, n ≤ 100.

Nous avons réuni sur les figures des deux pages précédentes, les courbes représentant les temps d'exécution des différentes méthodes sur 16 processeurs. Pour chacune d'entre elles, la plus grande dimension (fig.IV.1) atteinte en abcisse, correspond à la limite avant saturation de l'espace mémoire : rappelons que ce dernier est en théorie de 16 MOctets, ce qui laisse, une fois stocké le programme sur chaque processeurs, de l'ordre de 13 MOctets pour le stockage des entiers.

Les matrices considérées ont des coefficients choisis à l'aide d'une formule de récurrence (d'après [Knu]) du type :

$$\begin{cases} u=774755,\ a=62605,\ b=113218009,\ r=536870912, \\ \text{Pour i de 1 à n et j de 1 à n+1} \\ \quad u=(au+b) \bmod r,\ A[i,j] = u \bmod B. \end{cases} \qquad (4.2)$$

avec la borne B sur les données, comprise entre 100 et 10^7. Les nombres premiers utilisés sont quant à eux tous compris entre p=67000019 et q=67006253 (il en faut 308 pour une matrice de dimension 704). Pour chaque valeur de B, nous avons testé 5 matrices différentes : les temps d'exécution restent dans tous les cas à 4% de ceux que nous présentons.

Le commentaire des figures se retrouvera dans les deux paragraphes suivants, le premier concernant les implantations n'utilisant que les possiblités des transputers : donc celles de la méthode directe et p–adique.
Le cas des restes chinois est naturellement traité à part, il permet seul d'accéder facilement à l'unité vectorielle.

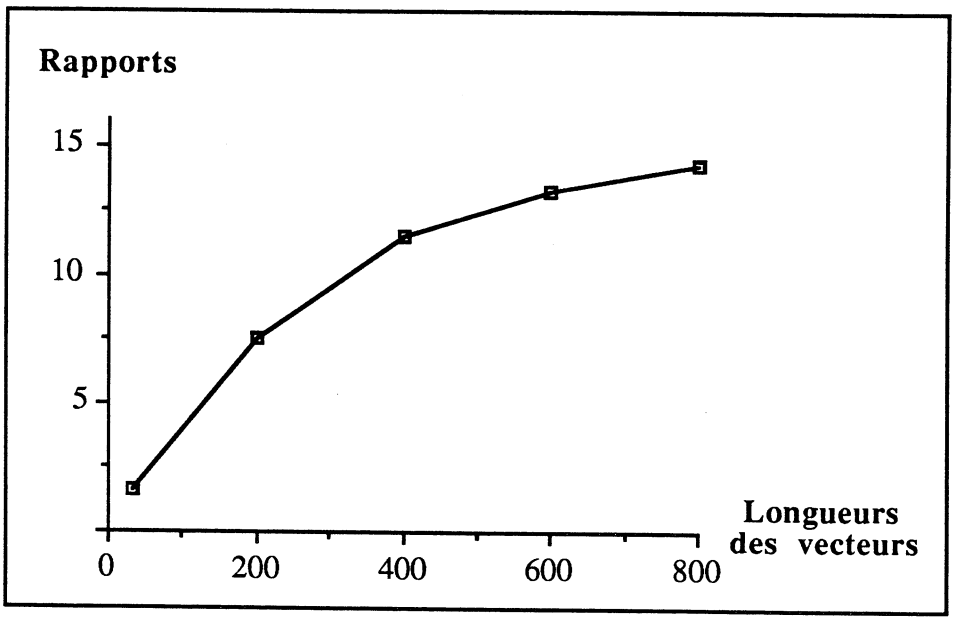


Figure IV.2 : Rapports des temps d'exécution du Saxpy modulo p
sur le transputer au format entier, et des temps d'exécution
sur l'unité vectorielle au format flottant .

La courbe tracée en pointillés sur la figure IV.1, donnant le comportement de Rem–Seq sans vectorisation des calculs, n'a pas été à proprement parler mesurée, mais seulement extrapolée à partir de résultats obtenus pour les résolutions modulo sur les transputers : en multipliant les temps d'exécution connus des résolutions par les quantités de nombres premiers à utiliser. N'oublions pas d'ailleurs, que le transputer travaille au format 32 bits, la borne sur les nombres premiers est donc plus restrictive qu'au format de l'unité vectorielle,

$$p \leq \sqrt{2^{31} - 1} < 46341.$$

Puisqu'elles ne tiennent pas compte des remontées, ces approximations sont certainement au–dessous de la réalité : mais dans une bien faible mesure d'après les rapports que nous donnons plus loin (§IV.2, fig.IV.20). Pour n=256, par exemple, on sait que l'exécution demanderait au moins 2024s, chiffre supérieur aux 1914s de la méthode p–adique, mais qui reste tout de même bien inférieur aux 7h40 d'une résolution directe (dans le même temps, la figure IV.2 peut nous fournir une limite supérieure, particulièrement pessimiste, de 3h10).

• "Explosion des coefficients"

De même que préserver la stabilité des calculs doit être une constante préoccupation, si l'on utilise le calcul numérique pour résoudre des problèmes de l'algèbre linéaire, les résolutions en arithmétique exacte, conduisent immanquablement à des données de tailles très importantes. L'expression "explosion des coefficients", ne mérite t'elle pas son emploi courant en calcul formel pour décrire le phénomène, si l'on pense qu'il faut 250 fois plus de mémoire pour stocker le vecteur solution d'un système de dimension 700, sous sa forme rationnelle qu'au format flottant ?

A titre d'exemple, nous donnons ci–dessous le numérateur et le dénominateur de la première composante x[1], du vecteur solution d'un système de dimension 704, dont les coefficients initiaux sont bornés par 100. Ces deux entiers nécessitent chacun pour être écrit, 2412 chiffres décimaux (soit quelques 1000 octets). Remarquons qu'ils sont premiers entre eux (le dénominateur est le déterminant de la matrice), et que leurs 1965 premiers chiffres respectifs sont identiques :

$$\boxed{\begin{aligned} x[1] &= 8.76\ldots\ 10^{2411} \,/\, 8.76\ldots\ 10^{2411} \\ &= 1 + 1.487\ldots\ 10^{-1966} \end{aligned}} \quad . \qquad (4.3)$$

Notons aussi, que pour la plupart des dimensions, les matrices formées à l'aide de (4.2) conduisent à des entiers dont les valeurs atteignent la borne d'Hadamard : aucun zéro n'est obtenu dans leurs décompositions si n est égal à 480, 640 ou 704. Pour n=512, on prend par défaut 29 nombres premiers superflus ($p \geq 67000019$), et 34 si n=576. Remarquons aussi que si $n \leq 700$ et $B \leq 95700$, un seul zéro suffit pour interrompre les résolutions modulo, la condition d'arrêt est donc particulièrement simple à tester. De même que si $B \leq 6.10^{12}$, on peut se contenter de deux zéros consécutifs.

876228619770839153182743242028730981988404709141633561755130326479925814266439552067476048816009527431032736164526998496457357284088921132489707371190088777828047407912098340719350115406182293834767415578795653537066591030777474045792539854452400146592842371802815723576350667195315885216397966503685807624103243226860185631849851530159722425132191278755734605411565168654613023749378399798718955274418475593736207428043861949478530973832670859993004105363149829603872277600555975440186037379925978580599551009504406373890000670048748641650015003778087749751207375180893402341323115492260855560814473585894669461423156553024419643609087858587936998290161299851973796631492155945283528204246698989447384190841255037265996414376048685814792268049910491700431153683584087770388059800396123435935475116738817280372858050474315795206630222913808333365148116507355716122404175664170917155197595339838951701509872483598138934875095937283708320177421436905549291424474756655798003463514832538339756937817686155546377362210517710962815012695630541178789470982094824845977245823709358995960588288756884763908969114822316534051025616656153210518073002426895153508938488381866928374569947467842195038957471709502427196016820357522900037721615636315498849152154014710252327982360312810590894417924803800020399340561502133568470420094317173251405198382797000033310046293536919728534478317987737043963645248725011786692526015052493862766713131712615875882427365738862573797100166201436855045024049332392435323452365805604396498079835182958726484594573863340685733934489324019700251907944439968068119786074882078747528718085393805790186896652605620477176717668292820615134467537650943611986385182715777209473621434046944091377501393293504554359408141750557789351325565105774114009885888071211707865745919999327408370864116555558361986116850189707850841973100341835650751496385052149105466006984118502545507669729041256502932610074400297462462160303553963654574332079448921316292776508272093130931811360874893792717797461148476621372385676643523294942716443851579899245604797235676887306299161179052825863998823089253203333829755990708806906826117609299162887063089701025666427305384305463581789036539483870752599362524960553900206416176577695366224352800633778830936377814120671220404189500601413230528893215114744184552626458174894905442948439971388049053108285052682774833936622132974025949905711931556747469561655215455389189 0

Figure IV.3 : Exemple de numérateur obtenu pour une des composantes du vecteur solution d'un système de dimension 704.

876228619770839153182743242028730981988404709141633561755130326479925814266439552067476048816009527431032736164526998496457357284088921132489707371190088777828047407912098340719350115406182293834767415578795653537066591030777474045792539854452400146592842371802815723576350667195315885216397966503685807624103243226860185631849851530159722425132191278755734605411565168654613023749378399798718955274418475593736207428043861949478530973832670859993004105363149829603872277600555975440186037379925978580599551009504406373890000670048748641650015003778087749751207375180893402341323115492260855560814473585894669461423156553024419643609087858587936998290161299851973796631492155945283528204246698989447384190841255037265996414376048685814792268049910491700431153683584087770388059800396123435935475116738817280372858050474315795206630222913808333365148116507355716122404175664170917155197595339838951701509872483598138934875095937283708320177421436905549291424474756655798003463514832538339756937817686155546377362210517710962815012695630541178789470982094824845977245823709358995960588288756884763908969114822316534051025616656153210518073002426895153508938488381866928374569947467842195038957471709502427196016820357522900037721615636315498849152154014710252327982360312810590894417924803800020399340561502133568470420094317173251405198382797000033310046293536919728534478317987737043963645248725011786692526015052493862766713131712615875882427365738862573797100166201436855045024049332392435323452365805604396498079835182958726484594573863340685733934489324019700251907944439968068119786074882078747528718085393805790186896652605620477176717668292820615134467537650943611986385182715777209473624340469440913775013932935045543594081417505577893513255651057741140098858880712117078657459199993274083708641165555583619861168501897078508419731003418356507514963850521491054660069841185025455076697290412565029326100744002974624621603035539636545743320794489213162927765055609731054747275290719385070493892535177791246978364982566101675038781823097825830996906829211268775036560114343083121382016869154535057459443929418073079236199814804316264821456004059193365680489997920036525197660509349321940457131492995169251836752635436606078790344079240469567337370483093423595107516301141244362316267081693544066690636595548556803486676592823263397068947674298341662750012661790734037515361062417751835717930748604609832579

Figure IV.4 : Exemple de dénominateur obtenu pour une des composantes du vecteur solution d'un système de dimension 704.

