Modélisation de phénomène électromagnétiques hyperfréquences sur calculateurs parallèles

Christian Vollaire

To cite this version:

Christian Vollaire. Modélisation de phénomène électromagnétiques hyperfréquences sur calculateurs parallèles. Autre. Ecole Centrale de Lyon, 1997. Français. NNT : . tel-00139905

HAL Id: tel-00139905

https://theses.hal.science/tel-00139905

Submitted on 3 Apr 2007

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

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

N° D'ORDRE : E.C.L. 97-26 ANNEE 1997

THESE

Présentée devant

L'ECOLE CENTRALE DE LYON

Pour obtenir le grade de
DOCTEUR
(Arrêté du 30/03/1992)

Spécialité: Génie Electrique

Préparée au sein de

L'ECOLE DOCTORALE

ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE

DE LYON

par

Christian VOLLAIRE

MODELISATION DE PHENOMENES ELECTROMAGNETIQUES HYPERFREQUENCES SUR CALCULATEURS PARALLELES

Soutenue le 29 septembre 1997 devant la commission d'examen :

JURY : MM

J. L. COULOMB	Professeur - ENSIEG - LEG	**Président - Rapporteur**
M. AUBOURG	Chargé de recherche CNRS - IRCOM	**Rapporteur**
M. BUFFAT	Professeur - ECL - LMFA	**Examinateur**
M. MOUSSAOUI	Professeur - ECL - MIS	**Examinateur**
A. NICOLAS	Professeur - ECL - CEGELY	**Examinateur**
L. NICOLAS	Chargé de recherche CNRS - ECL - CEGELY	**Examinateur**
P. THOMAS	Docteur - EDF - DER	**Examinateur**

A Isabelle,

REMERCIEMENTS

Ce travail a été réalisé au Centre de Génie électrique de Lyon. Je remercie Monsieur le Professeur A. NICOLAS, directeur du CEGELY et Monsieur le Professeur Ph. AURIOL, directeur de la formation doctorale, qui m'ont accueilli dans ce laboratoire.

Je remercie Monsieur le Président et Messieurs les membres du jury pour l'honneur qu'ils m'ont fait en acceptant de juger ce travail.

J'exprime ma profonde gratitude et ma reconnaissance envers Monsieur Laurent NICOLAS, chargé de recherche CNRS, pour l'aide qu'il m'a apportée tout au long de ce travail.

Je remercie également tout le personnel du laboratoire (chercheurs, techniciens et personnel administratif) et mes camarades chercheurs pour l'ambiance, agréable et favorable au travail, qu'ils ont su créer.

Je remercie enfin mes parents pour leur inconditionnel soutien tout au long de ces longues années d'études.

RESUME

L'objectif de ce travail est le portage efficace, sur différents calculateurs parallèles, d'une formulation permettant la modélisation de phénomènes électromagnétiques hyperfréquences en régime harmonique et en espace libre. Cette démarche découle du besoin accru de modéliser des dispositifs toujours plus grands et plus complexes qui ne peuvent être résolus sur des calculateurs scalaires classiques.

Les équations de Maxwell en régime fréquenciel sont discrétisées en 3 dimensions par la méthode des éléments finis couplée à des conditions aux limites absorbantes.

Les calculateurs parallèles qui servent de support à l'implantation de la formulation sont à mémoire partagée ou distribuée. Le portage sur un CRAY C98 (mémoire partagée) est fait à partir du code séquentiel. Le parallélisme est introduit par l'adjonction manuelle de directives interprétées à la compilation. Les résultats ainsi obtenus tendent à prouver que la parallélisation des algorithmes séquentiels est une granulité adaptée à ce type d'architecture. Néanmoins, le type de stockage du système matriciel est modifié afin d'accroître les performances vectorielles. L'utilisation de calculateurs à mémoire distribuée (ferme de stations, CRAY T3E) demande un modification complète des algorithmes de sorte à exploiter au mieux les ressources de ces calculateurs en distribuant le stockage de la matrice éléments finis et en minimisant le nombre de passages de messages.

L'expérience acquise montre que le calcul parallèle permet la modélisation de géométries réalistes en 3D mais que l'utilisation de tels supports informatiques requière une bonne adéquation algorithmes - architecture cible.

Des exemples de problèmes de diffractions par des objets de grande taille sont exposés. Une application de cette méthode aux problèmes d'antennes de Vlasov de forte puissance est également décrite. Le calcul du champ lointain calculé à partir du champ proche est en bon accord avec les résultats expérimentaux.

ABSTRACT

The purpose of this work is the efficient implementation on parallel computers of a formulation for the modelling of unbounded frequency domain microwave problems. Only parallel computation actually enables to model real devices because it reduces the computation time and mainly arranges enough memory.

Maxwell's equations are discretized in three dimensions (3D) using finite elements method coupled with absorbing boundary conditions. The formulation has been implemented on parallel computers with shared or distributed memory.

The sequential package has been first implemented on a CRAY C98 (shared memory). The parallelism is introduced by adding manually compiler directives and vector performances are increased by changing the data storage mode. The results show that the parallelization of sequential algorithms can give good performances on this type of architecture.

The use of parallel distributed memory computers (cluster of workstations, CRAY T3E) requires a deep modification of the algorithms in order to obtain good performances. For example, the finite element matrix is distributed on all processors and the number of messages passing is also minimized.

Some examples of answers of realistic devices enlighted with a plane wave are shown. Application to open waveguides is also presented. The computed far field is in good agreement with those obtained by measures.

TABLE DES MATIERES

Introduction 1

Chapitre I. Généralités sur le parallélisme 5

I 1. Introduction 5
I 2. Caractéristiques matérielles 5
I 2 1. Calcul intensif 5
I 2 2. Evolution des microprocesseurs 6
I 2 3. Evolution des architectures 8
I 2 4. Réseaux d'interconnexions 10
I 3. Logiciels et parallélisme 14
I 3 1. Librairies scientifiques 14
I 3 2. Les compilateurs 15
I 3 3. Outils d'aide à la programmation parallèle 16
I 3 4. Langages de programmation parallèle 17
I 4. Concepts de base du parallélisme 19
I 4 1. Modèles de parallélisme 19
I 4 2. Techniques d'utilisation 20
I 4 3. Granulité 21
I 4 4. Dépendance de données 21
I 4 5. Parallélisme au niveau des boucles 23
I 4 6. Parallélisme au niveau des procédures 24
I 5. Mesures de performances 25
I 5 1. Loi d'Amdhal 25
I 5 2. Bancs d'essais (*Benchmarks*) 25
I 6. Conclusion 26

Chapitre II. Formulation du problème 27

II 1. Introduction 27
II 2. Equations de Maxwell en régime harmonique 28
II 2 1. Equations 28
II 2 2. Régime harmonique 29
II 2 3. Equation vectorielle des ondes 30
II 2 4. Conditions aux limites 31
II 3. Formulation par éléments finis 32

II 3 1. Introduction 32
II 3 2. Formulation par la méthode de Galerkin 33
II 3 3. Fonction de pénalité 33
II 3 4. Conditions aux limites absorbantes 34
II 3 5. Formulation complète en champ total 36
II 4. Implantation de la formulation 36
II 4 1. Préparation de la structure de la matrice 36
II 4 2. Assemblage 37
II 4 3. Traitement des symétries 37
II 4 4. Conditions aux limites sur les conducteurs électriques parfaits 38
II 4 5. Symétrisation du système matriciel 40
II 4 6. Résolution du système matriciel 41
II 5. Calcul parallèle 42
II 5 1. Contraintes dues à la formulation 42
II 5 2. Problème test 43

Chapitre III. Description des machines cibles 45
III 1. Introduction 45
III 2. CRAY C98 45
III 2 1. Configuration matérielle 45
III 2 2. Modèle de programmation 45
III 2 3. Analyse des performances et optimisation 50
III 3. Ferme de stations 52
III 3 1. Aspect matériel 52
III 3 2. Choix du modèle de programmation 52
III 3 3. Aspect logiciel 53
III 3 4. Outils d'analyse de performances et *debogage* 54
III 4. CRAY T3E 54
III 4 1. Aspect matériel 55
III 4 2. Aspect logiciel 57
III 4 3. Outils d'analyse de performances 59
III 4 4. Optimisation monoprocesseur 59
III 5. Conclusion 60

Chapitre IV. Implantation sur le CRAY C98 61
IV 1. Introduction 61
IV 2. Code parallèle et performances 61
IV 2 1. Parallélisation automatique 61

IV 2 2. Stockage 63
IV 2 3. Assemblage 64
IV 2 4. Traitement des symétries 67
IV 2 5. Conditions aux limites sur les conducteurs électriques parfaits 69
IV 2 6. Symétrisation du système matriciel 69
IV 2 7. Résolution du système d'équations 71
IV 2 8. Analyse des performances globales du code 78
IV 3. Problème réaliste 81
IV 4. Conclusion 82

Chapitre V. Implantation sur la ferme de stations 83
V 1. Introduction 83
V 2. Code parallèle 83
V 2 1. Assemblage 83
V 2 2. Traitement des symétries et des conditions aux limites sur les cep 85
V 2 3. Symétrisation du système matriciel 87
V 2 4. Résolution du système d'équations 88
V 3. Problème réaliste 106
V 4. Conclusion 106

Chapitre VI. Implantation sur le CRAY T3E 108
VI 1. Introduction 108
VI 2. Code parallèle 108
VI 2 1. Code parallèle muni de PVM 109
VI 2 2. Code parallèle muni de SHMEM 112
VI 3. Problème réaliste 119
VI 4. Conclusion 120

Chapitre VII. Exemples de calcul 121
VII 1. Introduction 121
VII 2. Exemples 121
VII 2 1. Diffraction par un cylindre 121
VII 2 2. Diffraction par un avion 122
VII 2 3. Guide d'ondes ouverts 125

Conclusion 129

Bibliographie 133

Annexe 1. PVM 140

Annexe 2. Configuration matérielle des machines cibles 142

Annexe 3. Algorithmes du GC préconditionné et de QMR préconditionné 144

INTRODUCTION

Depuis quelques années, l'utilisation des moyens informatiques de plus en plus performants a contribué à l'essor de l'analyse numérique des problèmes physiques rencontrés dans le domaine des sciences pour l'ingénieur. Grâce à l'évolution de la puissance de calcul des ordinateurs, il est possible de prévoir le comportement d'un système et ainsi de l'optimiser avant la construction d'un prototype. Le gain en terme de temps et de coût pour le développement d'un produit est donc devenu considérable. Un autre aspect intéressant et souvent essentiel de la modélisation réside aussi dans la possibilité de connaître des grandeurs difficilement mesurables.

La modélisation de problèmes physiques consiste à résoudre les équations qui les régissent. Dans cette étude, nous nous intéressons aux phénomènes électromagnétiques dont le comportement est dicté par les équations de Maxwell. Nous cherchons en particulier à résoudre des problèmes de diffraction électromagnétique hyperfréquence (0.1 Ghz - 300 GHz) en régime harmonique. Ainsi, la formulation mise en oeuvre permet, par exemple, de calculer la réponse électromagnétique d'un objet illuminé par une onde plane ou de calculer les champs électromagnétiques dans et aux abords d'antennes ou de guides d'ondes ouverts hyperfréquence. Le calcul analytique des solutions ne pouvant être effectué que pour quelques cas, les solutions sont approchées numériquement. Pour ce faire, la méthode des éléments finis (EF) nodaux est utilisée couplée à des conditions aux limites absorbantes (CLA) afin de traiter des problèmes ouverts. Les champs électromagnétiques lointains sont déterminés à partir de méthodes intégrales (section radar, gain d'antenne, ...)

La modélisation de tels dispositifs en trois dimensions, à l'aide de la méthode des éléments finis, impose un maillage volumique dans tout le domaine d'étude même si celui-ci est réduit de par l'utilisation des conditions aux limites absorbantes. De plus, dix noeuds par longueur d'onde sont nécessaires pour obtenir un résultat d'une bonne précision. La formulation adoptée conduisant à trois inconnues complexes par noeud, les temps de calcul et surtout l'espace mémoire requis ne permettent pas le traitement de certaines géométries réalistes dans

cette gamme de fréquence. A l'heure actuelle, seul le calcul parallèle est en mesure de fournir les ressources informatiques nécessaires. Cette étude porte donc sur la parallélisation des algorithmes utilisés dans la méthode des éléments finis appliquée à nos équations.

L'utilisation du calcul parallèle requiert une culture informatique ainsi qu'une méthodologie de programmation spécifiques qui sont liées à l'architecture du calculateur cible. En effet, une bonne connaissance architecturale et logicielle de la machine, sur laquelle est développé le code, est nécessaire à l'obtention de performances en rapport avec ses capacités théoriques. La principale difficulté liée à l'utilisation du calcul parallèle réside dans le choix des algorithmes de chaque étape du code qui doivent permettre d'utiliser au mieux les ressources de la machine. La formulation a donc été portée sur deux types de calculateurs parallèles les plus représentatifs du marché actuel et même futur.

Le premier est le CRAY C98 de l'IDRIS (Institut Des Ressources en Informatique Scientifique), qui est un calculateur parallélo-vectoriel de type MIMD à mémoire partagée (multiprocesseurs). Tous les processeurs ont accès à une mémoire centrale et l'échange de données entre les processeurs se fait par accès à la même adresse mémoire. Le code séquentiel développé sur station de travail a été porté sur le CRAY C98 et le parallélisme introduit par l'adjonction de directives spécifiques interprétées à la compilation. Les résultats obtenus en termes de performances parallèles ont montré que cette démarche suffisait pour exploiter correctement les ressources de ce calculateur. En effet, la parallélisation des différentes boucles du code séquentiel s'est avérée être une granulité tout à fait adaptée à cette architecture. Néanmoins, certaines modifications structurelles du code ont été nécessaires afin d'augmenter les performances vectorielles du programme qui sont fondamentales sur un tel calculateur. Dans cette étude sera montré, en détails, comment porter efficacement un code séquentiel utilisant la méthode des éléments finis sur ce type de calculateur parallèle.

Le deuxième type de calculateur parallèle utilisé est de type MIMD à mémoire distribuée (multiordinateurs). La ferme de stations de l'Ecole Centrale de Lyon, ainsi que le CRAY T3E de l'IDRIS ont servi de support à cette implantation. Chaque processeur possédant sa propre mémoire, les échanges de données se font par passages de messages. Une réécriture complète du code a été nécessaire pour obtenir de bonnes performances parallèles. En effet, les

algorithmes séquentiels ne se prêtaient pas du tout à une bonne utilisation des ressources notamment au niveau de l'occupation mémoire. D'autre part, sur ce type d'architecture, il est nécessaire de minimiser le nombre de passages de messages qui introduisent un important surcoût qui fait chuter les performances parallèles du code. Une granulité de parallélisme beaucoup plus grande que celle employée sur le CRAY C98 a donc été adoptée.

Les constructeurs de super-calculateurs ayant tendance à s'orienter vers des architectures à mémoire distribuée, les algorithmes testés dans cette étude peuvent servir à orienter judicieusement les choix pour les développements de nouveaux codes. De ce fait, les programmes séquentiels ainsi mis au point se paralléliseraient facilement, ce qui est devenu aujourd'hui une étape nécessaire dans la vie d'un code de calcul. Le but de cette étude est donc de mettre en évidence d'une part l'apport du calcul parallèle dans le domaine de la modélisation numérique en électrotechnique et, d'autre part, de dégager des directives générales d'écriture de codes de calcul utilisant la méthode des EF en vue d'un portage sur un type de calculateur parallèle donné. En effet, il faut arriver à trouver la meilleure adéquation algorithmes / machines cibles, ceci étant le problème crucial de l'utilisation du calcul parallèle. De plus, ces directives peuvent être généralisées à quelque formulation que ce soit, même temporelle, les différentes phases d'un code de calcul utilisant la méthode des EF restant les mêmes.

Le premier chapitre est relatif aux généralités sur le calcul parallèle et introduit les principales notions fondamentales nécessaires à la compréhension des chapitres suivants. Dans un premier temps, les principales caractéristiques matérielles liées au calcul parallèle sont présentées. Une seconde partie est consacrée aux principaux outils logiciels rencontrés sur le marché. Les concepts de bases essentiels sont ensuite exposés. La dernière partie traite des possibilités de mesures de performances en calcul parallèle.

Dans le second chapitre, les équations de Maxwell régissant les problèmes de propagation d'ondes sont décrites brièvement. Des conditions aux limites absorbantes sont, par la suite, introduites et couplées à la formulation. Dans le cadre de notre étude, une formulation faible des équations est présentée, préalable à une discrétisation par EF. Les algorithmes utilisés pour l'implantation efficace de cette formulation et les limites en terme de taille des problèmes pouvant être résolus sur des calculateurs scalaires classiques sont enfin exposés.

Une description des différentes machines cibles et de leurs modèles de programmation est proposée dans le troisième chapitre.

Les chapitres quatre, cinq et six détaillent l'implantation et, par là-même, les algorithmes parallèles développés sur chaque machine cible. Les performances parallèles sont présentées sur un même problème test, pour chaque machine, afin de faciliter la comparaison.

Le dernier chapitre propose quelques exemples de géométries réalistes ainsi qu'une validation d'un calcul de guide d'onde ouvert de forte puissance à partir de mesures expérimentales. Ce dernier point montre bien l'apport du calcul parallèle dans la prédiction du comportement de systèmes complexes impossibles à modéliser à l'aide de calculateurs scalaires monoprocesseur.

Enfin, la conclusion synthétise les remarques issues de chaque chapitre pour exprimer quelques directives issues de l'expérience acquise durant cette étude et ouvre des perspectives pour un travail futur sur des méthodes non expérimentées ici.

CHAPITRE I
GENERALITES SUR LE PARALLELISME

I 1. Introduction

Dans ce chapitre sont exposés les concepts de base liés au calcul parallèle tant au niveau architecture que logiciel. Ce chapitre s'adresse surtout aux physiciens qui utilisent des méthodes numériques pour la simulation de phénomènes physiques et qui, comme nous, sont amenés à utiliser le calcul parallèle afin de pouvoir modéliser des systèmes plus gros, plus complexes et par la même réalistes. En effet, l'utilisation du calcul parallèle requiert une culture informatique ainsi qu'une méthodologie de programmation spécifiques liées à l'architecture du calculateur sur lequel l'application est développée.

Dans un premier temps, les principales caractéristiques matérielles liées au calcul parallèle sont présentées. Une seconde partie est consacrée aux principaux outils logiciels rencontrés sur le marché. Les concepts de bases essentiels sont ensuite exposés. La dernière partie traite des possibilités de mesures de performances en calcul parallèle.

I 2. Caractéristiques matérielles

I 2 1. Calcul intensif

Depuis les premiers temps de l'informatique, l'utilisation des ordinateurs s'est étendue à beaucoup de domaines scientifiques ou non. Cette demande accrue de calcul numérique se ressent tant au niveau des industriels qu'à celui du grand public. De nos jours, grâce à l'évolution de la puissance des ordinateurs, il est possible de prévoir le comportement d'un prototype sans avoir à le construire. De ce fait, le gain en terme de temps et de coût pour le développement d'un produit est devenu considérable. Les sciences qui mettent en oeuvre de tels calculs ont évolué de manière impressionnante. Elles agissent comme un catalyseur, entraînant toujours vers le haut la demande en puissance de calcul. Pour répondre à ce besoin, les fabriquants d'ordinateurs ont dû entreprendre des recherches dans plusieurs directions. Il

leur a fallu augmenter la puissance de calcul des microprocesseurs soit en augmentant l'intégration (donc la rapidité) au niveau d'un composant, soit en faisant évoluer l'architecture de la puce elle-même. Il semblerait que cette technologie (semi-conducteurs à base de silicium) ait atteint ses limites physiques. Pour cette dernière raison, une autre façon d'augmenter la puissance de calcul d'un ensemble est de faire exécuter simultanément par plusieurs unités de calculs les opérations à effectuer. Actuellement, les utilisateurs de super-calculateurs sont confrontés à un choix difficile quant au type d'architecture à adopter. En effet, sont disponibles sur le marché un grand nombre de calculateurs dont le choix nécessite une bonne adéquation entre l'architecture et le type d'applications.

I 2 2. Evolution des microprocesseurs

I 2 2 1. CISC

Jusqu'à une époque récente, l'évolution des processeurs allait dans le sens d'une complexité grandissante (C*ompleted* I*nstruction* S*et* C*omputer*: CISC). Ceci permet de garder une compatibilité avec les logiciels déjà existants. Certains constructeurs d'ordinateurs n'étant pas obligés de respecter cette règle se sont orientés vers d'autres architectures de puce.

I 2 2 2. RISC

L'architecture RISC (R*udesse* I*nstruction* S*et* C*omputer*) doit son apparition au fait, que malgré l'utilisation de langages évolués, une prédominance d'instructions simples peut être mise en évidence dans les codes générés. Les instructions microcodées sont rarement utilisées du fait de leur manque de généralité. Cette technologie qui bénéficie des avancées en matière de technique de compilation utilise donc des RAM plus rapides d'accès que des ROM comme un *cache* d'instructions. Les concepteurs de processeurs RISC tentent de limiter le temps d'exécution des instructions à un cycle d'horloge en émulant les autres instructions à l'aide de séquences basées sur les instructions élémentaires. Les accès mémoire se font à l'aide de registres et d'instructions limitées (*load/store*). Ceci est dû au fait qu'une instruction qui s'exécute en un cycle d'horloge ne peut pas calculer les adresses de ses opérandes et ceux-ci doivent donc se trouver dans les registres avant l'opération. Une lecture en mémoire consiste donc à utiliser une adresse placée en registre pour chercher un mot en mémoire et le placer dans un autre registre. Toutes les instructions travaillent sur des opérandes.

I 2 2 3. Vectoriel

Ce type de processeur permet le traitement vectoriel de flux d'opérandes scalaires. Un tel processeur est composé de plusieurs étages successifs de traitement. Le flux de données doit être continu et si possible organisé de manière contiguë en mémoire. Chaque unité de traitement va donc réaliser une opération élémentaire sur le flux de données qui progresse dans le *pipeline*. Une fois le *pipeline* rempli, il en sort un résultat à chaque pas de progression.

Illustrons cette technique par un exemple. Soit la multiplication de deux vecteurs notés A et B longs chacun de N opérandes. Le résultat est stocké dans un vecteur noté C. Cette opération peut être décomposée en micro-instructions [1], [8]: comparer les exposants, décaler les mantisses, multiplier les mantisses et ajuster l'exposant du résultat. Les N éléments des deux vecteurs vont progresser dans un *pipeline* à 4 étages comme l'illustre la figure (1.1).

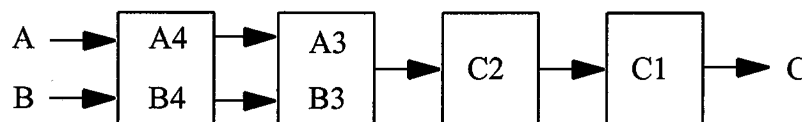


Fig. 1.1: Pipeline à 4 étages.

Soient N_{op} le nombre de micro-instructions nécessaires au calcul de l'opération souhaitée (4 dans notre exemple), N la taille des vecteurs et T le temps entre chaque pas de progression. Le temps nécessaire au traitement séquentiel de cette opération est donné par l'équation (1.1).

$$T_{seq} = N N_{op} T \tag{1.1}$$

Pour le traitement vectoriel, le premier résultat ne sera disponible qu'au bout du temps T_o appelé temps de chargement du *pipeline*. Par la suite, un résultat par pas de progression sera délivré. Par conséquent, le temps nécessaire au traitement vectoriel est dicté par l'équation (1.2).

$$T_{vec} = T_o + N T \tag{1.2}$$

Le gain obtenu par traitement vectoriel est donc (1.3):

$$G_v = \frac{T_{seq}}{T_{vec}} = \frac{N_{op}}{1 + \frac{T_o}{N T}} \tag{1.3}$$

Cette expression, qui n'est évidemment qu'approximative, permet de mettre en évidence plusieurs principes relatifs à la vectorisation: quand N augmente, le gain tend vers sa valeur maximale; cette valeur est proportionnelle à N_{op}; si N est trop faible le terme T_o devient prédominant dans l'expression du gain par vectorisation. L'accès aux données par le processeur vectoriel est un facteur déterminant en terme de performances. Certains constructeurs ont choisi de faire accéder directement le *pipeline* à la mémoire pour la lecture des données et le rangement des résultats. Une telle technique demande une bande passante mémoire très élevée. L'autre choix consiste à équiper le processeur de registres vectoriels de taille fixe (8 registres vectoriels de 128 mots sur le CRAY C98) et d'accès par le *pipeline* très rapide. Les vecteurs de grandes tailles doivent alors être découpés à la taille des registres vectoriels, ce qui est un facteur limitatif quant aux performances maximales.

I 2 3. Evolution des architectures

La taxonomie de M. J. Flynn [2], datant de 1966, classifie les architectures d'ordinateurs à partir des concepts de flots de données et de flots d'instructions. Un flux de contrôle ou d'instructions est une séquence d'instructions exécutées par l'ordinateur, tandis qu'un flux de données est une séquence de données servant à exécuter un flux d'instructions. En combinant les quatre possibilités que constituent le flot unique/multiple de données/instructions, les catégories suivantes sont obtenues:

I 2 3 1. SISD (*Single Instruction, Single Data stream*)

Les ordinateurs séquentiels monoprocesseurs sont, pour la plupart, des machines de ce type. Ces ordinateurs possèdent une seule unité de contrôle mais peuvent être dotés de plusieurs unités opératives.

I 2 3 2. SIMD (*Single Instruction, Multi Data stream*)

Ce type d'architecture est caractérisé par l'unicité de son unité de contrôle. Elle possède plusieurs processeurs élémentaires ainsi qu'un réseau d'interconnexions processeur à processeur ou processeur à mémoire. L'unité de contrôle commande à tous les processeurs d'exécuter, de manière synchrone, des instructions pas à pas sur des données locales. Le réseau d'interconnexions (grille, cube, hypercube, ...) permet l'échange de données entre les processeurs. Ce type de machine peut utiliser jusqu'à plusieurs milliers de processeurs

"élémentaires". Le développement d'algorithmes parallèles adaptés aux machines de type SIMD peut dans certains cas être très simple s'il est possible de fractionner en un ensemble de sous-problèmes identiques le problème à traiter [3]. La granulité ainsi que le type de parallélisme sont dépendants de l'organisation physique des unités de traitement (réseau de communications).

I 2 3 3. MISD (M*ulti* I*nstruction*, S*ingle* D*ata stream*)

Ce type d'architecture n'est pas utilisé à l'heure actuelle.

I 2 3 4. MIMD (M*uti* I*nstruction*, M*ulti* D*ata stream*)

Cette catégorie regroupe les systèmes multiprocesseurs qui utilisent plusieurs flux de contrôles et plusieurs flux de données. Ce type d'architecture consiste donc à interconnecter un certain nombre d'ordinateurs complets. Sa caractéristique principale repose sur la façon dont l'information est partagée par les différentes unités de contrôle: via une mémoire centrale (multiprocesseurs) ou via un réseau de communications servant de support aux passages de messages (multi-ordinateurs). Malgré le caractère asynchrone de leur fonctionnement et la difficulté qui en découle à les programmer, les calculateurs de type MIMD sont très répandus sur le marché. En effet, ils supportent un parallélisme de grain plus fort que celui utilisé par les machines de type SIMD. Ils sont donc d'une utilisation plus générale. Comme il est dit plus haut, la façon dont l'information est partagée par les différentes unités de contrôle implique encore un découpage de cette catégorie.

MIMD à mémoire globale physiquement partagée:

Dans ce type de machine, tous les processeurs accèdent à la mémoire centrale partagée par le biais d'un réseau d'interconnexions. L'échange d'informations entre les processeurs se fait donc par l'accès aux mêmes adresses dans la mémoire centrale. De ce fait, le facteur prédominant qui limite les performances est la bande passante de la mémoire. De plus le système doit permettre de gérer les accès mémoire quand plusieurs processeurs doivent modifier la même adresse mémoire au même moment. Il doit également permettre de savoir si la donnée présente à une adresse mémoire est valide au moment où elle est lue, c'est à dire si elle n'a pas déjà été modifiée par un autre processeur. Ce dernier point sera développé ultérieurement dans le chapitre relatif aux machines cibles. A titre d'exemples peuvent être cités le CRAY C94-98 ou encore Alliant FX 2800, Convex, ...

MIMD à mémoire distribuée:

Ces calculateurs sont en fait constitués de plusieurs ordinateurs reliés entre eux par un réseau de communications à haut débit. Chaque noeud de calcul possède sa propre mémoire et l'échange de données ainsi que les synchronisations se font au travers du réseau d'interconnexions (IPSC 860 d'Intel, NCUBE modèle 2, ferme de stations de travail, ...).

MIMD à mémoire distribuée virtuellement partagée:

Cette catégorie utilise des processeurs possédant chacun leur espace mémoire. Pour émuler une mémoire partagée, l'espace adressable vu par le programmeur est unique. Cette fonction est généralement gérée au niveau *hardware* par un système spécifique à chaque machine. Citons pour exemples le KSR1 de Kendall Square Research + système Allcache ou encore le CRAY T3E + SHMEM.

La classification de Flynn qui présente certains intérêts reste toutefois insuffisante pour classifier un certain nombre de super-calculateurs modernes. En effet, il est difficile de classer les processeurs vectoriels suivant que l'on considère les données comme un flux (SIMD) ou séparément (SISD). Néanmoins, cette classification présente l'intérêt d'être indépendante de toute considération architecturale précise.

I 2 4. Les réseaux d'interconnexions

L'interconnexion peut connecter des processeurs à des modules mémoire, ou encore, des processeurs entre eux [4]. Si les connexions sont directes entre processeurs, le réseau d'interconnexion est en général statique, tandis qu'il est dynamique si les connexions se font par l'intermédiaire d'une mémoire commune partagée en bancs. Dans ce dernier cas, il est nécessaire d'utiliser un algorithme de routage. La commande du réseau est différente suivant que l'échange d'informations se fait de manière synchrone (SIMD) ou asynchrone (MIMD). Enfin, le nombre de connexions permet aussi de classifier les machines. En effet, différents types d'interconnexions peuvent être envisagée [5]: chaque processeur peut être relié par des connexions fixes (réseau complet fixe) ou non (réseau complet réarrangeable) à tous les autres processeurs ou à tous les bancs mémoire, certaines permutations entre processeurs (ou banc mémoire) étant permises par le réseau.

I 2 4 1. Réseaux dynamiques

Parmi les réseaux dynamiques, plusieurs catégories peuvent être distinguées:

Les réseaux sans blocage: un réseau est sans blocage si, pour tout ensemble de liaisons établies, toute entrée inactive a accès à tout sortie inactive. Ces réseaux autorisent donc le débit maximum entre les différentes unités en entrée et en sortie. C'est Clos [5] qui a énoncé les règles d'existence de tels réseaux (condition de non-blocage). Les réseaux sans blocage ont une taille importante qu'il est difficile de minimiser sans affaiblir cette notion. De ce fait, presque aucune réalisation pratique ne peut être trouvée.

Les réseaux réarrangeables: pour palier au problème mis en évidence au paragraphe précédent, la structure du réseau est recalculée à chaque étape. Certaines restrictions sont néanmoins nécessaires. Le nombre d'entrées doit être égal au nombre de sorties et le fonctionnement doit être synchrone. Un tel réseau peut se bloquer étant donné que pour établir une liaison entre une entrée inactive et une sortie inactive, des liaisons existantes peuvent être modifiées.

Les réseaux avec blocage: un réseau possède un blocage si la condition de réarrangeabilité n'est pas vérifiée. Il n'est donc pas toujours possible de relier une entrée à une sortie.

Les réalisations pratiques de ce type de réseau d'interconnexions sont très répandues, citons pour exemple:

- Les bus: les unités fonctionnelles sont rassemblées autour d'un bus. Un contrôleur de bus gère l'accès aux ressources.

- Les réseaux crossbar: ce type de réseaux, appelé aussi matrice de points de croisement, est le modèle d'interconnexions dynamiques le plus simple. Toutes les lignes d'entrée croisent toutes les lignes de sortie et à chaque croisement est placé un interrupteur. Une connexion entre une entrée quelconque et une sortie quelconque est toujours possible (fig. 1.2).

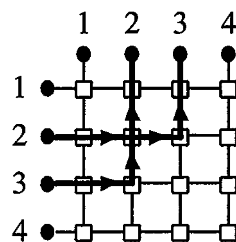


Fig. 1.2: Connexions entre points sur un crossbar.

I 2 4 2. Réseaux statiques

Dans la littérature, un réseau de processeurs est souvent représenté par un graphe non orienté dont les arêtes sont les liens de communications entre les noeuds et les processeurs. Voici quelques définitions utiles à la compréhension de ce paragraphe.

Degré d'un noeud: nombre d'arêtes adjacentes.

Degré d'un réseau: maximum des degrés de tous les noeuds (degré moyen noté Δ si les noeuds n'ont pas tous le même degré).

Excentricité d'un sommet: la plus grande distance de ce sommet à un sommet quelconque.

Diamètre noté D: maximum des excentricités des sommets du réseau.

Les principales réalisations pratiques sont les suivantes:

- Réseaux linéaires et anneaux: cette topologie est très utilisé dans les multiprocesseurs à mémoire distribuée. Pour un réseau linéaire, chaque noeud possède deux voisins sauf le dernier et le premier. Si le réseau contient P processeurs, son diamètre est de P (fig. 1.3).

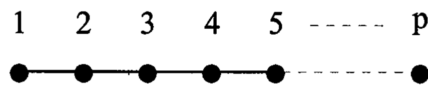


Fig. 1.3: Réseau linéaire.

Dans un anneau, chaque processeur possède 2 voisins. Pour un anneau de P processeurs le diamètre est de P/2 (fig. 1.4).

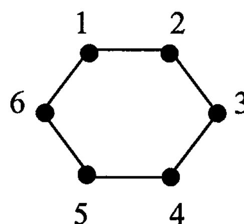


Fig. 1.4: Réseau en anneau de 6 processeurs et de diamètre (D) égal à 3.

- Grilles: soit une grille comportant n1 lignes et n2 colonnes. Chaque processeur possède 4 voisins excepté les processeurs situés sur les premières et dernières lignes/colonnes. La communication entre 2 sommets opposés nécessite de traverser n1 + n2 - 2 liens (fig. 1.5).

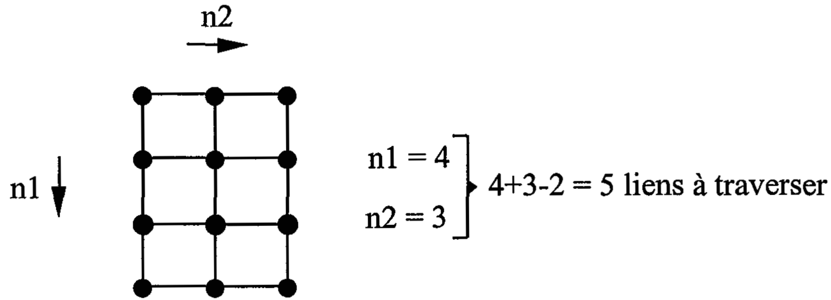


Fig. 1.5: Grille.

- Grilles toriques: une grille torique est obtenue en reliant entre eux les processeurs des premières et dernières lignes à ceux des premières et dernières colonnes d'une grille. Cette topologie est très répandue du fait de la simplicité de mise en oeuvre de certains algorithmes (matrices, ...). Son degré est égal à 4 et son diamètre à n1/2+n2/2 (fig. 1.6).

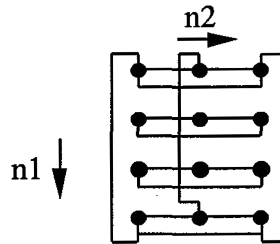


Fig. 1.6: Grille torique.

En extrapolant, l'hypergrille est définie. Elle possède une dimension de n et $\prod_{k=1}^{n} q_k$ noeuds (q_k noeuds sur chaque dimension k (1, ..., n)). Chaque processeur possède 2n voisins.

- Hypercube: l'hypercube de degrés d est une topologie à 2^d noeuds où chaque noeud a exactement d voisins. Un cube de degré d est défini à partir de 2 cubes de degré d-1, où les noeuds correspondant sont reliés. Le diamètre d'un hypercube de degré d est d (fig. 1.7).

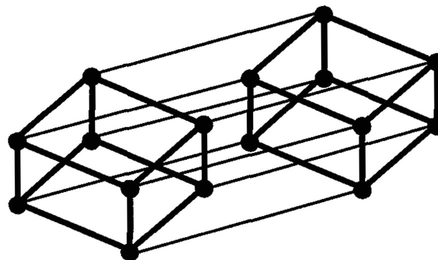


Fig. 1.7: Hypercube de degré 4.

- Arbres binaires: ce type de structure a l'avantage de représenter naturellement des machines hiérarchisées, ou bien des stratégies maître-esclave. Un arbre binaire complet est un graphe à 2^n-1 sommets. Le degré moyen des processeurs est égal à 3, la racine a un degré 2 et les noeuds terminaux un degré 1. Le diamètre est de 2 [$\log_2$ (p)], où p est le nombre de processeurs. Tous les noeuds sont à une distance inférieure à $\log_2$ (p) de la racine (fig. 1.8).

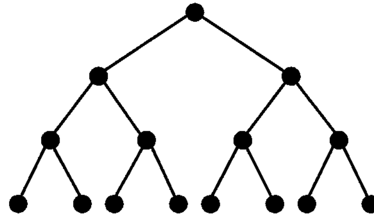


Fig. 1.8: Arbre binaire de profondeur 3.

Remarque: dans la pratique, les réseaux de processeurs sont souvent de degrés bornés assez faibles. Compte tenu de l'évolution des algorithmes pour des structures régulières, sans limitation du nombre de processeurs, il est nécessaire de pouvoir créer une topologie donnée à partir d'un réseau de degré plus faible. Par exemple, beaucoup de travaux ont été menés sur des réseaux où l'on connecte en hypercube des groupes de processeurs déjà reliés par d'autres réseaux (anneau, grille, arbre, ...).

I 3. Logiciels et parallélisme

Pour tirer profit des différents calculateurs parallèles, il est indispensable que ceux-ci soient dotés d'un environnement logiciel adapté à leur architecture physique. Dans cette partie, les principaux outils mis à la disposition des programmeurs sur la plupart des plates-formes parallèles sont présentés [5], [6], [7].

I 3 1. Librairies scientifiques

La plupart des librairies existantes peuvent être classées en trois catégories: librairies mathématiques, librairies de programmation et librairies graphiques. Lors du développement d'une application, il est conseillé au programmeur de s'appuyer sur les librairies fournies par le constructeur car celles-ci sont en général bien adaptées à l 'architecture de la machine.

Le premier niveau de librairies scientifiques est constitué par l'ensemble des BLAS (B*asic* L*inear* A*lgebra* S*ubroutines*). LAPACK et LINPACK utilisent les BLAS qui possèdent trois

niveaux: les BLAS 1 traitent des opérations entre vecteurs et utilisent typiquement O(N) opérations flottantes par appel, les BLAS 2 traitent des opérations entre matrices et vecteurs avec $O(N)^2$ opérations flottantes par appel, et les BLAS 3 traitent des opérations entre les matrices et utilisent $O(N)^3$ opérations flottantes. Les BLAS sont principalement utilisés par les programmeurs qui peuvent ainsi développer de façon efficace en s'appuyant sur ces briques élémentaires. Les constructeurs, quant à eux, se limitent généralement à leur utilisation dans les *benchmarks*.

I 3 2. Les compilateurs

La plupart des codes développés sur les calculateurs parallèles sont écrits dans des langages de haut niveau tel que le C, C++, FORTRAN, ... Le passage du programme au code machine, devant utiliser au mieux l'architecture de la machine cible, se fait par l'intermédiaire du compilateur. L'utilisation simple et optimale des super-calculateurs dépend fortement des compilateurs disponibles. De ce fait, de par l'arrivée des techniques vectorielles et la multiplication des architectures parallèles, les compilateurs modernes doivent faire appel à des techniques de plus en plus sophistiquées pour tirer profit de la puissance du matériel.

I 3 2 1. Optimisation automatique

Afin de libérer le programmeur d'un certain nombre de tâches, les compilateurs ont évolué pour réaliser celles-ci automatiquement. En général, ce type de compilateur effectue son travail uniquement sur les boucles. La figure (1.9) présente, de façon générale, les différentes phases de la compilation.

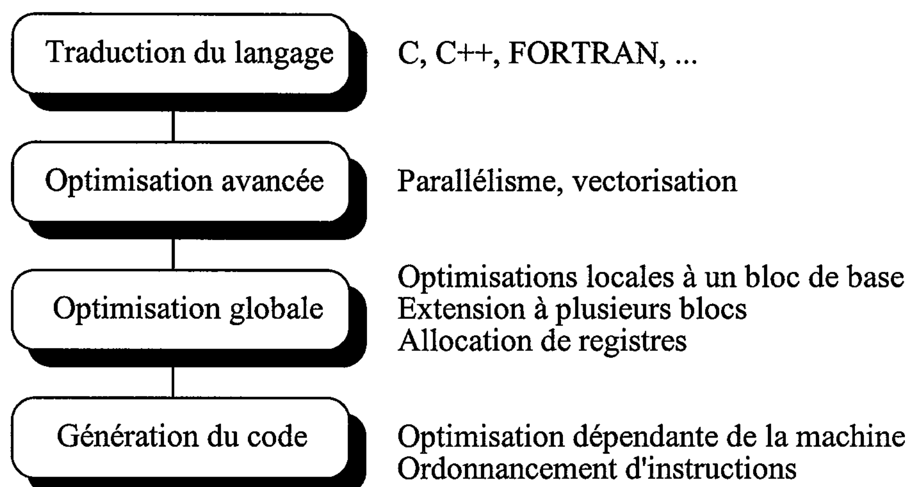


Fig. 1.9: Etapes de compilation.

En fonction de leur nature, scalaire, vectorielle ou parallèle, différentes optimisations peuvent être réalisées.

- Scalaire: éliminer les sous-expressions communes, remplacer les références à un pointeur par des références aux tableaux associés, déplacer du code, assembler des *if* consécutifs en *if-then-else,* éliminer les scalaires par substitution, reconnaître et convertir des variables d'induction, convertir les scalaires en vecteurs ou encore propager les copies de variables.

- Vectorielles et parallèles: détecter les boucles vectorisables et parallélisables, analyser les restructurations de boucles possibles, réordonner les instructions pour une meilleure efficacité, diviser les boucles ou encore classer les réductions et les récurrences.

- Parallèles: minimiser les synchronisations, gérer la répartition de la charge grâce à des techniques de distribution dynamiques ou statiques des itérations ou bien minimiser le surcoût dû au parallélisme.