IV.1 LES METHODES DIRECTE ET P-ADIQUE

• La mémoire

En regard de la mémoire, les méthodes directe et p–adique ont un comportement tout à fait similaire. Nous montrons sur la première figure, la place nécessaire à l'exécution de chacune, en fonction de la dimension du problème traité. Rappelons que d'après la théorie, il faut disposer de $n^3(\log B+\log n/2)/6P$ mots pour la première et de $n^2(1+\log B+\log nB)/P$ pour la deuxième. Comme prévu, les besoins de la résolution directe sont plus faibles pour les petites dimensions. Mais leur accroissement est beaucoup plus rapide.

Pour n=256, il faut 180 KOctets par processeur, soit près de 3MO au total. Pour la méthode p–adique le total est d'environ 6MO et pour n=368. Ces chiffres correspondent aux plus grandes dimensions pouvant être traitées.

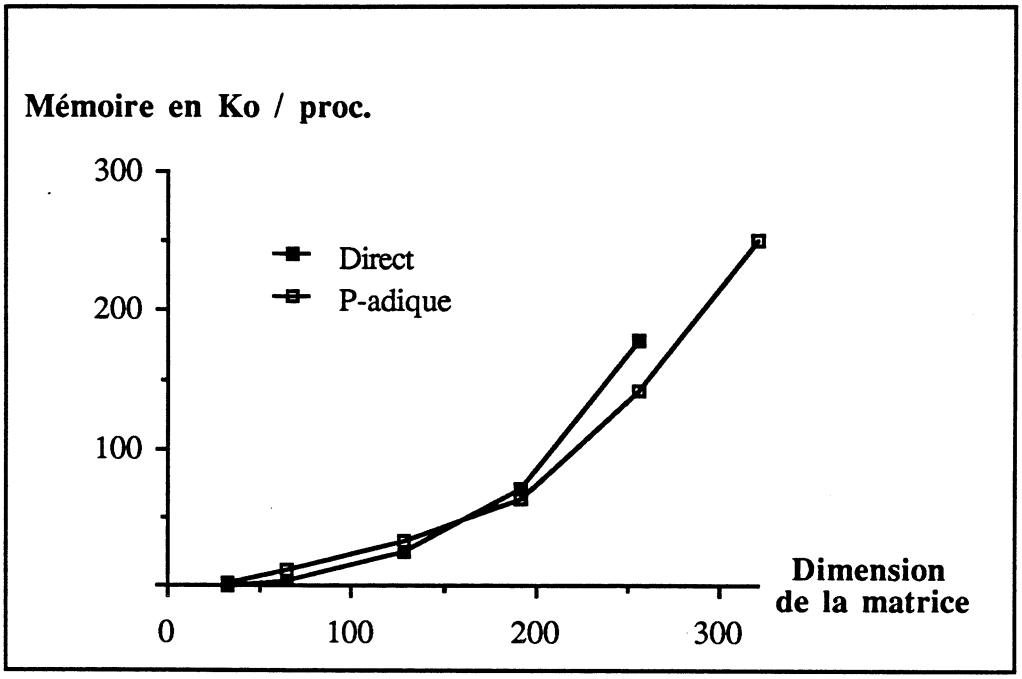


Figure IV.5 : mémoire utilisée en milliers d'octets par processeurs (P=16, B=100).

Mais répétons–le, ce n'est pas la place mémoire elle–même qui intervient dans les temps d'exécution mais sa gestion. Pour mettre en évidence l'importance de cette dernière, on appelera **mémoire virtuelle** nécessaire à une exécution, la place mémoire dont il faudrait disposer, pour assurer le stockage de tous les coefficients intermédiaires. **Le coût de la gestion mémoire est clairement lié à cette quantité si l'arithmétique est en précision infinie.**

La figure IV.6 nous montre les rapports des valeurs de cette dernière pour la résolution directe, par celles mesurées pour la méthode p–adique. Les rapports croient linéairement.

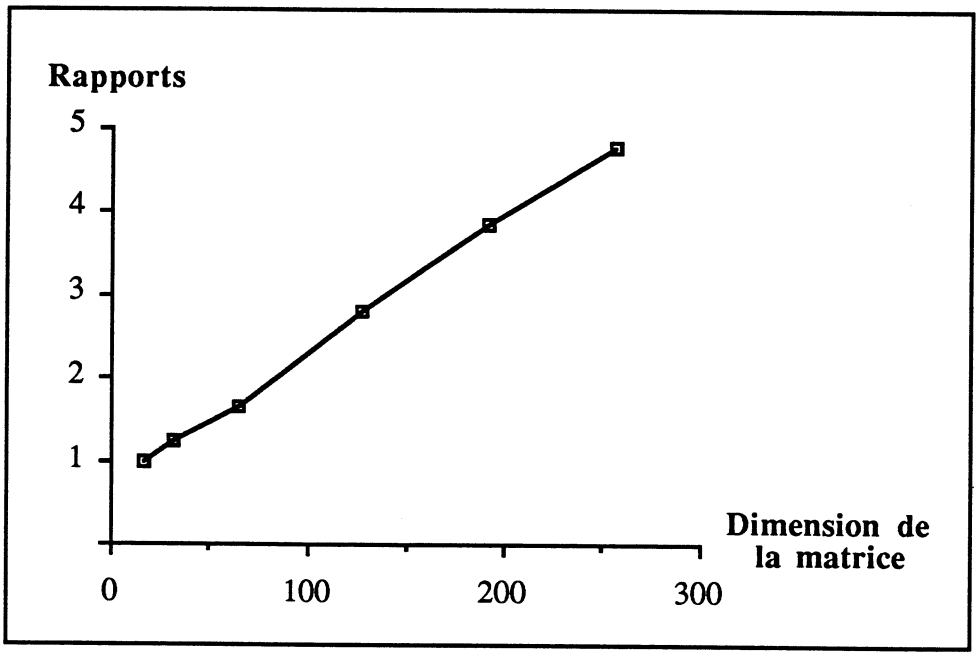


Figure IV.6 : Rapports des mémoires virtuelles de la résolution directe et de celles de la méthode p–adique (P=16, B=100).

Cette mémoire virtuelle est particulièrement considérable : elle est de l'ordre de 3.7 GigaOctets pour n=256 et la méthode à 2 pas. Et de 1.5 GO si n=320 pour une résolution p–adique.

Le coût de sa gestion est complexe à modéliser, mais il peut être mis en évidence en mesurant l'efficacité des méthodes.

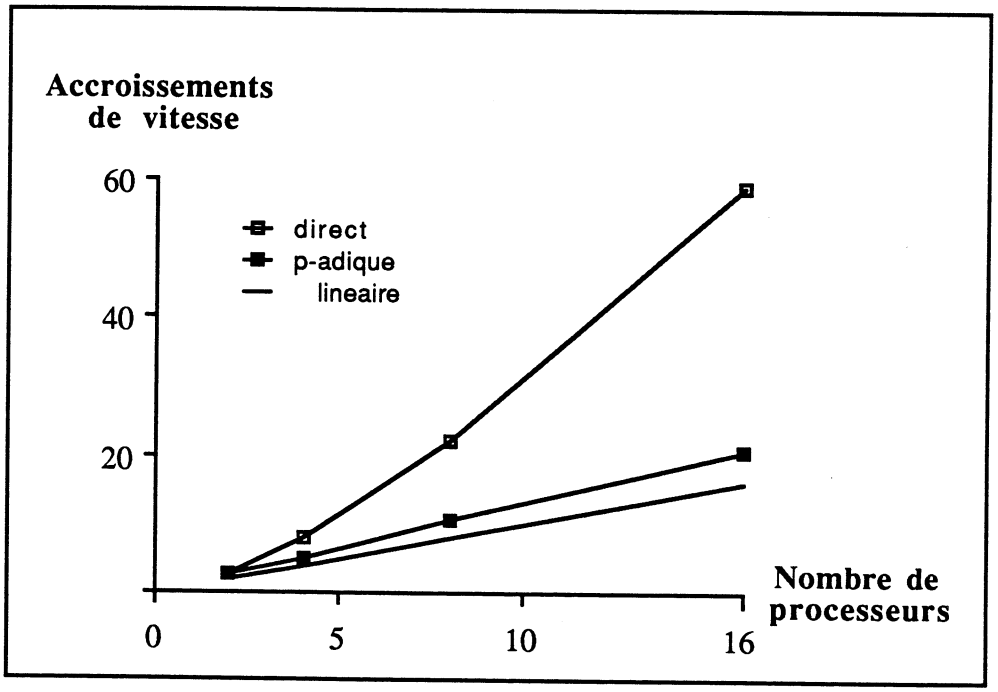
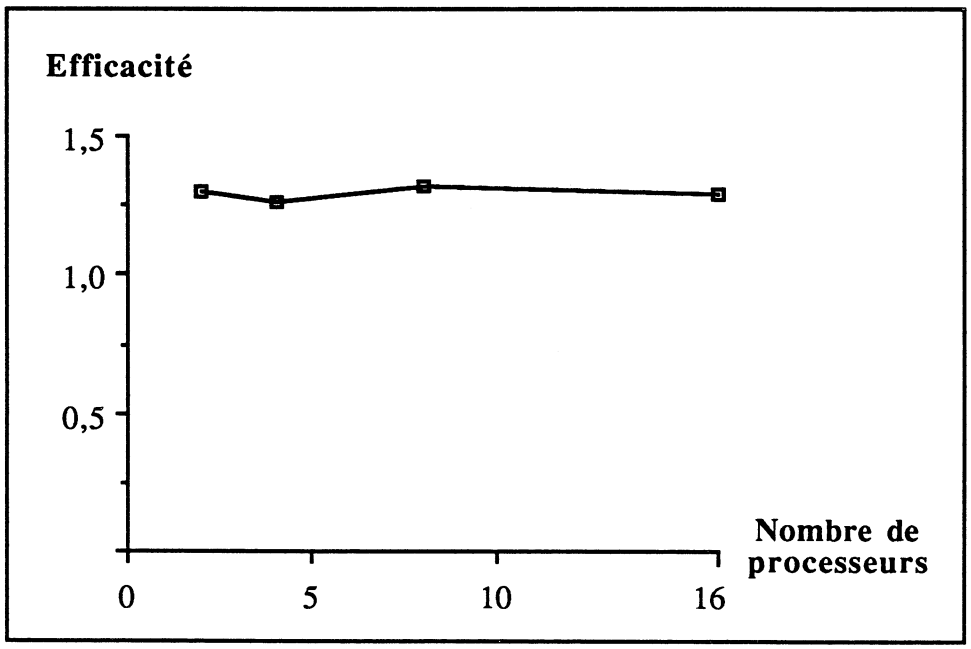


Figure IV.7 : Accroissements de vitesse pour une matrice de dimension 128, B=3000.