- Accès mémoire et utilisation des *caches*: marquer comme *cachables* les scalaires étendus en vecteurs, utiliser des techniques de blocs pour maximiser l'efficacité des données contenues dans les caches, extraire des boucles les groupes d'instructions qui calculent toujours la même valeur à chaque itération ou encore découper les boucles trop longues en boucles plus petites pour tenir dans les caches.

Ces quelques exemples montrent bien la difficulté que représente la mise au point d'un compilateur de ce type. Ces compilateurs travaillant presque exclusivement sur les boucles, les premiers essais de développement pour le parallélisme n'ont pas été très concluants. En effet, d'autres éléments sont à prendre en compte dans ce cas, notamment l'analyse inter-procédurale qui est encore trop complexe pour être intégrée dans un compilateur. La transformation du code séquentiel en code exécutable parallélo-vectoriel nécessite donc une intervention du programmeur faute de quoi les performances ainsi obtenues sont médiocres. De plus, ces compilateurs ne permettant pas d'exprimer directement un algorithme parallèle, les programmeurs se voient obligés de formuler leur algorithme sous une forme séquentielle, ce qui s'avère être une contrainte assez lourde dans certains cas.

I 3 3. Outils d'aide à la programmation parallèle

Un certain nombre d'outils d'aide à la programmation parallèle sont disponibles. Ils peuvent être de différents types: automatiques comme FPP et FMP sur le CRAY C94-98, interactifs

comme ATEXPERT, XBROSE et ATSCOPE sur le CRAY C94-98 ou encore extensibles au réseau comme PVM, MPI, ...

L'analyse inter-procédurale, difficile à automatiser aujourd'hui, est essentielle. L'avantage des outils cités précédemment réside dans le contrôle plus complet laissé au programmeur sur le choix à faire lors d'une résolution de dépendance ou sur la granulité de parallélisme à adopter. Il est possible d'utiliser les compilateurs comme des outils d'aide à la parallélisation. En effet, l'optimiseur fournit généralement une liste des boucles non optimisées, il est alors possible de comprendre la dépendance ou le problème bloquant la parallélisation et d'agir en conséquence, par exemple en ajoutant manuellement des directives.

Les environnements logiciels tels que PVM ou encore MPI permettent l'utilisation d'un réseau hétérogène composé de machines parallèles et monoprocesseur comme une ressource de calcul unique. Ces logiciels sont donc intéressants par leur capacité à paralléliser une application sur un réseau hétérogène. Bien adaptés au calcul distribué, ils donnent généralement des résultats plus intéressants sur des applications à grain important qui ne nécessitent pas beaucoup de passages de messages.

I 3 4. Langages de programmation parallèle

Les façons d'exprimer la nature parallèle d'une application le plus efficacement sont très diverses. Le programmeur doit souvent faire un compromis entre l'exploitation efficace des ressources de la machine et la portabilité de son code. En effet, dans un premier temps, l'application est développée à l'aide d'un langage standard tel que le C, C++, FORTRAN, ... Celui-ci est indépendant de la machine et n'utilise pas les possibilités de celle-ci de façon optimale. Par la suite, le compilateur extrait le parallélisme inhérent à l'application. Pour poursuivre l'optimisation, l'adjonction de directives ou d'extensions au programme est nécessaire. Ceci a pour conséquence l'éloignement du code optimisé de la version standard indépendante de la machine cible.

Les paragraphes qui suivent décrivent brièvement certains langages qui optimisent le code en le rendant le plus indépendant possible de l'architecture de la machine cible tout en conservant une efficacité raisonnable.

I 3 4 1. Langages conventionnels

Cette approche reprend le principe de l'utilisation d'un compilateur détectant et exploitant le parallélisme contenu dans le programme. Mais, comme cela est dit plus haut, ce type de compilateur travaille généralement au niveau des boucles. La reconnaissance du parallélisme à haut niveau d'abstraction nécessite l'intervention du programmeur. Cette approche présente néanmoins l'avantage de n'apporter aucune modification au code source.

I 3 4 2. Extensions de bas niveaux (#pragma, ...)

Cette méthode permet, sans trop modifier le programme, de tenir compte du parallélisme en créant différents processus et synchronisations. Evidemment les directives ajoutées varient selon la machine cible. De ce fait, la compréhension et la portabilité d'une machine à l'autre s'en trouvent fortement diminuées. En revanche, avec une bonne utilisation, leur efficacité peut s'avérer très élevée. Ces techniques sont basées sur l'adjonction de directives et d'appel à des librairies orientées multitraitements ou parallélisme.

I 3 4 3. Extensions de hauts niveaux (FORTRAN 90, HPF, ...)

Un bon exemple de ce type de langage est l'évolution du FORTRAN 77: le FORTRAN 90. Ce langage prend en compte certains éléments des aspects vectoriels et parallèles du calcul scientifique. Cette évolution est, en fait, un sous ensemble du F77 dont certaines instructions ont été éliminées (IF arithmétique, indices de boucles non entiers, ...).

Les principales nouveautés résident dans l'ajout d'instructions de contrôle plus complètes (SELECT, CASE pour les choix multiples, DO/END, DO, EXIT pour les boucles, ...). La récursivité, l'introduction des modules, le rassemblement des données et les opérations sur ces types et sous-programmes peuvent être utilisés. De plus, le F90 permet d'allouer dynamiquement des tableaux (ALLOCATE, DEALLOCATE). Les pointeurs sont introduits (POINTER), ainsi que les manipulations sur les tableaux.

Toutefois, cette nouvelle norme reste limitée car elle ne permet, par exemple, que le parallélisme sur les données (programmation vectorielle et parallélisme SIMD). Aucune fonctionnalité de gestion de la mémoire partagée, de passages de messages ou encore de parallélisme de boucles complexes n'est prévue.

Il faut signaler une dernière évolution du FORTRAN appelée HPF (*High Performance* FORTRAN). Celle-ci est en cours de développement et devrait comprendre la notion de mémoire virtuelle partagée (physiquement distribuée ou non). De ce fait, le même type de

programmation sur les machines parallèles à mémoire partagée et sur celles à mémoire distribuée devrait être possible.

I 3 4 4. Langages dédiés au parallélisme de données

Ces langages permettent de tirer profit du parallélisme de données sur des architectures différentes. Ils doivent posséder les caractéristiques suivantes: utilisation, comme base, d'un langage classique et ajout d'instructions permettant l'exploitation de façon explicite du parallélisme sur les données. Le FORTRAN D qui est une variante du FORTRAN en est un exemple. Il permet de spécifier la décomposition de données à l'aide d'instructions comme DECOMPOSITION, ALIGN, ... Le problème se pose alors à deux niveaux: l'alignement des données pour permettre de réduire les coûts de communications, ainsi que la distribution des données sur la machine physique en tenant compte de sa topologie et de son architecture.

En conclusion, il apparaît que l'environnement de travail standard de l'utilisateur de machines parallèles est en train de se mettre en place. En effet, les constructeurs tendent à développer un environnement indépendant de la machine cible. Les progrès réalisés en terme de portabilité et de qualité des logiciels permettent de penser que cet environnement sera un jour opérationnel.

I 4. Concepts de base du parallélisme

I 4 1. Modèles de parallélisme

Il existe deux notions de parallélisme: homogène et hétérogène. De ces deux concepts découlent un portage et une utilisation des processeurs du calculateur radicalement différents. Il faut noter qu'en calcul scientifique, la plupart des applications utilisent un parallélisme homogène, tandis que la programmation hétérogène est surtout présente au niveau du système d'exploitation (UNIX), dans les applications temps réel et dans les protocoles de communications [5].

I 4 1 1. Parallélisme hétérogène

Le parallélisme réside dans l'exécution en parallèle de tâches (instructions différentes). Ce type de programmation consiste à séparer les actions à réaliser, donc les fonctions du programme. Cette programmation est appelée programmation concurrente par opposition à la

programmation parallèle. Elle est présente dans les algorithmes de type maître/esclave, producteur/consommateur. Les outils de base d'une telle programmation sont le processus allégé, le *thread* d'exécution, le verrou, le sémaphore ou encore la barrière. Du fait de la nature asynchrone de cette programmation, leur utilisation peut s'avérer très complexe.

I 4 1 2. Parallélisme homogène

Dans ce type de programmation, une même tache est appliquée à des données différentes (S*ingle* P*rogram,* M*ulti* D*ata*). Le parallélisme s'applique donc sur les données et cette programmation est dite parallèle.

I 4 2. Techniques d'utilisation

Il existe trois façons d'exploiter le parallélisme d'une application donnée: parallélisme de contrôle, de flux et sur les données.

I 4 2 1. Parallélisme de contrôle

Ce parallélisme met en oeuvre l'exécution simultanée de plusieurs tâches plus ou moins indépendantes. Généralement, chaque processus est affecté à un processeur de la machine pour toute la durée de son exécution. Une application réelle, à cause de certaines dépendances, nécessite des synchronisations entre les tâches et parfois des mises à jour de données communes. Dans ce cas, les actions sur N processeurs ne s'exécuteront pas N fois plus vite.

I 4 2 2. Parallélisme de flux

Ce parallélisme reprend le concept du pipeline.

I 4 2 3. Parallélisme de données

Ce parallélisme est basé sur l'application d'une même suite d'opérations sur des données différentes. Ce type de programmation est surtout rencontré dans des applications utilisant du calcul matriciel. La même action est donc répétée sur différents éléments de la structure traitée. Ce type de programme peut se dérouler de manière synchrone, sur des architectures SIMD par exemple, ou asynchrone, sur des architectures MIMD. Sur une machine de type MIMD, le fonctionnement sera de type SPMD mais nécessitera une duplication du code sur chaque processeur et éventuellement une lecture des données par processeur.

I 4 3. Granulité

Ce paramètre est très important car de son choix découle l'efficacité d'une application parallèle sur une architecture donnée. La figure (1.10) schématise les différents niveaux de parallélisme possibles.

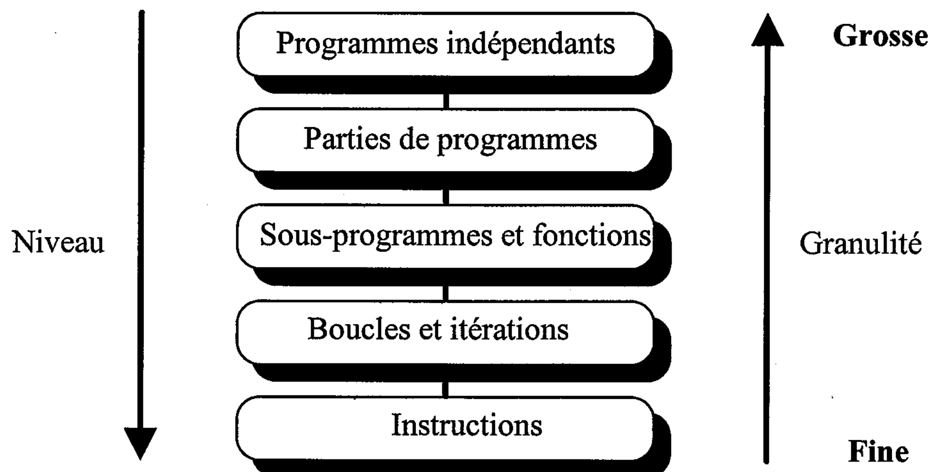


Fig. 1.10: Niveau de parallélisme et granulité.

En général, les architectures de type SIMD se prêtent bien à une granulité fine (instructions) tandis que les MIMD supportent mieux une granulité beaucoup plus grosse. En effet, les machines de type SIMD, qui fonctionnent en mode synchrone et opèrent sur des données différentes avec une seule unité de contrôle, intègrent naturellement la synchronisation dans le flux d'exécution. En revanche, du fait de son fonctionnement asynchrone, une machine de type MIMD doit gérer les synchronisations entre tâches par des mécanismes plus complexes.

Les synchronisations, quelles que soient leurs mises en oeuvre, introduisent un surcoût (*overhead*), variable suivant les machines et parfois prohibitif. De ce fait, l'adéquation granulité-architecture est un facteur prédominant quant aux performances parallèles du code. Néanmoins d'autres facteurs sont à prendre en compte, et ceci sera développé dans le chapitre relatif aux descriptions des machines cibles.

I 4 4. Dépendances des données

Ce problème est rencontré sur les calculateurs parallèles à mémoire partagée où le parallélisme est réalisé par distribution des itérations des boucles sur les différents

processeurs. Il existe une dépendance de données si une boucle modifie une case mémoire à une itération et que, lors d'une autre itération, cette case mémoire est lue ou écrite. Par contre il n'y a pas dépendance si une seule itération utilise une variable ou encore si plusieurs itérations lisent une variable sans la modifier. Les dépendances sont la cause majeure de la déficience des compilateurs à paralléliser certaines boucles. Il est donc important pour le programmeur de savoir les identifier quand il se retrouve confronté à une boucle non optimisée par le compilateur. Les dépendances plus significatives sont présentées ci-dessous.

I 4 4 1. Dépendance de flux

- Dépendance avant:

```
for(i = 0, i < n, i + +)
 {
  a(i) = a(i + 1);
 }
```

Dans cet exemple, a(i) est utilisé dans une itération et modifié dans la suivante. Il est toutefois possible de paralléliser cette boucle soit en synchronisant la boucle par affectation de la bonne valeur avant sa modification, soit en insérant un tampon et en effectuant deux boucles. La première remplit le tampon avec a et la deuxième met à jour a avec le tampon.

- Dépendance arrière:

```
for(i = 0, i < n, i + +)
 {
  a(i) = a(i - 1);
 }
```

a(i) est défini lors d'une itération et est modifié à la suivante. Ce type de dépendance est difficile à paralléliser et est très répandue. La seule manière de paralléliser cette boucle consiste à synchroniser les itérations de façon à garantir l'ordre d'exécution des écritures / lectures.

- Variable d'induction:

```
x = 0;
for(i = 0, i < n, i + +)
 {
  x = x + i;
  a(x) = b(i) + c(i);
 }
```

x est un variable d'induction. Il est possible de paralléliser cette boucle en écrivant simplement x en fonction de i (x = (i*(i+1))/2). x devient donc indépendant de sa valeur précédente et la boucle est donc parallélisable.

- Réduction:

```
for(i = 0,i < n,i + +)
 {
  x=x+a(i);
 }
```

x dépend de sa valeur à une autre itération, cette opération sur x est appelée une réduction. Cette boucle peut être parallélisée en ajoutant un tableau tampon de longueur égale au nombre de processeurs. Chaque processeur effectue une addition partielle de a sur le terme du tableau tampon correspondant à son numéro. Une autre boucle permet de réaliser la sommation totale à partir de tous les termes du tableau tampon.

I 4 4 2. Dépendance de contrôle

```
for(i = 0,i < n,i + +)
 {
  if(a(i)>0)goto etiquette;
  a(i)=a(i) + k;
 }
etiquette :
```

Il n'existe pas de moyen facile de paralléliser cette boucle car celle-ci contient un branchement de sortie.

Dans le chapitre relatif au modèle de programmation sur le CRAY C 94-98, plusieurs techniques de parallélisation et de vectorisation de boucles complexes seront présentées. En effet, le niveau de granulité choisi sur cette machine se situe au niveau des boucles. En revanche, un grain beaucoup plus gros a été adopté sur des architectures parallèles à mémoire distribuée.

I 4 5. Parallélisme au niveau des boucles

Cette granulité est la plus exploitée par les compilateurs. Dans le domaine de l'optimisation automatique, des travaux sont toujours en cours pour étendre l'efficacité des compilateurs à des boucles possédant des dépendances complexes ainsi qu'à l'analyse inter-procédurale.

Suivant la nature du compilateur, les résultats obtenus peuvent fortement varier. En règle générale, un fichier contenant la liste des boucles non optimisées est fournie par le compilateur. L'intervention du programmeur est ensuite nécessaire pour éventuellement remédier au problème. Dans la plupart des cas, quand au moins deux boucles sont imbriquées, la plus intéressante à paralléliser est la plus externe. La ou les boucles internes sont alors vectorisées.

I 4 6. Parallélisme au niveau des procédures

Il est souvent nécessaire de pouvoir exploiter un parallélisme de grain plus fort que celui relatif aux boucles. Ce parallélisme est souvent présent au niveau de l'appel de fonctions ou de sous-programmes dans des boucles. Actuellement les compilateurs automatiques n'offrant pas la possibilité d'effectuer une analyse inter-procédurale, ce travail reste donc à la charge du programmeur (outils spécifiques).

Pour illustrer cette technique, voici un exemple simple de transformation. La boucle du programme principal appelant est transmise au sous programme appelé. Au départ, la boucle du sous-programme peut être parallélisée (j) tandis que celle du programme principal ne peut l'être. Or la parallélisation de la boucle du sous-programme est pénalisante du fait du peu de travail à effectuer et du surcoût introduit par le parallélisme. La méthode consiste donc à transmettre la boucle du programme principal (i) au sous-programme et d'inverser les boucles i et j dans le sous-programme de manière à paralléliser la boucle nécessitant le plus de travail (j).

```
programme principal ( initial )
for(i = 0,i < n,i++)
 {
  appel sous-programme(A,b,i);
 }

sous-programme(A,b,i)
for(j = 0,i < 3,j++)
 {
  A(i,j)=A(i-1,j)+b;
 }
```

```
programme principal ( modifié )
appel sous-programme(A,b,i);

sous-programme(A,b,i)
for(j = 0,i < 3,j++)
 {
  for(i = 0,i < n,i++)
   {
    A(i,j)=A(i-1,j)+b;
   }
 }
```

I 5. Mesures de performances

Afin de quantifier le gain obtenu à l'aide du calcul parallèle pour traiter une application donnée, quelques notions, en plus des évaluations classiques de performances, peuvent être trouvées dans la littérature [2 - 6].

I 5 1. Loi d'Amdhal

Soit un programme informatique opérant sur une architecture parallèle, celui-ci est donc décomposé en deux parties: une partie séquentielle (Ps) et une partie parallèle (Pp). D'une manière générale, tout processus dont la vitesse change est pénalisé en terme de vitesse moyenne par les passages où sa vitesse est minimale. Dans le cas du parallélisme, l'accélération notée S (*speedup*) sur une architecture munie de Np processeurs est limitée par la loi énoncée par Gene Amdhal (1.4) [6]. Avec Pp = (1-Ps) la partie parallèle du programme.

$$S = \frac{1}{(Ps + \frac{1 - Ps}{Np})} \tag{1.4}$$

En considérant le temps d'exécution monoprocesseur (Ts) du programme et Tp le temps d'exécution sur P processeurs, la loi d'Amdhal est donnée par (1.5).

$$S = \frac{Ts}{Tp} = \frac{1}{Ps + \frac{Pp}{Np}} \tag{1.5}$$

D'après la formule (1.5), un code opérant sur une architecture munie de beaucoup de processeurs doit comporter une partie parallèle importante pour conserver une bonne accélération. De ce fait, deux types de machines sont présents sur le marché: des architectures comportant peu de processeurs très puissants et des architectures composées de beaucoup de processeurs moins puissants. Pour chaque application il existe une architecture cible optimale que la loi d'Amdhal permet en partie de prévoir.

I 5 2. Bancs d'essais (*benchmarks*)

Afin de pouvoir évaluer les performances d'un programme, il est nécessaire de définir tout d'abord quelques unités de mesures fréquemment utilisées pour ce genre de détermination.

- Le temps CPU : temps d'occupation par l'unité centrale pour traiter le programme. Les entrées/sorties et l'exécution des autres programmes sont exclues. Ce temps est composé du temps utilisateur et du temps système.

- Le nombre moyen de cycles d'horloge par instruction: ce temps dépend grandement du jeu d'instruction utilisé par le programme.

- Le MIPS: million d'instructions par secondes. Cette unité de mesure ne permet pas de comparer les performances de machines différentes. De plus, celle-ci dépend fortement du jeu d'instructions utilisé par le code.

- Le MFLOPS: cette unité de mesure peut être de deux formes différentes. Le MFLOPS crête qui correspond au nombre théorique d'opérations flottantes par seconde et le MFLOPS soutenu qui est le nombre atteint par un programme donné.

Ces différentes unités de mesures de performances sont souvent utilisées par les *benchmarks*. Ceux-ci permettent une classification des différents ordinateurs du marché. Les bancs de test peuvent être classifiés en deux grandes catégories: mesures des performances scalaires d'un code (MIPS) et mesures des performances en calcul flottant (MFLOPS). Dans la plupart des cas, les *benchmarks* sont faits de façon à indiquer des puissances de calcul atteignables par certaines applications et se rapprochant au maximum des performances idéales de la machine. Le choix du *benchmark* utilisé pour évaluer les performances d'une machine doit donc tenir compte des fonctionnalités dont l'utilisateur se servira le plus souvent.

I 6. Conclusion

Dans ce premier chapitre, les principales notions relatives au parallélisme tant au niveau matériel que logiciel ont été présentées. La programmation parallèle demandant tour à tour un haut niveau d'abstraction et une connaissance détaillée de l'architecture cible, ces principes de base devraient permettre au lecteur une compréhension plus aisée des chapitres à venir.

A l'heure actuelle, en programmation parallèle, les possibilités logicielles permettant de s'affranchir d'une bonne connaissance de l'architecture interne de la machine sont souvent limitées. De ce fait, ce type de programmation nécessite une gymnastique algorithmique souvent difficile devant permettre de trouver la meilleure façon de porter une méthode numérique sur un calculateur parallèle donné. Le choix du type de parallélisme, du modèle programmation (SPMD, MS, ...), de la bibliothèque de passage de message ou encore de la granulité sont donc à la charge du programmeur.

CHAPITRE II
FORMULATION DU PROBLEME

II 1. Introduction

La résolution d'un problème électromagnétique consiste à déterminer les champs électriques et/ou magnétiques dans une région de l'espace. Ces champs doivent satisfaire les équations de Maxwell ainsi que les conditions aux limites appropriées sur les contours du domaine d'étude. Le calcul analytique des solutions peut être quelquefois effectué pour des géométries simples. Pour la plupart des problèmes, la solution doit être approchée numériquement. Dans cette étude, la méthode des éléments finis nodaux est utilisée. Cette technique a la particularité de générer des matrices creuses et souvent symétriques définies positives. Ces caractéristiques constituent un avantage en terme de stockage et de résolution du système matriciel. Néanmoins, l'utilisation d'une discrétisation de type éléments finis (EF) nécessite un maillage volumique en trois dimensions de toutes les régions de l'espace. Pour contourner cet obstacle, et afin de traiter des problèmes ouverts de diffraction, la formulation est couplée à des conditions aux limites absorbantes (CLA), permettant ainsi une troncature du domaine d'étude au voisinage immédiat du système étudié.

La formulation présentée permet d'une part la résolution de problèmes de diffraction électromagnétique hyperfréquence en régime harmonique. Ainsi, la réponse électromagnétique d'un objet illuminé par une onde plane peut être calculée. D'autre part, la formulation légèrement modifiée permet également de calculer les champs électromagnétiques dans et aux abords de guides d'ondes ouverts et d'antennes.

Dans cette partie, les équations de Maxwell régissant les problèmes à résoudre sont d'abord décrites brièvement. Des CLA sont ensuite introduites et couplées à la formulation. Une formulation faible des équations est présentée puis discrétisatisée par EF. Les algorithmes utilisés pour l'implantation efficace de cette formulation et les limites en terme de taille des problèmes pouvant être résolus sur des calculateurs scalaires classiques sont enfin exposés.

II 2. Equations de Maxwell en régime harmonique

II 2 1. Equations

Les équations de Maxwell traduisent plusieurs phénomènes: la conservation de la charge, la loi de Faraday et les caractéristiques des milieux [9]. Sous leur forme différentielle, elles sont données par (2.1), (2.2), (2.3) et (2.4).

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} - \boldsymbol{J}_m \quad (2.1) \qquad \nabla \cdot \boldsymbol{B} = q_m \quad (2.2) \qquad \text{(loi de Faraday)}$$

$$\nabla \times \boldsymbol{H} = \frac{\partial \boldsymbol{D}}{\partial t} + \boldsymbol{J}_e \quad (2.3) \qquad \nabla \cdot \boldsymbol{D} = q_e \quad (2.4) \qquad \text{(conservation de la charge)}$$

Dans ces équations, q_e (Coulombs/m^3) est la charge électrique,
q_m qui n'a pas de signification physique est la charge magnétique,
$\boldsymbol{J}_e$ (A/m^2) est la densité de courant électrique,
$\boldsymbol{J}_m$ qui n'a pas de signification physique, est la densité de courant magnétique,
$\boldsymbol{B}$ (Weber/m^2) est l'induction magnétique,
$\boldsymbol{H}$ (A/m^2) est le champ magnétique,
$\boldsymbol{D}$ (Coulombs/m^2) est l'induction électrique et
$\boldsymbol{E}$ (volts/m) est le champ électrique.

A ces équations il convient d'ajouter les lois de comportement caractérisant les milieux dans lesquels les champs existent ainsi que les relations liées aux matériaux. Pour des milieux linéaires, sont définis:

$$\boldsymbol{D} = \varepsilon \boldsymbol{E} \quad (2.5) \qquad \text{(caractéristique des milieux diélectriques)}$$

$$\boldsymbol{B} = \mu \boldsymbol{H} \quad (2.6) \qquad \text{(caractéristique des milieux magnétiques)}$$

$$\boldsymbol{J}_e = \sigma \boldsymbol{E} \quad (2.7) \qquad \text{(caractéristique des milieux conducteurs)}$$

ε est la permittivité du milieu. On note $\varepsilon_r = \frac{\varepsilon}{\varepsilon_0}$ la permittivité relative du milieu et ε_0 la permittivité du vide. μ est la perméabilité du milieu. On note $\mu_r = \frac{\mu}{\mu_0}$ la perméabilité relative

du milieu et μ_0 la perméabilité du vide. σ est la conductivité du milieu. Pour certains problèmes, les conducteurs, considérés comme parfaits, seront caractérisés par $\sigma=\infty$ et les diélectriques parfaits par $\sigma=0$.

II 2 2. Régime harmonique

Si la variation des champs électromagnétiques est sinusoïdale (harmonique) en fonction du temps, ceux-ci, de même que l'opérateur de dérivation par rapport au temps, peuvent être exprimés à l'aide de quantités complexes à partir des transformations intégrales de Fourrier. Les équations de Maxwell (2.1), (2.2), (2.3) et (2.4) s'expriment donc comme suit:

$$\nabla\times\mathbf{E}=-\mathrm{j}\omega\mathbf{B}-\mathbf{j}_\mathrm{m} \qquad (2.8) \qquad \nabla\cdot\mathbf{B}=\mathrm{q}_\mathrm{m} \qquad (2.9)$$

$$\nabla\times\mathbf{H}=\mathrm{j}\omega\mathbf{D}+\mathbf{j}_\mathrm{e} \qquad (2.10) \qquad \nabla\cdot\mathbf{D}=\mathrm{q}_\mathrm{e} \qquad (2.11)$$

En considérant un milieu linéaire dépourvu de charge d'espace (q=0) et sans pertes électriques ($\sigma=0$), les équations de Maxwell deviennent alors:

$$\nabla\times\mathbf{E}=-j\omega\mu\mathbf{H}-\mathbf{j}_\mathrm{m} \qquad (2.12) \qquad \nabla\cdot(\mu\mathbf{H})=0 \qquad (2.13)$$

$$\nabla\times\mathbf{H}=j\omega\varepsilon\mathbf{E}+\mathbf{j}_\mathrm{e} \qquad (2.14) \qquad \nabla\cdot(\varepsilon\mathbf{E})=0 \qquad (2.15)$$

Remarques:

1. Il est possible d'étendre l'étude aux milieux avec pertes diélectriques et/ou magnétiques, ceci en introduisant une permittivité et/ou une perméabilité complexe:

$$\varepsilon=\varepsilon'-\mathrm{j}\varepsilon'' \quad \text{avec} \quad \varepsilon'=\varepsilon_r'\,\varepsilon_0 \quad \text{et} \quad \varepsilon''=\varepsilon_r''\,\varepsilon_0 \qquad (2.16)$$

$$\mu=\mu'-\mathrm{j}\mu'' \quad \text{avec} \quad \mu'=\mu_r'\,\mu_0 \quad \text{et} \quad \mu''=\mu_r''\,\mu_0 \qquad (2.17)$$

Les termes ε'' et μ'' traduisent respectivement les pertes diélectriques et magnétiques.

2. L'étude des métaux conducteurs homogènes de propriétés $(\varepsilon,\mu_0,\sigma)$ peut être effectuée en remplaçant celui-ci par un diélectrique $(\tilde{\varepsilon},\mu_0)$ dont la permittivité complexe est: $\tilde{\varepsilon}=\varepsilon+\mathrm{j}\dfrac{\sigma}{\omega}$.

3. La seule restriction rencontrée pour ce formalisme en régime harmonique concerne les matériaux comportant des pertes en régime temporel (variables en fonction du temps). Il est alors impossible de trouver une relation linéaire entre **D** et **E** ou entre **B** et **H** [11].

II 2 3. Equation vectorielle des ondes

Afin de faciliter la résolution du problème, les équations (2.8) (2.10) sont découplées, permettant ainsi d'obtenir une équation unique à une inconnue **E** ou **H** [12]. En prenant le rotationnel de (2.8) aux points où $\mathbf{J}_m = 0$, il vient:

$$\nabla\times\nabla\times\mathbf{E} = \omega^2\mu\varepsilon\mathbf{E} - j\omega\mu\mathbf{J}_e \quad (2.18)$$

qui s'écrit encore:

$$\nabla\times\nabla\times\mathbf{E} - k^2\,\mathbf{E} = -j\omega\mu\mathbf{J}_e \quad (2.19)$$

avec k = constante de propagation du champ électromagnétique dans le milieu considéré $(k^2 = \omega^2\varepsilon\mu)$. L'équation (2.19) est l'équation des ondes. L'équation en champ magnétique **H** est obtenue de la même façon à partir de (2.10) aux points où $\mathbf{J}_e = 0$:

$$\nabla\times\nabla\times\mathbf{H} - k^2\,\mathbf{H} = -j\omega\varepsilon\mathbf{J}_m \quad (2.20)$$

La relation vectorielle $\nabla^2\mathbf{X} = \nabla(\nabla\cdot\mathbf{X}) - \nabla\times\nabla\times\mathbf{X}$ appliquée à (2.19) et à (2.20) conduit à:

$$\nabla^2\mathbf{E} + k^2\,\mathbf{E} = -j\omega\mu\mathbf{J}_e \quad (2.21)$$

$$\nabla^2\mathbf{H} + k^2\,\mathbf{H} = -j\omega\varepsilon\mathbf{J}_m \quad (2.22)$$

Cette forme d'équation est appelée équation vectorielle de Helmholtz avec un terme source au second membre.

Remarque:

Les équations (2.19) et (2.20) contiennent implicitement les conditions de divergence nulle des champs, respectivement, **E** et **H**. Ceci peut être démontré en prenant la divergence de ces équations aux points où $\mathbf{J}_m$ et, respectivement, $\mathbf{J}_e$ sont nuls. En revanche, les équations (2.21) et (2.22) ne contiennent pas ces conditions et sont donc moins générales que (2.19) et (2.20).

II 2 4. Conditions aux limites

II 2 4 1. Conditions aux interfaces

Afin de permettre le traitement de problèmes complexes, l'analyse est étendue à un milieu homogène par morceaux. Les équations de Maxwell sont alors écrites dans chaque milieu et des conditions d'interfaces à la traversée d'un milieu leur sont adjointes.

Soient deux milieux définis comme sur la figure (2.1):

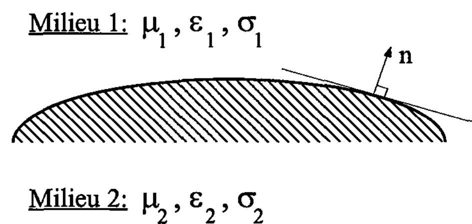


Fig. 2.1: Interface entre deux milieux de propriétés différentes.

Les conditions d'interfaces issues des équations de Maxwell, à traduire sur la grandeur dans l'équation unique sont:

$\mathbf{n}\cdot\mathbf{B}_1=\mathbf{n}\cdot\mathbf{B}_2$ (continuité de la composante normale de l'induction magnétique),(2.23)

$\mathbf{n}\times\mathbf{H}_1=\mathbf{n}\times\mathbf{H}_2$ (continuité de la composante tangentielle du champ magnétique),(2.24)

$\mathbf{n}\cdot\mathbf{D}_1=\mathbf{n}\cdot\mathbf{D}_2$ (continuité de la composante normale de l'induction électrique), (2.25)

$\mathbf{n}\cdot\mathbf{J}_{e1}=\mathbf{n}\cdot\mathbf{J}_{e2}$ (continuité de la composante normale du courant électrique), (2.26)

$\mathbf{n}\times\mathbf{E}_1=\mathbf{n}\times\mathbf{E}_2$ (continuité de la composante tangentielle du champ électrique). (2.27)

Remarques:

1. S'il existe un courant surfacique électrique ou magnétique sur une surface qui est l'interface entre deux milieux 1 et 2, alors les composantes tangentielles des champs **E** et **H** sont discontinues. Ceci implique que:

$$\mathbf{n}\times(\mathbf{E}_1-\mathbf{E}_2)=-\mathbf{J}_m \qquad (2.28)$$

$$\mathbf{n}\times(\mathbf{H}_1-\mathbf{H}_2)=-\mathbf{J}_e \quad (2.29)$$

2. Lors d'une résolution en champ électrique, à l'interface entre deux milieux dont l'un est un conducteur électrique parfait (cep), le champ électrique est nul dans ce dernier. De ce fait, l'équation (2.27) devient:

$$\mathbf{n}\times\mathbf{E}=0 \quad (2.30)$$

ce qui s'écrit encore:

$$\mathbf{n}\times\nabla\times\mathbf{H}=\mathbf{0} \quad (2.31)$$

II 2 4 2. Comportement du champ à l'infini

Lors du traitement de problèmes ouverts, une condition relative au comportement des champs à l'infini doit être forcée. En effet, la condition de régularité [10] et la donnée de la source ne garantissent pas l'unicité de la solution de l'équation d'helmholtz (2.21), (2.22) qui admet comme solutions des ondes convergentes ou divergentes.

La condition de rayonnement introduite par Sommerfeld [10] a pour effet de garantir que la fonction d'onde, solution de l'équation d'helmholtz, se comporte à l'infini comme une onde sortante. Notons que les solutions rayonnantes de l'équation d'helmholtz vérifient automatiquement la condition de régularité.

La condition de rayonnement de Sommerfeld ne garantit pas que la fonction d'onde solution de l'équation d'helmholtz soit aussi solution de l'équation vectorielle des ondes (2.19), (2.20). En effet, l'équation d'helmholtz ne contient pas implicitement la condition de divergence nulle. La condition énoncée par Silver-Müller [10] est l'équivalent de la condition de Sommerfeld pour des ondes obéissant à l'équation vectorielle des ondes.

II 3. Formulation par éléments finis

II 3 1. Introduction

Les principes relatifs à la discrétisation par EF étant généraux et forts bien connus, ceux-ci ne seront donc pas rappelés dans cette étude. Par contre, certains concepts annexes tels que le choix des variables d'états, le forçage à zéro de la divergence du champ ou encore le choix des

conditions aux limites absorbantes sont brièvement rappelés car ils sont plus ou moins spécifiques à notre formulation.

Les équations du problème sont en général reformulées avant le traitement par une méthode type EF soit en utilisant **E** ou **H** comme variables d'état, soit en utilisant des potentiels scalaires ou vecteurs comme variables d'état: **E** et **H** sont alors déterminés par dérivation numérique. Les formulations en potentiel scalaires et vecteurs éludent les problèmes de solutions parasites et de mauvais conditionnement des matrices résultant de la discrétisation de l'équation des ondes en champ **E** ou **H**. Ces approches nécessitent la définition de jauges pour assurer l'unicité de la solution. Cependant, en présence de milieux inhomogènes, la définition de jauges ne garantit plus cette dernière condition. Des conditions additionnelles doivent être imposées aux potentiels sur les interfaces des matériaux [21].

Etant donné que les approches en potentiels vecteurs requièrent toujours la définition de jauges et que les conditions d'interfaces sont formulées plus facilement lors d'une approche directe en terme de champ, la formulation adoptée est écrite directement en champ E ou H.

II 3 2. Formulation par la méthode de Galerkin

Une formulation dans la méthode des EF par un principe de résidus pondérés a été préférée à un principe variationnel car, pour les équations qui régissent notre problème, cette méthode s'avère plus souple [22]. La méthode de Galerkin est une méthode de résidus pondérés dans laquelle la fonction poids (ou test) fait partie de l'espace des solutions admissibles [23], [24].

La formulation EF est donc obtenue en appliquant la méthode de Galerkin aux équations. A partir de (2.20) et après une intégration par parties, il vient:

$$\iint_v [(\nabla \mathrm{W} \times \nabla \times \mathbf{H}) - \mathrm{W} k^2 \mathbf{H}] dv - \iint_s [\mathbf{n} \times (\mathrm{W} \nabla \times \mathbf{H})] ds = 0 \tag{2.32}$$

où W est la fonction de pondération ou fonction poids.

II 3 3. Fonction de pénalité

En deux dimensions (plan x,y), lors de la résolution de l'équation d'Helmholtz en champ électrique, ce dernier ne comporte qu'une composante invariante selon z. Par conséquent, la dérivée du champ électrique par rapport à z est nulle, entraînant ainsi $\nabla \cdot \mathbf{E} = 0$. Le champ est donc à divergence nulle (unicité de la solution).

En trois dimensions, en appliquant directement la méthode des EF nodaux à l'équation d'Helmholtz, rien ne vient forcer à zéro la divergence du champ. Par conséquent, des modes parasites numériques apparaissent dans la solution. Pour palier à ce problème, il faudrait ajouter à la formulation une deuxième équation du type $\nabla \cdot \mathbf{E} = 0$ (pour une résolution en champ électrique). Afin de garder un problème géré par une seule équation, un terme vectoriel, comportant la condition de divergence nulle du champ, est ajouté à l'équation unique:

$$\nabla(\nabla \cdot \mathbf{E}) \qquad \text{(en champ électrique)} \tag{2.33}$$

Une justification rigoureuse de cette démarche peut être trouvée dans [13].

La méthode de Galerkin est appliquée à la fonction de pénalité. A partir de (2.33) mais en champ H, et en utilisant plusieurs identités vectorielles, celle-ci devient:

$$-\iint_{v} [\nabla W \nabla \mathbf{H}]\, dv + \iint_{s} [W \mathbf{n} \nabla \mathbf{H}]\, ds \tag{2.34}$$

En sommant (2.32) et (2.34), la formulation globale en champ **H** est donnée par (2.35):

$$\begin{aligned} &\iint_{v} \left[(\nabla W \times \nabla \times \mathbf{H}) + W k^2 \mathbf{H} \right] dv - \iint_{v} [(\nabla W)(\nabla \cdot \mathbf{H})]\, dv \\ &+ \int_{Sext+Scond} W \; \mathbf{n} \cdot \nabla \cdot \mathbf{H}\, ds - \int_{Sext} [\mathbf{n} \times (W \nabla \times \mathbf{H})]\, ds = 0 \end{aligned} \tag{2.35}$$

Deux type de surfaces sont à considérer: Les surfaces externes (*Sext*) et les surfaces des conducteurs électriques parfaits (*Scond*). Le même formalisme peut être obtenu en champ **E**.

II 3 4. Conditions aux Limites Absorbantes (CLA)

II 3 4 1. Introduction

La propagation en milieu libre impose un aspect non borné du problème à traiter. L'emploi d'une méthode finie, utilisant une discrétisation volumique de la région d'analyse, nécessite la troncature du domaine d'étude. Cette frontière, physiquement fictive, doit absorber les ondes sortantes sans créer de réflexions qui, en réalité, n'existent pas. Pour respecter ce dernier point, des conditions aux limites sont imposées sur cette frontière, permettant de la rendre transparente pour les ondes sortantes. Ces Conditions aux Limites sont dites Absorbantes (CLA).

Généralement, ces CLA se présentent sous la forme d'opérateurs différentiels ou intégraux locaux ou non. Les opérateurs non locaux, qui présentent l'avantage de pouvoir placer la frontière aussi près que possible de la structure étudiée, ont l'inconvénient de détruire le caractère creux et symétrique des matrices générées par les méthodes finies. Les conditions aux limites locales, quant à elles, préservent le caractère creux des matrices EF. Cependant, ces opérateurs locaux ne sont que des approximations de la condition exacte et introduisent donc une erreur dans la solution du problème [14-19]. Dans cette partie, la CLA de type Engquist-Majda est rappelée. Celle-ci s'applique à des domaines d'étude parallélépipédiques qui s'avèrent être économiques en terme d'inconnues pour la plupart des géométries.

II 3 4 2. CLA de type EM

Cette condition aux limites utilise la transformée de Fourrier pour écrire toute onde comme une somme continue d'ondes planes. Une approximation de l'opérateur pseudo-différentiel est ensuite introduite pour annihiler certaines de ces ondes [10]. Une onde obéissant à (2.19) et se propageant dans la direction rentrante du domaine, peut s'écrire comme une somme continue sur les fréquences et sur les angles d'incidences, d'ondes planes. Engquist et Majda [14] ont montré comment annuler une telle onde. Dans l'opérateur qui effectue cette opération apparaît un radical qui le classe comme un opérateur pseudo-différentiel [10]. Afin de faciliter le couplage des CLA avec une méthode finie, l'opérateur est approché par un opérateur local: polynôme d'interpolation ou fraction rationnelle. De ce fait la CLA générée n'est plus exacte et la frontière doit être située au moins à une demie longueur d'onde de l'objet. Toutefois, il est possible d'obtenir des CLA qui absorbent l'onde dans une certaine plage d'angles d'incidences en prenant des approximations d'ordre élevé [18] [77]. La CLA vectorielle utilisée est (2.36):

$$\mathbf{n} \times \nabla \times \mathbf{H} = jk\mathbf{H}_t - \frac{j}{2k} \nabla_t^2 \mathbf{H}_t \tag{2.36}$$

II 3 4 3. Couplage des CLA avec la méthode des éléments finis

Le couplage des CLA et de la méthode des éléments finis consiste à substituer un terme de l'équation (2.35) par l'approximation (2.36). L'intégrale surfacique (Sext) sera donc remplacée. Afin d'alléger l'écriture, nous adopterons, pour les CLA, la notation suivante issue de (2.36) qui sera appliquée aux composantes tangentielles:

$$\mathbf{n}\times\nabla\times\mathbf{H}\cong g_{ABC}(\mathbf{H})=\mathrm{j}\,\mathrm{k}\,\mathbf{H}_t-\frac{\mathrm{j}}{2\mathrm{k}}\nabla_t^2\,\mathbf{H}_t \qquad (2.37)$$

II 3 5. Formulation complète en champ total

Afin d'obtenir la formulation entière en terme de champ total, il faut introduire l'équation (2.36) sous sa forme pondérée (Galerkin) dans l'équation (2.35). Pour le traitement de problèmes de diffraction, la prise en compte de l'onde incidente se fait sur la surface extérieure. Finalement, l'entière formulation, pour des problèmes de diffraction, en terme de champ total H est donnée par l'équation (2.37) [10].

$$\begin{aligned}&\iint_{v}\left[(\nabla \mathrm{W}\times\nabla\times\mathbf{H})+\mathrm{W}\,\mathrm{k}^2\,\mathbf{H}\right]dv-\iint_{v}\left[(\nabla \mathrm{W})(\nabla\cdot\mathbf{H})\right]dv+\int_{Sext+Scond}\mathrm{W}\;\;\mathbf{n}\cdot\nabla\cdot\mathbf{H}\,ds\\&-\int_{Sext}\mathrm{W}\,g_{ABC}(\mathbf{H})\,ds=\int_{Sext}\mathrm{W}\left[\,g_{ABC}(\mathbf{H}_i)-\mathbf{n}\times\nabla\times\mathbf{H}_i\,\right]ds\end{aligned} \qquad (2.37)$$

$\mathbf{H}$ est le champ total et $\mathbf{H}_i$ le champ incident.

Pour la résolution de problèmes de guides d'onde (2.38), où la source est à l'intérieur du domaine d'étude, le second membre de (2.37) est remplacé par une intégration surfacique du courant de source $\mathbf{J}_m$ [10]:

$$\begin{aligned}&\iint_{v}\left[(\nabla \mathrm{W}\times\nabla\times\mathbf{H})+\mathrm{W}\,\mathrm{k}^2\,\mathbf{H}\right]dv-\iint_{v}\left[(\nabla \mathrm{W})(\nabla\cdot\mathbf{H})\right]dv+\int_{Sext+Scond}\mathrm{W}\;\;\mathbf{n}\cdot\nabla\cdot\mathbf{H}\,ds\\&-\int_{Sext}\mathrm{W}\,g_{ABC}(\mathbf{H})\,ds=\int_{Scond}\mathrm{W}\,\mathbf{J}_m\,ds\end{aligned} \qquad (2.38)$$

Le même formalisme peut être développé pour l'équation en champ électrique.

II 4. Implantation de la formulation

Dans cette partie les différents algorithmes utilisés pour la mise en oeuvre du code séquentiel résolvant les équations décrites précédemment sont exposés. Le code séquentiel se décompose comme suit:

II 4 1. Préparation de la structure de la matrice

Cette opération, préalable à un assemblage par contributions élémentaires, consiste à initialiser les tableaux nécessaires au stockage de la matrice EF. Le type de stockage adopté est dit MORSE: afin de réaliser une économie en terme d'espace mémoire, seuls les termes non nuls sont stockés. La figure (2.3) illustre cette technique pour le stockage d'une matrice 4x4 quelconque. Le vecteur noté 'Terme' contient la valeur de tous les termes non nuls de la matrice, 'Rang' indique le numéro de colonne de chaque terme et 'N_p_colonne' le rang dans 'Terme' du premier terme de chaque ligne.

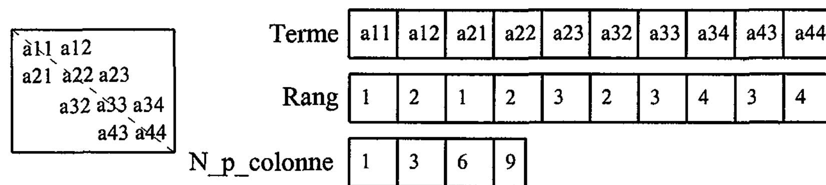


Fig. 2.3: Stockage MORSE pour une matrice 4x4 quelconque.

La préparation de la structure de la matrice EF consiste en la construction de 'Rang' et 'N_p_colonne' et en l'initialisation de 'Terme'. Cette opération est aussi appelée assemblage symbolique. Si la matrice est symétrique, il est possible de ne stocker que la partie inférieure de la matrice et ainsi de reduire pratiquement de moitié l'espace mémoire requis.

II 4 2. Assemblage

L'assemblage de la matrice EF est réalisé par contributions élémentaires. Les matrices relatives aux éléments finis sont assemblées tour à tour, les valeurs ainsi calculées sont ensuite disposées dans la matrice globale (remplissage du vecteur 'Terme') dont la structure a été établie à l'étape précédente. Les termes issus d'une matrice élémentaire peuvent être stockés n'importe où dans cette structure. Si l'élément assemblé contient un élément surfacique se trouvant sur la frontière extérieure, les CLA sont adjointes; si cet élément surfacique appartient à un cep, alors des CL appropriées doivent être forcées. La position des termes de chaque matrice élémentaire dans la structure de la matrice principale dépend de la numérotation issue du mailleur. Un algorithme de renumérotation de type 'Cuthill-Mckee' est utilisé afin de rapprocher les termes non nuls de la diagonale.

II 4 3. Traitement des symétries

La méthode utilisée consiste à imposer les conditions aux limites sur les surfaces se trouvant sur un plan de symétrie par une modification globale de la matrice EF après assemblage. Sur un plan de symétrie, lors d'une résolution en champ magnétique, H doit satisfaire l'équation (2.46) et l'équation (2.47) sur un plan d'antisymétrie.

$$\mathbf{n}\cdot\mathbf{H}=0 \qquad (2.46) \qquad\qquad \mathbf{n}\wedge\mathbf{H}=0 \qquad (2.47)$$

Les normales aux surfaces extérieures du domaine d'étude n'ont qu'une seule composante du fait que celui-ci est parallélépipédique. Par exemple, sur un plan de symétrie, si **n** = **x**, de (2.46) il découle $\mathbf{H}_x = 0$. La ligne correspondante est donc forcée à 0 et le terme diagonal à 1. Pour conserver une matrice EF de structure symétrique, la colonne de même rang que la ligne correspondant à $\mathbf{H}_x$ est aussi forcée à 0. Sur un plan d'antisymétrie, pour **n** = **x,** à partir de (2.47) il vient $\mathbf{H}_y = 0$ et $\mathbf{H}_z = 0$. La même démarche est suivie mais pour les deux lignes et colonnes correspondant à $\mathbf{H}_y$ et $\mathbf{H}_z$.

II 4 4. Conditions aux limites sur les conducteurs électriques parfaits

Les conditions aux limites sur les cep sont implicites lors d'une résolution en champ H et explicites en champ E. Etant donné que les normales aux surfaces des cep sont quelconques, les modifications à apporter à la matrice s'avèrent plus complexes mais la démarche utilisée reste la même afin de conserver la caractère symétrique de la matrice EF. Soit la condition aux limites à imposer donnée par (2.48), plusieurs cas sont à envisager suivant le nombre de composantes nulles du vecteur normal n.

$$\mathbf{n}\wedge\mathbf{E}=0 \qquad (2.48)$$

Si $\mathbf{n}=\begin{pmatrix}\mathbf{n}_x\\ \mathbf{n}_y\\ \mathbf{n}_z\end{pmatrix}$ il faut imposer sur les lignes correspondantes $\begin{cases}\mathbf{E}_x:\mathbf{n}_y\mathbf{E}_z-\mathbf{n}_z\mathbf{E}_y=0\\ \mathbf{E}_y:\mathbf{n}_z\mathbf{E}_x-\mathbf{n}_x\mathbf{E}_z=0\\ \mathbf{E}_z:\mathbf{n}_x\mathbf{E}_y-\mathbf{n}_y\mathbf{E}_x=0\end{cases}$ (2.49)

Les autres cas où une ou plusieurs des composantes de la normale sont nulles se déduisent de (2.49). Trois cas sont à envisager (2.50):

$$\begin{bmatrix}\mathbf{n}_z & 0 & -\mathbf{n}_x\\ -\mathbf{n}_y & \mathbf{n}_x & 0\\ 0 & -\mathbf{n}_z & \mathbf{n}_y\end{bmatrix}\times\begin{bmatrix}\mathbf{E}_x\\ \mathbf{E}_y\\ \mathbf{E}_z\end{bmatrix}=\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{matrix}\text{cas 1}\\ \text{cas 2}\\ \text{cas 2}\end{matrix} \qquad (2.50)$$

Dans ce système, 2 cas différents de modifications sont à prévoir. Le troisième cas est rencontré quand une ou deux des composantes de la normale sont nulles, il convient alors de fixer une des composantes du champ E à 0. Le même algorithme que précédemment est utilisé.

Cas 1: soit un système matriciel dont la ligne (2) doit être modifiée selon le cas 1 ($a=n_y$ et $b=-n_x$), la figure (2.4) illustre cet exemple.

$$
\begin{matrix} (1) \\ (2) \\ (3) \\ (4) \\ (5) \\ (6) \end{matrix}
\begin{bmatrix}
A11 & 0 & A13 & A14 & A15 & A16 \\
0 & a & 0 & b & 0 & 0 \\
A31 & 0 & A33 & A34 & A35 & A36 \\
A41 & b & A43 & A44 & A45 & A46 \\
A51 & 0 & A53 & A54 & A55 & A56 \\
A61 & 0 & A63 & A64 & A65 & A66
\end{bmatrix}
\times
\begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \\ X5 \\ X6 \end{bmatrix}
=
\begin{bmatrix} S1 \\ 0 \\ S3 \\ S4 \\ S5 \\ S6 \end{bmatrix}
$$

Fig. 2.4: Matrice symétrique modifiée et erronée pour le cas 1.

La matrice ainsi modifiée respecte la condition aux limites et garde une structure symétrique. En revanche, la modification de la ligne (2) et de la colonne (2) a entraîné une erreur dans les équations relatives aux autres lignes à cause de l'existence du terme b. De ce fait, il faut de nouveau apporter une modification globale à la matrice. A partir de l'équation de la ligne 2, il vient: $X2 = -(b/a)\, X4$.

En introduisant cette égalité dans les équations relatives aux autres lignes, cela donne:

Ligne (1) $$A11 \cdot X1 + A13 \cdot X3 + \left(A14 - \frac{b}{a} A12\right) \cdot X4 + A15 \cdot X5 + A16 \cdot X6 = S1 \quad (2.51)$$

Ligne (3) $$A31 \cdot X1 + A33 \cdot X3 + \left(A34 - \frac{b}{a} A32\right) \cdot X4 + A35 \cdot X5 + A36 \cdot X6 = S3 \quad (2.52)$$

Ligne (4) $$\left(A41 - \frac{b}{a} A12\right) \cdot X1 + b \cdot X2 + \left(A43 - \frac{b}{a} A32\right) \cdot X3 + \left(A44 + \frac{b^2}{a} - \frac{2b}{a} A42 + \frac{b^2}{a^2} A22\right) \cdot X4 + \left(A45 - \frac{b}{a} A52\right) \cdot X5 + \left(A46 - \frac{b}{a} A62\right) \cdot X6 = S4 - \frac{b}{a} S2 \quad (2.53)$$

Ligne (5) $$A51 \cdot X1 + A53 \cdot X3 + \left(A54 - \frac{b}{a} A52\right) \cdot X4 + A55 \cdot X5 + A56 \cdot X6 = S5 \quad (2.54)$$

Ligne (6) $$A61 \cdot X1 + A63 \cdot X3 + (A64 - \frac{b}{a} A62) \cdot X4 + A65 \cdot X5 + A66 \cdot X6 = S6 \qquad (2.55)$$

En redistribuant les termes des équations (2.51) à (2.55), il apparaît que les équations des lignes (1), (3), (4), (5) et (6) ont retrouvé leur forme initiale. La figure (2.5) montre le système matriciel symétrique incluant les conditions aux limites sur la ligne (2).

$$\begin{matrix}(1)\\(2)\\(3)\\(4)\\(5)\\(6)\end{matrix}\begin{bmatrix} A11 & 0 & A13 & A14-(b/a)A12 & A15 & A16 \\ 0 & a & 0 & b & 0 & 0 \\ A31 & 0 & A33 & A34-(b/a)A12 & A35 & A36 \\ A41-(b/a)A12 & b & A43-(b/a)A12 & A44+(*) & A45-(b/a)A12 & A46-(b/a)A12 \\ A51 & 0 & A53 & A54-(b/a)A12 & A55 & A56 \\ A61 & 0 & A63 & A64-(b/a)A12 & A65 & A66 \end{bmatrix} \times \begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \\ X5 \\ X6 \end{bmatrix} = \begin{bmatrix} S1 \\ 0 \\ S3 \\ S4-(b/a)S2 \\ S5 \\ S6 \end{bmatrix}$$

$$* = \frac{b^2}{a} - \frac{2b}{a} A42 \ \frac{b^2}{a^2} A22$$

Fig. 2.5: Matrice symétrique modifiée pour le cas 1.

<u>Cas 2</u>: soit un système matriciel dont la ligne (4) doit être modifiée selon le cas 2. En appliquant la même méthode que précédemment, la matrice symétrique totalement modifiée et incluant la condition aux limites sur sa ligne (4) est donnée par la figure (2.6).

$$\begin{matrix}(1)\\(2)\\(3)\\(4)\\(5)\\(6)\end{matrix}\begin{bmatrix} A11 & A12 & A13-(b/a)A41 & 0 & A15 & A16 \\ A21 & A22 & A23-(b/a)A42 & 0 & A25 & A26 \\ A31-(b/a)A41 & A32-(b/a)A42 & A33+(*) & b & A35-(b/a)A45 & A36-(b/a)A46 \\ 0 & 0 & b & a & 0 & 0 \\ A51 & A52 & A53-(b/a)A45 & 0 & A55 & A56 \\ A61 & A62 & A63-(b/a)A46 & 0 & A65 & A66 \end{bmatrix} \times \begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \\ X5 \\ X6 \end{bmatrix} = \begin{bmatrix} S1 \\ S2 \\ S3-(b/a)S4 \\ 0 \\ S5 \\ S6 \end{bmatrix}$$

$$* = \frac{b^2}{a} - \frac{2b}{a} A43 \ \frac{b^2}{a^2} A44$$

Fig. 2.6: Matrice symétrique modifiée pour le cas 2.

II 4 5. Symétrisation du système matriciel

Le système matriciel est symétrisé pour permettre l'utilisation d'un solveur symétrique, ce qui permet de réduire l'espace mémoire utilisé. Ceci est d'autant plus vrai si un préconditionnement de type Cholesky incomplet est utilisé. En effet, l'espace mémoire alors nécessaire est de une fois la taille de la matrice EF avant sa symétrisation si seulement la partie inférieure de la matrice de préconditionnement est construite ou de 1.5 fois la taille de la matrice EF avant sa symétrisation si la matrice de préconditionnement est construite, dans son intégralité. Ce souci d'économie en terme d'espace mémoire est prédominant dans cette étude car notre but est de traiter des problèmes complexes et par là même gros en terme d'inconnues.

La dissymétrie de la matrice EF étant faible, sa symétrisation est effectuée par l'addition de celle-ci avec sa matrice transposée. Cette méthode approximative, beaucoup moins coûteuse en nombre d'opérations qu'une multiplication par la matrice transposée, O (N) opérations au lieu de $O(N)^3$, donne néanmoins de bons résultats. De plus, la matrice EF garde la même structure. Le système de départ A x = b est transformé en $1/2\ (A + A^t)\ x = b$. La nouvelle matrice EF $(A + A^t)$ est symétrique définie positive alors que la matrice initiale A était définie positive. Quand cela est possible et dans un souci de simplification, la matrice EF est stockée sous la forme de deux matrices: partie inférieure et partie supérieure (colonnes → lignes). Donc, l'addition peut se faire à l'aide de seulement deux boucles imbriquées: la boucle externe opère sur les lignes des deux matrices et l'interne sur les colonnes.

II 4 6. Résolution du système matriciel

La matrice EF étant creuse et symétrique, une méthode itérative est utilisée pour résoudre le système matriciel. Le Gradient Conjugué (GC) muni d'un préconditionnement diagonal ou de Cholesky incomplet a ainsi été implanté [34] [annexe 3]. D'une manière générale, le préconditionnement consiste en une amélioration de l'algorithme au sens du nombre d'itérations. Soit une matrice M connue et facilement inversible, si celle-ci a un comportement proche de A, la matrice $M^{-1}.A$ aura un comportement proche de la matrice unité. La résolution du système $M^{-1}.A\ x = M^{-1}.b$ par une méthode itérative se fera en un nombre d'itérations inférieur à n. Il est prouvé que, par exemple, le GC converge en n itérations, n étant le nombre de lignes [57], [58].

Pour un préconditionnement diagonal, la matrice de préconditionnement est élaborée par inversion des termes diagonaux de la matrice EF: ceci entraîne un mauvais préconditionnement mais se révèle en revanche rapide et peu coûteux en espace mémoire. Le préconditionnement est réalisé par une multiplication vecteur-vecteur. Une multiplication matrice-vecteur est, entre autres, nécessaire à chaque itération pour le calcul du nouveau vecteur résidu.

L'utilisation du préconditionnement de Cholesky incomplet nécessite aussi une multiplication matrice-vecteur ainsi que deux autres étapes: la construction de la matrice de préconditionnement et la descente-montée pour la résolution du système. La plupart des techniques de préconditionnement sont basées sur la préparation d'une matrice dite de 'préconditionnement' élaborée à partir d'une méthode directe. En fait, l'utilisation d'une méthode directe a pour effet de remplir la matrice (ou la bande) et ceci entrainela perte du caractère creux de la matrice: les termes de remplissage sont rejetés, c'est à dire que la matrice de préconditionnement garde la même structure que la matrice EF. Avec une méthode de type Cholesky qui dans son utilisation complète a pour effet de remplir seulement la bande de la matrice, le préconditionnement sera d'autant plus efficace que l'algorithme de renumérotation aura rapproché les termes non nuls de la diagonale [76]. La matrice de préconditionnement à donc un comportement proche de A^{-1} ce qui a pour effet d'augmenter fortement la convergence du solveur itératif [34], [58]. Le problème majeur rencontré pour l'implantation de telles techniques réside dans l'accès aux termes de la matrice par colonnes et/ou dans la connaissance d'autres lignes [34], [58].

II 5. Calcul parallèle

II 5 1. Contraintes dues à la formulation

La formulation présentée précédemment entraîne plusieurs contraintes:

1) 3 inconnues complexes par noeuds,

2) 10 noeuds par longueur d'onde pour obtenir une bonne précision,

3) au moins 0.5 longueur d'onde entre l'objet et la frontière extérieure,

4) domaine d'étude parallélépipédique,

5) matrice EF non symétrique à cause des CLA,

6) CL sur les cep implicites en champ H mais explicite en champ E.

Ces remarques imposent des restrictions quant à la taille maximale des problèmes qui peuvent être traités ainsi qu'au temps CPU nécessaire sur des calculateurs scalaires classiques (HP 9000/712).

A titre d'exemple, prenons un domaine d'étude cubique de 3 longueurs d'ondes de côté. En vertu de la remarque 2, cette géométrie conduit à un problème maillé avec 27000 noeuds. L'utilisation d'hexaèdres du premier ordre, qui génèrent une largeur de bande d'à peu près 280 (valeur comprenant les 3 coordonnées), conduit à 7560000 inconnues complexes. Un complexe double précision étant stocké sur 16 octets, 121 Mo de mémoire sont nécessaires rien que pour stocker la matrice EF avant sa symmétrisation. Notons que cette géométrie ne fait que 30 cm de côté à 3 Ghz. Par conséquent, il paraît tout à fait impossible de pouvoir modéliser certains dispositifs réalistes dans cette gamme de fréquence (diffraction d'une onde plane par un aéronef, ...). Cet exemple montre la nécessité d'utiliser des calculateurs parallèles qui sont à l'heure actuelle les seuls ordinateurs disposant d'un espace mémoire suffisant.

II 5 2. Problème test

Afin de pouvoir calculer l'accélération sur plusieurs processeurs pour un problème donné, il est indispensable de pouvoir traiter celui-ci sur un processeur (chapitre relatif à la ferme de stations). De plus, l'utilisation d'outils d'analyses de performances entraîne souvent un surcoût qui devient prohibitif quand la taille des problèmes augmente. La comparaison des performances parallèles du code sur les différentes machines a conditionné le choix du problème test. Ce choix a été effectué de sorte que le problème puisse être calculé sur un processeur de la ferme de stations, ce qui est en terme d'espace mémoire le cas le plus défavorable.

Le problème test est donc un cylindre parfaitement conducteur maillé avec 10000 noeuds (hexaèdres du premier ordre) et comportant une symétrie. La figure (2.7) montre le module du champ H sur les différentes frontières et dans un plan de coupe. La fréquence de l'onde incidente est de 3 Ghz et la longueur du cylindre est égale à 10 cm et son rayon à 1 cm.

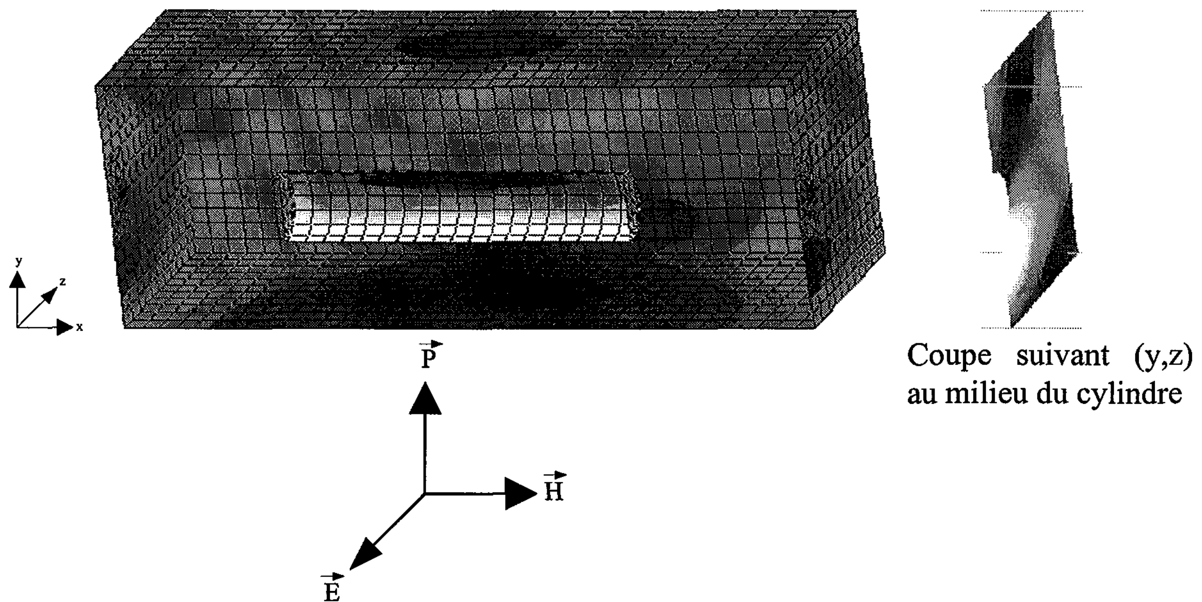


Fig. 2.7: Module du champ H.

Notre formulation couplée à des tétraèdres du premier et deuxième ordre a la particularité de générer des matrices mal conditionnées car la convergence des différents solveurs est alors beaucoup plus longue. Pour l'exemple ci-dessus, maillé avec différents types d'éléments mais avec un nombre de noeuds à peu près constant, le tableau (2.1) donne le nombre d'itérations et le temps de calcul pour les différents solveurs (Gradient Conjugué et QMR munis du préconditionnement diagonal).

	GC et precond. diagonal		QMR et précond. diagonal	
	Itérations	Temps CPU	Itérations	Temps CPU
Tétraèdres ordre 2	481	2668 s	889	5810 s
Tétraèdres ordre 1	713	2976 s	1260	5654 s
Hexaèdres ordre 1	204	2101 s	302	3053 s

Tab. 2.1: Convergence des solveurs en fonction du type d'éléments.

Pour des raisons d'économie de temps CPU sur les différentes machines, le problème test sera maillé avec des hexaèdres du premier ordre.

CHAPITRE III
DESCRIPTION DES MACHINES CIBLES

III 1. Introduction

Afin d'effectuer une implantation efficace de notre formulation sur les différentes machines cibles qui sont à notre disposition, une bonne connaissance de leurs architectures et des outils logiciels dont elles sont équipées est primordiale. Dans ce chapitre sont présentés le CRAY C98 (IDRIS), la ferme de stations (ECL), le CRAY T3E (IDRIS) ainsi que leurs modèles de programmation. L'Institut Des Ressources en Informatique Scientifique (IDRIS) est une unité du CNRS et a en sa possession plusieurs super-calculateurs. Elle permet aux industriels comme aux chercheurs d'avoir accès à de telles ressources informatiques assorties d'une aide personnalisée.

III 2. CRAY C98

III 2 1. Configuration matérielle

Ce super-calculateur parallélo-vectoriel est de type MIMD à mémoire partagée. Tous les processeurs ont accès à une mémoire centrale. Il est parmi les plus puissants de sa catégorie. Ce type de machine semble, actuellement, en déclin à cause des limitations de la bande passante mémoire qui ne permet pas de connecter plus de 8 processeurs à la même mémoire. Ses principales caractéristiques sont les suivantes: 8 processeurs vectoriels, 8 registres vectoriels de 128 Mots, 1 Gflops/s de puissance théorique par processeur, 4 Go de mémoire RAM, 120 Go de *swap* et une bande passante mémoire de 122.9 Go/s (annexe 2).

III 2 2. Modèle de programmation

III 2 2 1. vectorisation [41]

Comme il a été dit dans le premier chapitre, une instruction d'un code scalaire est relative à

une ou deux données, alors que pour un code vectoriel, une instruction traite un groupe de données (vecteurs). La vectorisation est en principe la première optimisation à réaliser.

En FORTRAN, les boucles DO, DOWHILE se vectorisent. Si plusieurs sont imbriquées, la plus interne est candidate à la vectorisation. La boucle vectorisée doit contenir au moins une référence à un tableau/vecteur.

Un code vectoriel n'introduit pas d'erreur par rapport à un code scalaire s'il n'y a pas de conflits de dépendances. En règle générale, il y a un conflit de dépendance entre deux opérations E/L (écriture/lecture) quand la dépendance, c'est à dire l'ordre, n'est pas respectée dans la version vectorielle. Le même problème est rencontré lors de la parallélisation. Les conflits de dépendances étant la cause principale d'inhibition de la vectorisation, la partie qui suit présente quelques cas de figures où ceux-ci peuvent être levés soit par le pré-processeur FPP, soit, dans des cas plus complexes, par le programmeur.

- Elimination d'un conflit de dépendances par restructuration de la boucle:

<u>Version avec un conflit</u>

```
REAL A(N), B(N), C(N)
DO I = 2, N-1
        A(I) = B(I-1) + C(I)   conflit sur B
        B(I) = B(I+1) -2
ENDDO
```

<u>Version avec la boucle restructurée</u>

```
REAL A(N), B(N), C(N)
DO I = 2, N-1
        B(I) = B(I+1) -2
        A(I) = B(I-1) + C(I)
ENDDO
```

- Elimination d'une dépendance ambiguë:

<u>Version ne pouvant être vectorisée</u>

```
REAL A(100), B(100), C(100)
COMMON // J
DO I = 2, 100
        A(I) = B(I)
        C(I) = A(I-J)   conflit si J<0
ENDDO
```

<u>Version pouvant être vectorisée</u>

```
  REAL A(100), B(100), C(100)
  COMMON // J
CDIR$ IVDEP
  DO I = 2, 100
        A(I) = B(I)
        C(I) = A(I-J)
  ENDDO
```

L'introduction de la directive IVDEP (*Ignoring* V*ector* DEP*endencies*) permet de palier à ce problème si le programmeur est sûr que $J \geq 0$.

- Elimination d'une dépendance par inversion de boucles:

Une inversion de boucle permet d'augmenter la longueur des vecteurs servant à la vectorisation. Ce procédé peut créer un conflit de dépendance ou des conflits mémoire. Cette technique sert aussi à l'élimination de dépendances: l'exemple de deux boucles opérant sur un tableau multidimensionnel et contenant une récurrence à une dimension est donné à titre d'exemple.

Version non vectorisable

```
SOUBROUTINE ORIGINAL (N)
REAL A(500, 500)
DO I = 2, N
      DO J = 2, N
            A(I, J) = A(I, J-1) * 2   récurrence
      ENDDO
ENDDO
```

Version vectorisable

```
SOUBROUTINE MODIFIED (N)
REAL A(500, 500)
DO J = 2, N
      DO I = 2, N
            A(I, J) = A(I, J-1) * 2
      ENDDO
ENDDO
```

- Elimination d'une dépendance par scission de boucle:

Cette technique consiste à scinder la boucle en une partie vectorisable et une partie scalaire.

Version non vectorisable

```
SOUBROUTINE ORIGINAL (N)
REAL A(1000) B(1000), C(1000)
DO 10 I = 2, N
      A(I) = A(I-1) * C(I)   récurrence
      B(I) = A(I) * C(I-1) ** 2
      C(I-1) = SQRT ( A(I) ) - B(I)
10 CONTINUE
END
```

Partie vectorisable en caractères gras

```
SOUBROUTINE MODIFIED (N)
REAL A(1000) B(1000), C(1000)
DO 10 I = 2, N-1
      A(I) = A(I-1) * C(I)
10 CONTINUE
DO 20 I = 2, N
      B(I) = A(I) * C(I-1) ** 2
      C(I-1) = SQRT ( A(I) ) - B(I)
20 CONTINUE
END
```

Afin de connaître les boucles non optimisées par le pré-processeur FPP, il convient d'observer le fichier de sortie (nom.m), et d'essayer d'éliminer les conflits par une intervention manuelle. D'autres problèmes peuvent inhiber la vectorisation, notamment l'appel à un sous-programme (CALL) ou à une fonction non vectorielle, les instructions d'E/S, les branchements arrières, les GOTO assignés et calculés ainsi que certaines instructions conditionnelles (IF arithmétique à 3 branches). Dans ces cas-là, seule une restructuration du code à la charge du programmeur est à même de résoudre ces difficultés.

D'autres types d'optimisations sont réalisées par FPP, notamment l'*inlining*, le déroulement de boucles (partiel ou non), l'utilisation de bibliothèques CRAY, la concaténation de boucles, le linéarisation de boucles, ... Ces techniques ne sont pas présentées dans cette étude car FPP réalise très bien ces opérations et, par conséquent, ces optimisations ne requièrent pas l'intervention du programmeur.

III 2 2 2. Parallélisation [40]

Le CRAY C98 est de type MIMD à mémoire partagée et supporte plusieurs types de parallélisme de granulité forte ou moyenne. L'optimisation parallèle d'un code peut se faire manuellement ou automatiquement. Dans la pratique, les meilleures performances sont obtenues en combinant les deux méthodes. La parallélisation d'une application augmente forcément son temps CPU ainsi que sa place en mémoire. En principe, seules les parties du programme les plus consommatrices en temps CPU sont parallélisées. Il faut veiller à ce que le surcoût induit par le parallélisme (*overhead*) ne soit pas supérieur au gain apporté par ailleurs. Dans le cas du parallélisme à granulité forte, il est préférable de limiter le nombre d'activations de tâches. Lors de la parallélisation de boucles, il faut veiller à ne pas réduire la longueur des vecteurs, car, alors, la parallélisation casse la vectorisation. Enfin, l'équilibrage de charge (*load balancing*) est une opération fondamentale en calcul parallèle. Il peut être réalisé de manière statique en découpant le travail à effectuer en tâches de même volume, ou de manière dynamique (automatique, par verrous, par affectation à la prochaine tâche libre ...).

- Le *macrotasking*

Cette technique permet d'exploiter un parallélisme de granulité forte (exécution parallèle de sous programmes). La plus petite granulité exploitable est de l'ordre de 1/100 ^{ème} de seconde. En fait, l'utilisation du *macrotasking* permet la gestion de tâches, de barrières, d'événements

et de verrous à l'aide de la bibliothèque CRAY *macrotasking*. La détection du parallélisme ainsi que l'examen de la portée des variables sont à la charge du programmeur. Cette technique rendant le programme non portable, elle n'est plus préconisée par CRAY, et elle n'a donc pas été utilisée dans cette étude.

- Le *microtasking* et l'*autotasking*

Le *microtasking* permet l'exploitation d'un parallélisme de granulité moyenne (1/100000 ^{ème} de seconde). Cette technique contraint le programmeur à préciser des directives au niveau des boucles à paralléliser. En fait, le *microtasking* n'est quasiment plus employé que via la parallélisation automatique (*autotasking*). Cette dernière méthode correspond au *microtasking* automatique. Le pré-processeur FPP détecte le parallélisme en déterminant les régions parallèles ainsi que la portée des variables dans celles-ci, et insère des directives qui seront interprétées lors de la compilation. Une fois le travail de FPP effectué, le processeur FMP interprète les directives introduites et transforme les régions parallèles en code parallèle. Cette méthodologie permet d'aller plus loin dans l'analyse de la portée des variables, notamment à l'aide du logiciel ATESCOPE. En effet, l'ensemble des variables que peut référencer une tâche se divise en plusieurs sous ensembles: privée, partagée avec au moins une autre tâche, ou encore de contrôle. L'intervention du programmeur est donc souhaitable sur les parties du code n'ayant pas été parallélisées (*microtasking*). Une portée doit être attribuée aux variables repérées *unknown* par ATSCOPE. Cette démarche peut éventuellement entraîner des résultats erronés. Les directives introduites par FPP (@) ou manuellement par le programmeur ($) sont principalement:

CMIC$ DO ALL [paramètres] [distribution du travail] est une directive de parallélisation de la boucle, avec [paramètres] qui fixent principalement la portée des variables de la boucle et [distribution du travail] qui définit la distribution des itérations de la boucle.

CMIC$ PARALLEL et END PARALLEL définissent respectivement le début et la fin d'une région parallèle dans laquelle chaque processeur exécute un block du code.

CMIC$ DO PARALLEL et END DO indiquent que la boucle sera exécutée en parallèle. Les paramètres et la distribution du travail sont les mêmes que pour DO ALL.

CMIC$ GUARD et END GUARD délimitent une région critique, à l'intérieur d'une région parallèle, dans laquelle le code est exécuté par un seul processeur à la fois.

CMIC$ WAIT et SEND délimitent une portion de code qui doit être exécutée séquentiellement à l'intérieur d'une boucle parallélisée.

CMIC$ TASKCOMMON entraîne le changement par FMP du block COMMON en blocks locaux à chaque tâche. Chaque processeur possède donc sa propre copie des variables déclarées dans ce COMMON.

III 2 3. Analyse des performances et optimisation

L'optimisation parallèle d'une application se fait grâce aux outils d'analyse de performances. Ceux-ci permettant de connaître les parties non optimisées d'un code, l'utilisateur peut alors concentrer ses efforts sur ces zones.

III 2 3 1. ATEXPERT

Ce logiciel analyse la parallélisation basée sur des directives. Il permet d'évaluer l'accélération en mode dédié, de repérer les parties du code où l'exécution passe le plus de temps, les zones non parallèles (zones séquentielles) ainsi que le surcoût introduit par le parallélisme (*overhead*). Son utilisation impliquant un fonctionnement monoprocesseur, les performances sont évaluées sur 2, 3, ... et 8 processeurs. Cette méthode est très pénalisante en terme de temps CPU consommé car le parallélisme réel de l'application est nul, donc aucun bonus n'est attribué à l'application. En effet, le fonctionnement des chaînes de *batch* à l'IDRIS permet l'économie de temps CPU pour une application bien parallélisée sous la forme de bonus venant se retrancher au temps CPU réellement consommé. C'est pour cette raison que toutes les mesures de performances sont effectuées sur le problème test de 10000 noeuds (hexaèdres du premier ordre).

Le surcoût dû au parallélisme peut être de plusieurs types:

- *Iteration Overhead* (IO), c'est le temps requis pour le démarrage d'une nouvelle itération depuis le test d'occurrence de la dernière. Ce temps tient aussi compte des conflits mémoire et des contentions sur les registres partagés.

- *Load Imbalancing* (LI): c'est le rapport entre les temps CPU d'exécution d'une distribution parfaite de tâches et la distribution réelle en exploitation.

- *Begin Parallel*, *Slave Arrival*, et *Wait Slave*: sont relatifs au temps introduit par l'initialisation des régions parallèles (passages des données aux esclaves, duplication des données, synchronisations, ...).

Si l'*IO* est fort, il faut augmenter la taille du grain, tandis qu'il faut la diminuer si l'*LI* est fort. La détermination du meilleur compromis est appelée le pavage. Le plus souvent, la solution réside dans le mode GUIDED (128) pour les boucles internes et CHUNKSIZE (n) sur les boucles externes, avec n à régler suivant l'application.

La minimisation du temps séquentiel requiert un effort sur la vectorisation. L'utilisation de PROFVIEW, qui est un outil d'aide à la vectorisation (*microtasking*), permet de repérer les boucles les plus consommatrices en terme de temps CPU. Néanmoins il est nécessaire de limiter les E/S formatées, qui sont beaucoup moins rapides, et éventuellement de restructurer les boucles (utiliser les techniques citées précédemment).

III 2 3 2. *Job Accounting* (JA)

Ce logiciel donne l'accélération effectivement obtenue lors de l'exécution (temps *user* total / temps *user* connecté). Sur une machine en exploitation comme le CRAY C98 de l'IDRIS, ces résultats ne sont pas reproductibles car elle fonctionne en temps partagé. Le temps système indiqué dans le fichier issu d'une soumission avec JA indique l'*overhead* dû au parallélisme ainsi que le temps consacré aux E/S. Le rapport (temps système + temps *user*) / temps passé représente l'accélération en mode dédié.

III 2 3 3. Hardware Parallel Monitoring

Cet outil permet l'analyse des paramètres hardware au niveau du programme principal. Par conséquent, il donne le nombre de Mflops, le nombre d'opérations scalaires et vectorielles, le temps perdu à cause des conflits mémoire, ... Il permet d'estimer le rapport puissance développée par le code / puissance théorique de la machine. Son utilisation doit être spécifiée lors de la soumission.

Il existe bien d'autres outils développés par CRAY permettant une analyse, et par là même une optimisation de tous les paramètres du code: temps d'exécution au niveau des instructions et sous-programmes (PROFVIEW), analyse des paramètres *hardware* au niveau des sous-routines (PERFVIEW), analyse des performances au niveau *hardware* des lignes de codes et des boucles (JUMPVIEW), ... Cette multitude d'outils permet au programmeur d'optimiser son code à tous les niveaux.

III 3. Ferme de stations

III 3 1. Aspect matériel

Ce calculateur, de type MIMD à mémoire distribuée, est la propriété de l'Ecole Centrale de Lyon. Il fait partie d'une gamme de matériel permettant de disposer d'une puissance de calcul considérable pour un prix raisonnable. Pour cette raison, ce type de calculateur est maintenant assez répandu. Par contre, son modèle de programmation impose souvent une restructuration complète des algorithmes utilisés par les codes séquentiels.

Ces principales caractéristiques sont les suivantes: 10 stations de travail DEC ALPHA 300 X cadencées à 175 Mhz, 64 Mflops/s (LINPACK) par station de travail, 64 Mo de RAM et 1 Go de *swap* par station de travail et réseau de communication en anneau (fibre optique: FDDI) (fig. 3.1).

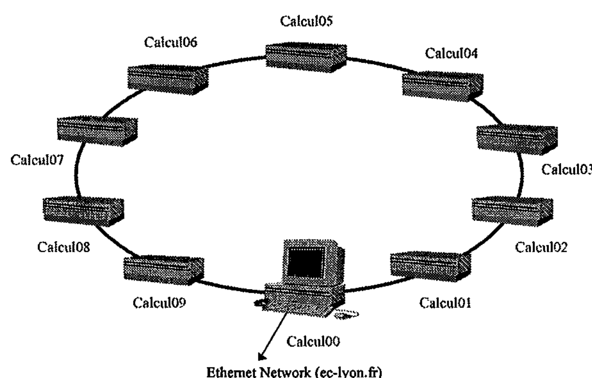


Fig. 3.1: Ferme de stations en anneau FDDI.

III 3 2. Choix du modèle de programmation

La ferme de stations supporte plusieurs modèles de programmation: SPMD (1 modèle), MPMD (couplage de modèles) et Maître/Esclave. La mise en oeuvre d'une application se fait par échanges explicites de messages. De ce fait, la gestion des communications et les synchronisations sont à la charge du programmeur. Dans cette étude, le logiciel PVM (Parallel Virtual Machine) sert de support à l'échange de messages entre les stations de travail.

Le choix du modèle de programmation se fait suivant différents critères intrinsèques [42]: type d'applications à programmer, environnement matériel et logiciel et habitudes de programmation.

En ce qui concerne le modèle MPMD, celui-ci ne convient pas à notre application car les différentes tâches à exécuter ne peuvent l'être que de manière séquentielle (assemblage, conditions aux limites, ...). Pour adopter une granulité plus fine, un parallélisme de données doit être mis en oeuvre.

Le modèle M/E pose des problèmes d'utilisation de la mémoire. En effet, il n'est pas possible de stocker toute la matrice EF sur une seule station de travail. Ceci aurait pour effet de détruire le principal avantage relatif au calcul parallèle pour notre étude en diminuant l'espace mémoire disponible.

Le modèle SPMD apparaît donc comme étant celui qui correspond le mieux aux habitudes de programmation. Le même programme se déroule donc sur tous les noeuds de calcul. La matrice EF étant répartie sur tous les processeurs, l'utilisation de l'espace mémoire disponible est ainsi optimisée. Si un processeur à besoin d'une donnée dont il ne dispose pas, il doit la réclamer à celui qui la possède.

III 3 3. Aspect logiciel

La ferme de stations est équipée principalement des logiciels suivants: compilateurs FORTRAN, C et C++, librairies de passages de messages (PVM, MPI), logiciels d'analyse de performances (PARAGRAPH, XPVM).

III 3 3 1. PVM

D'une manière générale, PVM peut permettre soit d'employer toutes les stations d'un réseau, éventuellement pendant les heures creuses, soit d'utiliser efficacement certains ordinateurs multiprocesseurs à mémoire répartie soit encore de servir de support pédagogique pour la formation au parallélisme. PVM est un logiciel composé d'un *deamon* et d'un ensemble de bibliothèques [43]. Son succès provient essentiellement de sa facilité de portage (30 architectures différentes) et de la configuration de la machine virtuelle. Il est utilisable en FORTRAN, C, C++. PVM n'est pas un langage, c'est un ensemble d'utilitaires et de bibliothèques qui offrent au programmeur des primitives avec lesquelles celui-ci pourra développer des mécanismes appropriés. Certaines actions peuvent être entreprises au niveau de la programmation afin d'optimiser les communications: éviter les synchronisations intempestives (barrières, réceptions bloquantes, ...), limiter les 'tamporisations' de messages,

transférer les grands volumes de données en mode PVMROUTEDIRECT. Ces recommandations ont été appliquées au code dès sa conception.