• Efficacité

Pour caractériser les implantations de la première partie, nous avions calculé les efficacités, en précisant qu'il fallait sous–entendre que le nombre d'opérations effectuées restait constant. De la même façon ici, il faut remarquer que la gestion mémoire dépend en coût du nombre de processeurs utilisés : une exécution sur P processeurs dispose de P fois plus de mémoire qu'une exécution sur un seul processeur. Au cours de cette dernière la mémoire est donc plus rapidement saturée (pour une même dimension de problème). Ce qui explique les courbes a priori bien surprenantes de la figure IV.7, qui donne les accroissements de vitesse obtenus (rapport du temps séquentiel par le temps parallèle). La méthode directe demande 60 fois moins de temps pour s'exécuter sur 16 processeurs que sur un seul. Le rapport est de 20 pour la méthode p–adique.



IV.8 : Efficacité de la méthode p–adique (n=128, b=3000).

La courbe de ces rapports, divisée par le nombre de processeurs, nous donne donc la coube des "efficacités", elle nous est donnée figure IV.8. Les quantités sont presque constantes égales à 1.25,

$$T_{par} \approx \frac{T_{seq}}{5/4\ P}\ . \qquad (4.4)$$

C'est le coût de la gestion mémoire qui domine ce rapport.

• Temps d'exécution

Les rapports des temps d'exécution de la méthode directe par ceux de la méthode p–adique, figure IV.9, varient en fonction de la dimension de la matrice comme variaient les rapports des mémoire virtuelles : l'accroissement est quasi linéaire. Rappelons qu'en théorie, les termes dominants en n sont respectivement $O(n^5\log^2 n)$ et $O(n^3\log^2 n)$ pour l'arithmétique, soit un rapport

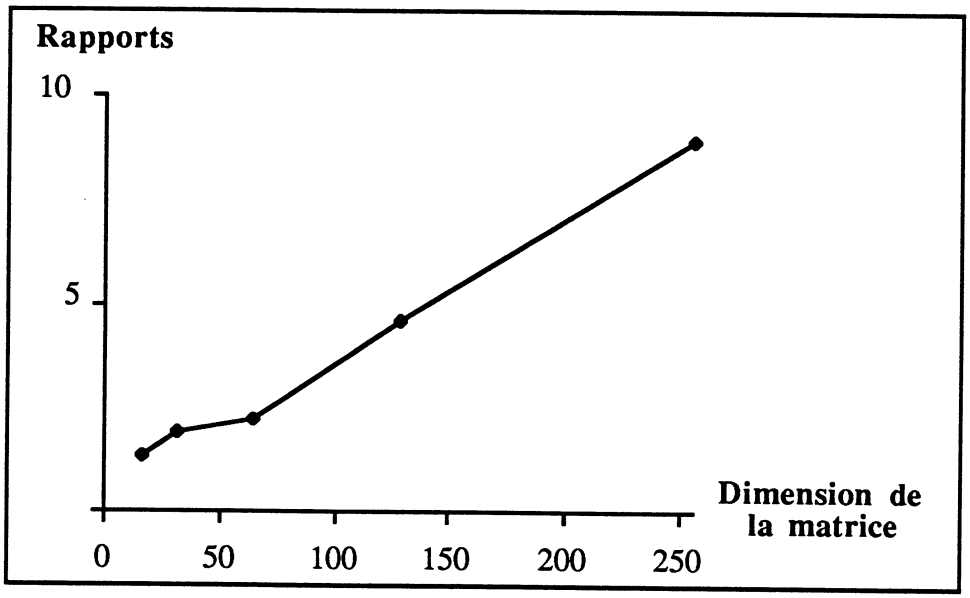


Figure IV.9 : Rapports des temps d'exécution de la méthode directe, et de ceux de la méthode p–adique (P=16, B=100).

nlogn. Pour les communications on a $O(n^3 \log n)$ et $O(n^2 \log n)$, ce qui conduit à un rapport linéaire. Les résultats théoriques paraissent donc sous–estimer la méthode p–adique (il faut toujours garder à l'esprit la gestion mémoire) qui donne néanmoins de bien meilleurs résultats. La figure IV.1 suffisait à nous en persuader. Nous avons vu que son exécution se décompose en trois phases distinctes. Le coût de l'exécution de l'algorithme de Jordan est en pratique négligeable : inférieur à 11 secondes pour les dimensions que nous avons traitées. Quant aux deux phases principales que représentent les itérations et le processus de remontée, elles ont en théorie des coûts équivalents.

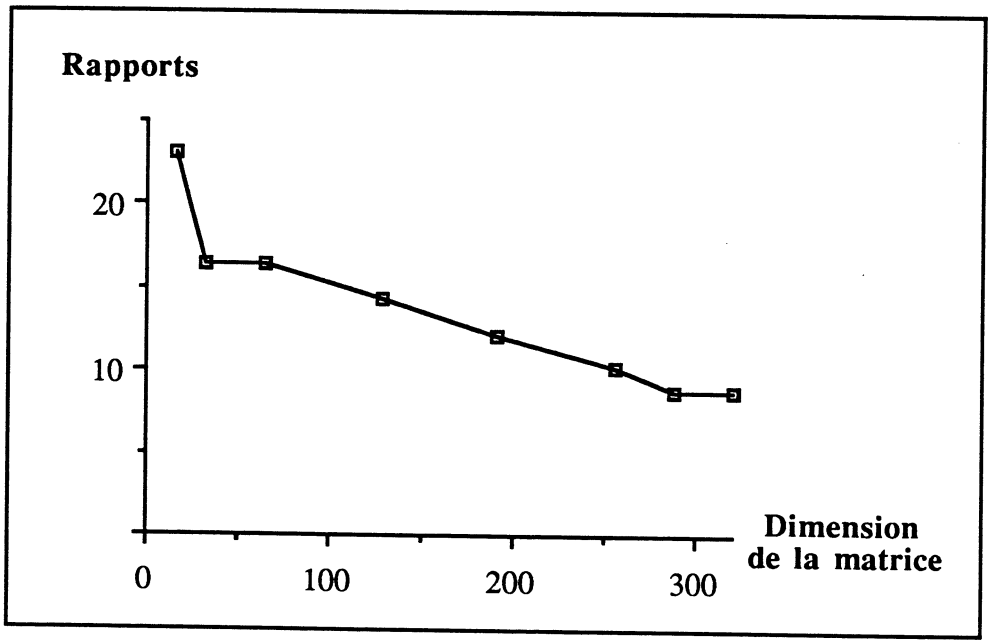


Figure IV.10 : Résolution p–adique, rapports des temps d'exécution de Jordan et des itérations par ceux de la remontée.

En réalité, la figure IV.10 nous montre que la remontée prend de plus en plus d'importance : elle représente 4% du total pour les petites dimensions et environ 10% pour une dimension 320. Nous avons vu, en effet, que les entiers apparaissant au cours des produits matrice–vecteur sont bornés par 2nB, ce qui est relativement faible. Leurs tailles sont au contraire maximales en fin d'algorithme.

Nous terminons ce paragraphe en montrant l'évolution des coûts quand la taille des données augmente. Les termes dominants ont pour les deux méthodes un facteur $\log^2 B$, il provient pour la méthode directe des tailles des entiers rencontrés, O(klogkB) à une étape k donée : et influence plus nettement l'accroissements des temps d'exécution (figure IV.11) que pour la méthode p–adique.

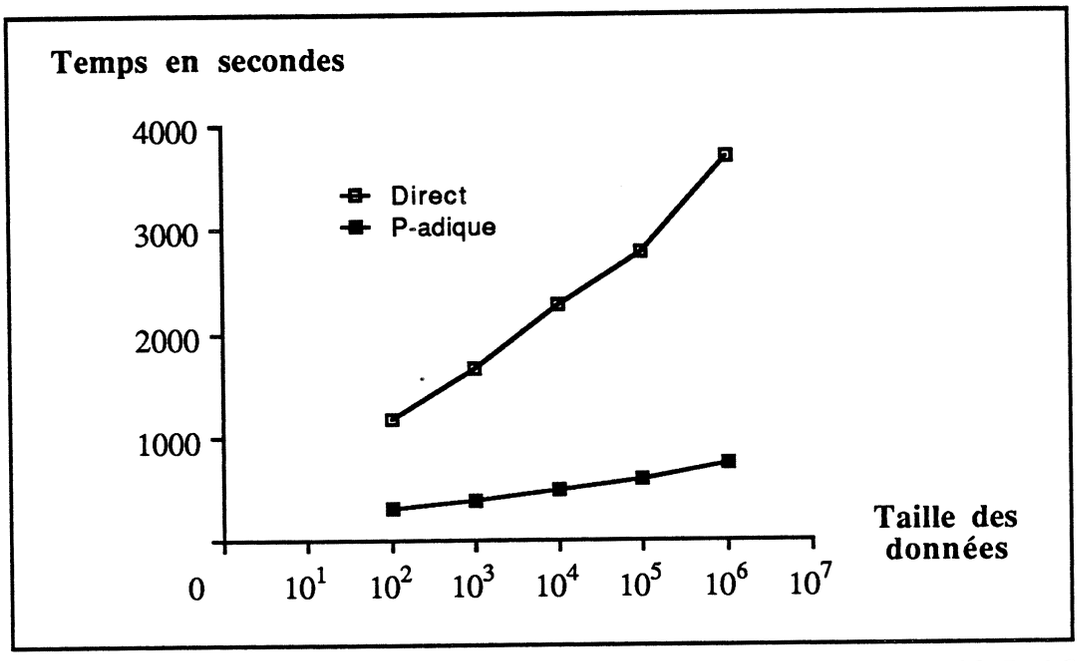


Figure IV.11 : Temps d'exécution en fonction de la taille des données (n=128, P=16).