III 3 4. Outils d'analyse de performances et *debogage*

III 3 4 1. XPVM

XPVM est un environnement de développement autour de PVM. Il offre les fonctionnalités suivantes: obtention de traces, de *débogage* et d'analyse de performances. Cet outil est intéressant pour l'apprentissage de PVM, pour la mise au point des programmes et pour l'analyse de certaines caractéristiques du code. Néanmoins, son utilisation introduisant un surcoût important, il ne peut être utilisé sur de gros problèmes.

III 3 4 2. PARAGRAPH

Cet outil permet une analyse post-mortem de traces d'exécution de programmes PVM. Il génère notamment des informations sur les processus, les communications (nombres, volume, ...), ainsi que des données statistiques récapitulatives de toute l'exécution du code (moyenne, extremum, ...). Son utilisation implique certaines restrictions comme la manipulation et la traduction de fichiers post-mortem très volumineux pour des applications utilisant de nombreuses fois les ressources de PVM. De ce fait, l'analyse des performances à l'aide de PARAGRAPH se borne au problème test maillé avec 10000 noeuds (hexaèdres du premier ordre).

Le calcul de l'accélération du code sur un problème nécessite la résolution de celui-ci sur une seule station de travail. Par conséquent, les accélérations pour toutes les phases du code sont calculées pour le problème test qui est en fait le plus gros problème pouvant être résolu sur un processeur.

III 4. CRAY T3E

Le CRAY T3E de l'IDRIS est une machine massivement parallèle composée de 256 processeurs DEC ALPHA interconnectés par un réseau de communications. Ce calculateur parallèle est de type MIMD à mémoire distribuée. Par conséquent, il supporte les mêmes modèles de programmation que la ferme de stations. Ce type de super-calculateur paraît être la nouvelle tendance que les constructeurs ont adopté car seul le calcul massivement parallèle,

ou du moins les calculateurs parallèles à mémoire distribuée, semble encore avoir une bonne marge de progression en terme de puissance de calcul.

III 4 1. Aspect matériel

III 4 1 1. Architecture

Le CRAY T3E est une machine massivement parallèle dont l'architecture est à mémoire distribuée (fig. 3.2) mais globalement adressable [66].

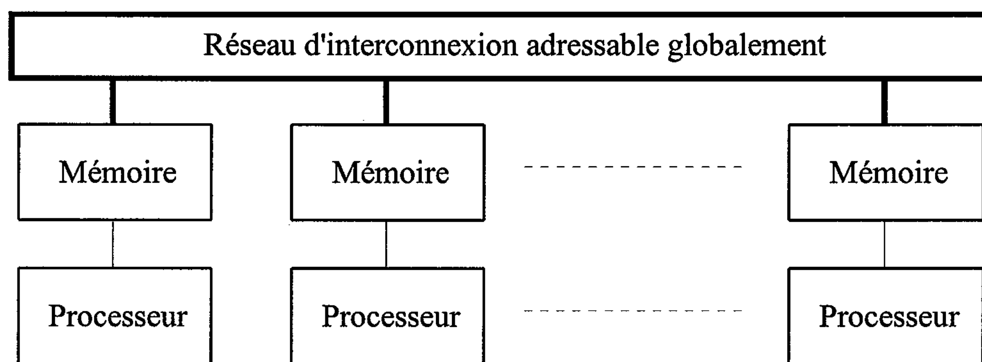


Fig. 3.2: architecture du CRAY T3E.

III 4 1 2. Réseau d'interconnexion

Le réseau d'interconnexion du CRAY T3E est un tore 3D (fig. 3.3). Chaque lien bidirectionnel, composé de deux liens unidirectionnels, possède une forte bande passante ainsi que des mécanismes pour cacher les latences. Une topologie en tore 3D présente plusieurs avantages: d'une part le nombre de connexions physiques reste raisonnable; d'autre part, grâce au bouclage des connexions, les transferts entre noeuds éloignés sont rapides; enfin, en cas de défaillance d'un lien et étant donné l'existence de plusieurs chemins, il y a automatiquement circulation dans la direction opposée entraînant une bonne tolérance aux pannes.

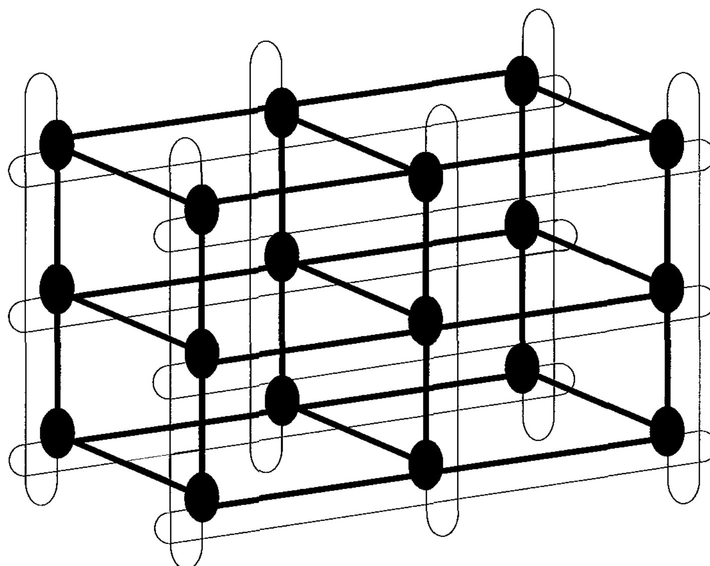


Fig. 3.3: Tore 3D (3x3x3).

Afin d'éviter des situations de blocage, le CRAY T3E est doté d'un système de gestion de canaux virtuels. Cela permet notamment de contourner les cas où 2 processeurs essaient simultanément de s'envoyer l'un l'autre des données, et où tous les processeurs dans une dimension donnée essaient simultanément d'envoyer une donnée au noeud qui suit.

III 4 1 3. Noeuds

Un noeud de calcul est composé de deux processeurs, d'une interface réseau et d'un système de transfert de données. Chaque processeur possède trois types de numérotations: physique, logique et virtuelle. Le numéro physique de chaque processeur est unique. C'est le système d'exploitation qui gère la configuration logique de manière à garder des noeuds de réserve pouvant remplacer n'importe quel noeud défaillant. Ce dernier est alors retiré de la table logique. De ce fait, tous les processeurs physiques ne font pas partie de la configuration logique.

A chaque application, le système d'exploitation attribue un groupe de noeuds qui possèdent une numérotation virtuelle convertie en numérotation logique par le système. Une partition peu être de plusieurs formes: grille unidirectionnelle, grille bidirectionnelle, grille tridimensionnelle, ... Ceci peut donc poser un problème quant à la répétitivité des performances parallèles.

Le noeud de calcul est organisé autour du microprocesseur DEC ALPHA 21164 (ou EV5). Ses principales caractéristiques sont les suivantes: cadencement à 300 Mhz, 128Mo de RAM/

processeur, jusqu'à 4 instructions / cycle d'horloge, 2 unités *pipelinées* de traitement des entiers, 2 unités *pipelinées* de traitement des flottants et de la division flottante, 600 Mflops/s et 1200 MIPS. Ce microprocesseur possède un cache d'instructions et un cache de données primaire de 8 Ko chacun ainsi qu'un deuxième niveau de cache de données de 96 Ko. Entre ce cache secondaire et la mémoire se situent 6 *Streams buffers* qui servent à anticiper les accès mémoire. Leur utilisation dans un code utilisant SHMEM pouvant entraîner une incohérence de données, le code doit respecter une structure séparant les phases de calcul et les phases de communications par des barrières. La mémoire est organisée en 8 bancs gérant chacun 8 Mmots. Le débit des transferts à la mémoire peut atteindre 600 Mo/s soutenus en lecture.

III 4 2. Aspect logiciel

III 4 2 1. Environnement système

Chaque processeur est équipé d'un micro-noyau (système UNICOS MAX à micro-noyaux) demandant peu d'espace mémoire pour lui-même. Il permet le contrôle des communications inter-processeurs, la gestion des allocations de mémoire ainsi que celle des interruptions. Les ressources sont gérées par le système d'exploitation en groupes administratifs: développement, production, ...

Quand une partition est attribuée à une application, les processeurs lui sont entièrement dédiés: pas de temps partagé. Une partition peut être composée de 1 à 256 processeurs. A l'intérieur d'une partition l'échange des données se fait par passages de messages. Les mécanismes implicites de CRAFT qui gèrent un parallélisme de données ne sont actuellement pas supportés par le CRAY T3E [69].

III 4 2 2. Environnement utilisateur

Sur le CRAY T3E de l'IDRIS, plusieurs compilateurs sont disponibles: C, C++, F90. De plus, plusieurs bibliothèques scientifiques sont à la disposition des utilisateurs: LAPACK, BLAS (version monoprocesseur), ScaLAPACK (version multiprocesseurs), ... Ces ressources n'ont pas été utilisées dans notre étude à cause du type de stockage requis pour les matrices.

A l'heure actuelle, un seul modèle de programmation est disponible sur cette machine: la gestion des communications est explicite, soit par échange de messages (PVM, MPI), soit par

échange de données (SHMEM) en utilisant le mécanisme d'adressage global du CRAY T3E. Au sein d'une même application ces deux techniques peuvent être mélangées.

III 4 3 3. SHMEM (SHared MEMory)

PVM est supporté en natif sur le CRAY T3E dans une version optimisée quelque peu différente de la version du domaine public. Etant donné le surcoût introduit par PVM, CRAY fournit une bibliothèque de communications non portable à très faible temps de latence. SHMEM s'appuie sur le concept de machine à mémoire physiquement distribuée et adressable globalement [69]. L'utilisation de PVM impose des copies dans des tampons intermédiaires, la construction de message ou encore un dialogue entre processeurs. SHMEM s'appuie simplement sur des copies ou des chargements de mémoire à mémoire sous la responsabilité du programmeur [70].

SHMEM opère sur des données qui sont à la même adresse physique sur chaque processeur qui en détient une copie. Pour ce faire, ces données doivent être déclarées en SAVE ou dans un COMMON en FORTRAN, en *static* en C, ou encore à l'aide de la bibliothèque 'malloc.h' en C pour une allocation dynamique à la même adresse sur chaque processeur (fonction shmalloc). Le transfert d'une donnée de mémoire à mémoire est effectuée sans que le processeur récepteur en soit averti. De ce fait, la synchronisation est à la charge du programmeur.

Les fonctions SHMEM peuvent être de 4 types:

- Les communications point à point permettent de copier ou charger des éléments contigus en mémoire avec un pas constant ou variable. La synchronisation est très importante et est à la charge du programmeur car le retour d'une fonction de copie ne garantie pas la fin de l'opération.

- Les communications collectives permettent de diffuser une variable, de collecter un nombre, distinct par processeur ou constant, d'éléments d'une variable source. Les processeurs impliqués dans ces opérations sont définis dans les variables des ces fonctions. L'utilisation de ce type d'instructions est fortement préconisée du fait de leur rapidité et de leur sûreté de fonctionnement. Une synchronisation de tous les processeurs participant à l'opération est nécessaire avant l'appel à celle-ci pour s'assurer que les données sont valides sur chacun des processeurs.

- Les fonctions de réductions opèrent sur des données de même type. Elles peuvent consister en une somme, un produit, un calcul du minimum, maximum, fonction logique, ... Les synchronisations, fondamentales pour ces opérations, sont ici encore à la charge du programmeur.

- Les fonctions de synchronisations peuvent être des barrières ou encore des attentes d'événements. Les barrières assurent aussi bien la fin des copies distantes que la synchronisation d'un ensemble de processus.

Comme il est dit plus haut, il est possible de mélanger PVM et SHMEM dans une même application. En règle générale, PVM est conservé dans les parties de code où les passages de messages n'introduisent pas trop d'*overhead* c'est à dire où beaucoup de calculs et peu de passages de gros messages sont effectués. SHMEM doit supplanter PVM pour les portions de code où les temps de communications sont critiques.

III 4 3. Outil d'analyse de performances

Le logiciel APPRENTICE permet en premier lieu de connaître précisément la répartition du temps CPU utilisé par les différentes parties du programme. Les performances du code données par APPRENTICE ne sont pas fiables mais les rapports entre les différents temps le sont. Par conséquent, et étant donné que les processeurs sont dédiés à une application, le temps de restitution et la connaissance du nombre d'opérations permet de calculer le nombre de Mflops/s que le code a atteint.

III 4 4. Optimisation monoprocesseur

Afin d'augmenter les performances monoprocesseur d'un code, il est nécessaire de réaliser certaines modifications [67, 68]. Il est préférable de focaliser les travaux sur les parties du code les plus consommatrices en temps CPU (APPRENTICE). Les optimisations à réaliser peuvent être de plusieurs ordres. Dans ce paragraphe sont décrites les principales modifications qui ont permis d'augmenter les performances monoprocesseur de notre code:

- La lecture d'adresses dans la mémoire doit se faire avec un pas de 1. Quand cette condition n'est pas remplie, il suffit, la plupart du temps, soit d'inverser les boucles, soit de dérouler la boucle incriminée.

- Eviter la lecture simultanée de deux références associées à la même ligne de cache. Pour ce faire, il est conseillé d'insérer un tableau muet (*pading*) entre les deux tableaux devant être accédés. Cette technique n'est praticable que sur des données allouées statiquement.

- Aligner un tableau avec le premier mot de la ligne de *cache* à l'aide de directives telles que CDIR$ cache_align. Cette méthode permet d'éviter de charger les 4 mots de la ligne de *cache* si une référence mémoire ne se trouve pas déjà dans le *cache*.

- Ecrire de façon contiguë dans la mémoire et minimiser le nombre les flux d'écriture au nombre de trois.

- Eviter les divisions par des constantes dans les boucles. Il faut par conséquent effectuer la division (1/la constante) en dehors de la boucle, stocker le résultat dans une variable tampon et faire la multiplication par cette variable tampon dans la boucle.

Le lecteur intéressé pourra se reporter à la documentation [67, 68] disponible sur le serveur WEB [71] de l'IDRIS pour plus de détails. De plus, il est conseillé de s'appuyer sur les bibliothèques fournies par le constructeur pour effectuer certaines opérations. Les compilateurs disponibles sur le CRAY T3E sont riches en options d'optimisation. Il faut néanmoins les tester une à une et vérifier le résultat du calcul à chaque fois.

Certaines directives énoncées dans la documentation fournie par l'IDRIS ont été appliquées à notre code. Globalement, les performances monoprocesseur ont augmenté de 20 %.

III 5. Conclusion

A partir des informations regroupées dans ce chapitre, il est possible de choisir les algorithmes susceptibles d'exploiter au mieux les ressources de chaque machine. De plus, une bonne connaissance des outils logiciels disponibles sur chaque calculateur permet d'optimiser le code. Actuellement, les constructeurs de super-calculateurs semblent s'orienter vers des machines massivement parallèles ou, du moins, vers des architectures à mémoire distribuée. Avec de tels ordinateurs, le choix des algorithmes et, par là-même, l'intervention du programmeur sont beaucoup plus délicats que sur des architectures à mémoire partagée. Par conséquent, nos efforts se sont essentiellement portés sur l'implantation de notre formulation sur des calculateurs parallèles à mémoire distribuée.

CHAPITRE IV
IMPLANTATION SUR LE CRAY C98

IV 1. Introduction

De nombreux travaux relatifs à l'adaptation de codes séquentiels sur des calculateurs à mémoire partagée peuvent être trouvés dans la littérature. Ces études portent sur l'amélioration des algorithmes utilisés par différents codes éléments finis notamment au niveau des phases d'assemblage (décomposition de domaine, éléments finis indépendants, ...) [25-26] et de résolution du système matriciel (méthodes itératives, frontales, ...) [27-31].

Dans notre étude, la formulation présentée au chapitre II étant déjà opérationnelle sur calculateur scalaire (station de travail), le code développé en FORTRAN 77 a donc été porté tel quel sur le CRAY C98 de l'IDRIS. Comme il le sera démontré par la suite, la plupart des algorithmes utilisés sur un calculateur monoprocesseur n'ont pas besoin d'être entièrement modifiés pour une utilisation efficace sur une machine parallèle à mémoire partagée de type MIMD [32-33]. Ce type d'approche présente, en autre, l'avantage de n'introduire aucun calcul supplémentaire par rapport à un code séquentiel (pré-processing pour la décomposition de domaine, ...).

Dans ce chapitre sont présentés la parallélisation automatique puis manuelle du code, les différents types de stockages adoptés ainsi que les performances vectorielles et parallèles obtenues pour chaque type de stockage.

IV 2. Code parallèle et performances

IV 2 1. Parallélisation automatique

Un code devant évoluer sur un calculateur parallèle doit, en premier lieu, être modifié de façon à supprimer toute interactivité lors de son déroulement. Le programme ainsi transformé a été compilé à l'aide des outils d'*autotasking* (FPP, FMP).

La figure (4.1) montre que les performances parallèles ainsi obtenues sont médiocres sur le problème test. On remarque que le code présente un taux de parallélisme de seulement 16%.

La figure (4.2) donne la répartition du temps CPU pour les différentes étapes du code dans lesquelles le pré-processeur a apporté des modifications.

Le taux moyen de remplissage des registres vectoriels sur la totalité du temps d'exécution du programme est de 9.9 mots. Sachant que la longueur des registres vectoriels du CRAY C98 est de 128 mots, ce taux de remplissage moyen peut paraître peu élevé. De plus, cette étude met aussi en évidence des performances vectorielles assez médiocres (20 Mflops).

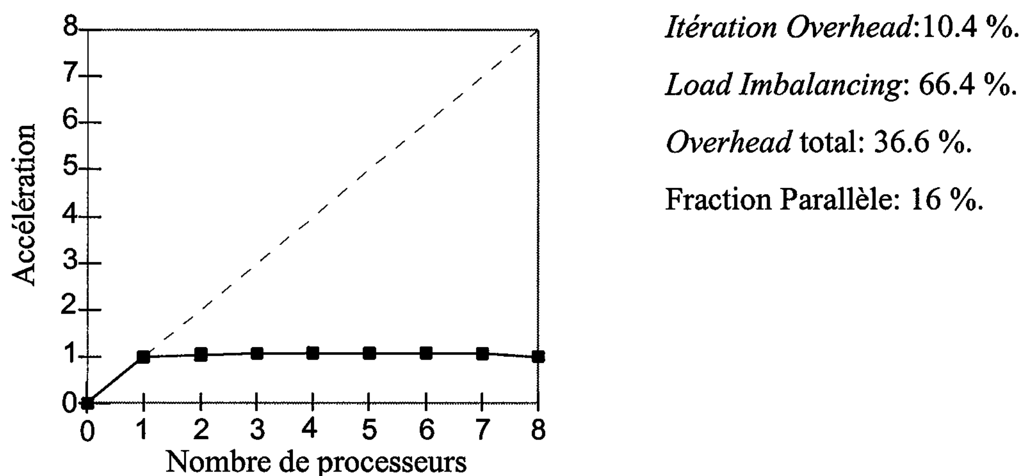


Fig. 4.1: Accélération pour le problème test après parallélisation automatique.

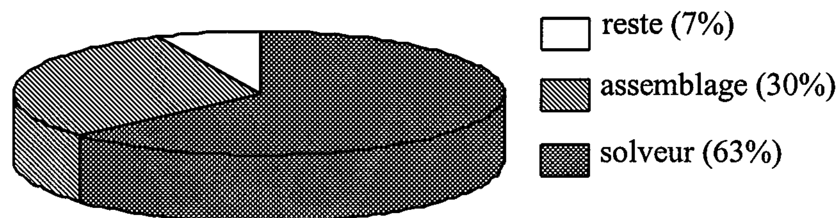


Fig. 4.2: Répartition du temps CPU pour le problème test après parallélisation automatique.

Une analyse plus fine des performances révèle que le pré-processeur a réalisé un important travail de vectorisation. La plupart des boucles ont été modifiées de telle sorte à favoriser la vectorisation (déroulement, inversion, IDVEP, ...). Par contre, au niveau de la parallélisation, peu de choses ont été réalisées. En effet, le pré-processeur n'a pu paralléliser la plupart des boucles car il n'a pu trouver la portée des variables dans celles-ci. L'étude de la portée des variables a donc été réalisée, à l'aide du logiciel ATSCOPE, et par la suite, chaque étape du code a pu être parallélisée en ajoutant manuellement des directives de parallélisation

(*microtasking*). Cette démarche a permis de garder le contrôle des algorithmes parallèles utilisés.

IV 2 2. Stockage

Dans la littérature, beaucoup de travaux relatifs au portage de codes sur ce type de machine ont été consacrés à l'optimisation du type de stockage de la matrice EF [1]. En effet, celui-ci a une grande influence sur les performances vectorielles du code, notamment durant la phase de résolution où beaucoup d'accès aux termes de la matrice sont réalisés. Il faut aussi garder à l'esprit que, bien que les performances parallèles soient très appréciables (et appréciées par l'IDRIS), le CRAY C98 est avant tout constitué de processeurs puissamment vectoriels. La parallélisation des différentes étapes nécessaires à la résolution du système matriciel a mis en évidence ce problème [82] et plusieurs types de stockages ont donc été implantés:

- Stockage MORSE: cette méthode est la plus économique en terme d'espace mémoire. En effet, une fois la matrice EF symétrisée, seule la partie inférieure de celle-ci servira pour les étapes suivantes. De ce fait, l'espace mémoire alloué pour la partie supérieure peut éventuellement servir au stockage de la matrice de préconditionnement dans le cas du préconditionnement de Cholesky incomplet.

- Stockage MORSE redondant [1], [27-28]: cette technique nécessite deux fois plus d'espace mémoire que le stockage MORSE car la matrice EF restera stockée dans son intégralité même après sa symétrisation. Il est réalisé de la même façon que le stockage MORSE avec en plus un vecteur qui permet de situer, pour chaque ligne, le terme se trouvant sur la diagonale. Ce type de stockage présente l'intérêt de conserver toute la ligne de la matrice. Ceci est un atout pour la parallélisation de la multiplication matrice-vecteur. De plus, son utilisation permet l'accès aux termes d'une même colonne de façon concourante dans la mémoire. Comme cela sera démontré par la suite, cet aspect est primordial pour le préconditionnement de Cholesky.

- Stockage type 'lignes de ciel': son utilisation permet l'obtention de très bonnes performances vectorielles (>100 *Mflops* pour la totalité du code sur le problème test). En revanche, étant donné que tous les termes présents à l'intérieur de la bande de la matrice EF sont stockés, son utilisation s'avère impossible sur des problèmes de grosse taille. Sur le problème test, la largeur de bande est d'environ 3000 pour seulement 280 termes non nuls avec des hexaèdres du premier ordre. L'augmentation de la taille du problème tend en outre à

élargir la bande de la matrice EF. De plus, l'utilisation du préconditionnement de Cholesky incomplet avec ce stockage revient en fait à effectuer la méthode de Cholesky (méthode directe) car il n'y a plus réjection des termes à l'intérieur de la bande. Pour ces différentes raisons, ce type de stockage n'a pu être utilisé pour la résolution de grands problèmes et n'est, par conséquent, pas présenté dans cette étude.

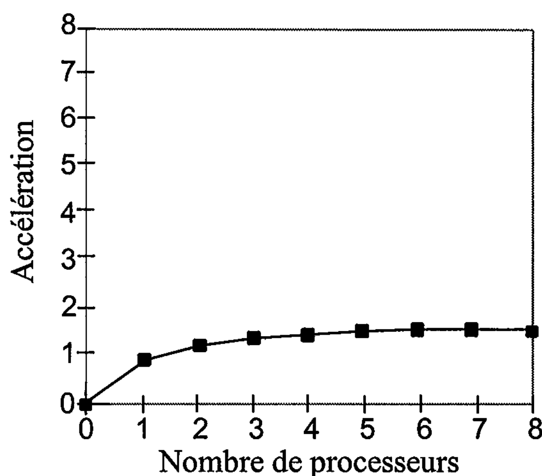
Les paragraphes qui suivent présentent les choix effectués pour la parallélisation de chaque étape du code, les performances parallèles et les temps de calcul en fonction du type de stockage (MORSE ou MORSE redondant).

IV 2 3. Assemblage

Avant de réaliser l'assemblage, le code procède à l'initialisation de différents tableaux alloués statiquement ainsi qu'à la lecture du fichier de données. Ces différentes opérations sont en grande partie vectorisées mais peu parallélisées. Elles apparaîtront dans la répartition globale du temps CPU dans la partie dénommée 'autre'.

IV 2 3 1. Assemblage symbolique

La construction des tables permettant le stockage de la matrice EF est une opération qui ne peut être parallélisée car elle contient implicitement une dépendance arrière. Il est possible d'évaluer, par exemple, les termes du vecteur 'rang' relatifs à la ligne 2 de la matrice, mais leur placement au sein du vecteur nécessite au préalable la même opération pour la ligne 1. De ce fait, cette opération est essentiellement vectorisée.



Overhead total: 38 %.

Fraction parallèle : 48.5 %.

Fig. 4.3: Accélération pour l'assemblage symbolique du problème test.

La figure (4.3) montre que les performances parallèles obtenues pour cette étape sur le problème test sont médiocres. Ceci est essentiellement dû à l'importance de la fraction séquentielle de cette partie du code. Notons que le type de stockage n'a pas d'incidence sur cette étape du fait que toute la structure de la matrice EF est préparée dans les deux cas. La construction du vecteur supplémentaire (position de la diagonale dans les tables 'Termes' et 'Rang'), nécessaire pour l'utilisation du stockage MORSE redondant, n'a que très peu d'influence sur la durée totale de cette étape (9700 cycles d'horloge au lieu de 9200 sur l'exemple de la figure 4.3).

IV 2 3 2. Assemblage de la matrice EF

La parallélisation de cette opération est réalisée de sorte que chaque processeur assemble une matrice élémentaire relative à un élément fini. Les paramètres se reportant à cette matrice doivent avoir une portée dite 'privée' par rapport à la tâche exécutée par chaque processeur. Ceux-ci sont donc déclarés dans un COMMON précédé de la directive CMIC$ TASKCOMMON. De ce fait, chaque processeur aura sa copie de ces paramètres (tableaux, indices, termes, ...). La boucle sur les EF est donc modifiée comme suit:

```
CMIC$ DO ALL private (indices) shared (données issues du mailleur)
        DO indice = 1 , nombre d'éléments
```

Chaque processeur assemble une matrice élémentaire et va placer les termes de celle-ci dans la matrice globale stockée en mémoire partagée. Lors de cette écriture, il est possible que deux processeurs tentent de modifier simultanément la même case mémoire dans le vecteur 'terme'. Pour éviter tout conflit, la partie du code modifiant la matrice EF est entourée de directives permettant la création d'une région critique (séquentielle):

```
CMIC$ GUARD
        terme(i) = terme(i) + coef
CMIC$ END GUARD
```

La figure (4.4) présente cette technique. La notion de mémoire privée à chaque processeur n'est pas physique; elle permet néanmoins d'illustrer le concept d'espace mémoire partagé accessible par seulement un processeur.

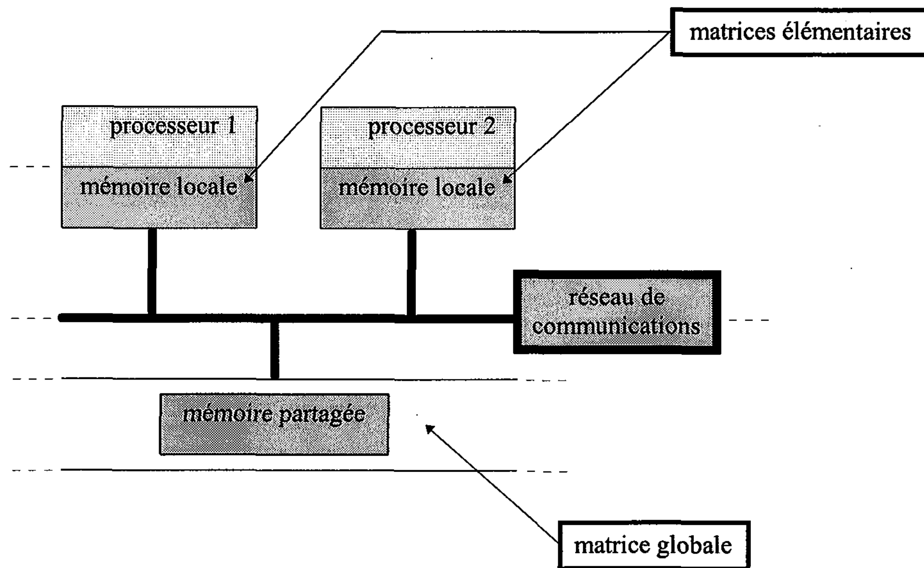
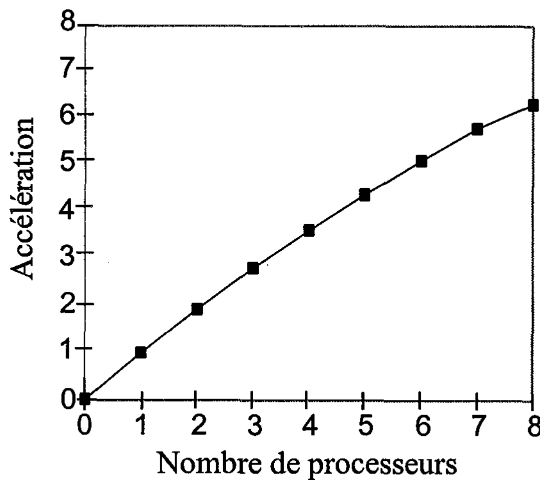


Fig. 4.4: Assemblage par contributions élémentaires.



Overhead total: 1.8 %.

Fraction parallèle : 96.4 %.

Fig. 4.5: Accélération pour l'assemblage du problème test.

L'accélération obtenue ainsi que les principales caractéristiques de la région parallèle sont présentées à la figure (4.5) sur le problème test. Les meilleures performances parallèles ont été obtenues, pour cette étape, avec une répartition parallèle du travail de type SINGLE. En effet, la répartition de charge étant bonne, toutes les itérations sont réalisées de façon concourante. Ici encore, le type de stockage n'a que très peu d'incidence.

IV 2 3 3. Décomposition des EF volumiques en EF surfaciques

Cette opération intervient après l'assemblage de la matrice. Elle consiste à établir la liste des EF surfaciques (numéro des noeuds, coordonnées ...) à partir de celle des EF volumiques afin de pouvoir introduire par la suite les différentes conditions aux limites. Cette étape est

constituée d'une boucle sur les EF volumiques incluant différents tests. La parallélisation se fait donc de la même façon qu'au paragraphe précédent: la boucle sur les EF est éclatée pour être réalisée en parallèle sur les différents processeurs. La portée des variables suit la même logique que précédemment: l'indice de boucle ainsi que les variables de tests sont privées à chaque processeur et les tableaux construits ainsi que les données issues du mailleur sont partagées. La modification des tableaux comportant les informations relatives aux EF surfaciques ainsi que leur comptage se fait dans une région critique de code.

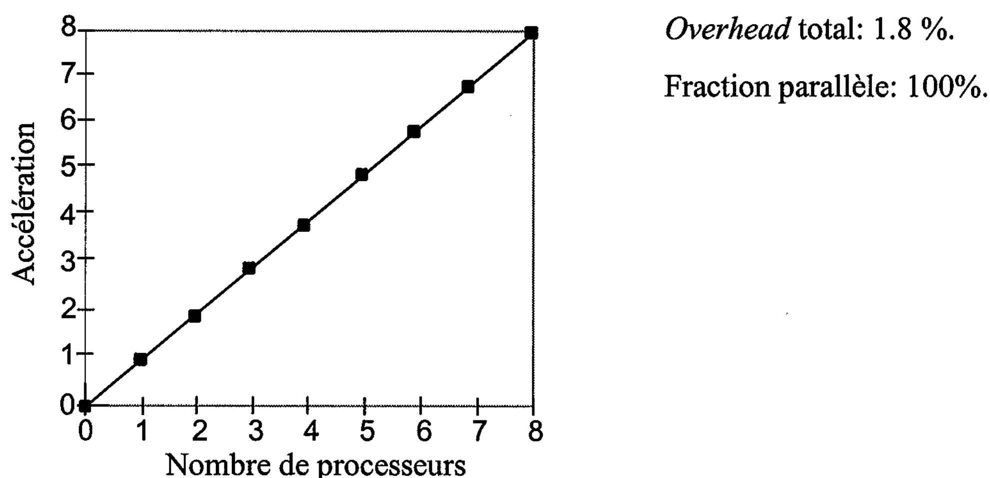


Fig. 4.6: Accélération pour la décomposition de EF volumiques en EF surfaciques du problème test.

A partir des performances parallèles de la figure (4.6), obtenues pour une répartition du travail de type SINGLE sur le problème test, il apparaît que la partie séquentielle de cette étape (modification des tableaux dans la région critique) étant réalisée très rapidement, celle-ci ne fait donc pas décroître les performances. Le type de stockage n'a pas d'influence sur cette partie du code.

Notons que la méthode utilisée reste la même que dans le code séquentiel car les performances obtenues sont très bonnes.

IV 2 4. Traitement des symétries

La modification de la matrice EF permettant l'application des conditions aux limites sur les plans de symétries / antisymétries consiste principalement en une boucle sur les éléments surfaciques du problème. Un test est effectué sur chaque élément surfacique afin de savoir si celui-ci se trouve sur un plan de symétrie / antisymétrie. Si cette condition est remplie, la modification des lignes / colonnes de la matrice EF est réalisée.

La parallélisation de cette étape est faite en répartissant sur tous les processeurs la boucle sur les éléments surfaciques (DO ALL). Chaque processeur va donc effectuer une partie des modifications à apporter à la matrice. Les tableaux contenant les informations relatives aux éléments surfaciques (table des éléments, coordonnées des noeuds, ...) sont partagés par tous les processeurs (*shared*) tandis que l'indice de boucle ainsi que les variables de test sont privés à chaque processeur (*private*). La modification de la matrice EF stockée en mémoire partagée devrait se faire dans une région critique du code afin d'éviter des conflits mémoire. La figure (4.7) montre l'exemple de deux processeurs qui fixent la condition illustrée par (2.46) sur les lignes 4 et 6 de la matrice EF.

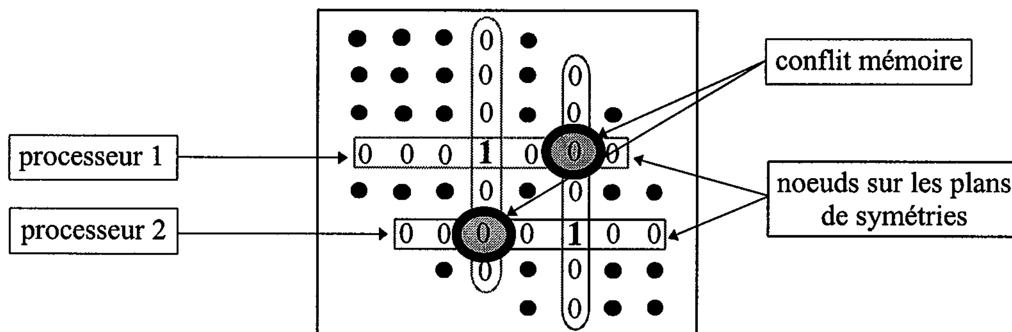
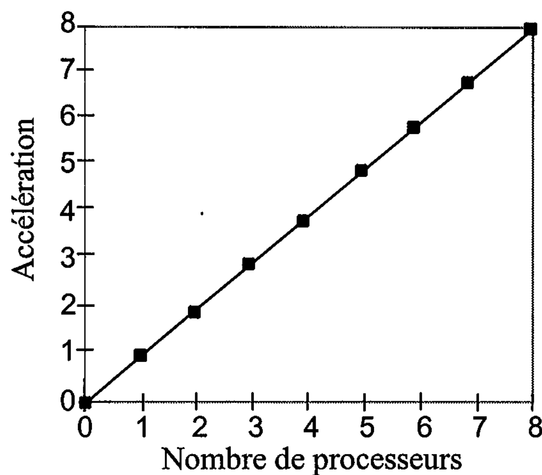


Fig. 4.7: Modification de la matrice EF par 2 processeurs.

Sur l'exemple à 2 processeurs de la figure (4.7), il apparaît que, si les 2 processeurs tentaient de modifier simultanément la même case mémoire, ce serait pour y apporter la même modification. De ce fait, la zone de code qui modifie la matrice EF se trouve dans une région parallèle. La figure (4.8) montre les performances ainsi que le pourcentage d'*overhead* relatifs à la parallélisation de cette opération sur le problème test qui comporte une symétrie.



Overhead total: 1.8 %.

Fraction parallèle: 100%.

Fig. 4.8: Accélération pour le traitement des symétries / antisymétries du problème test.

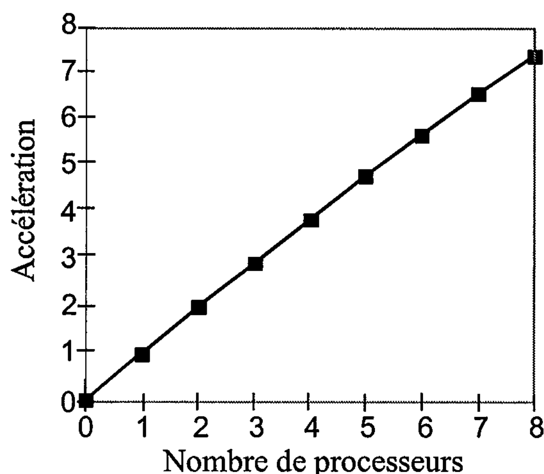
Les meilleures performances parallèles ont été obtenues avec une répartition du travail de type SINGLE. Le type d'assemblage n'a pas d'influence sur cette étape.

IV 2 5. Conditions aux limites sur les conducteurs électriques parfaits (cep)

Cette étape est parallélisée de la même manière que celle relative au traitement des symétries. La boucle sur les EF surfaciques est donc parallélisée mais la modification de la matrice globale se fait dans une région critique pour une résolution en champ E car, dans ce cas, il est possible que les modifications que deux processeurs tentent d'apporter simultanément à la matrice EF soient différentes.

La figure (4.9) montre que les performances parallèles pour l'application des conditions aux limites sur les cep sur le problème test pour une résolution en champ E sont très bonnes.

Pour cette étape, les meilleures performances parallèles ont été obtenues avec une répartition du travail de type SINGLE. Contrairement au traitement des symétries, la modification de la matrice EF dans la région critique (séquentielle) est ici pénalisante en terme d'accélération. Le type de stockage n'a pas d'influence sur cette partie du code.



Overhead total: 2.2 %.

Fraction parallèle : 98.5%.

Fig. 4.9: Accélération pour la prise en compte des conditions aux limites sur les cep pour le problème test (résolution en champ E).

IV 2 6. Symétrisation du système matriciel.

La symétrisation du système matriciel est réalisée par l'addition de la matrice EF avec sa matrice transposée.

Stockage MORSE: les deux matrices étant stockées indépendamment, cette opération ne nécessite que deux boucles imbriquées. La boucle externe opère sur les lignes de la matrice

EF et sur les colonnes de la matrice transposée, tandis que la boucle interne décrit les colonnes et les lignes de la matrice transposée. La parallélisation est effectuée en éclatant la boucle externe sur tous les processeurs:

CMIC$ DO ALL shared (nombre de lignes, tableaux relatifs aux deux matrices, ...)
private (indice, variables de test)

La boucle interne sur les colonnes est vectorisée grâce à l'adjonction de la directive CMIC$ IVDEP qui permet au compilateur d'ignorer une éventuelle dépendance dans cette boucle. La figure (4.10) illustre cette technique.

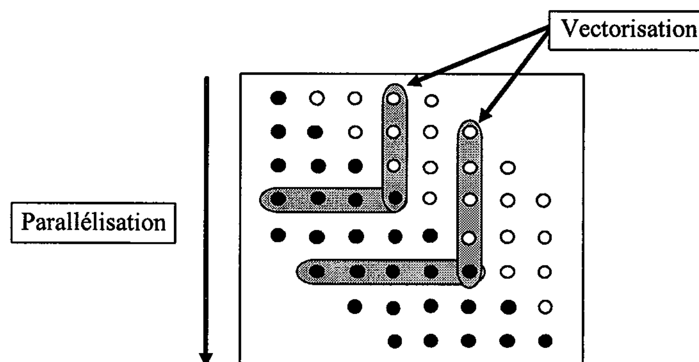
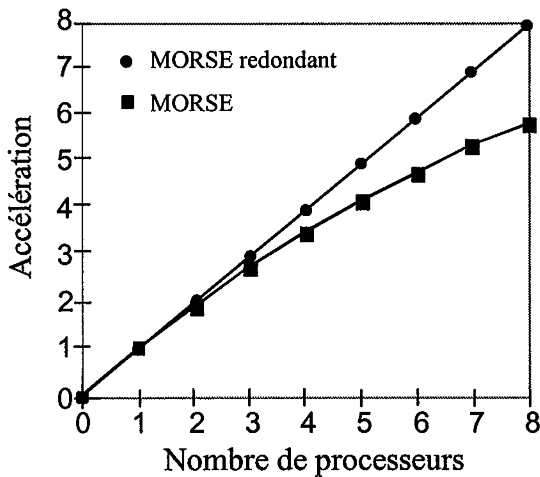


Fig. 4.10: Parallélisation et vectorisation de la symétrisation du système matriciel.

Plusieurs configurations de la répartition du travail ont été testées afin de déterminer le meilleur pavage: DO ALL SINGLE, DO ALL CHUNKSIZE (100), DO ALL CHUNKSIZE (1000), ... Le meilleur compromis a été trouvé pour un mode SINGLE. Evidemment, un autre choix pourrait s'avérer meilleur pour de plus gros problèmes. Mais, étant donnés les problèmes énoncés plus haut quant à l'utilisation de ATEXPERT, lors du traitement de géométries réalistes, cette opération restera en mode SINGLE.

Les performances parallèles obtenues pour cette opération sur le problème test (fig. 4.11) montrent que l'*overhead* introduit fait chuter les performances parallèles parce que chaque itération pour le stockage MORSE est très rapide (nombre de cycles d'horloge).

Stockage MORSE redondant: l'opération est plus complexe. En effet, pour chaque terme dont le rang est inférieur à la diagonale, il faut aller chercher son transposé dans la ligne qui le contient. L'opération est donc plus longue. La parallélisation est réalisée comme pour le stockage MORSE.



MORSE (1.1 e6 cycles d'horloge)

Overhead total: 1.8 %.

Fraction parallèle : 94.8%.

MORSE redondant. (1.7 e8 cycles d'horloge)

Overhead total: 1.8 %.

Fraction parallèle : 99.8%

Fig. 4.11: Accélération pour la symétrisation du système matriciel du problème test.

La figure (4.11) présente les performances parallèles, sur le problème test, qui montrent que l'*overhead* introduit pour le démarrage de chaque itération est négligeable devant la durée de celle-ci (MORSE redondant). Cette opération est plus longue avec l'utilisation du stockage MORSE redondant, mais ce surcoût est négligeable car cette étape ne représente qu'un très faible pourcentage du temps total d'exécution (fig. 4.19).

IV 2 7. Résolution du système d'équations

IV 2 7 1. Préconditionnement diagonal

Le préconditionnement est réalisé par une multiplication vecteur-vecteur. Cette opération est parallélisée en cassant la boucle sur les termes des vecteurs. Une multiplication matrice-vecteur étant nécessaire à chaque itération afin de calculer le nouveau vecteur résidu, celle-ci est réalisée en parallèle. Chaque processeur va effectuer une partie du travail en multipliant plusieurs lignes de la matrice par le vecteur. Ce dernier sera partagé par tous les processeurs ainsi que les tableaux de la matrice EF. La méthode diffère quelque peu suivant le type de stockage adopté.

Stockage MORSE: étant donné que seule la partie inférieure de la matrice EF est disponible à ce niveau du calcul, chaque processeur ne peut multiplier que les termes en sa possession par le vecteur. Les vecteurs résultats partiels, stockés en mémoire privée propre à chaque processeur, sont concaténés en mémoire partagée. Cette dernière opération est vectorisée. La figure (4.12) détaille ces opérations.

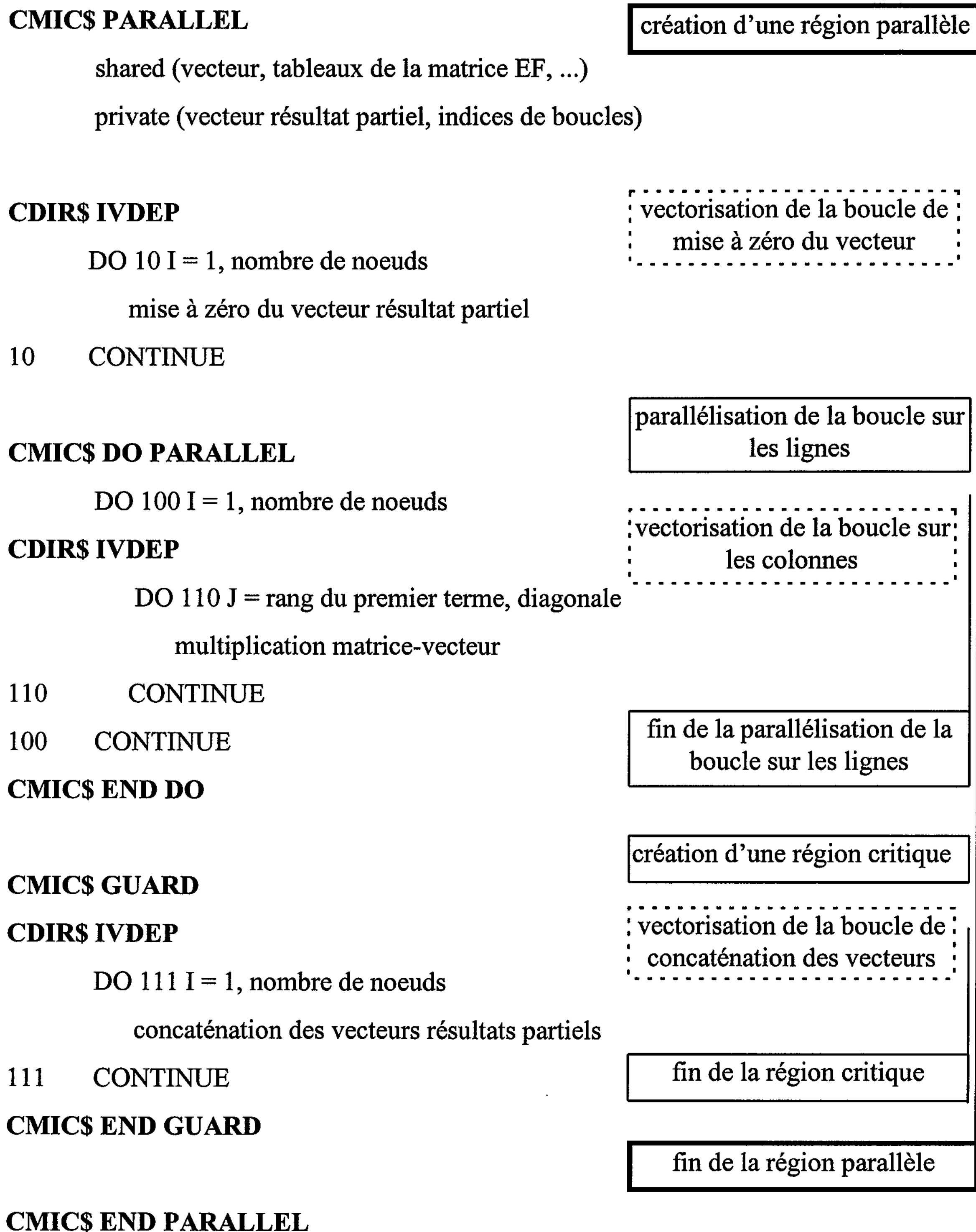


Fig. 4.12: Parallélisation de la multiplication matrice-vecteur.

La figure (4.13) illustre les opérations effectuées par deux processeurs lors de la multiplication matrice-vecteur. Dans cet exemple, la répartition du travail est réalisée de sorte que le processeur 1 opère sur les deux premières lignes et le processeur 2 sur les suivantes (DO PARALLEL CHUNKSIZE (2)). La mauvaise répartition de la charge paraît alors

évidente. Cependant lors du traitement de grands systèmes matriciels issus de méthodes finies, la matrice est creuse et bande, ce qui entraîne en fait une excellente répartition de charge.

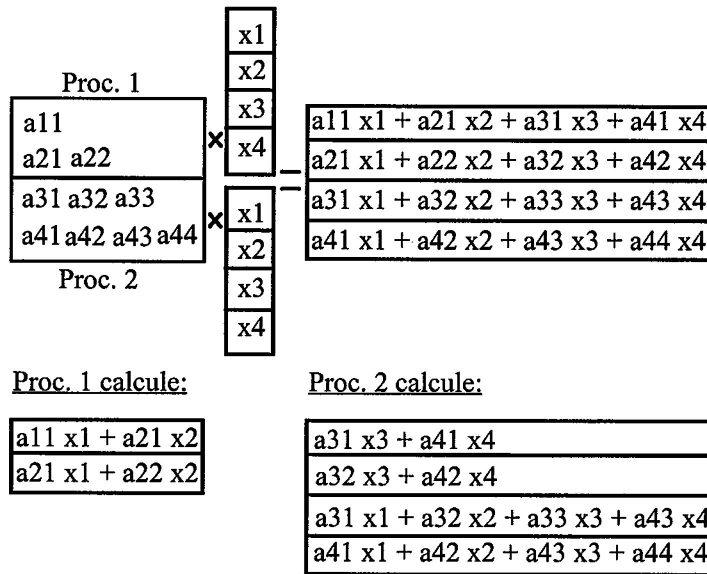
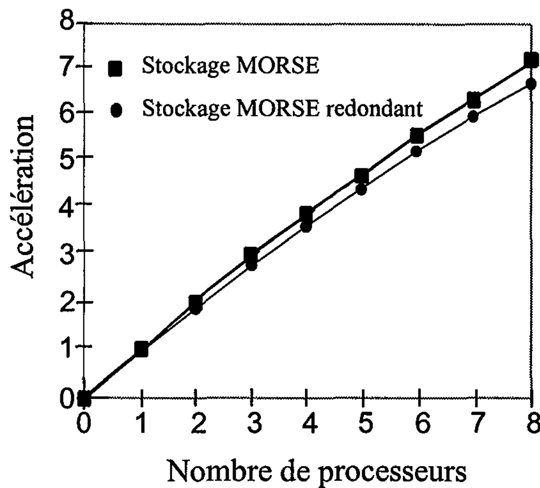


Fig 4.13: Multiplication matrice-vecteur sur 2 processeurs.

Une étude du pavage de la boucle 100 (fig. 4.12) a été nécessaire afin d'optimiser la répartition du travail sur les processeurs. Plusieurs modes ont été testés: DO PARALLEL SINGLE, DO PARALLEL CHUNKSIZE (10), DO PARALLEL CHUNKSIZE (100), DO PARALLEL CHUNKSIZE (1000). DO PARALLEL CHUNKSIZE (10000). Pour cette taille de problème, le mode DO PARALLEL CHUNKSIZE (1000) s'est avéré le plus performant. D'après les résultats obtenus (fig. 4.14) sur le problème test, il ressort que les performances parallèles obtenues pour le stockage MORSE sont assez bonnes malgré l'utilisation d'une région critique.

Stockage MORSE redondant: étant donné que la totalité de la matrice EF est disponible, chaque processeur peut calculer une partie du vecteur. Celui-ci réside en mémoire partagée car aucun conflit mémoire n'est possible. La parallélisation est donc effectuée en répartissant la boucle sur les lignes de la matrice sur les différents processeurs. La même étude du pavage a été réalisée, la même répartition (DO PARALLEL CHUNKSIZE (1000)) s'est avérée la plus performante.



Stockage MORSE (1.8e9 cycles d'horloge)

Overhead total: 3.5 %.

Fraction parallèle: 98.7 %.

Stockage MORSE redondant (8.7e8 cycles d'horloge)

Overhead total: 4.2 %.

Fraction parallèle: 97.7 %.

Fig. 4.14: Accélération pour la multiplication matrice-vecteur en mode CHUNKSIZE (1000) pour le problème test.

De la figure (4.14), on remarque que le type de stockage utilisé n'a que très peu d'influence sur les performances parallèles ainsi que sur le nombre de cycles d'horloge nécessaire au calcul (performances vectorielles).

IV 2 7 2. Préconditionnement de Cholesky incomplet

Ce type de préconditionnement permet de réduire considérablement le nombre d'itérations nécessaires à la résolution du système d'équations. En effet, pour le problème test, le GC muni du préconditionnement diagonal nécessite 224 itérations tandis que l'utilisation d'un préconditionnement de Cholesky incomplet permet de réduire ce nombre à 76. Toutefois, afin de pouvoir évaluer l'efficacité des deux méthodes de préconditionnement, il faut également comparer le temps CPU total utilisé par chacune d'elles pour résoudre un problème.

L'utilisation du préconditionnement de Cholesky incomplet dans l'algorithme du GC requiert une multiplication matrice-vecteur, qui est parallélisée de la même façon qu'au paragraphe précédent suivant le type de stockage, la construction de la matrice de préconditionnement ainsi qu'une phase de résolution d'un système matriciel constitué de deux matrices triangulaires supérieure et inférieure.

Si la matrice EF notée A est symétrique définie positive, il existe une matrice triangulaire inférieure unique notée L, telle que $A = L \times L^t$. Ses coefficients sont calculés suivant un algorithme par colonnes [27-29] (algo. 4.1):

pour j = 1 à nombre de lignes
(calcul du terme diagonal)

$$Ljj = \sqrt{Ajj - \sum_{k=1}^{j-1}(Ljk)^2}$$

pour i = j+1 à nombre de lignes
(calcul des coefficients de la colonne j)

$$Lij = \frac{1}{Lii}(Aji - \sum_{k=1}^{i-1} Lik\, Ljk)$$

fin

fin

Algo. 4.1: Construction de la matrice de Cholesky par colonnes.

Cet algorithme possède un parallélisme implicite: dès l'évaluation du coefficient diagonal Ljj, les coefficients Lij de la colonne j de la matrice L peuvent être calculés indépendamment. Cette méthode a déjà été exploitée sur une architecture à mémoire partagée [1], [27-28]. Toutefois, si les matrices sont connues par lignes (stockage MORSE), la recherche des termes d'une même colonne implique une perte de temps. Le stockage MORSE redondant permet de s'affranchir de cet inconvénient [27] car l'accès au terme d'une même colonne se fait en balayant les termes de rang supérieur à la diagonale de la ligne correspondante. La figure (4.15) illustre le déroulement de cet algorithme.

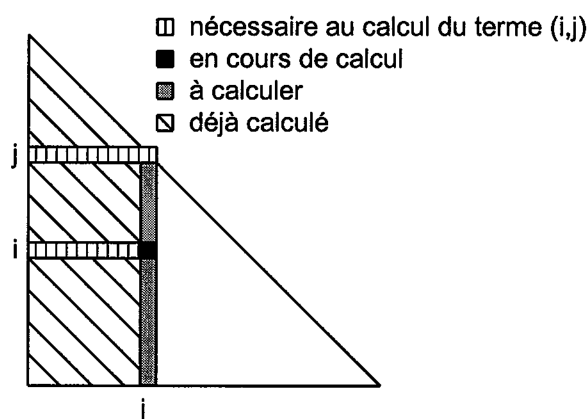


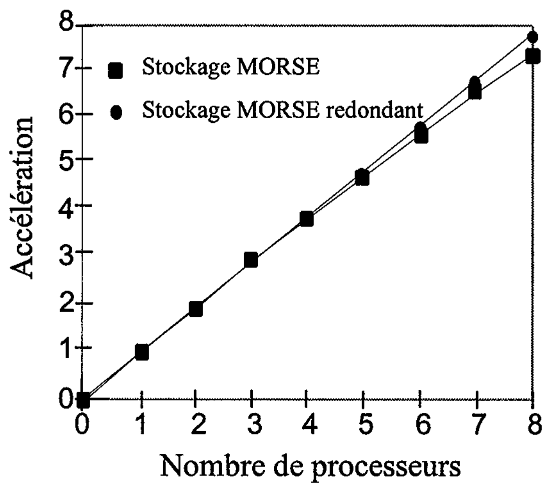
Fig. 4.15: Construction de la matrice de Cholesky par colonnes.

Le préconditionnement de Cholesky incomplet consiste donc en la construction de la matrice L de même structure que A. Ceci revient à rejeter les termes de remplissage.

Stockage MORSE: la matrice de préconditionnement est stockée à la place de la partie supérieure de la matrice EF car, dans ce cas, seule L est construite.

Stockage MORSE redondant: son utilisation requiert l'allocation d'un espace mémoire équivalent à celui nécessaire au stockage de la matrice EF entière. En effet, la matrice de préconditionnement est construite et stockée au travers du même stockage MORSE redondant que la matrice EF car l'étape suivante nécessite aussi l'accès aux termes de L par colonnes.

La parallélisation de cette étape est réalisée par la répartition des itérations de la boucle i de l'algorithme (4.1) sur les différents processeurs. La figure (4.16) montre les performances parallèles obtenues pour cette étape en fonction du type de stockage utilisé sur le problème test. Une répartition de type SINGLE du travail sur les différents processeurs a donné les meilleurs résultats en terme de répartition de charge.



Stockage MORSE (14.2e9 cycles d'horloge).
Overhead total: 8.5%.
Fraction parallèle: 99.9%.

Stockage MORSE redondant (2.4e9 cycles d'horloge).
Overhead total: 9.8%.
Fraction parallèle: 89.4%.

Fig. 4.16: Accélération pour la construction de la matrice de Cholesky pour le problème test.

Notons que la construction de la matrice de préconditionnement avec un stockage MORSE requiert un temps de calcul beaucoup plus important (nombre de cycles d'horloge) qu'avec un stockage MORSE redondant. Ceci est dû au temps nécessaire à la recherche des termes non nuls d'une même colonne, phase dont on s'affranchit lors de l'utilisation d'un stockage MORSE redondant. En fait, comme il sera présenté au paragraphe suivant, les performances vectorielles du code sont alors médiocres pour un stockage MORSE.

La matrice de Cholesky incomplète est construite lors de l'appel du solveur. A chaque itération, les systèmes $L.y = b$ puis $L^t.x = y$ sont résolus de la façon suivante afin de déterminer le nouveau vecteur résidu (algo. 4.2):

résolution du système triangulaire inférieur
pour i = 1 à nombre de lignes (n)

$$yi = (bi - \sum_{k=1}^{k=i-1} Lik\,yk) / Lii$$

fin

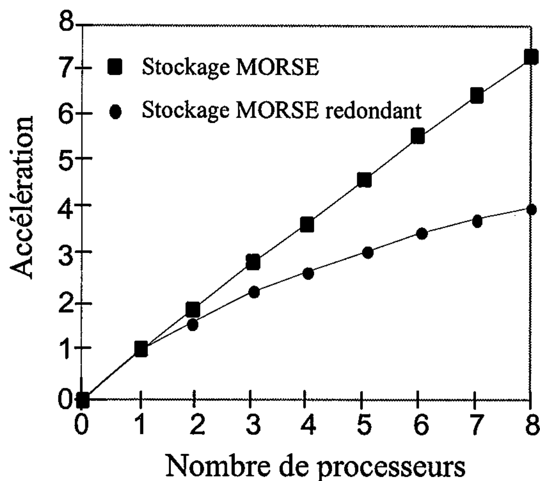
résolution du système triangulaire supérieur
pour i = nombre de lignes (n) à 1

$$xi = (yi - \sum_{k=i+1}^{k=n} Lki\,xk) / Lii$$

fin

Algo. 4.2: Descente-remontée pour la résolution du système.

La résolution du système triangulaire inférieur sera notée, par la suite, descente et celle du système supérieur, remontée. Lors de la descente ou de la remontée, le calcul du résultat de la ligne i nécessite la connaissance des résultats des lignes antérieures. De ce fait, cet algorithme est implicitement séquentiel (dépendance arrière). La parallélisation de cette étape se fait donc en répartissant les itérations de la boucle réalisant $\sum_{k=1}^{k=i-1} Lik\,yk$ pour la descente. Pour chaque produit à réaliser, la boucle est éclatée si le nombre d'itérations à effectuer (i-1-k) est au moins égal au nombre de processeurs alloués à l'application. La parallélisation de la remontée est effectuée de la même manière. Le stockage MORSE redondant pour la matrice de préconditionnement s'avère alors d'une grande efficacité pour la recherche des termes Lki.



Stockage MORSE (1.8e11 cycles d'horloge).
Overhead total: 9.2%.
Fraction parallèle: 99.9%.

Stockage MORSE redondant (3.7e9 cycles d'horloge).
Overhead total: 80%.
Fraction parallèle: 80.5%.

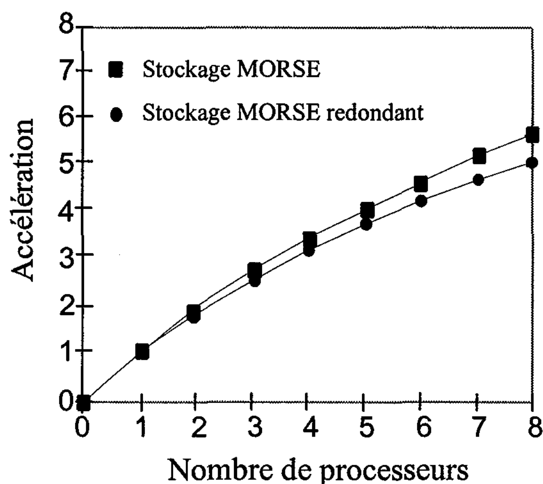
Fig. 4.17: Accélération pour la montée-descente pour le problème test.

De la figure (4.17) il ressort que l'overhead introduit par l'éclatement de la boucle, avec un stockage MORSE, est négligeable à cause du temps requis pour la recherche des termes Lki. Cette remarque n'est pas valable pour le stockage MORSE redondant. De ce fait, les performances parallèles s'en trouvent diminuées. Par conséquent, les performances vectorielles de cette étape avec un stockage MORSE redondant sont beaucoup plus élevées et le temps CPU s'en trouve d'autant diminué (nombre de cycles d'horloge).

IV 2 8. Analyse des performances globales du code

IV 2 8 1. Préconditionnement diagonal

Les performances globales du code muni du préconditionnement diagonal pour les deux types de stockage sont présentées sur la figure (4.18) sur le problème test.



Stockage MORSE (20 Mflops, 262s).
Remplissage des registres vectoriels: 8
Overhead total: 2.4%.
Fraction parallèle: 94.9%.

Stockage MORSE redondant (40.2 Mflops, 187s).
Remplissage des registres vectoriels: 15
Overhead total: 2.5%.
Fraction parallèle: 93.8%.

Fig. 4.18: Accélérations globales (préconditionnement diagonal) pour le problème test.

Certaines parties du code n'ont pu être ou ont été très peu parallélisées, soit à cause de leur nature (écritures et lectures de fichiers), soit à cause de leurs algorithmes (assemblage symbolique). La figure (4.19) montre la répartition du temps CPU lors de l'exécution du code sur le problème test et pour les deux types de stockage. La partie assemblage regroupe les étapes d'assemblage symbolique, de décomposition des éléments volumiques et d'introduction des conditions aux limites sur les cep.

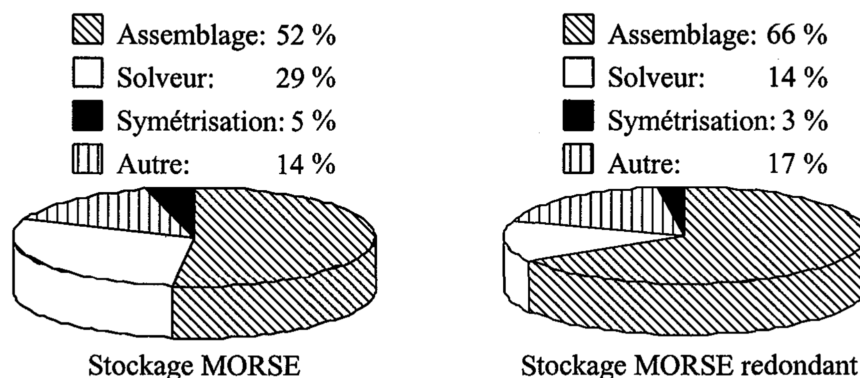
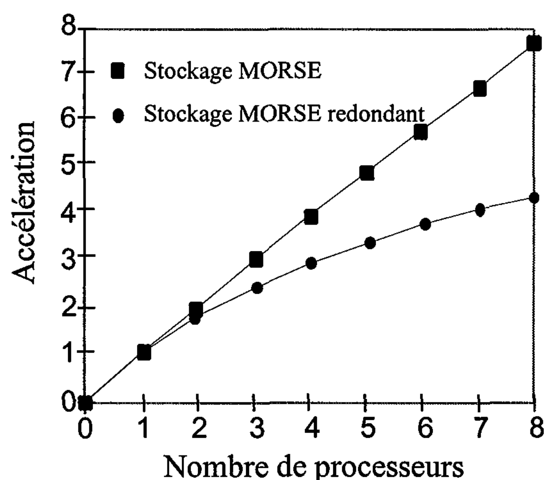


Fig. 4.19: Répartition du temps CPU (préconditionnement diagonal).

L'utilisation du stockage MORSE redondant n'a que très peu d'influence sur les phases d'assemblage, de symétrisation ainsi que sur la partie dénotée autre. En fait, la diminution de la proportion du temps consommé par le solveur avec un stockage MORSE redondant est induite par l'augmentation de ses performances vectorielles.

IV 2 8 2. Préconditionnement de Cholesky incomplet

Les performances globales du code muni du préconditionnement de Cholesky incomplet pour les deux types de stockage sont présentées sur la figure (4.20) pour le problème test.



Stockage MORSE (2.3 Mflops, 3544s).

Remplissage des registres vectoriels: 9.

Overhead total: 2.7%.

Fraction parallèle: 99.5%.

Stockage MORSE redondant (7 Mflops, 886 s).

Remplissage des registres vectoriels: 11.

Overhead total: 14.3%.

Fraction parallèle: 87.7%.

Fig. 4.20: Accélérations globales (préconditionnement de Cholesky incomplet) pour le problème test.

Le taux de remplissage moyen des registres vectoriels, sur la totalité du temps d'exécution du programme, est plus élevé avec un stockage de type MORSE redondant; il en est de même en ce qui concerne les performances vectorielles du code (quel que soit le type de

préconditionnement). Ceci est dû au fait que la taille du problème est trop faible pour l'utilisation de cette méthode et, par conséquent, que les flux de données ne sont pas assez longs pour obtenir des performances vectorielles convenables. La figure (4.21) montre la répartition du temps CPU pour le traitement du problème test avec le GC muni du préconditionnement de Cholesky incomplet pour le stockage MORSE et MORSE redondant.

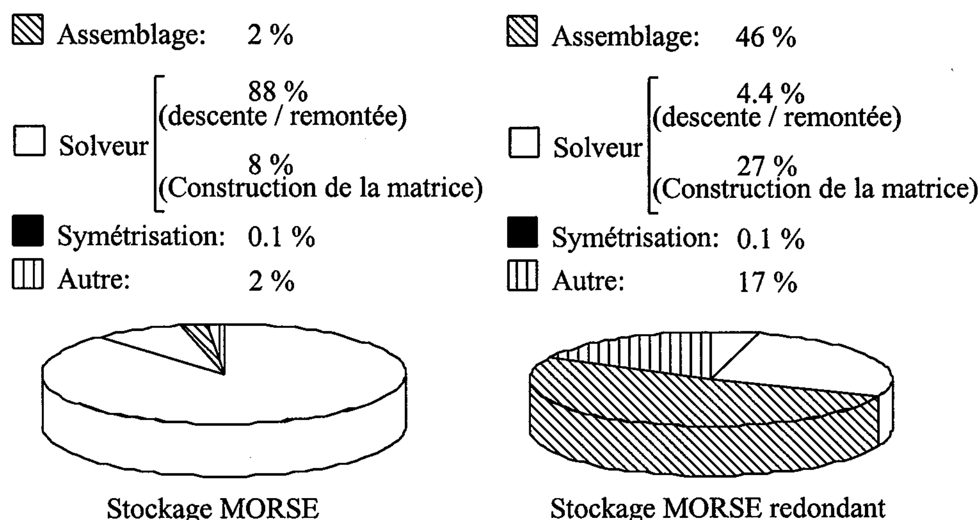


Fig. 4.21: Répartition du temps CPU (préconditionnement de Cholesky incomplet).

A partir des temps de calcul obtenus avec ces quatre méthodes (figures 4.18 et 4.20), le préconditionnement diagonal associé à un stockage MORSE redondant s'avère être la méthode la plus efficace. Comme il sera montré au paragraphe suivant, il faut relativiser cette affirmation et préciser qu'elle n'est valable que pour un problème de petite taille (10000 noeuds) maillé avec des hexaèdres du premier ordre. Le tableau (4.1) résume les performances obtenues pour le problème test pour une résolution sur 8 processeurs.

	Stockage	Temps total	*Mflops*	Mémoire	Itérations
Diagonal	Stockage MORSE	262 s	20	9 Mw	224
	Stockage MORSE_red	187 s	40.2	9 Mw	224
Cholesky	Stockage MORSE	3544 s	2.3	9 Mw	76
	Stockage MORSE_red	886 s	7	16 Mw	76

Tab. 4.1. Performances pour le problème test.

Etant donné que la matrice EF est dissymétrique au départ, l'utilisation d'un stockage MORSE redondant ne requiert pas d'espace mémoire supplémentaire si un préconditionnement diagonal est utilisé.

Le préconditionnement de Cholesky incomplet ne peut être utilisé que couplé à un stockage MORSE redondant à cause de des performances vectorielles obtenues. L'espace mémoire requis pour le code est alors pratiquement multiplié par 2 car l'obtention de bonnes performances vectorielles nécessite la construction de L et L^t.

IV 3. Problème de grande taille (diffraction par un cylindre)

Le CRAY C98 permet de traiter théoriquement des problèmes allant jusqu'à 350000 noeuds avec un stockage MORSE redondant et un préconditionnement de Cholesky incomplet (le système de *batch* limite les programmes à 400Mw). Afin de pouvoir comparer les méthodes, un même problème a dû être résolu par chacune d'entre elles. Cette opération est très coûteuse en temps CPU et c'est pour cette raison que le problème test comporte seulement 40000 noeuds (hexaèdres du premier ordre). Le tableau (4.2) présente les résultats obtenus pour une résolution sur 8 processeurs.

Plus loin dans cette étude, seront présentés des problèmes de grande taille mais qui n'ont été traités que par la méthode la plus performante (stockage MORSE redondant et préconditionnement de Cholesky incomplet).

	Stockage	Temps/proc.	*Mflops*	Mémoire	Itérations
Diagonal	Stockage MORSE	145 s	33	42 Mw	301
	Stockage MORSE_red	107 s	72	42 Mw	301
Cholesky	Stockage MORSE	Trop consommateur de temps CPU			
	Stockage MORSE_red	100 s	61	73 Mw	84

Tab. 4.2. Performances pour un problème de 40000 noeuds.

Le préconditionnement de Cholesky incomplet couplé au stockage MORSE n'a pu être utilisé du fait de ces mauvaises performances vectorielles. Pour cet exemple, le préconditionnement de Cholesky incomplet couplé à un stockage MORSE redondant s'avère

être la méthode la plus performante. L'augmentation de la taille du problème a entraîné une croissance des performances vectorielles totales du code à un niveau très acceptable. Cette tendance ne peut que se confirmer pour la résolution de problèmes encore plus grands.

IV 4. Conclusion

Cette étude montre la nécessité d'une intervention réfléchie de la part du programmeur afin d'obtenir de bonnes performances parallèles et vectorielles sur cette machine. En effet, le travail réalisé par le pré-processeur FPP s'avère limité en terme de parallélisation. Néanmoins, les outils d'analyse disponibles sur le CRAY C98 permettent à l'utilisateur, via des techniques de *microtasking*, d'optimiser un code. Cette démarche permet de garder le contrôle des algorithmes parallèles.

L'approche qui consiste à porter un code séquentiel sur ce type de machine sans pour autant le restructurer, éventuellement le réécrire, montre que de bonnes performances parallèles peuvent être obtenues. De ce fait aucun calcul supplémentaire dû au parallélisme (pré-processing pour une décomposition de domaine, ...) n'est introduit. En revanche, les performances monoprocesseur du code restent médiocres malgré les efforts réalisés pour améliorer la vectorisation (PROFVIEW). Ceci est dû au fait que les matrices EF sont creuses et, par conséquent, que les flots de données sont courts. Ceci est dû aussi et surtout au type de stockage (MORSE) utilisé. En effet, les opérations réalisées avec les données relatives à une ligne de la matrice peuvent être vectorisées, lors de la multiplication matrice-vecteur par exemple. Pour effectuer les mêmes opérations sur la ligne suivante, il faut accéder aux tableaux 'N_p_colonne', 'Rang' puis 'Terme', ce qui à pour effet de 'casser' la vectorisation.

Il existe plusieurs techniques de stockage permettant de palier à ce problème. Beaucoup de travaux, présents dans la littérature [35-39], portent sur des applications opérant sur des calculateurs munis de processeurs essentiellement vectoriels. Toutefois, d'après les différentes expérimentations effectuées jusqu'ici, il apparaît que seule l'utilisation du stockage MORSE redondant permet d'obtenir des performances vectorielles acceptables sans pour autant augmenter le besoin d'espace mémoire dans des proportions qui empêchent le traitement de géométries réalistes.

CHAPITRE V
IMPLANTATION SUR LA FERME DE STATIONS DEC ALPHA

V 1. Introduction

Comme il a été dit précédemment, le mode de programmation choisit (SPMD) impose une granulité assez grosse. Sur ce type de machine il est important d'essayer de minimiser le nombre de passages de messages car ceux-ci introduisent un surcoût très important (*overhead*). En effet, il est impossible de superposer la plupart des graphes de communications des différents algorithmes sur une configuration matérielle en anneau.

Les algorithmes de chaque étape ont été repensés pour répondre aux exigences citées précédemment. Le code a donc été entièrement réécrit. Le choix du langage s'est porté sur le C++ pour tirer profit de la programmation orientée objet: encapsulation des données, surdéfinition d'opérateurs, ... [44], [45]. L'expérience acquise à l'heure actuelle montre qu'il a fallu renoncer à un certain nombre de facilités relatives au C++, notamment au niveau syntaxique, afin d'obtenir de bonnes performances monoprocesseur en terme de Mflops/s.

V 2. Code parallèle

Le programme commence donc par s'exécuter sur le processeur originel de la machine virtuelle, puis il démarre des copies de lui-même sur les autres stations de la machine virtuelle. Une tâche est donc attribuée par processeur, et chaque tâche va rentrer dans un groupe. Le numéro d'entrée dans ce groupe servira à la répartition de charge, à identifier le processeur maître et les esclaves lors du fonctionnement M/E du solveur ou encore aux synchronisations collectives. Chaque copie du programme commence par effectuer une lecture du fichier de données (lecture parallèle). Cette technique requiert donc l'allocation temporaire de la mémoire correspondante sur chaque station. Contrairement aux méthodes, plus économiques en terme d'espace mémoire, qui consistent à lire les données au fur et à mesure des besoins, cette méthode permet d'éviter les goulots d'étranglement au niveau des E/S [46].

V 2 1. Assemblage

Afin de tirer profit de l'espace mémoire disponible sur un tel calculateur, la matrice EF doit être répartie sur tous les noeuds de calcul. A cet effet, la plupart des travaux rencontrés dans la littérature sont basés sur des techniques de décomposition de domaines [47-52], [63, 64], [79-81]. Ces approches apparaissent comme les plus naturelles sur un calculateur à mémoire distribuée. En revanche, elles introduisent des calculs supplémentaires pour effectuer cette décomposition. De plus, dans la plupart des problèmes que nous sommes amenés à résoudre, il n'existe qu'un seul domaine physique (diffraction par un cep, guide d'onde). D'autres techniques peuvent être trouvées, notamment certaines mettant en jeu des fonctionnements M/E: assemblage par contributions élémentaires [53], ou fragmentation de boucles et distribution du travail [54]. Ces méthodes n'optimisant pas l'occupation mémoire, elles ne sont donc pas appropriées.

Nous avons donc choisi d'assembler la matrice EF par degrés de liberté [55], [63]. Cette technique consiste à rechercher pour chaque noeud tous les EF l'incluant. Les trois lignes, qui correspondent aux trois coordonnées relatives à ce noeud, sont construites en une seule fois et ensuite compressées au travers d'un seul stockage MORSE (partie inférieure et supérieure de la matrice EF). Il n'est en effet pas possible de stocker la matrice EF à l'aide de 2 stockages MORSE afin de faciliter l'addition avec la matrice transposée comme dans le code séquentiel car, lors de la construction d'une ligne, la structure de la colonne correspondante n'est pas encore connue.

L'assemblage par degrés de liberté a demandé une restructuration des données. Dans le code séquentiel, la configuration du maillage se présente sous la forme de trois tableaux tab1, tab2, tab3 (fig. 5.1): tab1 comprend la liste des numéros des noeuds; tab2 contient le rang dans tab1 du premier noeud d'un élément et tab3 le rang dans tab1 du dernier noeud d'un élément. A partir de ces tables, un nouveau tableau a été élaboré (Nouvtab): chaque ligne correspond à un noeud et sur chaque ligne sont stockés tous les numéros des éléments incluant le noeud considéré. L'assemblage se fait donc en parcourant ce tableau à deux dimensions et en allant chercher dans les autres tableaux (tab1, tab2, tab3) les noeuds correspondant à chaque élément.

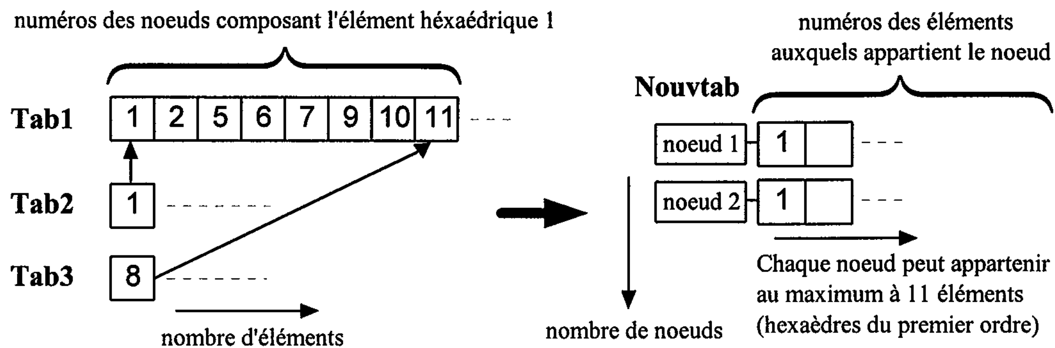


Fig. 5.1: Structure des données.

Aucun calcul préalable à l'assemblage n'étant requis contrairement aux techniques de décomposition de domaine ou d'assemblage symbolique, chaque processeur se voit alors attribuer, au départ de l'assemblage, un certain nombre de lignes. Ces opérations peuvent donc se dérouler indépendamment sur chaque noeud de calcul. Etant donné que la matrice est 'bande', la répartition de la charge peut se faire de manière statique au départ de l'assemblage en fonction du numéro du processeur dans la machine virtuelle. Elle sera pratiquement parfaite si on néglige la répartition des éléments surfaciques dans la matrice EF intervenant dans la phase d'introduction des CLA et CL. Cette démarche, qui est en fait une décomposition par le biais de la numérotation des noeuds, a une incidence sur les possibilités de choix des algorithmes des étapes suivantes, notamment pour la résolution du système d'équation.

La figure (5.5) montre que l'accélération obtenue avec cette méthode pour la phase d'assemblage sur le problème test est optimale.

V 2 2. Traitement des conditions aux limites sur les cep et les plans de symétrie

Pour cette phase, la même technique qu'au chapitre précédent est utilisée. Par conséquent, une modification globale de la matrice EF est requise. Chaque processeur effectue les modifications intervenant sur sa partie de la matrice. Evidemment, la répartition de la charge à ce niveau dépend de la distribution des éléments surfaciques, positionnés sur les cep ou sur les plans de symétries, dans la matrice EF. Par la suite, plusieurs passages de messages sont nécessaires pour maintenir le caractère symétrique de la matrice.

A titre d'exemple, un problème traité sur 2 processeurs, dans lequel un noeud numéroté i est placé sur un plan de symétrie, est présenté figure (5.2). En vertu de (2.46), la ligne numéro i et la colonne i de la matrice EF doivent être forcées à 0 et le terme diagonal à 1.

Dans le cas de symétries et d'antisymétries, seuls les numéros des colonnes à forcer à 0 sont envoyés. Dans le cas d'une résolution en champ **E**, pour le traitement des CL sur les cep, les numéros des colonnes ainsi que les coefficients sont transmis.

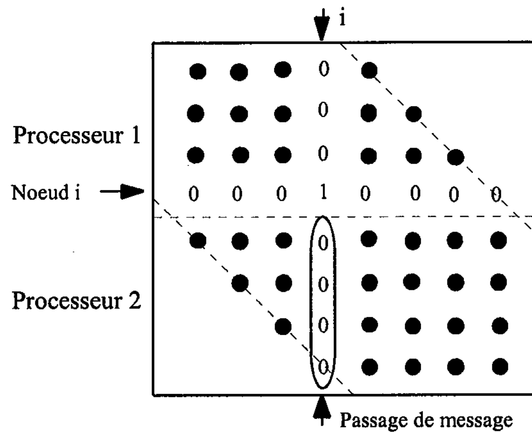


Fig. 5.2: Passage de message pour le traitement des symétries et des conditions aux limites.

V 2 2 1. Minimisation du nombre de passages de messages

Afin de minimiser le nombre de passages de messages, seuls les voisins immédiats d'un noeud de calcul reçoivent des messages de celui-ci. Ceci pose un problème quand le nombre de lignes par processeur n'est plus assez important pour garantir que les colonnes à forcer à 0 ne se prolongent pas dans la partie de la matrice détenue par le processeur suivant. Le même exemple que dans la figure (5.2) est repris figure (5.3) avec 4 processeurs pour illustrer un cas défaillant. Il est évident que ce problème est éludé quand chaque processeur dispose d'un nombre de lignes au moins égal à la moitié de la largeur de bande. Ceci dépend du type d'élément et de la taille du problème.

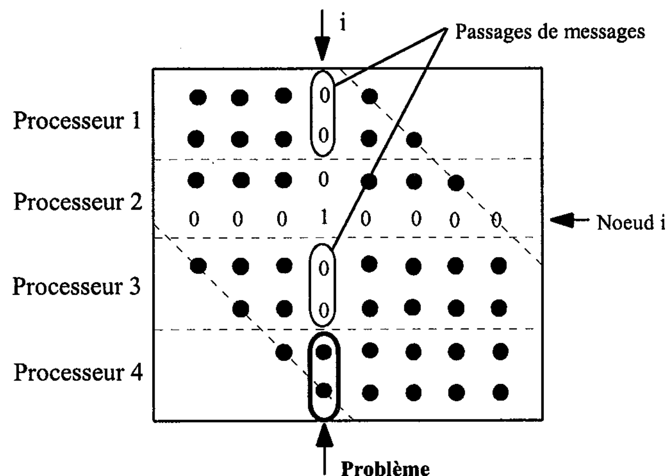


Fig. 5.3: Limitation du nombre de passages de messages.

La figure (5.5) montre que l'accélération obtenue avec la méthode de minimisation du nombre de passages de message pour la phase de traitement des conditions aux limites sur le problème test pour une résolution en champ **H** est pratiquement optimale. Toutefois ces performances se dégradent notoirement pour une résolution en champ **E**, où la CL sur les cep doit être imposée. L'incidence de cette étape sur les performances globales du code est négligeable, même lors d'une résolution en champ **E**, car elle représente moins de 2 % du temps CPU total (fig. 5.8). Par conséquent aucune autre technique n'a été essayée.

V 2 3. Symétrisation du système matriciel

Le système matriciel à résoudre étant faiblement dissymétrique et pour les mêmes raisons que celles citées précédemment, celui-ci est symétrisé par l'addition de la matrice EF avec sa transposée. Cette opération nécessite des passages de messages comme le montre la figure (5.4) sur un exemple à 2 processeurs. Sur chaque processeur, une boucle opère jusqu'à la diagonale sur les lignes de la partie de la matrice contenue sur ce même processeur. Pour chaque terme non nul, le terme transposé est recherché s'il est stocké sur le processeur. Les termes situés après la diagonale ayant un rang supérieur au numéro de la dernière ligne du paquet contenu par le processeur sont stockés pour envoi vers le processeur suivant.