IV.2 AVEC LE THEOREME DES RESTES CHINOIS

Si les différences sont déjà très marquées entre les deux premières méthodes, l'utilisation combinée du théorème des restes chinois et d'unités vectorielles se démarque plus encore : elle permet de réduire le coût de la résolution d'un système de dimension 256, par l'algorithme à 2 pas, dans un facteur 62. Les rapports avec la méthode p–adique sont bien moins importants, nous les donnons sur la figure IV.12.

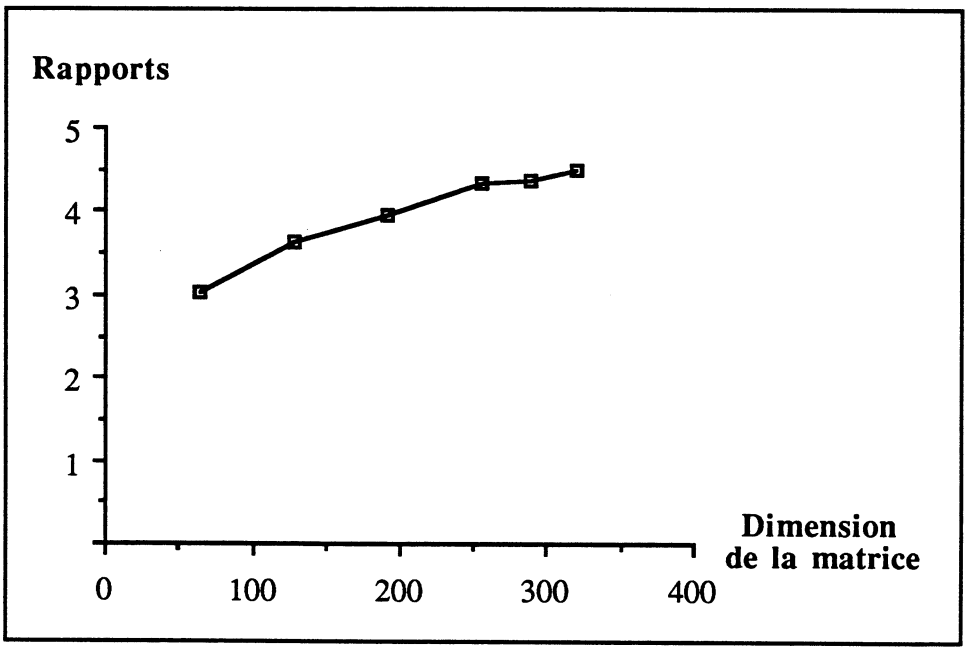


Figure IV.12 : Rapports des temps d'exécution de la méthode p–adique
et de ceux obtenus en utilisant le théorème des reste chinois (P=16, K=4, B=100).

• Mémoire

Le gain de temps est loin d'être le seul avantage de cette troisième technique. Elle est en effet la seule à permettre d'augmenter considérablement la dimension de la matrice A : il faut attendre des valeurs de n proches de 700 pour saturer l'espace mémoire (figure IV.1). La raison principale en étant que le coût de la remontée peut être, en pratique de même qu'en théorie, négligé (figure IV.19 et IV.20), les limitations dûes à l'arithmétique en précision infinie apparaissent donc plus tardivement.

La place mémoire requise avant la remontée n'amène aucun commentaire, elle est comme nous l'avons vu, inversement proportionnelle au nombre K de processeurs utilisés pour chacune des résolutions modulo. La figure IV.13 nous donne d'autre part, le nombre d'octets nécessaire au stockage de la solution, ce qui correspond au stockage maximum pendant la remontée. Ces chiffres ont été mesurés pour K=16, et sont analogues pour ses autres valeurs. En ce qui concerne

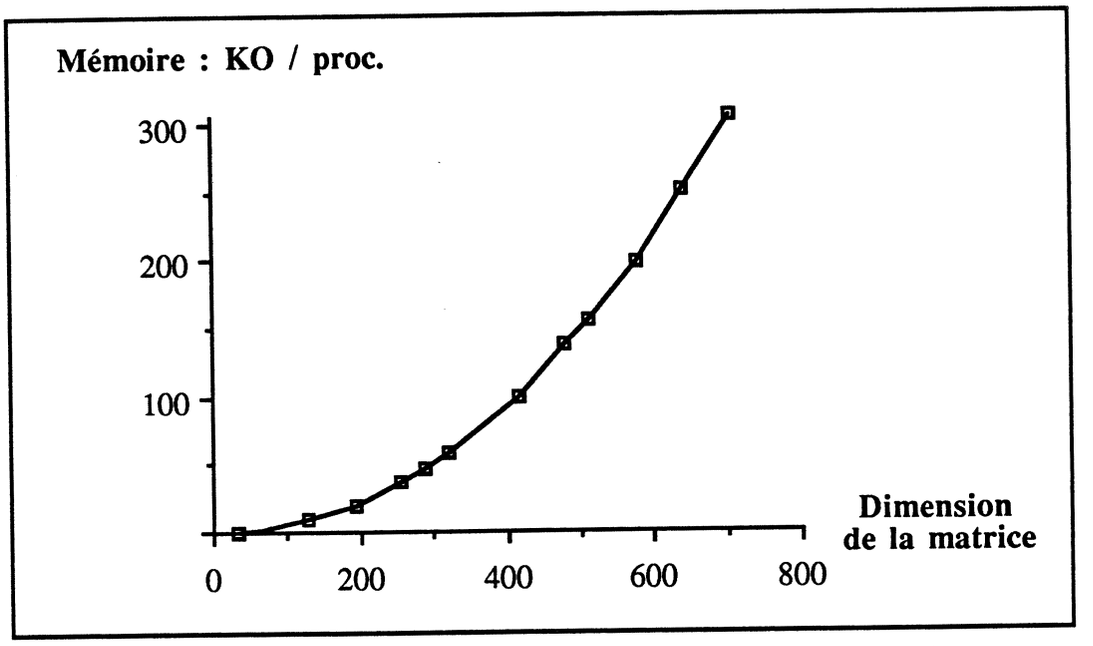


Figure IV.13 : Place mémoire occupée en fin d'exécution en milliers d'octets par processeur (P=K=16, B=100).

la mémoire virtuelle et donc le coût de la gestion mémoire, elle se limite à 650 KO environ pour n=256, ce qui est 360 fois moins important que pour une résolution directe et 75 fois moins que pour la méthode p–adique.
En particulier, il n'est pas étonnant que l'on retrouve alors des efficacités moins surprenantes que celle du paragraphe précédent. Nous n'avons pas tracé les courbes correspondantes, qui ne seraient pas sans rappeler celles des résolutions dans $\mathbb{Z}/p\mathbb{Z}$ (partie 1, figure III.14).

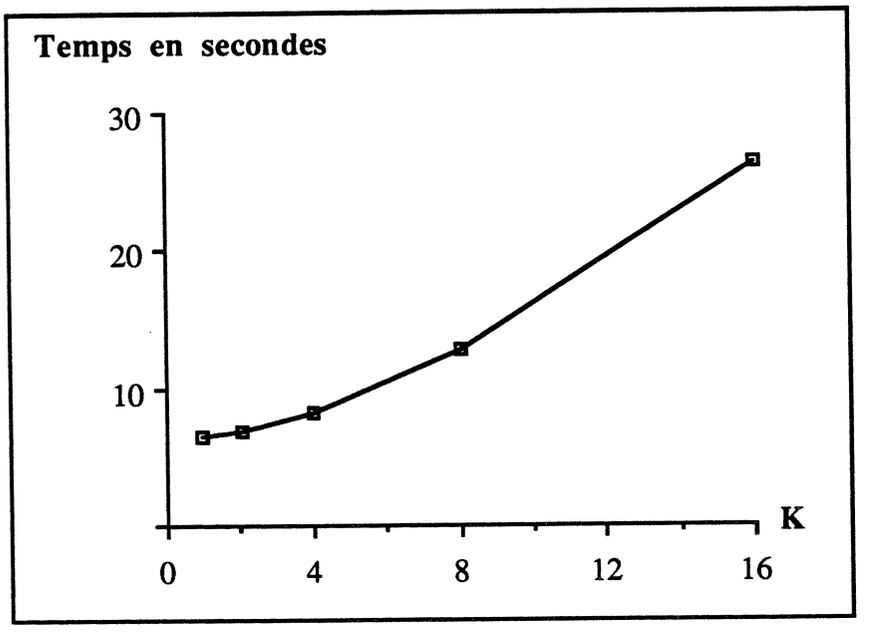


Figure IV.14 : Temps d'exécution en fonction du nombre de processeurs utilisés pour chacune des résolutions modulo (B=100, n=64).

• Temps d'exécution

Nous avons tracé, pour quatre dimensions différentes de matrice, les courbes donnant les temps d'exécution de l'algorithme RmS–NG en fonction de K, avec P=16.
Sur les deux premières (figure IV.14 et IV.15) la valeur K=1 est prise en compte : les résolutions modulo de dimensions n=64 et n=192 peuvent être traitées séquentiellement. Pour ces dimensions, et comme annoncé au §III.5, K=1 donne les meilleurs résultats.