Cette technique requiert une allocation supplémentaire d'espace mémoire correspondant à la moitié de celui requis par la matrice EF. En effet, toute la matrice EF (partie inférieure et supérieure) étant stockée en une seule fois, il est impossible de se servir de l'espace mémoire déjà alloué pour la matrice EF afin de stocker le résultat de l'addition. Afin de prévoir le volume d'espace mémoire qu'il faut alors allouer, un comptage des termes se situant en dessous et sur la diagonale à été réalisé lors de la phase d'assemblage. Ce surcoût d'espace mémoire temporairement alloué n'est pas pénalisant car il correspond à peu près à celui nécessaire, lors de l'assemblage, au stockage temporaire des données issues du mailleur.

Notons qu'un stockage MORSE redondant aurait pu être utilisé pour faciliter la construction de la matrice de Cholesky: toute la matrice serait alors restée stockée en mémoire après sa symétrisation. Cela aurait aussi eu pour effet de diminuer l'espace mémoire requis pendant l'addition avec la matrice transposée, mais aurait augmenté celui nécessaire au préconditionnement de Cholesky. De plus, étant donné que seuls les termes de la partie supérieure de la matrice devant être additionnés avec des termes stockés sur un autre

processeur sont envoyés, l'utilisation d'un stockage MORSE redondant aurait entrainé une augmentation considérable de la grosseur et du nombre de messages à envoyer.

La même méthode de minimisation de passages de messages qu'au paragraphe précédent est utilisée, c'est à dire que chaque processeur n'envoie des messages qu'à ses voisins immédiats.

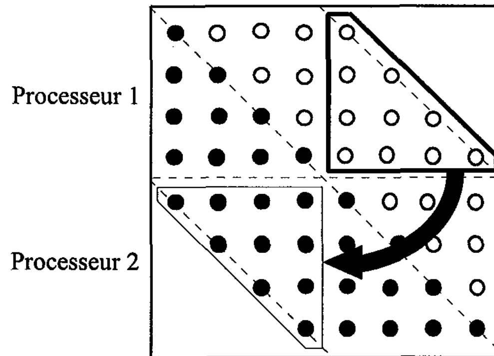


Fig. 5.4: Passage de message pour la symétrisation du système matriciel.

La figure (5.5) montre que les performances obtenues pour la phase de symétrisation du système matriciel sur le problème test sont médiocres car l'*overhead* introduit par le parallélisme est supérieur au gain apporté par ailleurs. Ici encore, le pourcentage de temps consacré à cette opération étant faible (Fig. 5.8), l'incidence sur les performances parallèles globales du code sont moindres.

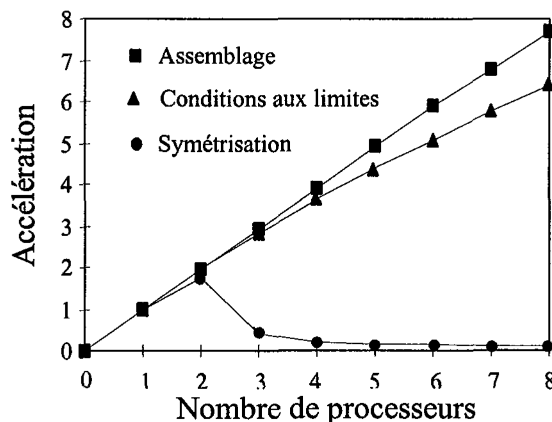


Fig. 5.5: Accélérations pour les phases d'assemblage, de traitement des conditions aux limites et de symétrisation pour le problème test (résolution en champ H).

Notons qu'à la fin de ces trois phases et avant l'appel au solveur, le deuxième membre du système matriciel doit être concaténé sur chaque processeur car chacun d'eux en détient une partie. La méthode utilisée est la même que pour la concaténation des vecteurs résidus partiels

nécessaire lors de la résolution avec un préconditionnement diagonal. Cette technique est décrite en détails dans le paragraphe qui suit.

V 2 4. Résolution du système linéaire d'équations

La résolution du système matriciel ne peut se faire que sur l'ensemble de la matrice. Contrairement aux techniques de décompositions de domaines qui opèrent le plus souvent par sous-domaines puis sur l'ensemble du système [47-52, 72-75], un assemblage par degré de liberté impose cette restriction. De plus, les méthodes [1], [28-29] s'appuyant sur le caractère creux des matrices EF pour déterminer des équations indépendantes entre elles dans le système matriciel à résoudre, n'ont pu être utilisées car elles demandaient une permutation des lignes. En effet, après un assemblage par degrés de liberté, la matrice EF est répartie sur tous les processeurs, et cette technique demanderait donc trop de passages de messages.

Le Gradient Conjugué (GC) [34], [56-58] ainsi que QMR (*Q*uasi-*M*inimum *R*ésidu basé sur l'algorithme de Lanczos) ont donc été implantés [59-62]. QMR permet une convergence sans oscillation et le GC est la méthode la plus utilisée pour ce type de problèmes [annexe 3]. A cause du caractère itératif, donc séquentiel, de ces méthodes, le choix de la granulité de parallélisme s'est porté sur les opérations matricielles nécessaires à l'obtention du vecteur résidu calculé à chaque itération. Les différents algorithmes parallèles mis en oeuvre à cet effet dépendent du type de préconditionnement.

V 2 4 1. Préconditionnement diagonal

La construction de la matrice de préconditionnement consiste en l'inversion des termes diagonaux de la matrice EF qui sont stockés dans un vecteur dupliqué sur chaque processeur. Le préconditionnement est donc réalisé par une multiplication vecteur-vecteur. Cette technique de préconditionnement ne nécessite aucun passage de message et ne requière que très peu d'espace mémoire supplémentaire. Par contre, elle génère un préconditionnement de mauvaise qualité.

La mise en oeuvre de méthodes telles que le GC ou QMR munis d'un préconditionnement diagonal impose une multiplication matrice-vecteur pour la détermination du vecteur résidu à chaque itération. La parallélisation réside donc dans ce produit matrice-vecteur. La matrice EF, à ce stade de traitement du problème, est symétrique: seule sa partie inférieure est donc encore stockée. Au départ, le vecteur initial à multiplier est dupliqué sur tous les processeurs.

Par conséquent, chaque noeud de calcul va multiplier sa partie de la matrice par le vecteur et obtenir ainsi un vecteur résidu partiel. La concaténation de ces vecteurs résidus partiels peut être effectuée de deux manières:

- Sur tous les processeurs en mode SPMD, c'est à dire que chaque processeur envoie à tous les autres son résultat partiel, puis chacun effectue la concaténation au fur et à mesure des réceptions. Si Nproc est le nombre de processeurs participant à l'opération, ce mode de fonctionnement impose ($(Nproc.)^2$ - Nproc.) passages de messages à chaque itération. Le volume de données transitant par le réseau s'avère très grand (fig. 5.23). De plus, le graphe de communications mis en jeu, chaque processeur communiquant avec tous les autres, ne se superpose absolument pas au réseau physique en anneau.

- Sur un seul processeur en mode Maître-Esclave, c'est à dire que tous les processeurs esclaves envoient leurs résultats partiels au maître qui effectue la concaténation puis renvoie le vecteur résultat total aux esclaves. Ce mode de fonctionnement diminue le nombre de passages de messages à (Nproc-1) x 2 (fig. 5.22). Le graphe de communications est donc un peu plus avantageux que dans le mode SPMD. En revanche, lorsque le maître calcule le vecteur résidu total, les esclaves sont en attente du résultat. Le temps d'attente *idle* (attente) est donc supérieur (fig. 5.18 et 5.19). Notons que pour les deux types de fonctionnement (SPMD et M/E), seuls les termes non nuls des vecteurs sont envoyés afin de réduire la taille des messages. Ceci est réalisé en *packtant* dans le message à envoyer le rang du premier terme non nul, le nombre de termes et ensuite les termes eux mêmes. Le *dépacktage* doit se faire dans le même ordre.

La figure (5.18) présente une trace PARAGRAPH pour le problème test traité sur 4 processeurs avec le GC et pour un fonctionnement M/E (le processeur 0 est le maître), la figure (5.22) le nombre de messages correspondant et la figure (5.26) une moyenne, sur la totalité du temps de fonctionnement, de l'état de chaque processeur.

Les figures (5.19), (5.23) et (5.27) présentent les même résultats pour un fonctionnement SPMD.

Le mode M/E peut apparaître plus avantageux d'après les figures (5.22) et (5.23) car le nombre de passages de messages est inférieur. De plus, la moyenne de l'état des processeurs (fig. 5.26 et 5.27) met en évidence une meilleure occupation des processeurs sur la totalité du temps d'exécution du code pour un mode M/E. Malgré cela, le mode SPMD reste plus performant en terme de temps CPU consommé car le temps d'attente des processeurs esclaves

dans le mode M/E est simultané pour tous ceux-ci et par là-même très pénalisant. De plus les messages renvoyés par le processeur maître sont de longueur égale à la taille de la matrice, donc plus grands que dans le mode SPMD où seuls les termes non nuls sont envoyés.

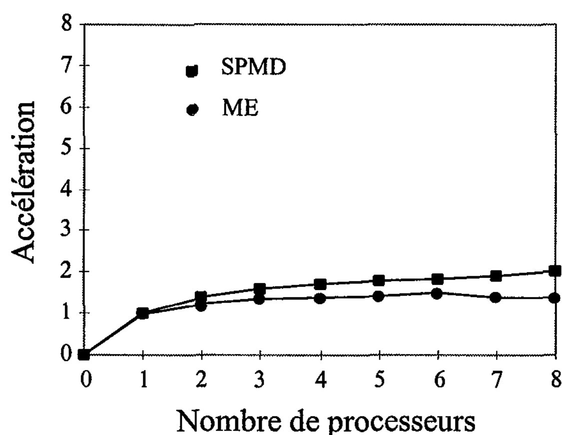


Fig. 5.6: Accélérations du solveur pour les 2 modes de fonctionnement pour le problème test.

La figure (5.6) montre que les performances du GC pour le problème test et pour les deux modes de fonctionnement (SPMD et M/E) sont médiocres. Par conséquent, les performances globales du code sont fortement pénalisées par la mauvaise accélération du solveur du fait que celui-ci représente la majorité du temps CPU utilisé (fig. 5.8). Cette même figure montre la prédominance du temps consacré aux passages de messages. En effet, pour un même problème, quand le nombre de processeurs passe de 4 à 8, la proportion du temps consommé par le GC augmente considérablement. La répartition est à peu près la même pour un fonctionnement SPMD et M/E, et ceci quel que soit le solveur utilisé.

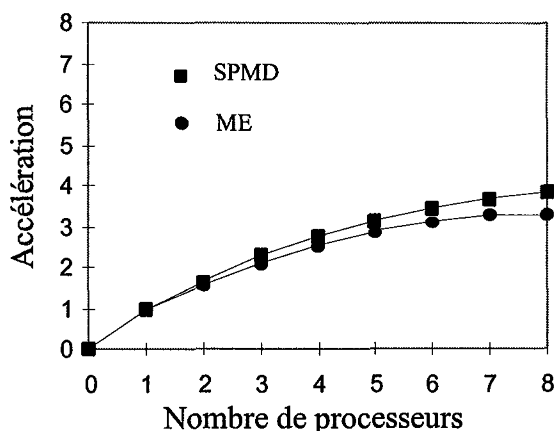


Fig. 5.7: Accélérations globales du code avec un préconditionnement diagonal pour le problème test.

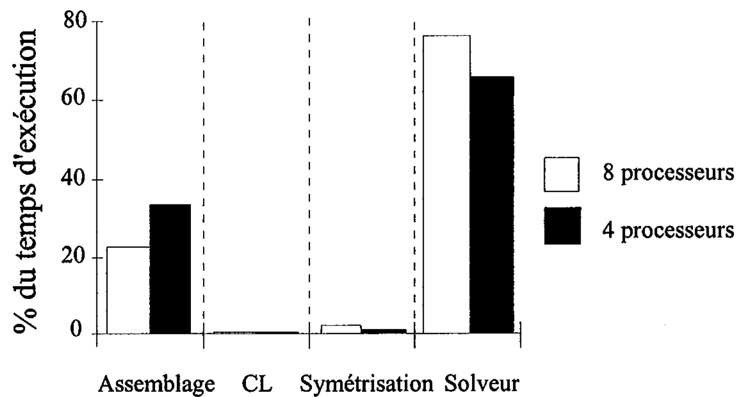


Fig. 5.8: Répartition du temps CPU avec un solveur muni d'un préconditionnement diagonal pour le problème test.

D'après les figures (5.6) et (5.8), un effort est à faire au niveau du solveur pour le rendre performant. Cette constatation amène plusieurs réflexions:

- Il faudrait augmenter la granulité de parallélisme de manière à diminuer le nombre de passages de messages. Une telle démarche s'avère impossible sans que soient revus les algorithmes des autres phases du code et notamment l'assemblage.

- Le code pourrait être porté sur une architecture plus adaptée au graphe de communications mis en jeu. Ceci fait l'objet du prochain chapitre.

- D'autres types de préconditionnement pourraient être employés afin d'accélérer la convergence du solveur de manière à diminuer son influence sur les performances globales du code. Comme il est dit en introduction, l'implantation d'autres types de préconditionnement implique la parallélisation d'autres algorithmes. De ce fait, il faut comparer d'une part la convergence des deux méthodes (GC et QMR) avec les différents préconditionnements et, d'autre part, les performances parallèles, c'est à dire le temps CPU consommé pour le traitement d'un problème pour un nombre donné de processeurs.

La figure (5.9) présente la convergence des deux types de solveurs munis du préconditionnement diagonal (GC et QMR) sur le problème test. Avec ce préconditionnement, le nombre d'itérations reste le même quel que soit le nombre de processeurs utilisés. Malgré la convergence 'douce' de la méthode QMR, le GC reste plus efficace sur ce type de problème (taille et type d'éléments) car QMR requiert plus de passages de messages [annexe 3].

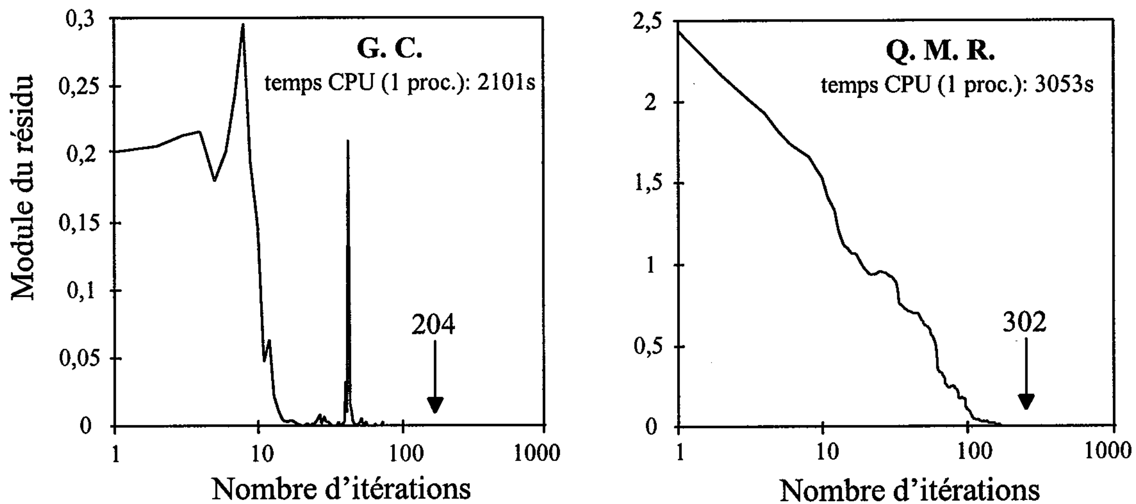


Fig. 5.9: Convergence du GC et de QMR munis d'un préconditionnement diagonal sur le problème test.

V 2 4 2. Préconditionnement de Cholesky incomplet

Le GC muni du préconditionnement de Cholesky incomplet requiert aussi une multiplication matrice-vecteur. Celle-ci est donc parallélisée de la façon décrite au paragraphe précédent. L'utilisation du préconditionnement de Cholesky incomplet impose la parallélisation d'autres algorithmes: construction de la matrice de préconditionnement et descente-remontée pour la résolution du système. Etant donné que la matrice de préconditionnement est construite par colonnes (algo. 4.1), et que la structure de la matrice est connue a priori, il est possible de construire L et L^t simultanément (ceci est assimilable à un stockage MORSE redondant). L'accès aux termes de L par colonnes se fera en accédant aux termes de L^t par ligne, ce qui accélérera la phase de remontée. L'espace mémoire requis est de 1.5 fois la taille de A avant sa symétrisation. Le même espace ayant été alloué pour la symmétrisation du système matriciel puis partiellement détruit, cette allocation n'est pas pénalisante.

La construction de la matrice de Cholesky, sur une architecture à mémoire distribuée, impose un grand nombre de passages de messages et de synchronisations car la construction du terme Lij nécessite la connaissance de la ligne i, du terme Ljj et de la ligne j. Si cette dernière n'est pas stockée sur le processeur considéré, un passage de message est nécessaire. En fait, l'algorithme opère de la manière suivante (algo. 5.1): Le processeur 1 qui a en mémoire la partie supérieure de la matrice débute le calcul. Dès qu'il a fini de calculer une ligne, il l'envoie aux autres processeurs qui pourront alors débuter leurs calculs sur les

colonnes. Une fois que le processeur 1 a terminé d'assembler sa partie de la matrice, le processeur 2 entame le même processus et envoie les lignes qu'il calcule. Les processeurs de rang supérieur à 2 s'en serviront pour assembler les termes de la colonne correspondante et le processeur 1 stockera les valeurs reçues dans la matrice L^t, et ainsi de suite ... La figure (5.10) illustre cette démarche sur une matrice 4x4 répartie sur 3 processeurs. Les différentes étapes numérotées de 1 à 5 représentent les opérations réalisées en parallèle. Le graphe de communications mis en jeu consiste en l'envoi de messages d'un processeur vers tous les autres.

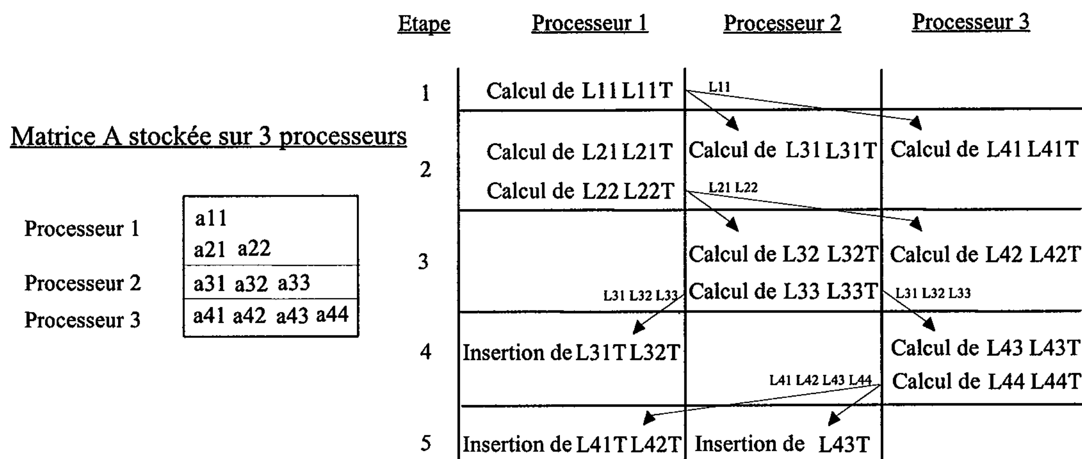


Fig. 5.10: Construction de la matrice de Cholesky incomplète.

```
boucle sur les processeurs
pour p = 1 à nombre de processeurs
        test pour déterminer quelle partie du code effectuer sur chaque processeur
        si p = numéro du processeur
                boucle sur les lignes du processeur
                pour j = 1 à nombre de lignes du processeur
                        initialisation du vecteur à émettre (longueur j)
                        calcul de Σ_{k=1}^{j-1} (Ljk)^2
                        stockage des Ljk dans le vecteur à émettre
                        calcul du terme diagonal Ljj
                        stockage de Ljj dans le vecteur à émettre
                        émission du vecteur aux autres processeurs
                        calcul des Lij sur ce processeur (i = j+1 à nb. de lignes stockées sur ce proc.)
                fin
        fin
        test pour déterminer quelle partie du code effectuer sur chaque processeur
        si p > numéro du processeur
                pour i = 0 à numéro de la première ligne de ce paquet
                        réception bloquante du vecteur
                        boucle sur les lignes du processeur
                        pour i = 1 à nombre de lignes stockées processeur
                                calcul des termes de Cholesky, si ils existaient déjà, de
                                la colonne j à l'aide des termes du vecteur réceptionné
                        fin
                fin
        fin
        test pour déterminer quelle partie du code effectuer sur chaque processeur
        si p < numéro du processeur
                pour i = 0 à (nb. de lignes total-(num. 1ère ligne+nb. de lignes stockées sur ce proc.))
                        réception bloquante du vecteur
                        insertion dans la matrice transposée de Cholesky des termes non nuls
                fin
        fin
fin
```

Algo. 5.1: Construction de la matrice de Cholesky sur une architecture à mémoire distribuée.

La matrice de Cholesky incomplète est construite lors de l'appel au solveur. A chaque itération les systèmes $L.y = b$ puis $L^t.x = y$ sont résolus. Lors de la descente ou de la remontée, le calcul du résultat de la ligne i nécessite la connaissance des résultats des lignes antérieures. Etant donné que chaque processeur détient une partie du résultat, cet algorithme est implicitement séquentiel. En effet, les processeurs de rang supérieur à 1 ne pourront débuter leurs calculs que lorsque le processeur 1 aura terminé le sien et envoyé son résultat partiel aux autres, et ainsi de suite. L'algorithme parallèle utilisé est le suivant (algo. 5.2):

boucle sur les processeurs
pour p = 1 à nombre de processeurs

test pour déterminer quelle partie du code effectuer sur chaque processeur
si p = numéro du processeur
Résolution du système triangulaire inférieur
pour i = 1 à nombre de lignes (n)

$$yi = (bi - \sum_{k=1}^{k=i-1} Lik\, yk) / Lii$$

fin
envoi du résultat aux autres processeurs
fin

test pour déterminer quelle partie du code effectuer sur chaque processeur
si p ≠ numéro du processeur
réception bloquante des vecteurs résidus partiels issus des autres proc.
concaténation des résultats partiels
fin
fin

Algo. 5.2: Descente sur une architecture à mémoire distribuée.

La même démarche est adoptée pour la résolution du système triangulaire supérieur. Cette étape est donc assez pénalisante en terme de performances parallèles. Bien que la convergence des solveurs (GC et QMR) soit fortement augmentée par ce préconditionnement, le surcoût introduit en terme de parallélisme est tel que cette méthode ne peut être utilisée au-delà de 4 processeurs. Les temps d'attente lors des réceptions bloquantes, dus à la mauvaise répartition de charge, dépassent alors le temps maximal admissible par PVM. La figure (5.20) présente une trace PARAGRAPH du traitement du problème test sur 4 processeurs avec le GC, la

figure (5.24), le nombre de messages correspondant et la figure (5.28), une moyenne, sur la totalité du temps de fonctionnement, de l'état de chaque processeur. Ces figures montrent la mauvaise répartition de charge induite par cet algorithme. L'excès *d'overhead* est principalement dû au nombre de passages de messages nécessaires à la construction de la matrice de préconditionnement.

Pour le problème test, l'accélération obtenue pour le GC et pour la totalité du code avec le préconditionnement de Cholesky incomplet est présentée figure (5.11) et la répartition du temps CPU pour les différentes étapes, figure (5.12). Ici encore, la prédominance du temps consacré aux passages de messages est évidente.

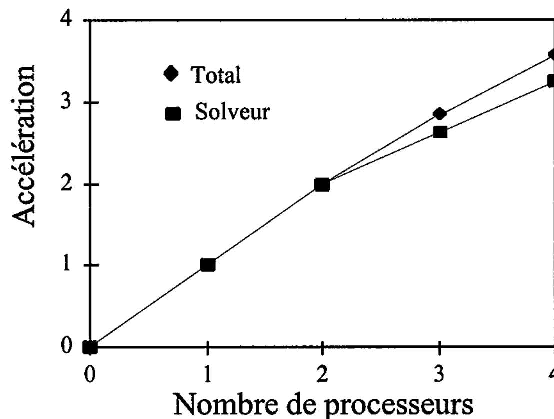


Fig. 5.11: Accélérations du solveur muni d'un préconditionnement de Cholesky incomplet et de la totalité du code pour le problème test.

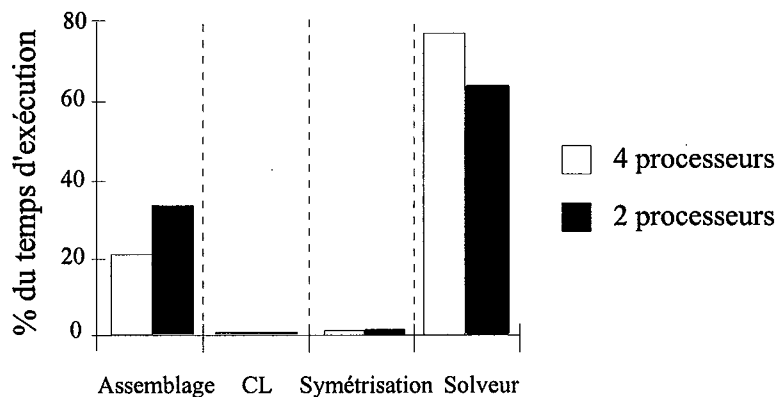


Fig. 5.12: Répartition du temps CPU avec un solveur muni d'un préconditionnement de Cholesky incomplet pour le problème test.

Malgré les mauvaises performances en terme de temps CPU (fig. 5.13) des solveurs GC et QMR munis du préconditionnement de Cholesky incomplet, cette technique reste néanmoins intéressante en terme de convergence sur des problèmes mal conditionnés. La figure (5.13)

présente les convergences obtenues sur le problème test. Il apparaît que la convergence est fortement améliorée par le préconditionnement de Cholesky incomplet. Cette dernière remarque est d'autant plus valable pour des problèmes maillés avec des tétraèdres. En effet, l'utilisation de tels éléments entraîne une convergence très lente du solveur.

Le nombre d'itérations nécessaires à la résolution du système d'équations ne dépendant pas du nombre de processeurs, à partir du temps CPU utilisé pour la résolution sur un processeur et des courbes d'accélérations du solveur, il apparaît que malgré la diminution conséquente du nombre d'itérations, le préconditionnement diagonal reste le plus performant. En effet, sur 4 processeurs et pour le traitement du problème test, avec un solveur (GC) muni du préconditionnement diagonal il faut 2101/2.7 = 778 s (fig. 5.9 et 5.7) pour résoudre le problème test. Le traitement par le GC muni du préconditionnement de Cholesky incomplet, nécessite 4680/3.57 = 1308 s (fig. 5.13 et 5.11). Le tableau (5.1) reprend ces résultats.

	Temps mono processeur	Accélération sur 4 processeurs	Temps par processeurs
Précond. diagonal	2101 s	2.7	778 s
Précond. de Cholesky	4680 s	3.57	1308 s

Tab. 5.1: Comparaison des performances du GC, pour une résolution sur 4 processeurs, en fonction du type de préconditionnement.

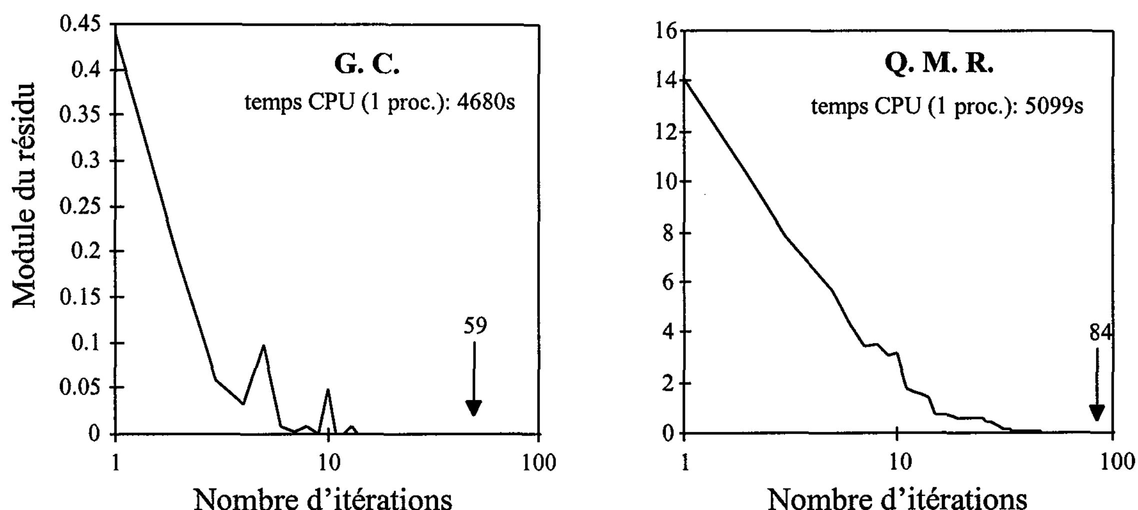


Fig. 5.13: Convergence du GC et de QMR munis d'un préconditionnement de Cholesky incomplet pour le problème test.

Le préconditionnement de Cholesky incomplet étant un technique difficilement adaptable à une architecture à mémoire distribuée, l'étude s'est donc poursuivie en essayant de modifier cette méthode pour diminuer le nombre de passages de messages nécessaires à la construction de la matrice ainsi qu'à la descente et remontée.

V 2 4 3. Préconditionnement de Cholesky incomplet 'par paquets'

Cette méthode originale [83] consiste, lors d'une résolution sur plusieurs processeurs, à calculer les termes de la matrice de préconditionnement seulement à partir des termes de la matrice EF disponibles sur le noeud de calcul considéré. L'algorithme utilisé est le même qu'au paragraphe précédent (algo. 5.1), mais, certains termes n'étant pas présents sur le processeur, les calculs ne sont qu'approximatifs. La matrice de préconditionnement a la structure présentée à la figure (5.14). Lors d'une résolution sur un processeur, cette méthode revient à effectuer le préconditionnement de Cholesky incomplet présenté précédemment.

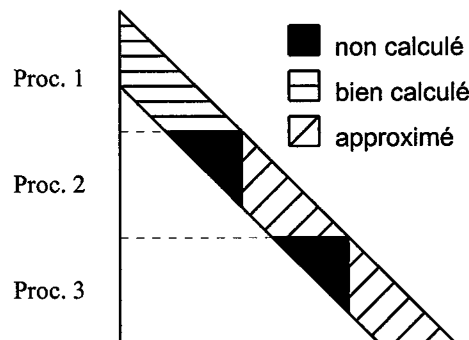


Fig. 5.14: Structure de la matrice de Cholesky 'par paquets'

L'erreur introduite par cette méthode a évidemment une incidence sur la convergence des algorithmes (GC et QMR) ainsi préconditionnés. Remarquons qu'il est possible d'introduire une erreur dans le calcul du préconditionnement car celui-ci n'a pas d'incidence sur la précision du résultat [annexe 3].

Etant donnée la forme de la matrice de préconditionnement, la descente et la remontée peuvent être effectuées en parallèle. En effet, le processeur 2 peut entamer son calcul sans avoir à attendre les résultats issus du processeur 1. Chaque processeur calcule donc indépendamment son morceau de vecteur résidu (algo. 5.3). Des passages de messages sont nécessaires à la fin de la descente et de la remontée afin de reconstituer sur tous les

processeurs le vecteur résidu. La même méthode (SPMD) que pour la concaténation du vecteur résidu lors du préconditionnement diagonal est utilisée.

Résolution du système triangulaire inférieur

pour i = 1 à nombre de ligne (n) du processeur

$$yi = (bi - \sum_{k=1}^{k=i-1} Lik\, yk) / Lii$$

fin

envoi du résultat aux autres processeurs

réception bloquante et concaténation des vecteurs résidus partiels issus des autres noeuds

Résolution du système triangulaire supérieur

pour i = nombre de lignes (n) du processeur à 1

$$xi = (yi - \sum_{k=i+1}^{k=n} Lki\, xk) / Lii$$

fin

envoi du résultat aux autres processeurs

réception bloquante et concaténation des vecteurs résidus partiels issus des autres noeuds

Algo. 5.3: Descente-remontée parallèles.

La figure (5.21) présente une trace PARAGRAPH du traitement du problème test sur 4 processeurs avec le GC, la figure (5.25) le nombre de messages correspondant et la figure (5.29) une moyenne, sur la totalité du temps de fonctionnement, de l'état de chaque processeur. Ces figures (5.21), (5.25) et (5.29), qu'il faut comparer aux figures (5.20), (5.24) et (5.28), mettent en évidence une bonne répartition de charge ainsi qu'une occupation du réseau de communications tout à fait acceptable. L'*overhead* introduit par le parallélisme reste lui aussi dans des proportions convenables.

Cette méthode implique que le nombre d'opérations à effectuer dépend du nombre de processeurs sur lesquels est traité le problème. Sur 1 processeur, cela revient à utiliser la méthode du préconditionnement de Cholesky incomplet décrite au paragraphe précédent. A partir de 2 processeurs, le nombre d'opérations à réaliser diminue du fait de la structure de la matrice de préconditionnement (fig. 5.14) et de la technique utilisée pour la descente et remontée. De la remarque précédente il découle que les performances des solveurs munis de ce type de préconditionnement ne peuvent être évaluées en termes d'accélération car la résolution sur 2 processeurs est 2.8 fois plus rapide que sur 1 processeur. Pour le problème test, les temps CPU obtenus pour le GC et pour la totalité du code avec ce préconditionnement

sont présentés figure (5.15) et la répartition du temps CPU pour les différentes étapes, figure (5.16). Il apparaît que pour cette taille de problème, la méthode perd de son efficacité à partir de 5 processeurs. L'augmentation de la taille du problème repousse cette limite comme cela sera montré par la suite.

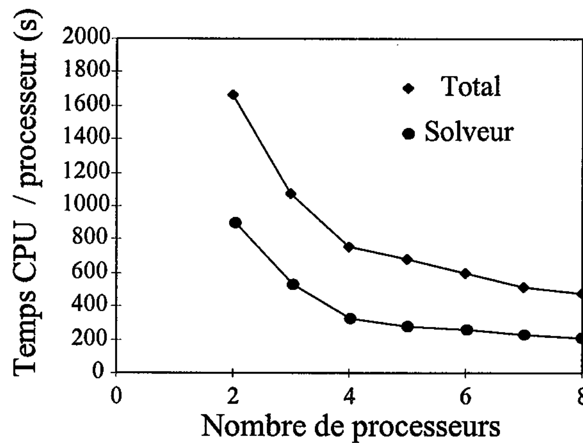


Fig. 5.15: Temps CPU par processeur pour le solveur et pour le code entier munis d'un préconditionnement de Cholesky incomplet 'par paquets'.

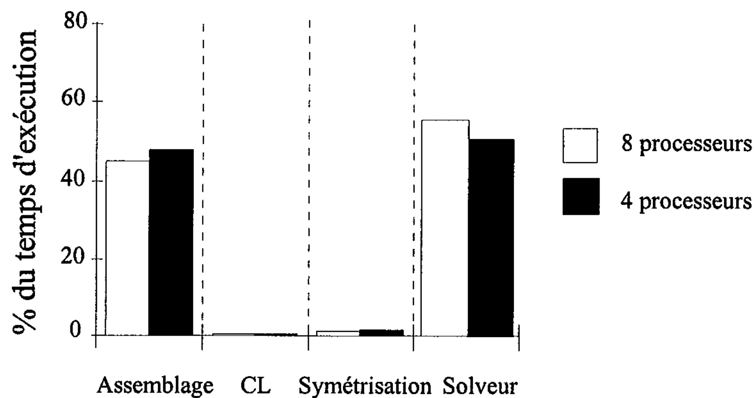


Fig. 5.16: Répartition du temps CPU avec un solveur muni d'un préconditionnement de Cholesky 'par paquets'.

Le nombre d'itérations nécessaires pour la résolution du système d'équations dépend évidemment du nombre de processeurs, la parallélisation dégradant la méthode mais pas la précision du résultat. La figure (5.17) présente la convergence des deux solveurs (GC et QMR) sur le problème test pour différents nombres de processeurs. Il apparaît cependant que l'augmentation du nombre de processeurs n'est pas très pénalisant en terme de convergence. A titre de comparaison, sur 4 processeurs, avec le GC muni du préconditionnement diagonal, il faut 778 s. Le GC muni du préconditionnement de Cholesky incomplet nécessite 1308 s et le GC muni du préconditionnment de Cholesky 'par paquets' requiert 754 s (fig. 5.15). Le tableau (5.2) reprend ces résultats.

	Temps mono processeur	Accélération sur 4 processeurs	Temps par processeurs
Précond. diagonal	2101 s	2.7	778 s
Précond. de Cholesky	4680 s	3.57	1308 s
Précond. de Cholesky par paquets	4680 s	/	754 s

Tab . 5.2: Comparaison des performances du GC, pour une résolution sur 4 processeurs en fonction du type de préconditionnement.

Le préconditionnement de Cholesky par 'paquets' semble être un bon compromis entre l'efficacité du préconditionnement et le nombre de passages de messages requis. Toutefois, il reste à vérifier si l'augmentation du nombre de processeurs au-delà de 8 ne dégrade pas outre mesure la méthode, et si cette technique conserve ses avantages sur une architecture plus performante au sens réseau d'interconnexions. Ceci fera l'objet du prochain chapitre.

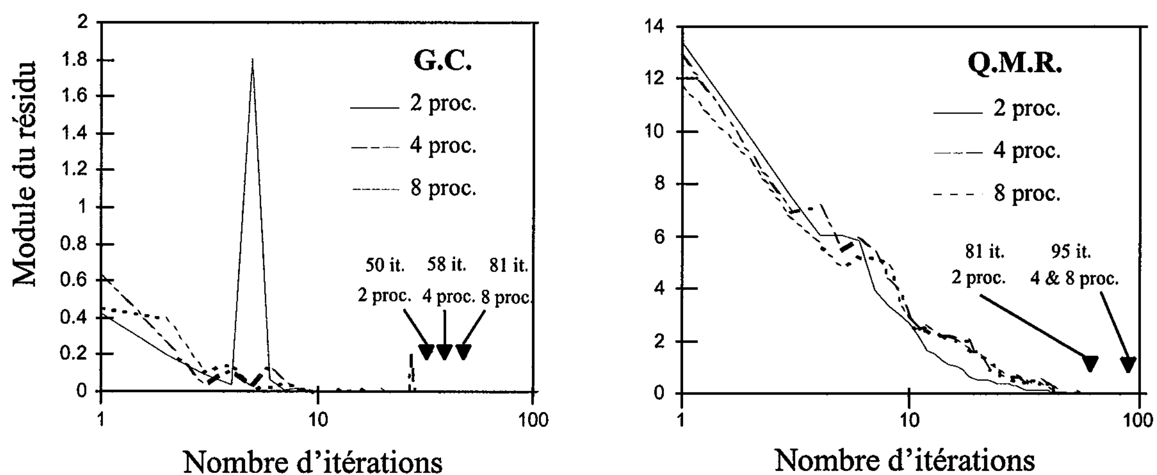


Fig. 5.17: Convergence du GC et de QMR munis d'un préconditionnement de Cholesky incomplet 'par paquets'

La méthode présentée dans ce paragraphe permet donc de diminuer fortement le nombre d'itérations sans pour autant introduire un *overhead* prohibitif. Elle apparaît donc comme un bon compromis entre un préconditionnement diagonal et un préconditionnement de Cholesky incomplet. Son utilisation requiert un espace mémoire total égal à 1.5 fois celui de la matrice EF avant sa symétrisation. Ce surcoût en terme d'espace mémoire, par rapport au préconditionnement diagonal n'est, de toute façon, pas pénalisant car une telle allocation temporaire à déjà été nécessaire dans les étapes antérieures du code.

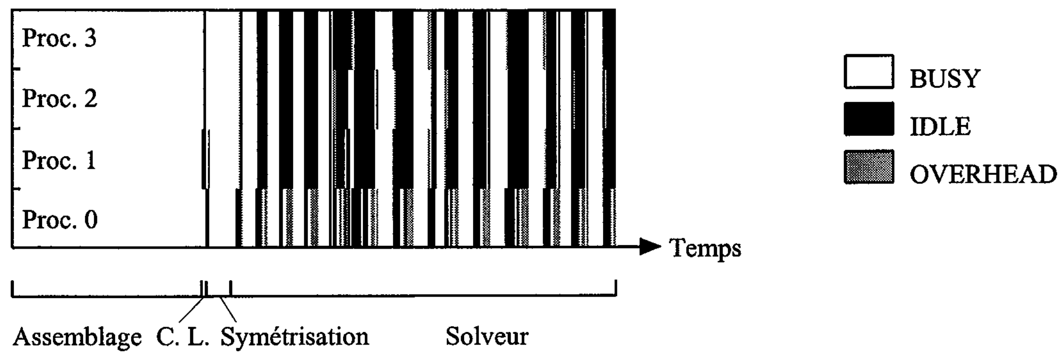


Fig. 5.18. Déroulement du code (précond. diag. et fonct. M/E).

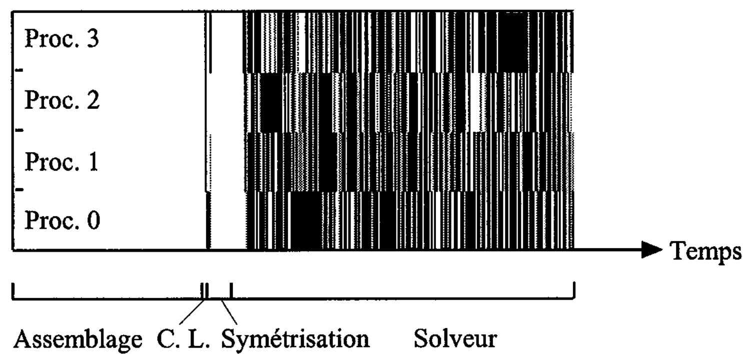


Fig. 5.19. Déroulement du code (précond. diag. et fonct. SPMD).

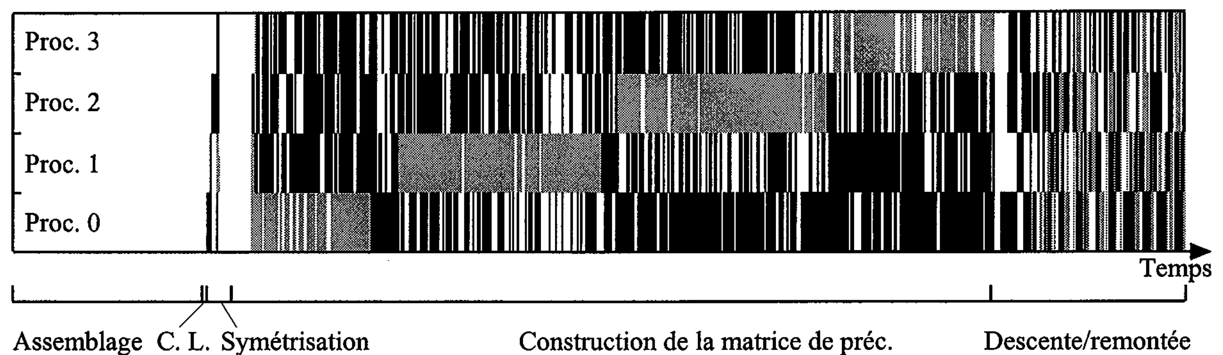


Fig. 5.20. Déroulement du code (précond. de Cholesky).

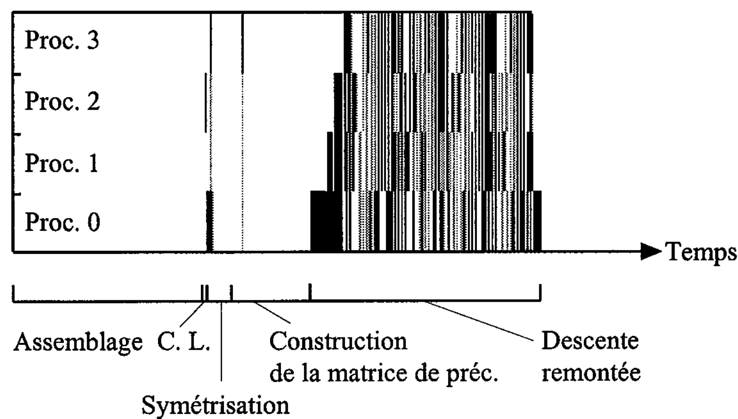


Fig. 5.21. Déroulement du code (précond. de Cholesky par paquets).

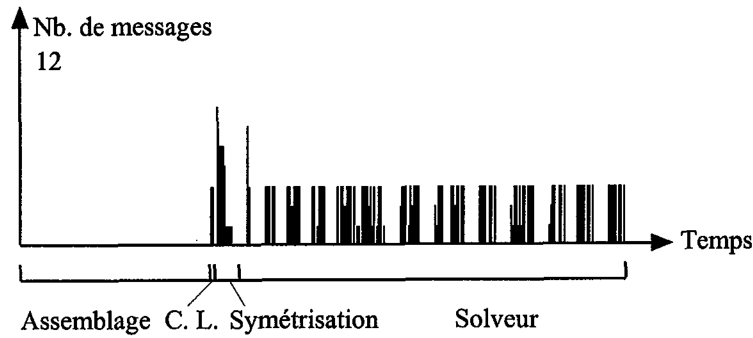


Fig. 5.22. Nombre de messages (précond. diag. et fonct. M/E).

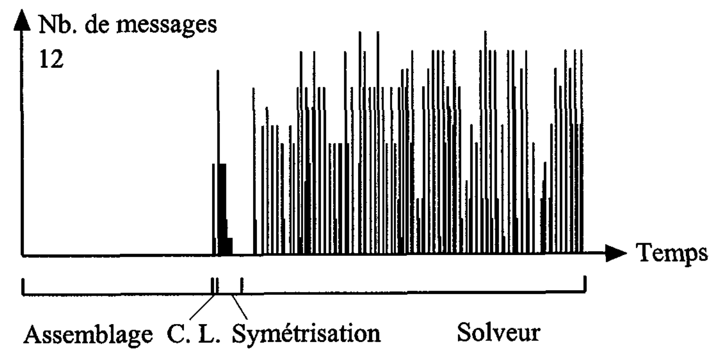


Fig. 5.23. Nombre de messages (précond. diag. et fonct. SPMD).

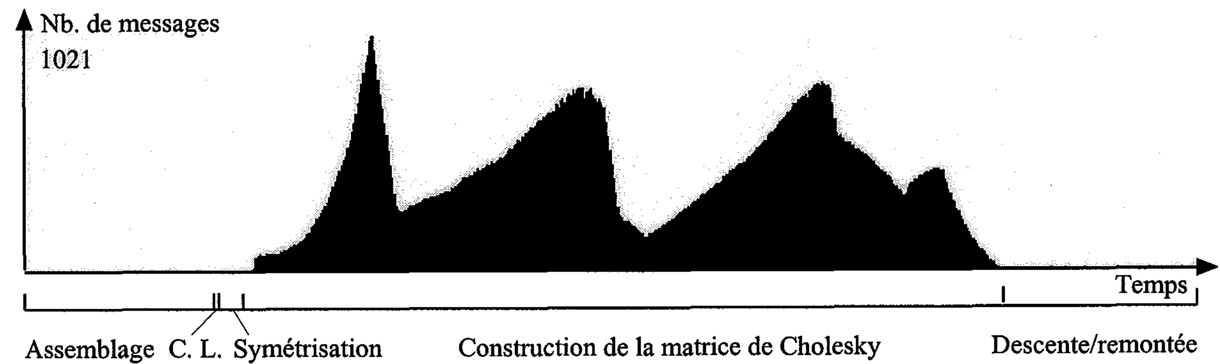


Fig. 5.24. Nombre de messages (précond. de Cholesky).

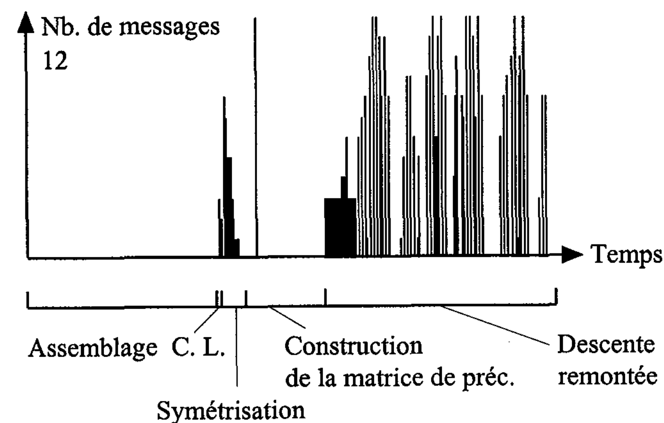


Fig. 5.25. Nombre de messages (précond. de Cholesky par paquets).

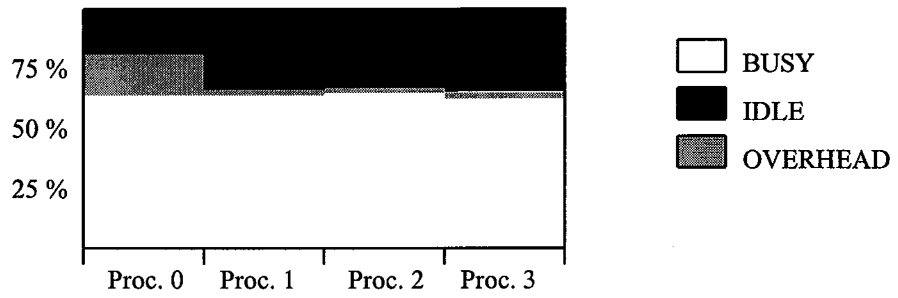


Fig. 5.26. Moyenne de l'état des proc. (précond. diag. et fonct. M/E).

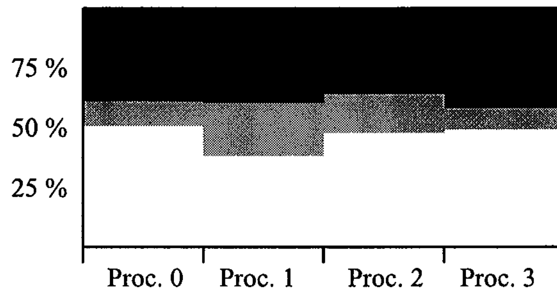


Fig. 5.27. Moyenne de l'état des proc. (précond. diag. et fonct. SPMD).

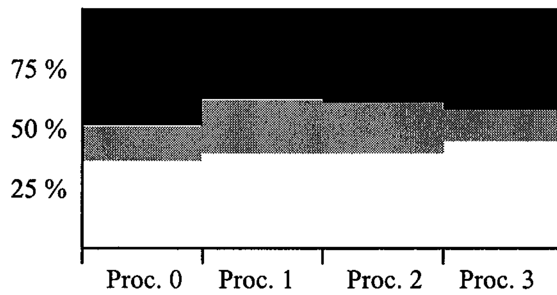


Fig. 5.28. Moyenne de l'état des proc. (précond. de Cholesky).

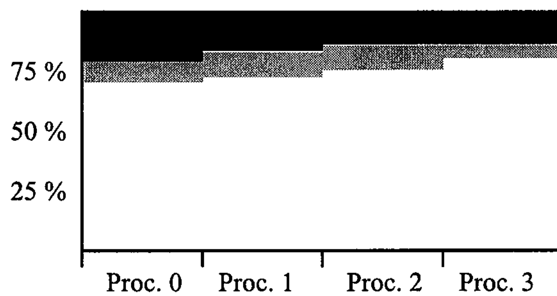


Fig. 5.29. Moyenne de l'état des proc. (précond. de Cholesky par paquets).

V 3. Problème de grande taille (diffraction par un cylindre)

Sur la ferme de stations, il est théoriquement possible de traiter des problèmes pouvant aller jusqu'à 80000 noeuds (hexaèdres du premier ordre). Afin de pouvoir comparer les performances des différentes machines, le même problème que sur le CRAY C98 est exposé: 40000 noeuds (hexaèdres du premier ordre). Le tableau (5.3) présente les résultats obtenus pour une résolution sur 4 et 8 processeurs.

	Temps/proc.	Itérations
Diagonal (4 proc.)	6975 s	269
Diagonal (8 proc.)	4386 s	269
Cholesky par paquets (4 proc.)	8643 s	82
Cholesky par paquets (8 proc.)	3949 s	102

Tab. 5.3: Performances pour un problème de 40000 noeuds

Le préconditionnement de Cholesky incomplet n'a pu être utilisé pour ce problème du fait de ses mauvaises performances. Pour cet exemple, le préconditionnement de Cholesky incomplet par paquets est la méthode la plus performante pour une résolution sur 8 processeurs.

V 4. Conclusion

Cette étude a mis en évidence la nécessité d'une restructuration fondamentale des algorithmes utilisés sur calculateurs séquentiels pour l'adaptation à un calculateur parallèle à mémoire distribuée. Evidemment, beaucoup d'algorithmes parallèles susceptibles de bien exploiter les ressources de tels ordinateurs n'ont pas été abordés dans cette étude. Néanmoins, de nombreux travaux disponibles dans la littérature permettent de faire le point sur les méthodes disponibles [47-52], [63, 64]. L'originalité de cette étude réside dans le fait qu'aucun calcul supplémentaire n'est requis par rapport à un code séquentiel (décomposition de domaines, éléments finis indépendants, ...).

A l'heure actuelle, seul le calcul parallèle est à même de fournir la puissance de calcul et l'espace mémoire nécessaires à la modélisation de géométries 3D réalistes en électrotechnique. De plus, les constructeurs de super-calculateurs s'orientant vers des

architectures à mémoire distribuées, il paraîtrait logique de devoir tenir compte de certaines directives lors du développement de nouveaux codes de calcul sur ordinateurs séquentiels. De ce fait, à partir d'un choix judicieux des algorithmes séquentiels, la parallélisation de ces codes, qui est une étape nécessaire, devrait se faire plus facilement. Au moment du développement d'un code séquentiel, le programmeur devra donc choisir des algorithmes susceptibles d'être facilement parallélisés (assemblage par degrés de liberté, traitement des symétries par modification de la matrice après assemblage, solveur itératif, ...).

CHAPITRE VI
IMPLANTATION SUR LE CRAY T3E

VI 1. Introduction

Le code développé sur la ferme de stations a été implanté sur le CRAY T3E de l'IDRIS. Bien que PVM soit supporté sur cette machine, il sera démontré par la suite que pour minimiser l'*overhead* introduit par les passages de messages, l'utilisation de SHMEM est nécessaire. Cette bibliothèque CRAY non portable permet l'émulation *hardware* d'espace mémoire physiquement distribué mais globalement adressable. De plus, une documentation spécifique mise à la disposition des utilisateurs permet d'effectuer une optimisation monoprocesseur du code à l'aide de différentes techniques [67, 68]. Les résultats exposés ici incluent les optimisations monoprocesseur sachant que celles-ci ont globalement apporté une réduction de l'ordre de 20 % du temps CPU, et ceci quelle que soit la méthode de résolution.

Dans ce chapitre sont exposées les performances du code parallèle muni de PVM pour les passages de messages, les modifications apportées dans certaines parties du code pour l'introduction de SHMEM ainsi que les performances alors obtenues. L'étude des performances parallèles (accélération, répartition du temps CPU, ...) est limitée à 8 processeurs dans cette partie pour des raisons d'économie de temps CPU et aussi à cause des limitations imposées par les méthodes de minimisation de passages de messages. En effet, le problème test ne comportant que 10000 noeuds, sa résolution ne peut être réalisée sur plus de 8 processeurs. Les répartitions des temps CPU et l'étude des surcoûts sont donnés pour 4 et 8 processeurs comme sur la ferme de stations.

VI 2. Code parallèle

Le code opérant sur la ferme de stations a donc été porté sur le CRAY T3E. Ces deux machines étant du même type et supportant les mêmes modèles de programmation, peu de modifications ont été apportées au code dans un premier temps.

VI 2 1. Code parallèle muni de PVM

VI 2 1 1. Assemblage, traitement des CL et symmétrisation du système matriciel

Les accélérations obtenues pour les différentes phases d'assemblage, de traitement des CL et de symmétrisation du système matriciel sont sensiblement équivalentes à celles obtenues sur la ferme de stations car ces étapes ne nécessitent que très peu de passages de messages.

VI 2 1 2. Solveur et code entier

- Préconditionnement diagonal

La figure (6.1) met en évidence de meilleures performances que sur la ferme de stations pour le code entier muni d'un préconditionnement diagonal (fonctionnement SPMD). Cette constatation paraît logique compte tenu des performances du réseau de communications. Toutefois cette amélioration n'est pas aussi grande que l'augmentation des performances du réseau aurait pu le laisser supposer.

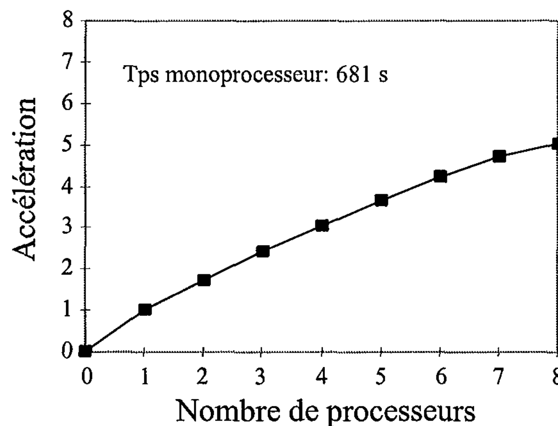


Fig. 6.1: Accélération pour le code entier muni du préconditionnement diagonal pour le problème test (PVM).

La figure (6.2) montre la répartition du temps CPU pour le code muni du préconditionnment diagonal. L'augmentation du nombre de processeurs est moins pénalisante que sur la ferme de stations.

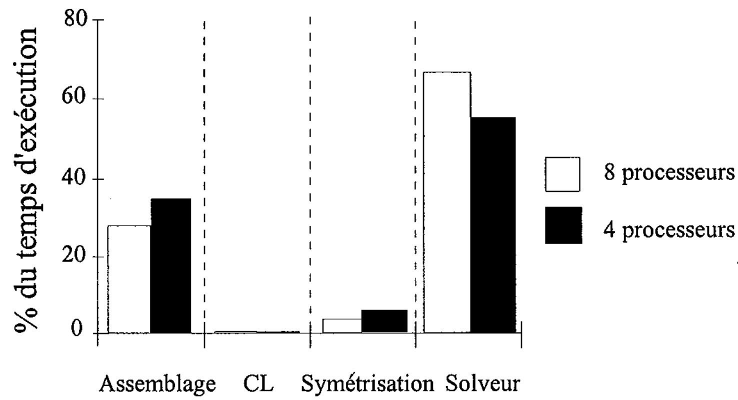


Fig. 6.2: Répartition du temps CPU pour le code entier muni du préconditionnement diagonal pour le problème test (PVM).

- Préconditionnement de Cholesky incomplet

Malgré les performances du réseau de communications, cette méthode ne fournit pas des résultats satisfaisants pour le traitement de grosses géométries. De plus, les mêmes limitations que sur la ferme de stations, quant au nombre de processeurs maximum utilisables, sont rencontrées et ceci malgré le positionnement de variables d'environnement de PVM.

- Préconditionnement de Cholesky incomplet par paquets

La figure (6.3) montre que cette méthode reste valable sur ce type d'architecture et pour cette taille de problème. Ici encore, la résolution sur 2 processeurs est 2.5 fois plus rapide que sur 1 seul. Les performances sont donc exprimées en terme de temps CPU.

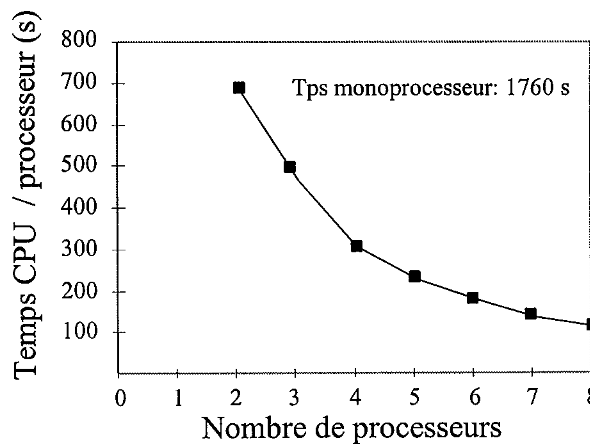


Fig. 6.3: Temps CPU par processeur pour le code entier muni du préconditionnement de Cholesky incomplet par paquets pour le problème test (PVM).

La figure (6.4) montre la répartition du temps CPU pour le code muni du préconditionnement de Cholesky incomplet par paquets. L'augmentation du nombre de processeurs est ici encore moins pénalisante que sur la ferme de stations.

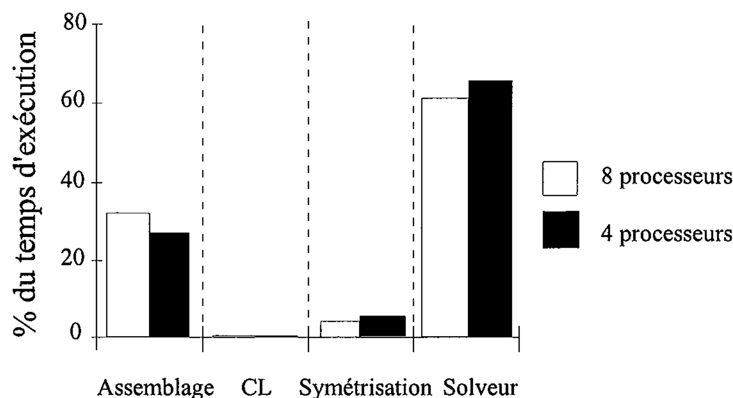


Fig. 6.4: Répartition du temps CPU pour le code entier muni du préconditionnement de Cholesky incomplet par paquets pour le problème test (PVM).

VI 2 1 3. Analyse de l'*overhead*

Les performances globales du code sont accrues (fig. 6.1 et 6.3) quelle que soit la méthode de préconditionnement. Le tableau (6.1) résume ces performances (temps CPU par processeur) pour les différents préconditionnements lors d'une résolution sur 1, 2, 4 et 8 processeurs. Le préconditionnement de Cholesky par paquets apparaît comme étant la méthode la plus performante pour cette taille de problème à partir de 8 processeurs. L'augmentation de la taille du problème ou la résolution de géométries plus complexes tend à diminuer ce nombre (chapitre VII).

	1 proc.	2 proc.	4 proc.	8 proc.
Préc. diag.	681 s	401 s	227 s	136 s
Préc. Cho. paq.	1760 s	702 s	321 s	127 s

Tab. 6.1: Performances du code muni de PVM pour les différentes méthodes de résolutions sur le problème test.

L'analyse du surcoût introduit par les passages de messages lors du déroulement du code (fig. 6.2, 6.4 et 6.5) montre que les efforts doivent se porter sur le solveur même si l'utilisation du préconditionnement de Cholesky incomplet par paquets en limite les effets. En effet, pour

les deux types de préconditionnement, et quel que soit le nombre de processeurs utilisés, la majorité des surcoûts introduits le sont dans la phase de résolution du système d'équations.

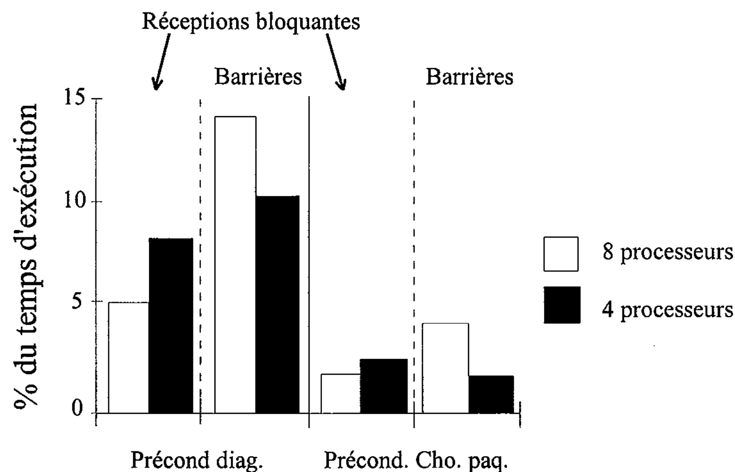


Fig. 6.5: *overheads* introduits par les passages de messages (PVM) pour les différentes méthodes de résolutions sur le problème test.

VI 2 2. Code parallèle muni de SHMEM

VI 2 2 1. Assemblage, traitement des CL et symmétrisation du système matriciel

En vertu des remarques du paragraphe précédent, une seule modification a été apportée à cette partie du code car les passages de messages ne sont pas pénalisants pour les phases de traitement des symétries et de symétrisation du système matriciel. La concaténation du deuxième membre sur tous les processeurs (en mode SPMD avec PVM) est réalisée à l'aide d'une fonction de réduction de SHMEM (shmem_complexd_sum_to_all). Le même principe étant utilisé pour container les vecteurs résidus partiels à chaque itération quand un préconditionnement diagonal est utilisé, la méthode est décrite en détail dans le paragraphe suivant. Les performances parallèles de ces étapes sont pratiquement les mêmes que sur la ferme de stations.

VI 2 2 2. Solveur

- Préconditionnement diagonal

SHMEM a supplanté PVM pour la phase de concaténation des vecteurs résidus partiels stockés sur les différents processeurs. En fait l'envoi en mode SPMD a été remplacé par une

fonction de réduction (shmem_complexd_sum_to_all). Les vecteurs résidus partiels doivent alors avoir une longueur égale au nombre de lignes de la matrice et être alloués avec une fonction de la bibliothèque 'malloc.h', ceci afin d'avoir la même adresse sur tous les processeurs. De plus, des vecteurs de travail doivent être aussi alloués et des synchronisations sont nécessaires pour s'assurer que ces vecteurs contiennent des données valides au moment du premier appel de la fonction. Par la suite, si ces vecteurs de travail ne sont utilisés simultanément que par une seule fonction de SHMEM, plus aucune initialisation n'est requise. Par ailleurs une autre synchronisation doit permettre de s'assurer que les données présentes dans les vecteurs à container sont valides sur chacun des processeurs au moment de l'appel de la fonction. Ceci permet aussi de séparer les phases de calcul des phases de passages de messages (utilisation sécurisée des *Streams buffers*).

Les performances obtenues pour le code entier muni du préconditionnement diagonal fonctionnant avec SHMEM sont exposées figure (6.6). La répartition du temps CPU (fig. 6.7) fait apparaître que les allocations dynamiques avec les fonctions de la bibliothèque 'malloc.h' introduisent une augmentation du temps CPU. Ces allocations n'étant pas nécessaires lors d'une résolution sur un processeur, elles peuvent être interprétées comme un surcoût et font donc chuter les performances parallèles. Néanmoins, celles-ci sont supérieures à celles obtenues avec PVM.

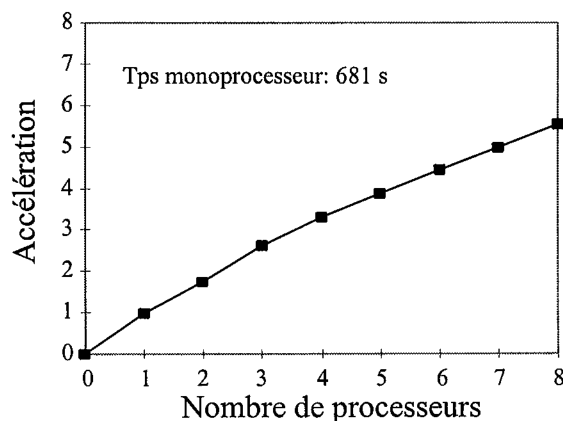


Fig. 6.6: Accélération pour le code entier muni du préconditionnement diagonal pour le problème test (SHMEM).

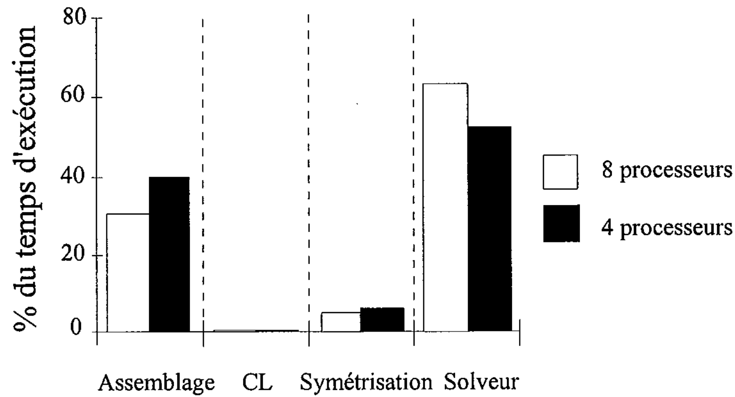


Fig. 6.7: Répartition du temps CPU pour le code entier muni du préconditionnement diagonal pour le problème test (SHMEM).

- Préconditionnement de Cholesky incomplet

Le remplacement de PVM par SHMEM dans la construction de la matrice de préconditionnement pose des problèmes de synchronisations. Le mécanisme général de cette étape reste le même que sur la ferme de stations mais, étant donné que les réceptions bloquantes dans SHMEM n'existent pas, tout un mécanisme de synchronisations a dû être mis au point.

Les synchronisations peuvent être résumées ainsi (algo. 6.1): le processeur devant envoyer une ligne aux autres processeurs doit attendre que ceux-ci soient prêts à la recevoir. Quand cette condition est remplie, il doit leur signaler que les données relatives à la ligne (longueur, numéro, ...) sont valides car les autres processeurs sont en attente de cette confirmation pour accéder à ces données. Une synchronisation collective de tous les processeurs est ensuite réalisée car le retour de SHMEM_PUT ne garantit pas la fin de l'opération. Une fonction collective est alors utilisée pour transférer les termes de la ligne (SHMEM_BROADCAST). Les autres processeurs peuvent alors dépaqueter ces données, puis signaler au processeur émetteur qu'ils sont prêts à réceptionner la ligne suivante. Ici encore toutes les données devant être transmises sont stockées dans des vecteurs alloués avec des fonctions de la bibliothèque 'malloc.h'. Les mêmes remarques que précédemment concernant les vecteurs de travail restent valables.

```
pour p = 1 à nombre de processeurs (boucle sur tous les processeurs)
        test pour déterminer quelle partie du code effectuer sur chaque processeur
        si p = numéro du processeur
                boucle sur les lignes du processeur
                pour j = 0 à nombre de lignes stockées sur ce processeur
                        calcul de  Σ(k=1..j-1) (Ljk)²

                        packtage des Ljk dans le vecteur à émettre
                        calcul du terme diagonal Ljj
                        packtage de Ljj dans le vecteur à émettre
                        shmem_wait_until (les autres proc. prêts à recevoir la ligne j ?)
                        shmem_put (données concernant la ligne j sont valides)
                        shmem_barrière (synchronisation de tous les processeurs)
                        shmem_broadcast (envoi de la ligne j)
                        pour i = j+1 à nombre de lignes stockées sur ce processeur
                                calcul des Lij sur ce processeur
                        fin
                fin
        fin
        test pour déterminer quelle partie du code effectuer sur chaque processeur
        si p > numéro du processeur
                pour i = 0 à numéro de la première ligne stockée sur ce processeur
                        if ( i = 0) shmem_put (ce processeur est prêt à recevoir la ligne 0)
                        shmem_wait_until (données concernant la ligne i sont valides ?)
                        récupération des paramètres de la ligne i
                        shmem_barrière (synchronisation de tous les processeurs)
                        shmem_broadcast (réception de la ligne)
                        dépaquetage de la ligne reçue
                        shmem_put (ce processeur est prêt à recevoir la ligne i+1)

                        boucle sur les lignes du processeur
                        pour j = 0 à nombre de lignes stockées sur ce processeur
                                calcul des termes de Cholesky, si ils existaient déjà, de
                                la colonne j à l'aide des termes du vecteur réceptionné
                        fin
                fin
        fin
        test pour déterminer quelle partie du code effectuer sur chaque processeur
        si p < numéro du processeur
                pour i = 0 à (nb. de lignes total-(num. 1ère ligne+nb de lignes stockées sur ce proc.))
                        if ( i = 0) shmem_put (ce processeur est prêt à recevoir la ligne 0)
                        shmem_wait_until (données concernant la ligne i sont valides ?)
                        récupération des paramètres de la ligne i
                        shmem_barrière (synchronisation de tous les processeurs)
                        shmem_broadcast (réception de la ligne)
                        dépaquetage de la ligne reçue
                        shmem_put (ce processeur est prêt à recevoir la ligne i+1)

                        pour j = 0 à nombre de lignes stockées sur ce processeur
                        insertion dans la matrice transposée de Cholesky des termes non nuls
                        fin
                fin
        fin
fin
```

Algo. 6.1: Construction de la matrice de Cholesky avec SHMEM.

Dans la phase de descente-remontée nécessaire à la résolution des systèmes à chaque itération, tous les processeurs doivent tour à tour envoyer la partie du résultat qu'ils ont calculée aux autres processeurs. La structure reste donc la même qu'avec PVM mais une synchronisation a été ajoutée de manière à séparer les phases de calcul des phases de communications et à s'assurer que les données contenues dans les vecteurs sont valides au moment de l'appel de la fonction. Une fonction collective de SHMEM a remplacé celle de PVM (algo. 6.2). Les vecteurs doivent être alloués à la même adresse sur tous les processeurs. Les mêmes remarques que précédemment restent valables en ce qui concerne les vecteurs de travail.

boucle sur les processeurs
pour p = 1 à nombre de processeurs

test pour déterminer quelle partie du code effectuer sur chaque processeur
si p = numéro du processeur

Résolution du système triangulaire inférieur
pour i = 1 à nombre de lignes (n)

$$yi = (bi - \sum_{k=1}^{k=i-1} Lik\, yk) / Lii$$

fin

fin

synchronisation de tous les processeurs
shmem_broadcast (envoi et réception du vecteur résultat partiel)

test pour déterminer quelle partie du code effectuer sur chaque processeur
si p ≠ numéro du processeur
dépaquetage et recopie du vecteur résultat partiel
fin

fin

Algo. 6.2: Descente pour la résolution du système avec SHMEM.

La même méthode est utilisée pour la résolution du système triangulaire supérieur.

La figure (6.8) montre que, malgré l'utilisation de SHMEM, cette méthode n'est pas adaptée aux architectures à mémoire distribuée. La répartition du temps CPU (fig. 6.9) met en évidence le même problème que pour un préconditionnement diagonal en ce qui concerne les allocations dynamiques avec les fonctions de la bibliothèque 'malloc.h'.

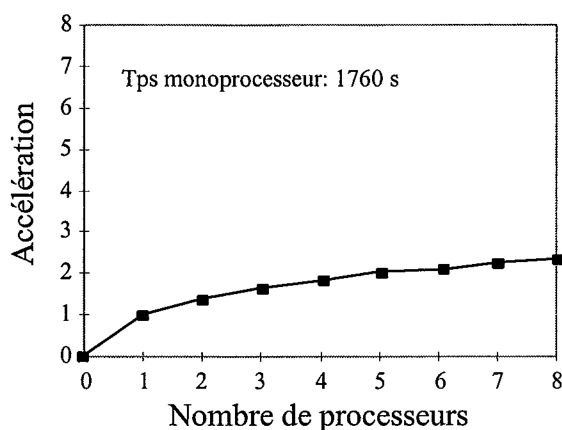


Fig. 6.8: Accélération pour le code entier muni du préconditionnement de Cholesky incomplet pour le problème test (SHMEM).

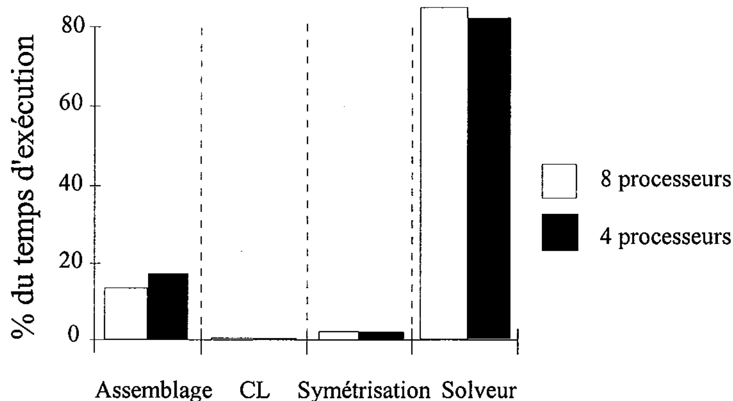


Fig. 6.9: Répartition du temps CPU pour le code entier muni du préconditionnement de Cholesky pour le problème test (SHMEM).

-Préconditionnement de Cholesky incomplet par paquets

Les modifications apportées dans cette partie se situent au niveau de la descente-remontée parallèle car la construction de la matrice de préconditionnement ne requiert aucun passage de message. Quand tous les processeurs ont fini de calculer leurs résultats partiels, la concaténation de ceux-ci est réalisée de la même manière que pour le préconditionnement diagonal (algo. 6.3). Toutes les remarques précédentes concernant les synchronisations et les différents vecteurs restent valables.

Résolution du système triangulaire inférieur

pour i = 1 à nombre de ligne (n) du processeur

$$yi = (bi - \sum_{k=1}^{k=i-1} Lik\, yk) / Lii$$

fin

synchronisation de tous les processeurs
shmem_complexd_sum_to_all (concaténation des vecteurs résultats partiels)

Résolution du système triangulaire supérieur

pour i = nombre de lignes (n) du processeur à 1

$$xi = (yi - \sum_{k=i+1}^{k=n} Lki\, xk) / Lii$$

fin

synchronisation de tous les processeurs
shmem_complexd_sum_to_all (concaténation des vecteurs résultats partiels)

Algo. 6.3: Descente-remontée parallèles avec SHMEM.

Les performances parallèles (fig. 6.10) et la répartition du temps CPU (fig. 6.11) montrent que l'utilisation de SHMEM permet d'augmenter considérablement les performances de la méthode.

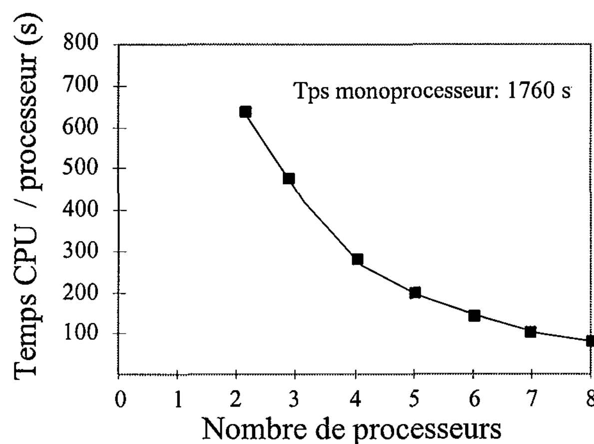


Fig. 6.10: Temps CPU par processeur pour le code entier muni du préconditionnement de Cholesky incomplet par paquets pour le problème test (SHMEM).

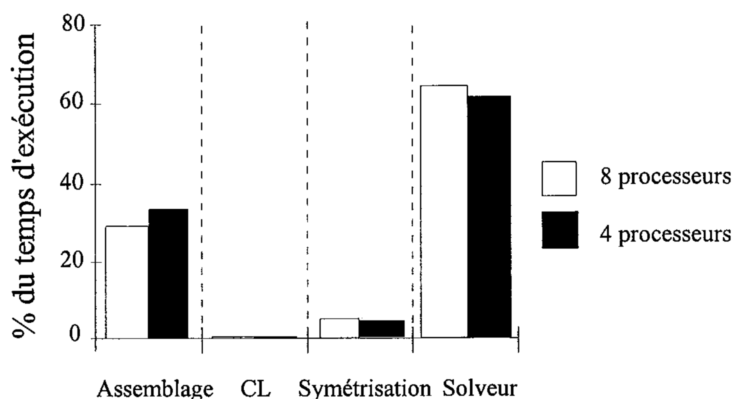


Fig. 6.11: Répartition du temps CPU pour le code entier muni du préconditionnement de Cholesky par paquets pour le problème test (SHMEM).

Le Tableau (6.2) résume les performances (temps CPU par processeur) du code muni de SHMEM pour la résolution du problème test sur 1, 2, 4 et 8 processeurs.

	1 proc.	2 proc.	4 proc.	8 proc.
Préc. diag.	681 s	390 s	201 s	123 s
Préc. Cho.	1760 s	1461 s	1035 s	807 s
Préc. Cho. paq.	1760 s	662 s	303 s	105 s

Tab. 6.2: Performances du code muni de SHMEM pour les différentes méthodes de résolutions sur le problème test.

Il apparaît que le préconditionnement de Cholesky incomplet par paquets est la méthode la plus performante pour une résolution sur 8 processeurs. Cette remarque est d'autant plus valable que les géométries sont complexes. L'utilisation de SHMEM permet de réduire les surcoûts dus aux passages de messages. Son utilisation rend le code non portable mais, comme il est montré dans le paragraphe suivant, le gain sur de gros problèmes est considérable.

VI 3. Problème de grande taille (diffraction par un cylindre)

Afin de pouvoir comparer les performances des différentes machines, le même problème que sur le CRAY C98 et que sur la ferme de stations a été résolu: 40000 noeuds avec des hexaèdres du premier ordre. Le tableau (6.3) présente les résultats obtenus. Ces performances sont présentées pour le code muni de PVM et de SHMEM et pour une résolution sur 8 processeurs.

	Temps CPU/proc.	Itérations
Diagonal (PVM)	1675 s	309
Diagonal (SHMEM)	1468 s	309
Cholesky par paquets (PVM)	1535 s	108
Cho. par paquets (SHMEM)	1232 s	108

Tab. 6.3: Performances pour un problème de 40000 noeuds

Le préconditionnement de Cholesky incomplet n'a pu être utilisé pour ce problème du fait de ces mauvaises performances parallèles. Pour cet exemple, le préconditionnement de Cholesky incomplet par paquets est la méthode la plus performante.

VI 4. Conclusion

L'utilisation de cette machine est fortement simplifiée par la documentation et l'aide fournies par l'IDRIS. L'optimisation monoprocesseur et l'optimisation des communications sont des phases délicates mais essentielles pour exploiter au maximum les possibilités du CRAY T3E. Lors de la phase de développement d'un code sur cette machine, il est conseillé, à chaque modification, de garder une version portable du programme (PVM).

Chaque processeur de ce calculateur étant doté de 128 Mo de RAM et d'un système d'exploitation à micro-noyau très peu consommateur en espace mémoire (16 Mo), il est possible de traiter un problème de 10000 noeuds sur 1 processeur: les tests ont montré que, pour traiter un tel problème sur 1 processeur, 111 Mo de mémoire étaient nécessaires (option Job Accounting). Le CRAY T3E étant doté de 256 processeurs et en tenant compte des surcoûts d'espace mémoire dus à l'utilisation de SHMEM, il est donc théoriquement possible de traiter des problèmes pouvant aller jusqu'à 2.10^6 noeuds.

CHAPITRE VII
EXEMPLES DE CALCULS

VII 1. Introduction

Dans ce chapitre, sont présentés plusieurs grands problèmes pour lesquels sont donnés le nombre de noeuds, le type d'éléments, le nombre d'itérations requis pour les résoudre ainsi que les temps de calcul. Pour les problèmes de diffraction, les objets sont modélisés comme des conducteurs électriques parfaits (cep). Les guides d'ondes ouverts sont eux aussi considérés comme étant des cep.

VII 2. Exemples

Cet exemple concerne la diffraction d'une onde plane par un cylindre cep (fig. 7.1). La résolution a été effectuée en champ **H** et ses principales caractéristiques sont les suivantes:

Domaine d'étude: 800x150x300 mm.

Objet (cylindre): 600 mm, ∅ 100 mm.

Nombre de noeuds: 51920.

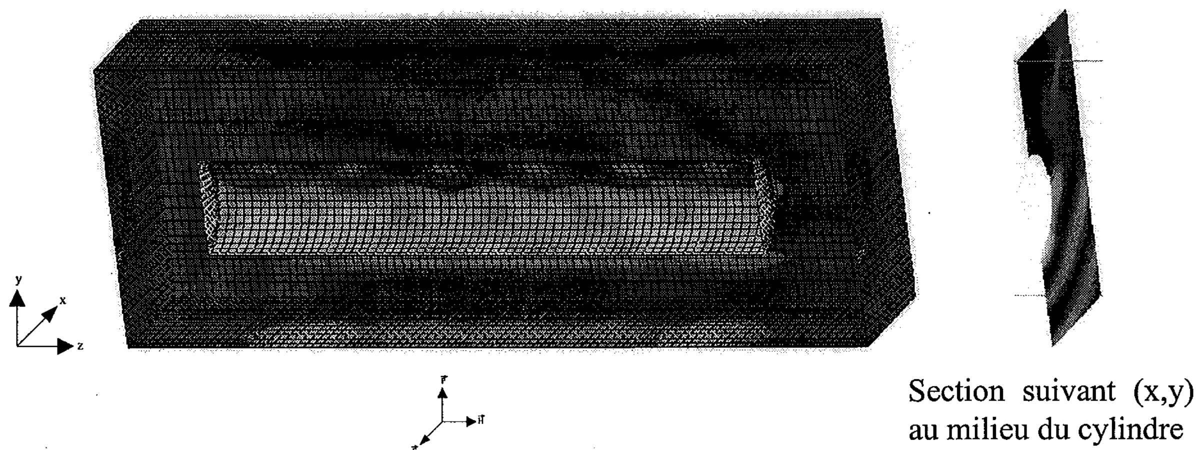
Nombre d'éléments: 46210 hexaèdres du premier ordre.