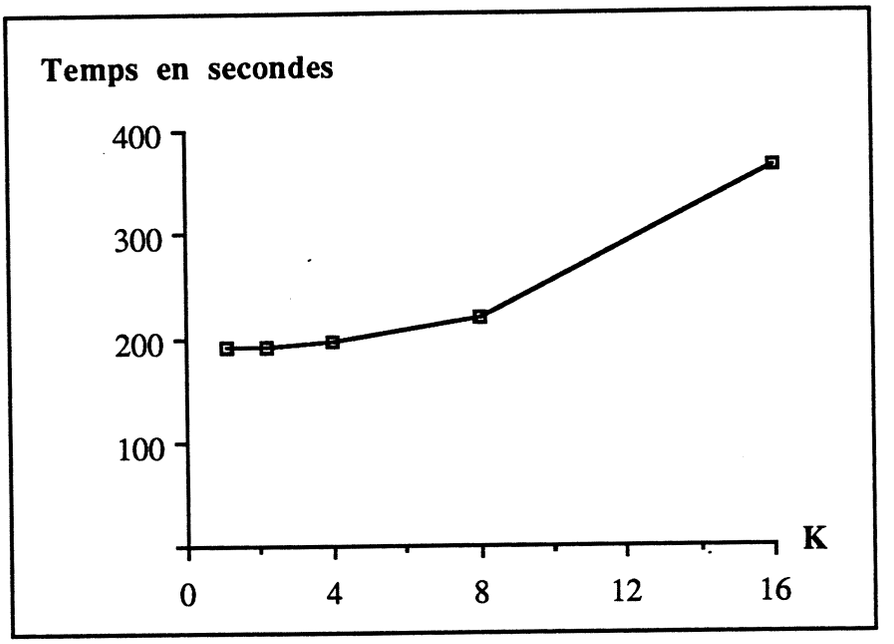


Figure IV.15 : Temps d'exécution en fonction du nombre de processeurs utilisés pour chacune des résolutions modulo (B=100, n=192).

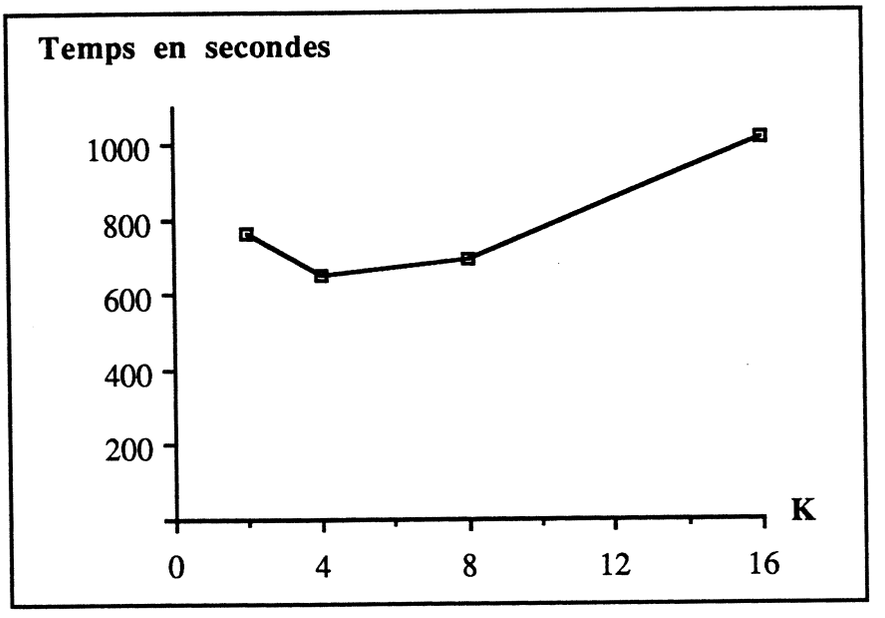


Figure IV.16 : Temps d'exécution en fonction du nombre de processeurs utilisés pour chacune des résolutions modulo (B=100, n=288).

Mais les petites valeurs de K limitent les dimensions traitées. Si K=1, n doit rester inférieur à 250. Et inférieur à 350 quand K=2. La valeur K=1 n'apparaît donc plus sur les deux courbes suivantes (figure IV.16 et IV.17) qui, il est important de le remarquer, ne sont plus strictement croissantes. Il est en effet nécessaire de trouver un compromis entre :

- le coût des communications de la phase des résolutions modulo, il croit avec K.
- le coût des communications de la phase de remontée qui, de même que le coût du test d'arrêt ($nh^2/2KP$), augmente quand K diminue (notons aussi que la longueur des vecteurs pendant les résolutions des systèmes triangulaires, décroit avec K).

Pour les exemples que nous avons traités ce compromis est toujours réalisé pour K=4.

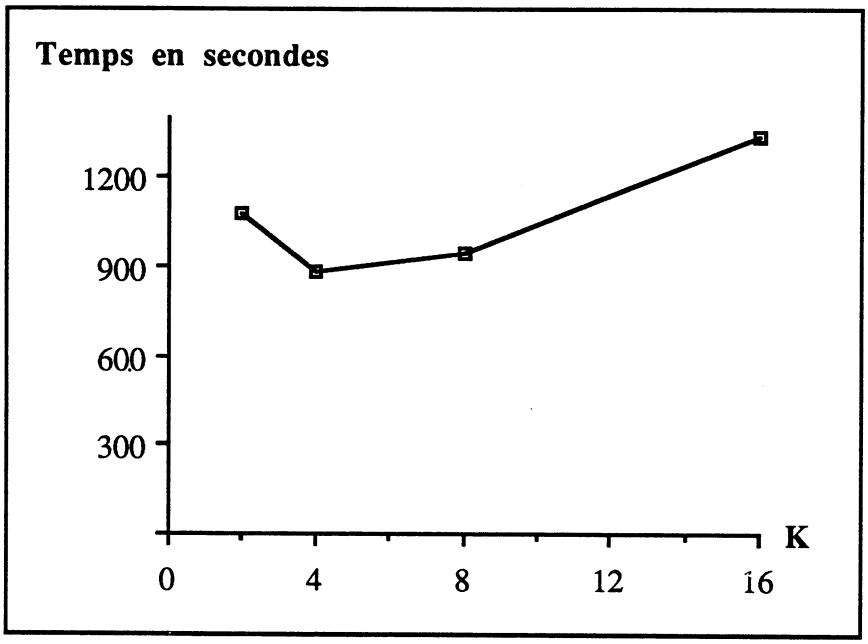


Figure IV.17 : Temps d'exécution en fonction du nombre de processeurs utilisés pour chacune des résolutions modulo (B=100, n=320).

Puisque les coûts arithmétiques sont identiques pour les différents K, les coûts des communications sont mis en évidence en soustrayant les coûts totaux de deux exécutions, avec deux valeurs distinctes de K. Pour simplifier, la courbe de la figure IV.18 a été obtenue à l'aide de résolutions n'incluant pas le test récursif.
En reprenant les coûts récapitulés au §III.5, la différence des temps d'exécutions avec K_1=16 et de ceux avec K_2=8 est donc donnée par

$$\frac{n^2h\,(K_1 - K_2)}{P} = \frac{n^2h}{2}\,, \qquad (4.5)$$

et, si K_2=4, par

$$\frac{3n^2h}{4}\,. \qquad (4.6)$$

C'est le rapport mesuré de (4.5) par (4.6) que l'on peut visualiser sur la figure IV.18. Il est donc en théorie, constant et égal à 2/3.

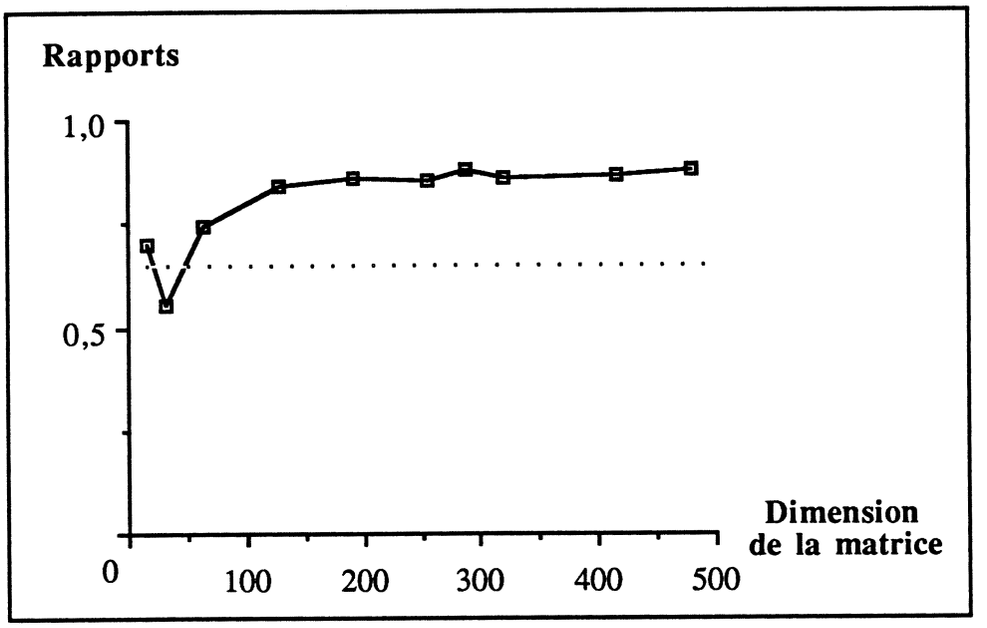


Figure IV.18 : Rapports des différences des temps d'exécution avec K=16 et K=8, et des différences des temps d'exécution avec K=16 et K=4. Le rapport théorique est représenté en pointillé (B=100).

En pratique, il tend de même à être constant : il se stabilise autour de 0,85. La faible différence avec la valeur 0,67 théorique étant due aux termes d'ordres inférieurs qui dépendent de P.

De même que l'on a pu le faire pour la méthode p–adique, il est intéressant d'observer les parts des deux phases principales des résolutions, dans le coût total.

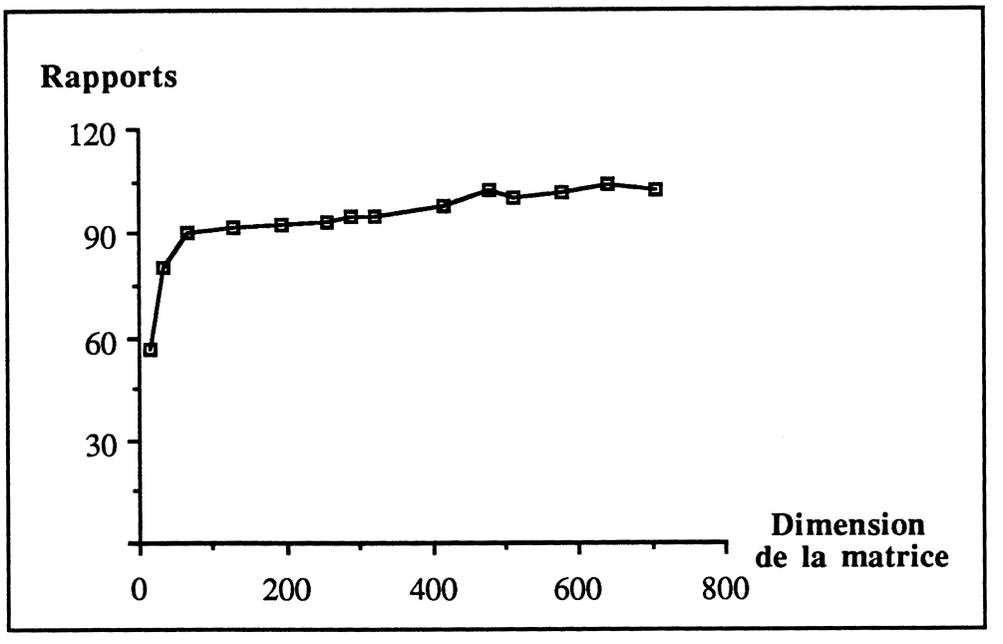


Figure IV.19 : Rapports des temps d'exécution des résolutions modulos, et de ceux de la remontée des coefficients, K=16 (P=16, B=100).

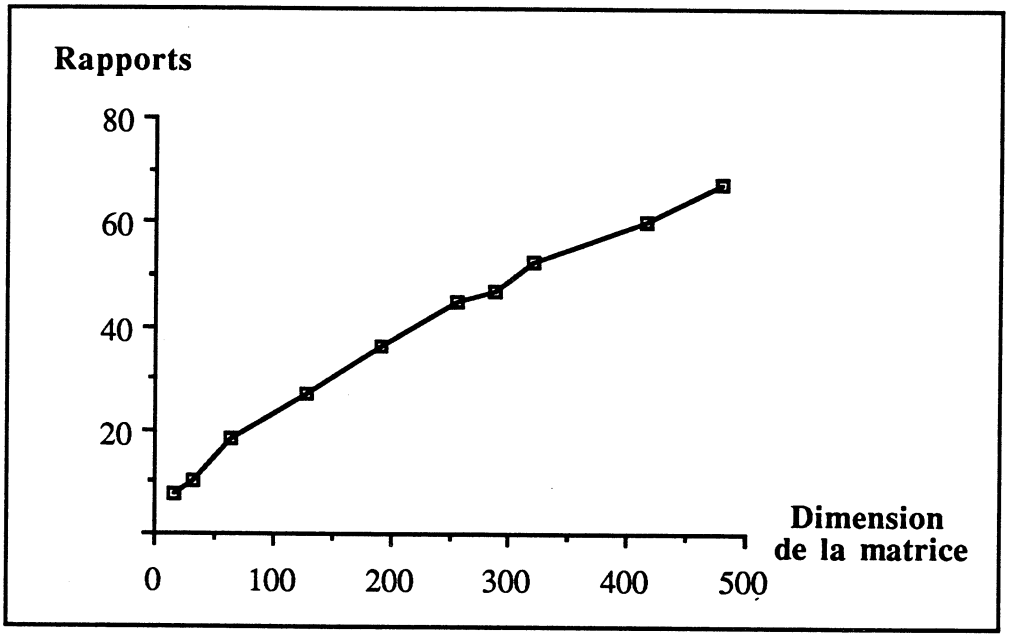


Figure IV.20 : Rapports des temps d'exécution des résolutions modulos, et de ceux de la remontée des coefficients, K=4 (P=16, B=100).

Les contributions des résolutions modulo et de la remontée sont respectivement $O(n^4 \log n)$ et $O(n^3 \log^2 n)$ pour l'arithmétique, le rapport est donc en $O(n/\log n)$. Ce qui apparaît plus nettement si K=4 que si K=16 (figures IV.19 et IV.20). Soulignons, qu'en tout état de chose, il est très important : de l'ordre de 100 dès que la dimension de A est supérieure à 400 (K=16). En particulier, il faut à peine plus de 2 minutes pour calculer le vecteur dont une composante a été donnée sur les figures IV.3 et IV.4, si bien sûr, ses résidus sont connus.

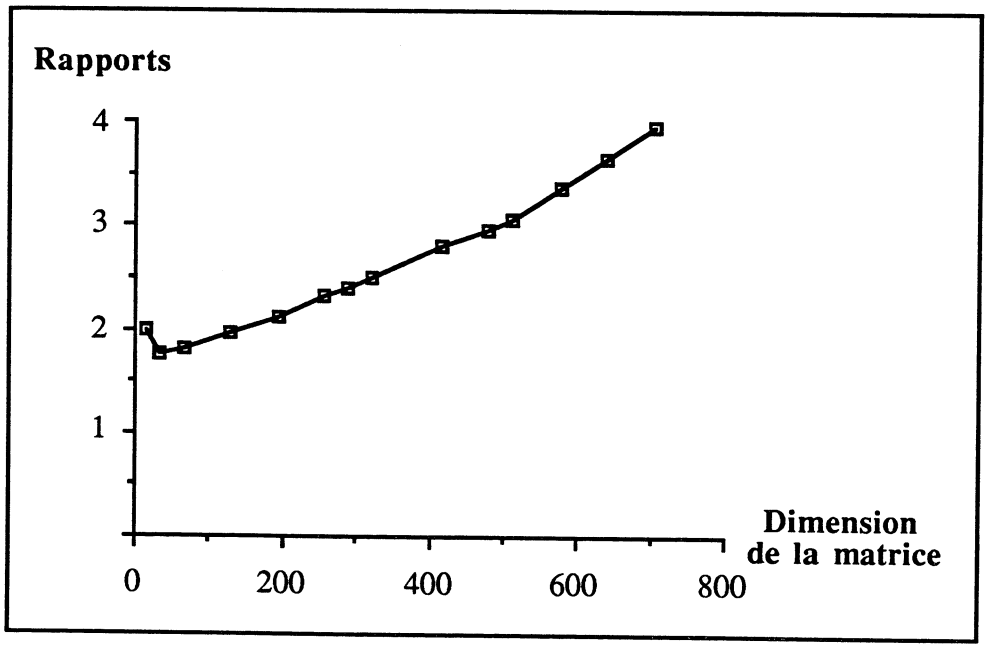


Figure IV.21 : Rapports du coût total de la phase de remontée et de son coût vectoriel, K=16 (P=16, B=100).

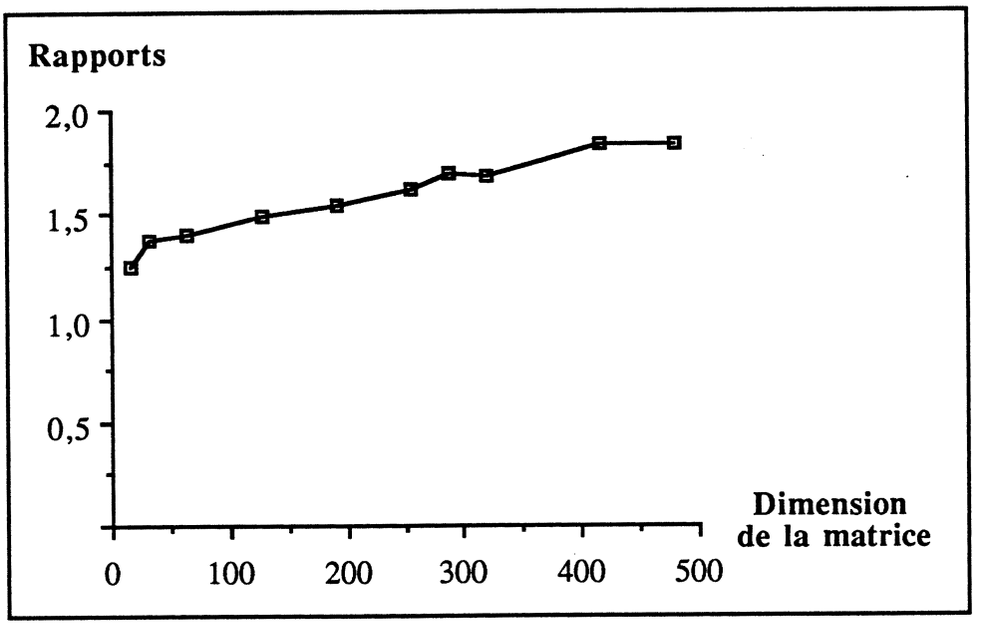


Figure IV.22 : Rapports du coût total de la phase de remontée
et de son coût vectoriel, K=16 (P=16, B=100).

Comme nous l'avions indiqué au début de ce chapitre, la remontée peut elle–même être en partie vectorisée. Les figures IV.21 et IV.22 nous montrent que l'unité vectorielle intervient pour une bonne part dans le coût total. Les rapports calculés seraient sans doute plus importants pour Rms–G (puisque les calculs effectués sur une longueur n et non plus n/P).

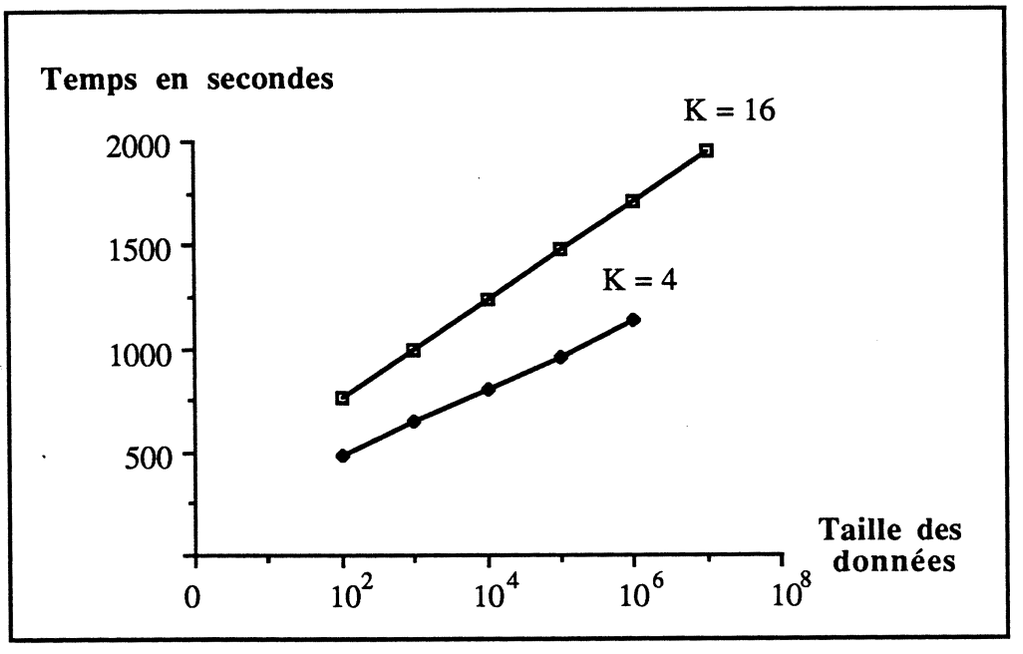


Figure IV.23 : Temps totaux d'exécution en fonction
de la taille des données (n=256, K=P=16).

• En faisant varier la taille des données

Terminons ici encore, en donnant l'évolution des temps d'exécution quand la borne B sur les coefficients de la matrice A augmente. Rappelons, à titre de comparaison, que pour les résolutions directe et p–adique, le terme dominant des coûts avait un facteur $\log^2 B$. Les mêmes accroissements se retrouve pour le processus de remontée sur la figure IV.24.
Le temps total des exécutions, quant à lui, augmente théoriquement comme logB. C'est obtenu avec précision sur la figure IV.23 aussi bien pour K=16 que pour K=4.

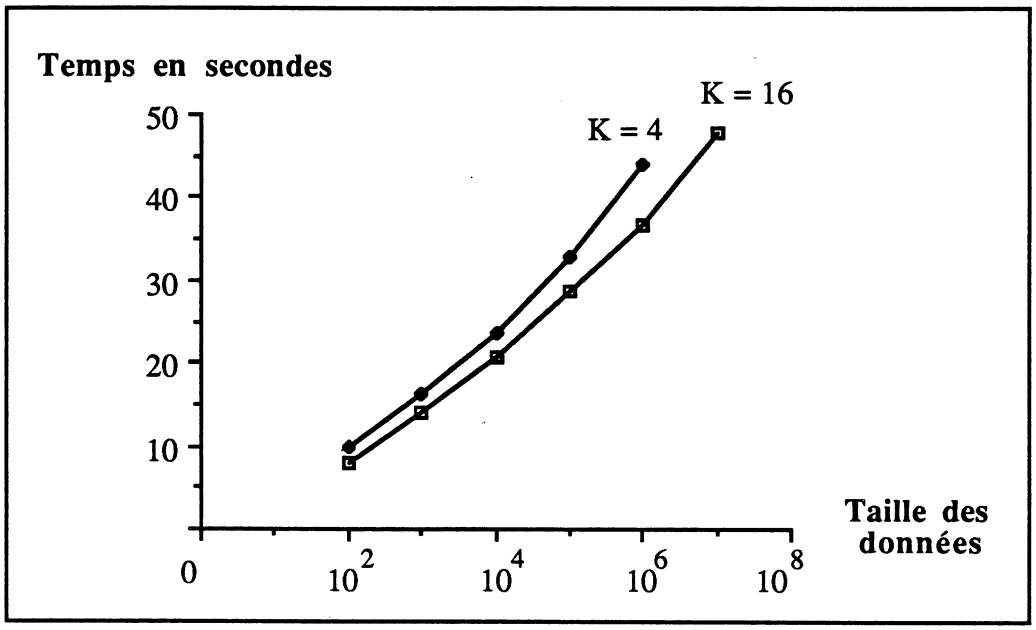


Figure IV.24 : Temps d'exécution de la remontée en fonction de la taille des données (n=256, K=P=16).

CONCLUSION

Les trois méthodes proposées dans la première partie, pour le calcul en parallèle, du noyau d'une matrice A d'ordre n à coefficients dans un corps fini $\mathbb{Z}/p\mathbb{Z}$: *diffusion, pipeline* et *pivots locaux*, conduisent à 6 algorithmes, puisque l'on peut ou non se servir d'une matrice identité bordant A au cours des éliminations.

Comme le laisse prévoir l'étude théorique, c'est l'utilisation des pivots locaux qui conduit aux meilleurs temps d'exécution, si A est telle que le nombre de coefficients nuls rencontrés au cours de l'élimination n'est pas trop important. Mais il est difficile de conclure sur la supériorité d'un des algorithmes dans le cas général, bien que l'algorithme de diffusion soit le seul à garder un coût de communication, C_d, à peu près constant quelle que soit la matrice : $C_d \approx \tau n^2/2$. Alors que ce même coût pour l'algorithme du pipeline ou des pivots locaux sans matrice identité, C_p, puisse varier dans un facteur P (le nombre de processeurs de l'hypercube) d'une matrice à l'autre : $\tau n^2/2 \leq C_p \leq \tau n^2 P/2$. Si la structure de la matrice A influe considérablement sur le comportement d'un algorithme, nous avons vu que sa répartition dans le réseau joue aussi un tout premier rôle : une étude complémentaire pourrait concerner la répartition (dynamique ?) de la charge de calcul entre les différents processeurs.

Nous avons montré que ces calculs modulo p présentent l'avantage indéniable de se vectoriser simplement et efficacement. C'est ce qui nous permet, après avoir mis au point une arithmétique de $\mathbb{Z}/p\mathbb{Z}$ sur des unités vectorielles de calcul flottant, de résoudre, par exemple, un système linéaire de dimension 1024 en moins d'une minute (élimination de Gauss, P=16, p=67000033).

Vient ensuite le problème de la résolution sur l'hypercube, de systèmes à coefficients entiers. Force est de constater en tout premier lieu, pour la classe des matrices dont les coefficients sont choisis au hasard, que les temps d'exécution de l'élimination de Gauss (la méthode directe), sont prohibitifs. Son coût arithmétique théorique $O(n^5 \log^2 n)$ est amplifié en pratique, par le coût de la gestion de la mémoire inhérent à l'utilisation d'une arithmétique à précision infinie. Surtout quand cette dernière s'effectue sur de très grands entiers. Pensez qu'il faudrait près de 3,7 GigaOctets ! pour stocker tous les coefficients intermédiaires calculés au cours de la résolution d'un système de dimension 256. Cette gestion de la mémoire conduit aussi à des accroissements de vitesse (les temps des exécutions séquentielles sur les temps des exécutions parallèles) supra–linéaires : la méthode directe peut demander 60 fois moins de temps pour s'exécuter sur 16 processeurs que sur un seul (n=128).

L'intérêt des méthodes utilisant des techniques de congruences (développements p–adiques ou théorème des restes chinois), est justement qu'elles permettent d'éviter de tenir compte de ces considérations : le coût tend à se limiter effectivement au coût arithmétique. Et, conséquence essentielle, les transferts de données se limitant à des transferts d'entiers modulo, le coût des communications est inférieur dans un facteur n, à celui de la méthode directe : $O(n^2 \log n)$ au lieu de $O(n^3 \log n)$. Si les calculs ne sont pas vectorisés, la résolution d'un système de dimension 256 dont les coefficients sont initialement bornés par 100, demande, que ce soit par les développements p–adiques ou par le théorème des restes chinois, un peu plus de 50 minutes. C'est à dire près de 9 fois moins que les 7h40 demandées par la méthode directe.

Alors que l'élimination de Gauss sature la mémoire (16 MégaOctets) dès la dimension n=256, la méthode p–adique permet d'atteindre n=368.

Seul le théorème des restes chinois nous autorise alors, à poursuivre les investigations. Et c'est le calcul vectoriel dans les corps finis, qui, combiné à l'utilisation parallèle de 16 processeurs, nous permet de trouver la solution rationnelle de systèmes linéaires de dimension 704 en 3h30.

Osons ici quelques extrapolations, à partir des temps des résolutions de systèmes linéaires par le calcul numérique [Don]. Soulignons (ceci justifie ces extrapolations) qu'en résolvant un système dans $\mathbb{Z}/p\mathbb{Z}$, on effectue plus d'opérations flottantes par unité de temps, qu'en le résolvant numériquement (le coût des communications est proportionnellement moins important). On sait qu'une résolution numérique effectue de l'ordre de $N=2n^3/3$ opérations flottantes. Et qu'il en faut $7n^3/3$ dans $\mathbb{Z}/p\mathbb{Z}$. En multipliant par le nombre de résolutions nécessaires pour appliquer le théorème des reste chinois, on a donc le coût E du calcul de la solution exacte,

$$E \approx \frac{7}{2} \left(n \log_p B + \frac{n}{2} \log_p n \right) N.$$

Sur le Cray Y–MP/832 (8 processeurs) par exemple, la solution d'un système de dimension 1024 dont les coefficients sont bornés par 100, pourrait être obtenue en à peine plus de 8 minutes. Et en 46 minutes sur un IBM 3090/600E VF (6 processeurs). Mais il faudrait patienter une trentaine de jours sur un micro–Vax II.

Si l'on ne se limite pas aux petites dimensions pour les problèmes traités, le parallélisme et les puissances de calcul auxquelles il peut être associé, nous apparait donc comme un outil indispensable au Calcul Formel, et à la gestion des données complexes qu'il demande de manipuler (1,4 MégaOctets sont nécessaires au stockage de la solution rationnelle pour n=704).

REFERENCES BIBLIOGRAPHIQUES

[All] "Alliant FX/series product summary", Alliant Computer Systems (1985).

[Bar] E.H.Bareiss, "Computational solution of matrix problems over an integral domain", J.Inst.Math.Applic. 10 (1972), 68-104.

[Ber] S.J.Berkowitz, "On computing the determinant in small parallel time using a small number of processors", Information Processing Letters 18 (1984).

[BH] J.R.Bunch & J.E.Hopcroft, "Triangular factorization and inversion by fast matrix multiplication", Math. Comp., vol. 28, 1974.

[BMcL] G.Birkhoff & S.MacLane, "A survey of Modern Algebra", 3rd Ed., Macmillan, New York, 1965.

[Bod] E.Bodewig, "Matrix Calculus", North–Holland, 1959.

[Cab] S.Cabay, "Exact solution of linear equations", Proc. Second. Symposium on Symbolic and algebraic Manipulation, ACM New York (1971).

[CCL] "CCLisp V1.0 Technical Summary", Gold Hill Computers, Inc., Cambridge, Mass.

[CL] S.Cabay & T.P.L.Lam, "Congruence techniques for the exact solution of integer systems of linear equations", ACM Trans. Math. Software 3, 386–397 (1977).

[CR] M.Cosnard & Y. Robert, "Implementing the nullspace algorithm over GF(p) on a ring of processors", Second International Symposium on Computer and Information Sciences, Istanbul (1987).

[Csa] L.Csanky, "Fast parallel inversion algorithms", SIAM J. Comput. 5 (4), 1976.

[CTV 1] M.Cosnard, B.Tourancheau & G.Villard, "Présentation de l'hypercube T20 de FPS", Journées Architectures C3 (1987), Sophia Antipolis Valbonne, Revue Bigre+Globule, n°56.

[CTV 2] M.Cosnard, B.Tourancheau & G.Villard, "Gaussian elimination on message passing architectures", in Supercomputing, E.N. Houtis et al. eds., Lecture notes in Computer Sciences 297, Springer Verlag (1988).

[CW] D.Coppersmith & S.Winograd, "Matrix multiplication via arithmetic progressions", Proc. 19th Annual ACM Symp. Theory Comp., 1987.

[Dix] J.D.Dixon, "Exact solution of linear equations using P–adic expansions", Numer.Math. 40, 137–141 (1982).

[Don] J.J.Dongarra, "Performance of various computers using standard linear equations software in a Fortran environment", Argonne National Laboratory MCS–TM–23, April 87.

[Fly] M.J.Flynn, 1966, "Very high–speed computing systems", Proc. IEEE, 54, 1901-1909.

[FPS] Floating Point Systems, "Programming the FPS T–series", Release C00, FPS Inc. Ed.

[Gei] G.A.Geist, "Efficient parallel LU factorization with pivoting on a hypercube multiprocessor", ORNL Preprint 6211 (1985).

[Gen] W.M.Gentleman, "Some complexity results for matrix computations on parallel processors", J.ACM 25, 1, 1978.

[GH] G.A.Geist & M.T.Heath, "Matrix factorization on a hypercube multiprocessor", Hypercubes Multiprocessors 1986, M.T.Heath Ed., SIAM, 1986.

[GHS] J.I.Gustafson, S.Hawkinson, K.Scott, "The architecture of a homogeneous vector supercomputer", Journal of Parallel and Distributed Computing, 3 (1986).

[GN] A.Gerasoulis & I.Nelken, "Gaussian elimination and Gauss–Jordan on Mimd architectures", LCSR–TR–105, Rutgers University (1988).

[GKa] B.Gregory & E.Kaltofen, "Analysis of the binary complexity of asymptotically fast algorithms for linear system solving", presented at the poster session of the 1987 European Conference on Computer Algebra in Leipzig, East Germany.

[GK] R.T.Gregory & E.V.Krishnamurthy, "Methods and applications of error–free computations", Springer–Verlag (1984).

[GMc] R.P.Gabriel & J.McCarthy, "Queue–based multi–processing Lisp", in Proc. 1984 Symposium on Lisp and Functional Programming, Austin, Texas (1984).

[Hal] R.H.Jr.Halstead, "Multilisp : A language for concurrent symbolic computation", ACM Toplas 7, 4 (1985).

[HAOS] R.H.Jr.Halstead, T.L.Anderson, R.B.Osborne & T.L.Sterling, "Concert : design of a multiprocessor development system", in 13th Annual Int. Symposium on Computer Architecture,

Tokyo, Japan (1986).

[HB] K.Hwang & F.Briggs, "Parallel processing and computer architecture", Mc Graw Hill, 84.
[ISS] I.C.F.Ipsen, Y.Saad, M.H.Schultz, "Complexity of dense linear system solution on a multiprocessor ring", Lin. Alg. Appl. 77, 1986.

[Inm] INMOS Limited, "Occam programming manual", Prentice–Hall International, London 84.

[Int] "iPSC System product summary", Intel Corporation, Beaverton, Oregon.

[JH] S.L.Johnsson & C.T.Ho, "Spanning graphs for optimum broadcasting and personalized communication in hypercubes", Technical Report 500, Comp.Sc.Dpt., Yale University, 1986.

[Knu] D.E. Knuth, "The art of computer programming", vol. 2, Addison Wesley (1969).

[KRS] E.V.Krishnamurthy, T.M.Rao & K.Subramanian, "P–adic arithmetic procedures for exact matrix computations", Proc. Indian Acad. Sci. 82A, 165–175 (1975).

[KT] S.Kuppuswami & B.Tourancheau, "Performance evaluation of the FPS T20", RR 708I, IMAG Grenoble, 1988.

[LC] G.Li & T.Coleman, "A new method for solving triangular systems on distributed memory message–passing multiprocessors", Tech. Rep., Comp. Sc. Dpt., Cornell University, Ithaca, New York, march 1987.

[LQRV] H.Le Verge, P.Quinton, Y.Robert & G.Villard, "A propos de la résolution d'un système linéaire dans un corps fini : algorithmes et machines parallèles", Colloque C3, Angoulème, Déc.88.

[MB] M.A.Morrison & J.Brillhart, "A method of factoring and the factorization of F7", Math. of Comput. 29, 129 (1975), 183–205.

[McC] M.McClellan, "The exact solution of systems of linear equations with polynomials coefficients", Journal of A.C.M.,vol. 20,1973, pp563–588.

[Mol] C.Moler, "Numerical comparisons of triangular solvers on the Intel iPSC", presented at the Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee, sept. 1986.

[Mul] J.M.Muller, Communication personnelle, 1988.

[Neu] W.Neun, "Implementation of the LISP arbitrary precision arithmetic for vector processor", Proc. First International Workshop on Computer Algebra and Parallelism, Grenoble juin 1988.

[OR 1] J.M.Ortega & C.H.Romine, "Parallel solution of triangular systems of equations", Tech. Rep. RM–86–05, Dpt. Appl. Math., Univ. of Virginia.

[OR 2] J.M.Ortega & C.H.Romine, "The ijk forms of factorization methods II. Parallel systems", Parallel Computing 7, 1988.

[Pon] C.G.Ponder, "Evaluation of performance enhancements in algebraic manipulation systems", Ph. Doct. Th. Berkeley Univ. of Calif., Report NO UCB/CSD 88/438, August 1988.

[PW] D.Parkinson & M.Wunderlich, "A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers", Parallel Computing 1 (1984).

[Ro] J.L.Roch, "Towards a parallel computer algebra co–processor", Proc. First International Workshop on Computer Algebra and Parallelism, Grenoble juin 1988.

[RSSV] J.L.Roch, F.Sénéchaud, F.Siebert & G.Villard, "Computer Algebra on a MIMD machine", 1988 International Symposium on Algebraic Computation, Roma, Italy.

[RT] Y.Robert & B.Tourancheau, "Block Gaussian elimination on a hypercube vector multiprocessor", RT 37, IMAG Grenoble, 1988.

[RTV] Y.Robert, B.Tourancheau & G.Villard, "Data allocation strategies for the Gauss and Jordan algorithms on a ring of processors", Information Processing Letter, to appear.

[Saa 1] Y.Saad, "Communication complexity of the Gaussian elimination on a multiprocessor", Lin. Alg. Appl. 77, 1986.

[Saa 2] Y.Saad, "Gaussian elimination on hypercubes", in Parallel Algorithms and Architectures, Eds M.Cosnard et all., North–Holland (1986) 5–18.

[SS] Y.Saad & M.H.Schultz, "Topological properties of hypercubes", IEEE Transactions on Computers, Vol.37, n°7 (1988).

[Sam] A.H.Sameh, "Numerical parallel algorithms", A Survey, D.Lawrie, A.Sameh ed., High Speed Computer and Algorithm Organization, Academic Press, New York, 1977.

[Sei] C.L.Seitz, "The Cosmic cube", Comm. ACM 28, 1, (1985) 22–33.

[Sor] D.C.Sorensen, "Analyse of pairwise pivoting in Gaussian elimination", MCS–TM–26, Argone National Laboratory, 1984.

[SW] Q.F.Stout & B.Wagner, "Intensive hypercube communication I : prearranged

communication in link–bound machines", CRL–TR–9-87, University of Michigan (1987).

[TI] H.Takahasi &Y.Ishibashi, "A new method for exact calculation by a digital computer", Information Processing in Japan, 1 (1961), 28–42.
[Ult] Ultrix–32 Version 2.2, Programmer's Manual, Digital Equipment Corporation Ed.

[Wat] S.M.Watt, "Bounded Parallelism in Computer Algebra", University of Waterloo, Dept. of Mathematics, RR CS–86–12 (1986).

[Wie] D.Wiedemann, "Solving sparse linear equations over finite fields", IEEE Transactions on Information Theory, IT–32, n° 1, (1986).

[WS] C.Whitby–Stevens, "The Transputer", Sigarch Newsletter 13,3 (1985).

[WW] M.C.Wunderlich & H.C.Williams, "A parallel version of the continued fraction integer factoring algorithm", The Journal of Supercomputing, 1, 217–230 (1987).

Résumé :

Nous nous intéressons à la résolution exacte de systèmes linéaires en parallèle, et traitons deux aspects de base du problème : le calcul du noyau d'une matrice dont les coefficients sont dans un corps fini, et la résolution de systèmes à coefficients entiers.

La partie théorique du travail, dans laquelle nous évaluons les coûts des différents algorithmes proposés, en mettant l'accent sur les coûts des communications. Est complétée par une comparaison expérimentale des méthodes sur un hypercube à 16 processeurs. Seule l'application du théorème des restes chinois, a rendu possible le calcul de la solution rationnelle de systèmes entiers d'ordres supérieurs à 700. En partie vectoriel, ce calcul demande moins de 4 heures.

Si l'on ne se limite pas aux petites dimensions pour les problèmes traités, le parallélisme et les puissances de calcul auxquelles il peut être associé, nous apparait donc comme un outil indispensable au Calcul Formel. Discipline maintenant propre à prendre le relai du Calcul Numérique quand, par exemple, l'instabilité des calculs se révèle trop importante.

Mots–clés : calcul formel, parallélisme, multi–processeur MIMD, calcul vectoriel, algorithme, systèmes linaires, solution rationnelle.