Fréquence de l'onde incidente: 3 GHz.

Le tableau (7.1) résume les performances obtenues sur les différentes machines pour la résolution de ce problème.

	Type de solveur	Tps/proc.	Itérations
CRAY C98 (8 proc.)	GC + Cholesky + MORSE redondant	461 s	187
Ferme (8 proc.)	GC + diagonal	4.1 h	783
CRAY T3E 8 proc.	GC + diagonal (shmem)	3456 s	766
CRAY T3E 64 proc.	GC + diagonal (shmem)	1434 s	766

Tab. 7.1: Performances pour la résolution du problème traitant de la diffraction par un cylindre.



Section suivant (x,y) au milieu du cylindre

Fig. 7.1: Module du champ magnétique (H).

VII 2 1 3. Remarque

Pour la résolution de ce problème sur la ferme de stations et sur le CRAY T3E, seul le préconditionnement diagonal a été utilisé car, cette géométrie étant simple, la convergence est très rapide. L'utilisation du préconditionnement de Cholesky incomplet par paquets n'est utile que pour des géométries complexes qui engendrent des matrices EF mal conditionnées.

VII 2 2. Diffraction par un avion

Cet exemple traite d'un avion modélisé comme un cep illuminé par une onde plane (fig. 7.2, fig 7.3 et fig. 7.4). La résolution a été effectuée en champ **H** et ses principales caractéristiques sont les suivantes:

Domaine d'étude: 17.5x8.5x5 m.

Objet (avion): 11.7m de longueur et 9.2 m d'envergure.

Nombre de noeuds: 51183.

Nombre d'éléments: 308928 tétraèdres du premier ordre.

Fréquence de l'onde incidente: 100 Mhz.

Le tableau (7.2) résume les résultats obtenus sur les différentes machines.

	Type de solveur	Tps/proc.	Itérations
CRAY C98 (8 proc.)	GC + Cholesky + MORSE redondant	2676 s (113 Mflops)	2777
Ferme (8 proc.)	GC + diagonal	22.3 h	7476
	GC + Cholesky par paquets	16.4 h	4902

Tab. 7.2: Performances pour la résolution du problème traitant de la diffraction par un avion.

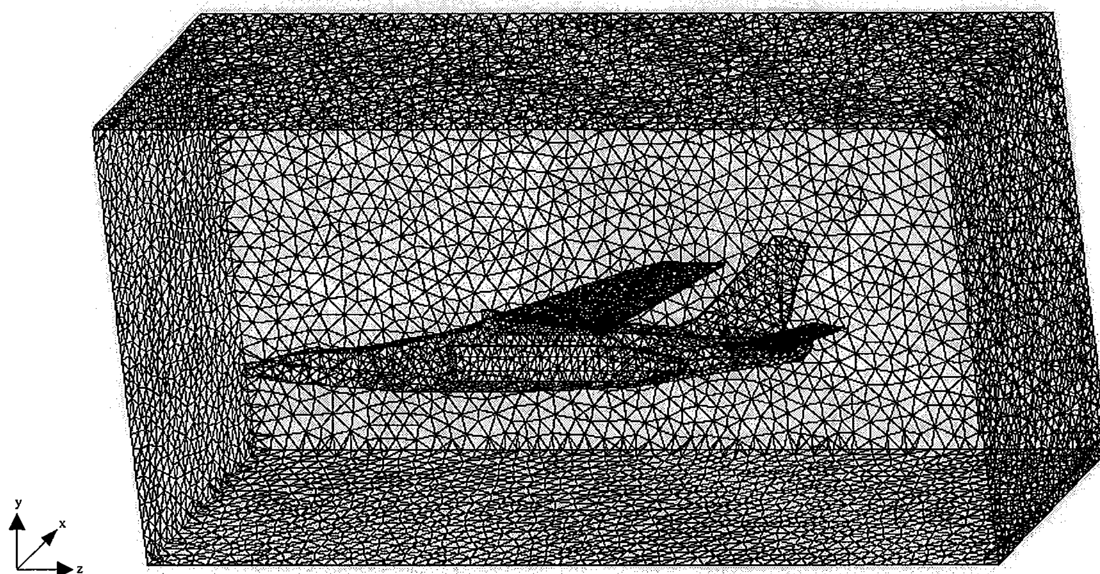


Fig. 7.2: Maillage du problème.

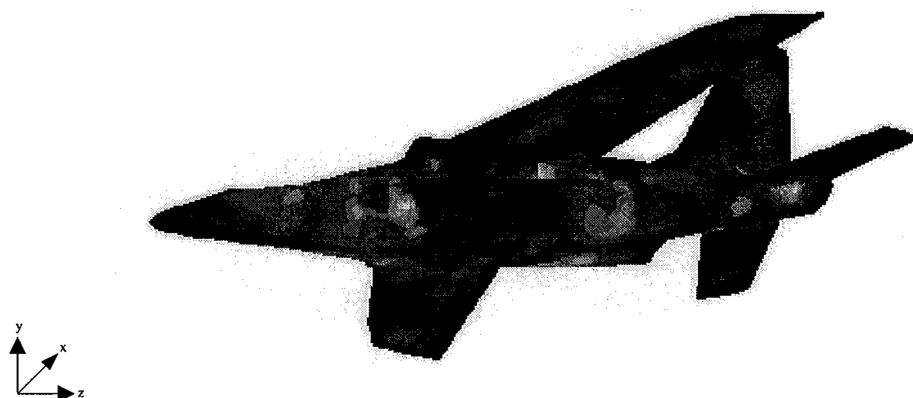


Fig. 7.3: Module du champ magnétique (H).

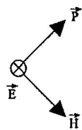


Fig. 7.4: Section suivant yz, module du champ magnétique (H) à l'instant 0.

VII 2 2 3. Remarque

L'utilisation du préconditionnement de Cholesky incomplet par paquets sur une géométrie complexe maillée avec des tétraèdres du premier ordre permet une diminution notable du nombre d'itérations et du temps de calcul. Notons que pour cet exemple, la fréquence de l'onde incidente n'est que de 100 Mhz à cause des grandes dimensions de l'avion. Une fréquence plus élevée demanderait un nombre de noeud plus grand.

VII 2 3. Guide d'ondes ouvert

Cet exemple concerne le calcul des champs électromagnétiques dans et aux abords d'un guide d'ondes ouvert tronqué a 60 degrés (fig. 7.5, fig.7.6 et fig. 7.7) [78]. Il s'agit en fait de la modélisation d'une antenne de Vlasov [84]. La résolution a été effectuée en champ **H** et ses principales caractéristiques sont les suivantes:

Domaine d'étude: 200x130x65 mm.

Objet (guide d'onde): 150 mm, ∅ 47.6 mm.

Nombre de noeuds: 62415.

Nombre d'éléments: 57168 hexaèdres du premier ordre.

Fréquence de la source en mode TM_{01}: 8.6 Ghz.

Le tableau (7.3) résume les résultats obtenus sur les différentes machines.

		Type de solveur	Tps/proc.	Itérations
Ferme de stations (8 proc.)		GC + diagonal	13.3 h	2931
		GC + Cholesky par paquets	11 h	2133
CRAY T3E	8 proc.	GC + diagonal	3.5 h	2902
	64 proc.		1.73 h	2902
	8 proc.	GC + Cholesky par paquets	3.25 h	2003
	64 proc.		1.6 h	2165

Tab. 7.3: Performances pour la résolution du problème traitant du guide d'ondes.

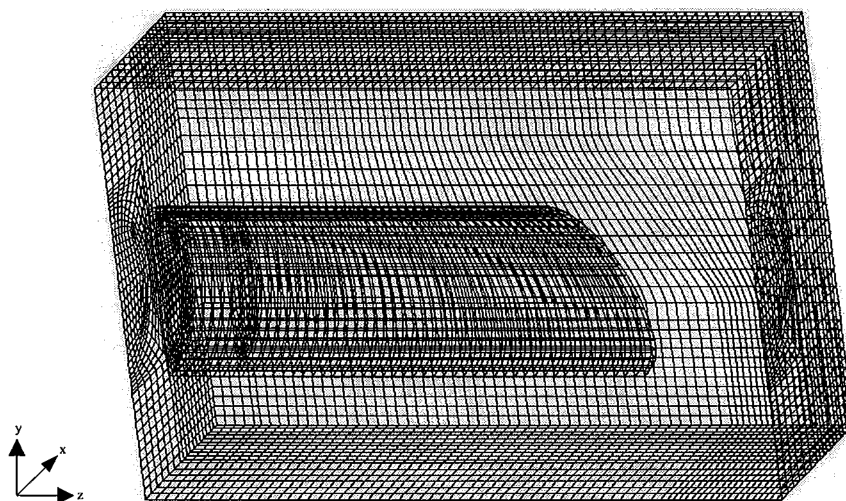


Fig. 7.5: Maillage du guide d'ondes tronqué à 60 degrés.

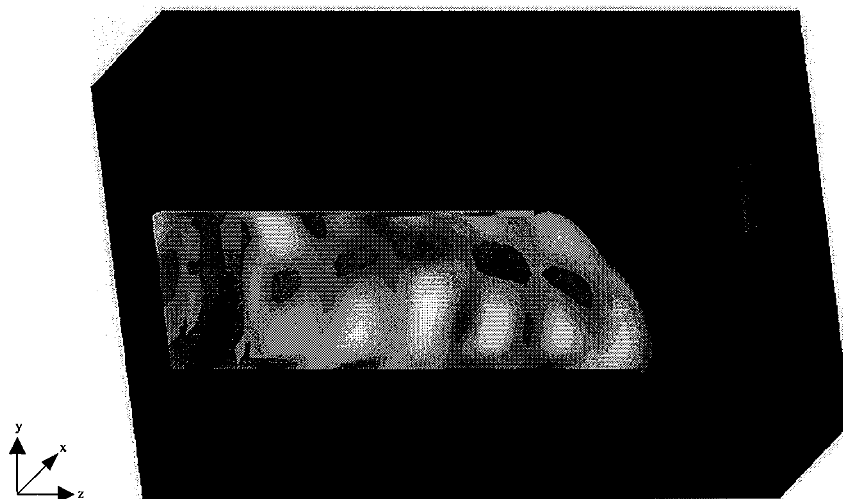


Fig. 7.6: Module du champ magnétique (H).

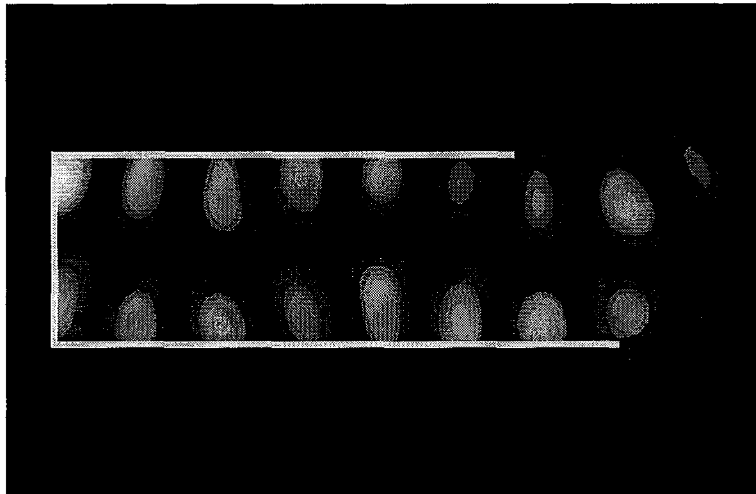


Fig. 7.7: Section suivant (y,z), module du champ magnétique (H) à l'instant 0.

Le calcul parallèle permet de modéliser réellement ce problème. A partir du champ proche et de méthodes intégrales [78], le champ lointain peut être déterminé et comparé aux mesures (fig. 7.8). Jusqu'alors, la modélisation permettait seulement de faire des approximations en 2D de ce système et les résultats obtenus étaient d'une précision médiocre [10]. Le calcul en 3D permet d'obtenir des résultats beaucoup plus proches des mesures et de valider à la fois celles-ci et la formulation.

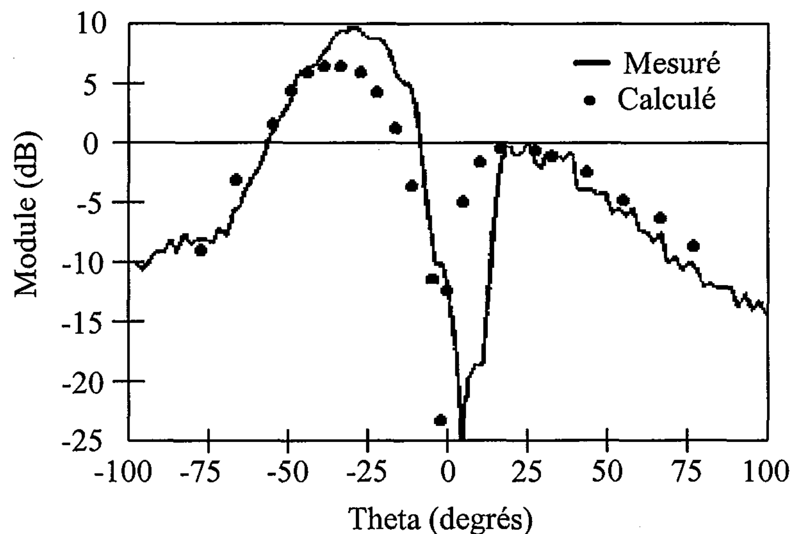


Fig. 7.8: Comparaison du module du champ lointain (H) mesuré et calculé (10 mètres) dans le plan yz pour un guide d'ondes tronqué à 60 degrés.

La bonne concordance entre les résultats de simulation et les mesures laisse à penser que le calcul du champ proche est juste. Par le biais de la simulation, il est donc possible d'accéder à

des paramètres difficilement mesurables (champ proche, champ à l'intérieur du guide, ...) ou encore d'espérer optimiser ce dispositif avant de le construire.

A titre d'exemples, les figures (7.9), (7.10) et (7.11) montrent les comparaisons entre les mesures et les calculs de champs lointains (H) pour des guides d'ondes tronqués à 90, 45 et 30 degrés. Notons que les caractéristiques de ces problèmes (fréquence, mode de la source, dimension du domaine d'étude) ainsi que le nombre de noeuds et d'éléments sont à peu près les mêmes que pour le guide tronqué à 60 degrés.

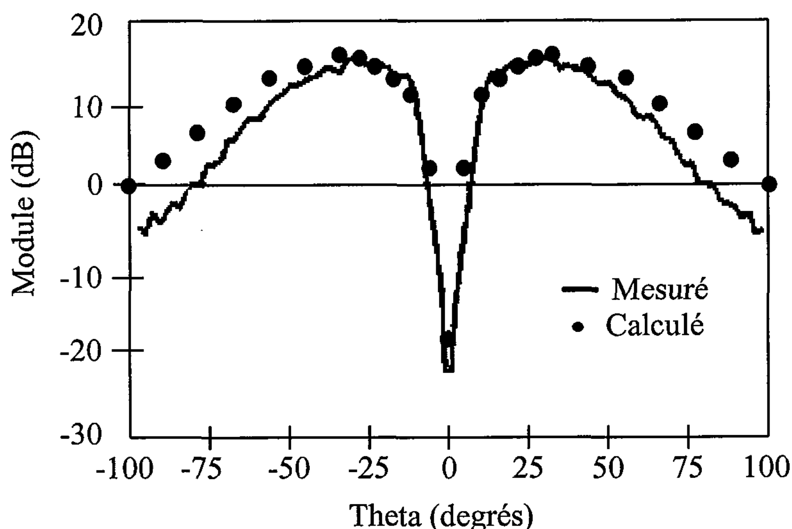


Fig. 7.9: Comparaison du module du champ lointain (H) mesuré et calculé (10 mètres) dans le plan yz pour un guide d'ondes tronqué à 90 degrés.

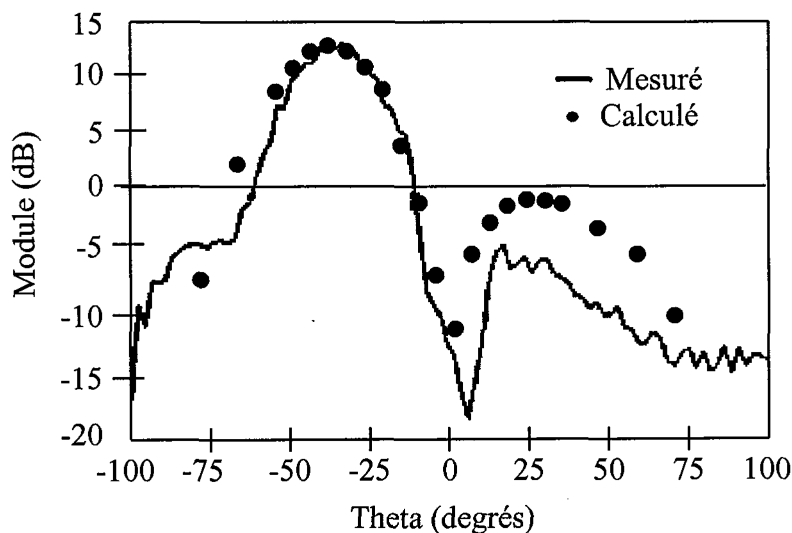


Fig. 7.10: Comparaison du module du champ lointain (H) mesuré et calculé (10 mètres) dans le plan yz pour un guide d'ondes tronqué à 45 degrés.

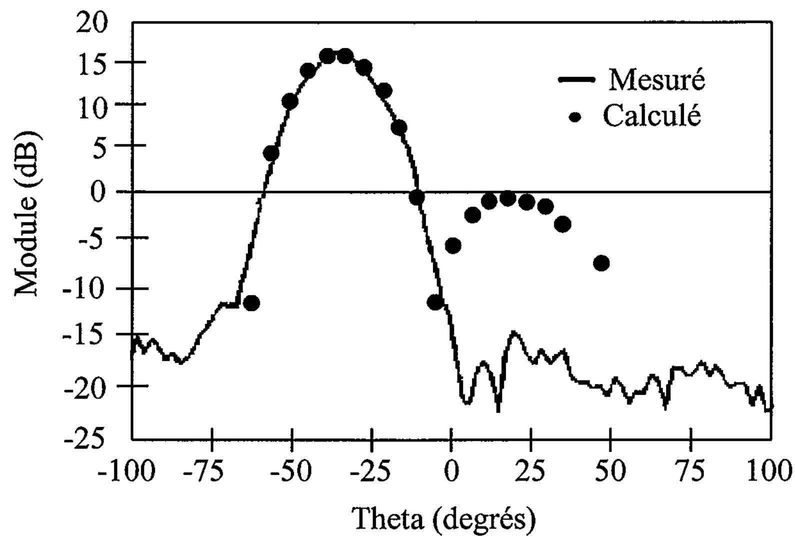


Fig. 7.11: Comparaison du module du champ lointain (H) mesuré et calculé (10 mètres) dans le plan yz pour un guide d'ondes tronqué à 30 degrés.

La différence entre les mesures et les résultats de simulation peuvent découler de plusieurs problèmes:

- Les guides sont considérés comme des cep.

- Le maillage sur la coupe du guide est constitué d'une seule couche d'éléments. Cela peu être un problème pour bien représenter les courant induits sur les angles.

- Pour les mesures en champs lointains des dispositifs sur pieds (antennes, ...) des câbles, ... viennent perturber les mesures. Ils ne sont pas pris en compte lors de la modélisation.

CONCLUSION

Dans cette thèse, une implantation, sur différents types de calculateurs parallèles, d'une formulation permettant la modélisation de problèmes de propagation d'ondes électromagnétiques en régime harmonique par la méthode des éléments finis a été présentée. Elle a été appliquée avec succès au calcul d'antennes de forte puissance et à l'analyse de problèmes de diffraction électromagnétique par une onde plane.

La méthode est basée sur la formulation de Galerkin de l'équation des ondes, en régime harmonique et en champ **E** ou **H**. L'utilisation des éléments finis nodaux impose l'introduction d'une fonction de pénalité pour éviter les modes parasites numériques. Des conditions aux limites absorbantes de type Engquist-Majda ont été adjointes à la formulation pour permettre le traitement de problèmes ouverts. Les domaines d'études sont alors parallélépipédiques ce qui permet une économie en terme d'inconnues pour la plupart des géométries. Les matrices générées par une telle méthode étant creuses et non symétriques, celles-ci doivent être symétrisées avant d'être résolues par des méthodes itératives. Ainsi le gradient conjugué a été utilisé couplé à différents types de préconditionnements.

L'étude de problèmes réalistes en trois dimensions et en hyperfréquences est rendue possible par l'utilisation du calcul parallèle qui est, à l'heure actuelle, la seule façon d'y parvenir. La formulation a donc été portée sur deux types de calculateurs parallèles qui sont les plus représentatifs du marché actuel.

L'implantation de la formulation sur le CRAY C98 s'est faite à partir du code séquentiel développé sur station de travail. La parallélisation automatique ne donnant pas de bons résultats, l'introduction du parallélisme s'est faite par l'adjonction manuelle de directives interprétées à la compilation. Cette technique a permis de garder le contrôle de la granulité de parallélisme adoptée. Néanmoins, un changement du mode de stockage de la matrice éléments finis a été nécessaire pour obtenir de bonnes performances vectorielles. L'assemblage a été réalisé par contributions élémentaires et le traitement des conditions aux limites ainsi que la symétrisation du système matriciel ont été parallélisés par distribution, à tous les processeurs, du travail à effectuer sur la matrice. Différents préconditionnements ont été couplés au gradient conjugué. Le préconditionnement diagonal, économique en espace mémoire et facilement parallélisable, s'est révélé peu adapté aux problèmes complexes générant des matrices mal conditionnées. Le préconditionnement de Cholesky incomplet, dans une version

par colonnes, s'est révélé plus performant mais plus consommateur d'espace mémoire à cause de la méthode de stockage.

Les résultats obtenus en terme de performances parallèles et vectorielles ont montré qu'une telle méthode suffisait à exploiter correctement les ressources du CRAY C98.

Les géométries traitées sur cet ordinateur comportent un nombre de noeuds bien inférieur à la limite théorique fixée par la taille maximum admissible pour le programme dans les chaînes de *batch*. Ceci est d'abord dû au problème de *post-processing* et de *pre-processing* de gros fichiers réalisés au laboratoire. Une solution consisterait à se servir de fichiers binaires qui sont beaucoup moins volumineux mais dont la génération et la lecture entre les stations de travail du laboratoire et le CRAY C98 posent quelques problèmes. D'autre part, le nombre d'heures dont nous disposions sur le CRAY C98 de l'IDRIS ne nous a pas permis de traiter beaucoup de problèmes réalistes une fois le développement du code fini.

Pour implanter la formulation sur des machines à mémoire distribuée, le code a été entièrement réécrit et les algorithmes remaniés, rendant ainsi l'occupation mémoire optimale en distribuant la matrice élément fini et en minimisant aussi le nombre de passages de messages.

Ainsi l'assemblage a été réalisé par degrés de liberté, évitant tout passage de message durant cette étape mais limitant les possibilités pour la phase de résolution. Le traitement des conditions aux limites et la symétrisation du système matriciel demandant des passages de messages sont des phases qui présentent de mauvaises performances parallèles, mais dont l'influence est négligeable du fait de la petite portion du temps de résolution total qu'elles représentent. Différents préconditionnements ont été couplés au gradient conjugué ainsi qu'au *quasi minimum residu*. Le préconditionnement diagonal, économique en espace mémoire, s'est avéré facile à mettre en oeuvre sur ce type d'architecture car seule une multiplication matrice-vecteur a dû être parallélisée. Mais, le préconditionnement généré et les performances parallèles obtenues se sont avérés décevants. Le préconditionnement de Cholesky incomplet s'est quant à lui montré inadapté à une architecture à mémoire distribuée à cause du nombre de passages de messages requis. Finalement, un préconditionnement original dit de Cholesky incomplet par paquets, s'est révélé être efficace sur tous types et tailles de problèmes.

Les problèmes traités sur la ferme de stations ont utilisé au maximum les ressources de ce calculateur. En effet, sur 8 processeurs il est théoriquement possible de traiter des problèmes maillés avec 80000 noeuds; cependant, à cause des ressources PVM requises sur chaque processeur, le nombre de noeuds ne peut en fait excéder 70000.

Le portage du code développé pour la ferme de stations sur le CRAY T3E de l'IDRIS a été assez simple dans un premier temps. Puis PVM a été remplacé par une bibliothèque non portable SHMEM pour obtenir des performances bien supérieures. Une optimisation monoprocesseur a également été réalisée. Un telle démarche rend le code non portable mais

est nécessaire, tout comme l'utilisation de SHMEM, pour utiliser au mieux les ressources du CRAY T3E. Les mêmes remarques que sur le CRAY C98 sont valables quant à la taille des problèmes pouvant être résolus sur cette machine.

En définitive, notre formulation à été implantée sur deux types de calculateurs parallèles les plus représentatifs du marché actuel. Bien que beaucoup de méthodes n'aient pu être testées notamment les techniques de décompositions de domaines, certaines règles peuvent être mise en évidence. Lors du développement de codes de calcul utilisant la méthode des éléments finis, le choix des algorithmes est déterminant pour une parallélisation éventuelle du programme. Sur des calculateurs à mémoire partagée, un simple portage d'un code séquentiel peut donner d'excellents résultats en apportant quelques modifications au programme. Les machines à mémoire distribuées ayant tendance à prendre le pas sur celles à mémoire partagée pour des raisons de prix et de possibilités matérielles, une attention toute particulière doit être portée quant au choix des algorithmes séquentiels, car la parallélisation sur ce genre de machine est beaucoup plus délicate. De ce point de vue, la méthode adoptée ici, sur les architectures à mémoire distribuée, présente l'avantage de pouvoir être explicitée facilement d'une manière séquentielle. Il n'en est pas de même pour les autres techniques.

Le calcul parallèle permet donc la modélisation de phénomènes réalistes en électrotechnique mais demande une culture et une méthodologie de programmation liées à l'architecture de la machine cible. Trouver la bonne adéquation machine / algorithmes est souvent un exercice difficile mais essentiel pour bien exploiter ces calculateurs.

A partir de l'expérience acquise, il est possible de paralléliser n'importe quelle formulation discrétisée par la méthode des EF. En effet, la plupart des formulations harmoniques en 3D conduisent à 3 inconnues complexes par noeuds (hyperfréquence: champ, magnétodynamique: potentiel, ...). Donc, la modélisation de systèmes en tous points réalistes ne peut se faire sans une parallélisation des codes mettant en jeu ces formulations. Les algorithmes développés dans cette étude peuvent donc être repris et employés à cette fin.

De plus, le développement d'une formulation temporelle en trois dimensions qui couplerait des éléments finis pour la discrétisation spatiale et des différences finies pour la discrétisation temporelle peut être envisagé. En effet, une telle formulation 3D demanderait de très grandes ressources informatiques (espace mémoire, temps de calcul,...). Mais, étant donné que les phases du code resteraient pratiquement les mêmes qu'en régime harmonique (assemblage, CL, résolution, multiplication matrice-vecteur, ...), son développement s'avère tout à fait envisageable en introduisant le parallélisme de la même manière que dans cette étude en fonction de la machine cible.

Un autre domaine dans lequel le calcul parallèle peut être d'une grande utilité est l'optimisation par des algorithmes stochastiques. En effet, de telles méthodes nécessitent souvent le calcul par EF de plusieurs dizaines voir centaines de configurations d'un même système. Le calcul parallèle peut être employé pour réduire le temps de retour soit en calculant, sur chaque processeur, des configurations indépendantes les unes des autres, soit en parallélisant le calcul EF de chaque configuration. Ce choix dépend évidemment de la taille de chaque problème à résoudre. De la même façon, en contrôle non destructif où sont mis en oeuvre des calcul paramétriques, le calcul parallèle peut être utilisé pour réduire les temps de calcul souvent prohibitifs.

Nous espérons avoir montré que le calcul parallèle est d'une grande utilité en électrotechnique. Il doit donc faire l'objet d'une recherche applicative en génie électrique sous peine de ne pouvoir traiter que des problèmes simples et de se voir distancer par les autres domaines des sciences pour l'ingénieur comme la mécanique des fluides où le calcul parallèle est couramment employé.

[1] H. MAGNIN, **Elements finis, parallélisme et calcul réparti: amélioration d'un logiciel de calcul de champs électromagnétiques pour l'électrotechnique**, Thèse *INPG*, 15 Mai 1991.

[2] J.P. BONATRE, **La programmation parallèle**, *Eyrolle.*

[3] M. COSNARD, D. TRYSTRAM, **Algorithmes et architectures parallèles**, *Interéditions.*

[4] J.L. JACQUEMIN, **Informatique parallèles et système multiprocesseurs**, *Hermes.*

[5] G. DEGHILAGE, **Architecture et programmation parallèle**, *Addison-Wesley.*

[6] M. COSNARD, N. NIVAT, Y. ROBERT, **Algorithmes parallèles**, *Masson.*

[7] M. GENGLER, S. UBEDA, F. DESPREZ, **Initiation au parallélisme, concepts et algorithmes**, *Masson.*

[8] G. PION, **Amélioration des logiciels d'élément finis par l'utilisation de méthodes d'agrégation et d'architecture parallèles**, Thèse *INPG*, 19 septembre 1984.

[9] J.P. PEREZ, R. CARLS, R. FLECHKINGER, **Electromagnétisme, fondements et applications**, *Masson*, 2nd édition.

[10] J.L. YAO BI N'guessan, **Méthode des éléments finis mixtes et conditions aux limites absorbantes pour la modélisation des phénomènes électromagnétiques hyperfréquences**, Thèse *ECLyon*, 24 janvier 1995.

[11] R.F. HARRINGTON, **Time harmonic electromagnetic field**, *Mc Graw-hill*, New York 1961.

[12] D.M. POZAR, **Microwave engineering**, *Addison-Wesley.*

[13] W.E. BOYSE, D.R. LYNCH, K.D. PAULSEN, G.N. MINERBO, "Nodal-Based Finite Elements Modeling of Maxwell's Equations", *IEEE. Trans. on Ant. and Prop.*, Vol. 40, n° 6, pp. 642-651, June 1992.

[14] B. ENGQUIST, A. MAJDA, "Absorbing Boundary Conditions for the Numerical Simulation of Waves", *Math. of comp.*, vol. 31, n° 139, pp. 629-651, July 1977.

[15] B. ENGQUIST, A. MAJDA, "Radiation Boundary Conditions for Acoustic and Elastic Waves Calculation", *P. and Applied Math.*, vol. XXXII, pp. 313-357, *Jhon Wiley and Sons*, Inc.

[16] D. GIVOLI, "Non-Reflecting Boundary Conditions", *Journal of Comp. Physics 94*, pp. 1-29, 1991.

[17] B. STUPFEL, "Absorbing Boundary Conditions on Arbitrary Boundaries for the Scalar

and Vector Wave Equations" *IEEE. Trans. on Ant. and Prop.*, vol. 42, n° 6, June 1994.

[18] V.N. KANELLPOULOS, J.P. WEBB, "3D Finite Elements Analysis of a Metallic Sphere Scattering: Comparaison of First and Second Order Absorbing Boundary Conditions", *J. Phys. III France 3 (1993)*, pp. 563-572, March 1993.

[19] R.L. HIGHTON, "Numerical Absorbing Boundary Conditions for the Wave Equation", *Math. of Comp.*, vol. 49, n° 179, pp. 65-90, July 1987.

[20] J.A. STRATTON, **Electromagnetic theory,** *Mc Graw-Hill,* New York 1941.

[21] I.D. MAYERGOYZ, "Some Remarks Concerning Electromagnetic Potentials", *IEEE. Trans. on Magn.*, vol. 29, pp. 1578-1583, 1993.

[22] R.L. FERRARI, R.L. NAIDU, "Finite Elements Modeling of High Frequency Electromagnetic Problems with Material Discontinuities", *IEE. Proc. A*, vol. 137, n° 6, pp. 313-370, 1990.

[23] S. RATNAJEEVAN, H. HASLL, **Computer aided analysis and design of electromagnetic devices,** *Elsevier.*

[24] P.P. SILVESTER, R.L. FERRARI, **Finite elements for electrical engineers,** *Cambridge university press*, 2nd édition.

[25] S. RATNAJEENVAN, SR. H. HOOLE, "Finite Element Electromagnetic Field Computation on the Sequent Symmetry 81 Parallel Computer", *IEEE. Trans. on Magn.*, vol. 26, n° 2, pp. 837-840, March 1990.

[26] R. K. AGARWAL, **Parallel Computers and Large Problems in Industry,** Computational Methods in Applied Sciences, 1992 *Elsevier Science Publishers B. V.*

[27] H. MAGNIN, J. L. COULOMB, "Amélioration des Performances Parallélo-vectorielles du Gradient Conjugué par Extension du Schéma de Stockage Matriciel", *J. Phys. III France 3,* pp. 519-529, March 1993.

[28] H. MAGNIN, J. L. COULOMB, "A Parallel and Vector Implementation of Basic Linear Algebra Subroutines in Iterative Solving of Large Sparse Linear Systems of Equations", *IEEE. Trans. on Magn.*, vol. 25, n° 4, pp. 2895-2897, July 1989.

[29] H. MAGNIN, J. L. COULOMB, R. PERRIN-BIT, "Parallel and Vectorial Solving of Finite Element Problems on a Shared-Memory Multiprocessor", *IEEE. Trans. on Magn.*, vol. 28, n° 2, pp. 1712-1715, March 1992.

[30] K. R. JAKSON, "A Survey of Parallel Numerical Methods for Initial Value Problems for Ordinary Differential Equations", *IEEE. Trans. on Magn.*, vol. 27, n° 5, pp. 3792-

3797, September 1991.

[31] I. S. DUFF, J. K. REID, "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations", *ACM Trans. on Math. Software,* vol. 9, n° 3, pp. 302-325, September 1983.

[32] C. VOLLAIRE, L. NICOLAS, A. NICOLAS, "Finite Elements for Scattering Problems on a Parallel Shared Memory Computer", 3rd international workshop on Electric and Magnetic Fields, Liège (Belgium), 6-9 May 1996.

[33] C. VOLLAIRE, L. NICOLAS, A. NICOLAS, "Finite Elements and Absorbing Boundary Conditions on Parallel Computers", *Numerical Methods in Engineering' 96, Wiley & Sons Ltd.*

[34] P. JOLY, **Mise en Oeuvre de la Méthode des Elements Finis**, *SMAI, Ellipes,* n°2, 1990.

[35] N. J. DISERENS, A. R. MAYHOOK, "Experience in the Use of Vector Processors for 3D Static Analysis", *IEEE. Trans. on Magn.*, vol. 26, n° 2, pp. 831-833, March 1990.

[36] Y. SAAD, "Krylov Subspace Method on Supercomputers", *SIAM J. Sci. Stat. Comput.,* vol. 10, n° 6, pp. 1200-1232, November 1989.

[37] O. WING, J. W. HUANG, "A Computation Model of Parallel Solution of Linear Equation", *IEEE. Trans. on Comp.,* vol. C-29, n° 7, pp. 632-638, July 1980.

[38] M. A. SRINIVAS, "Optimal Parallel Scheduling of Gaussian Elimination DAG'S", *IEEE. Trans. on Comp.,* vol. C-32, n° 12, pp. 1009-1017, December 1983.

[39] E. CHOW, Y SAAD, "ILUS: AN Incomplete LU Preconditionner in Sparse Skyline Format", *Supercomputer Institute Research Report,* UMSI 95/78, April 1995.

[40] D. GIROU, "Le multi-tâches sur machine CRAY", cours IDRIS, V. 1.3, Janvier 1994.

[41] G. GRASSEAU, H. DELOUIS, "Environnement FORTRAN sur le CRAY C98 de l'IDRIS, vectorisation-optimisation", cours IDRIS, V. 1.2.1, Février 1994.

[42] J. CHERGUI, J. ESCOBAR, D. GIROU, "Programmation par échange de messages (PVM) ", cours IDRIS, V. 2.3, Juin 1995.

[43] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, V. SUNDERAM, "PVM, Parallel Virtual Machine, a user's guide and tutorial for network parallel computing", the MIT Press, Cambridge, Massachusetts, London England.

[44] B. W. R. FORDE, R. O. FOSCHI, S. F. STIEMER, "Object-Oriented Finite Element Analysis", *Computers & Structures,* vol. 34, n° 3, pp. 355-374, 1990.

[45] E. J. SILVA, R. C. MESQUITA, R. R. SALDANHA, P. F. M. PALMEIRA, "An Object-Oriented Finite Element Program for Electromagnetic Field Computation", *IEEE Trans. on Mag.,* vol. 30, n° 5, pp. 3618-3621, September 1994.

[46] M. L. BARTON, "Three Dimensional Magnetic Field Computation in a Distributed Memory Parallel Processor", *IEEE Trans. on Mag.,* vol. 26, n° 2, pp. 834-836, March 1990.

[47] K. PREIS, G. VRISK, A. ZIEGLER, O. BIRO, CH. MAGELE, W. RENHART, K. R. RICHTER, "Distributed Processing of FEM in a Local Area Network", *IEEE Trans. on Mag.,* vol. 26, n° 2, pp. 827-830, March 1990.

[48] M. L. BARTON, J. R. RATTNER, "Parallel Computing and its Impact on Computational Electromagnetics", *IEEE trans. on Mag.,* vol. 28, n° 2, pp. 1690-1695, March 1992.

[49] X. L. LAI, O. B. WILAND, "Domain Decomposition for Indefinite Elliptic Problems", *SIAM J. Sci. Stat. Comput.,* vol. 13, n° 1, January 1992.

[50] K. IWANO, V. CUNGOSKI, K. KANEDA, H. YAMASHITA, "A Parallel Processing Method in FE Analysis Using Domain Division", *IEEE Trans. on Mag.,* vol. 30, n° 5, pp. 3598-3601, September 1994.

[51] Y. SAAD, A. V. MALVESKY, "P. SPARSLIB: A Portable Library of Distributed Memory Sparse Iterative Solvers", Supercomputer Institute Research Report, UMSI 95/180, September 1995.

[52] Y. SAAD, "Kyrlov Subspace Methods on Parallel Computers", Supercomputer Institute Research Report, UMSI 95/276, December 1995.

[53] G. J. BENDZSAK, T. W. MA, "Parallel Computation of 3D Electric and Magnetic Fields", *IEEE Trans. on Mag.,* vol. 27, n° 5, pp. 4205-4209, September 1991.

[54] A. CHATERJEE, J. L. VOLAKIS, D. WINDHEISER, "Parallel computation of 3D Electromagnetic Scattering Using Finite Elements and Conformal ABCs", *IEEE Trans. on Mag.,* vol. 30, n°5, pp. 3606-3609, September 1994.

[55] D. ZOIST, "Parallel Processing Techniques for FE Analysis: Stiffnesses, Loads and Stresses Evaluation", *Computers & Structures,* vol. 28, n° 2, pp. 247-260, 1988.

[56] C. F. SMITH, A. F. PETERSON, R. MITTRA, "The Biconjugate Gradient Method for Electromagnetic Scattering", *IEEE Trans. on Antennas and Propagation,* vol. 38, n° 6, pp. 938-940, June 1990.

[57] J. R. WHITEMAN, **The Mathematics of Finite Elements and Applications,** *Academic press,* London and New York 1973.

[58] A. JENNINGS, **Matrix Computation for Engineers and Scientitcs,** *Wiley.*

[59] R. W. FREUND, "A Transpose-Free Quasi-Minimal Residual Algorithm for Non-hermitian Linear System", *SIAM J. Sci. Compt.*, vol. 14, n° 2, pp. 470-482, March 1993.

[60] R. W. FREUND, N. M. NACHTIGAL, "QMR: A Quasi-Minimal Residual Method for Non-Hermitian Linear System", *Tech. Report NASA and USRA,* AMS(MOS): 65F10, 65N20, CR: G 1.3.

[61] R. W. FREUND, M. H. GUTKNECHT, N. M. NACHTIGAL, "An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices", *Tech. Report RIACS and NASA,* Armes Research Center, 90.45, November 1990.

[62] R. FREUND, "Conjugate Gradient Type Methods for Linear Systems with Complex Symmetric Coefficient Matrices", *SIAM J. Sci. Stat. Compt.,* vol. 13, n° 1, January 1992.

[63] C. VOLLAIRE, L. NICOLAS, and A. NICOLAS, "Finite Elements Coupled with Absorbing Boundary Conditions on parallel Distributed Memory Computer" *IEEE trans. on Mag*, vol. 33, n° 2, pp. 1448-1451, March 1997.

[64] R. LEE, V. CHUPONGSTIMN, "A Partitioning Technique for Finite Element Solution of Electromagnetic Scattering from Electrically Large Dielectric Cylinders", *IEEE trans. on Antennas and Propagation*, vol. 42, n° 5, pp. 737-741, May 1994.

[65] Y. SHIRLEY, CHOI-GROGAN, K. ESWAR, P. SADAYAPPAN, R. LEE, "Sequential and Parallel Implementations of Partitioning Finite Element-Method", *IEEE trans. on Antennas and Propagation*, vol. 44, n° 12, pp. 1609-1616, December 1996.

[66] D. GIROU, "Introduction au T3D (Aspects matériels et logiciels)", cours IDRIS, V. 1.2., Décembre 1995.

[67] D. GIROU, G. GRASSEAU, "Optimisations monoprocesseur sur T3D", cours IDRIS, V. 1.0., Octobre 1995.

[68] D. GIROU, G. GRASSEAU, "Optimisations monoprocesseur sur T3E", cours IDRIS, V. 1.0., Févier 1997.

[69] V. ALESSANDRINI, "Introduction aux CRAY T3D / T3E", cours IDRIS, V. 1.1., Février 1996.

[70] J. CHERGUI, E. GONDET, "Optimisation des communications sur T3D / T3E

(SHMEM) ", cours IDRIS, V. 1.5., Janvier 1996.

[71] http://www.idris.fr

[72] T. HORIE, H. KURAMAE, T. NIHO, "Parallel Electromagnetic-Mechanical Coupled Analysis Using Combined Domain Decomposition Method", *IEEE trans. on Mag*, vol. 33, n° 2, pp. 1792-1795, March 1997.

[73] G. MADER, F. H. UHLMANN, "A Parallel Implementation of a 3D-BEM-Algorithm Using Distributed Memory Algorithms", *IEEE trans. on Mag*, vol. 33, n° 2, pp. 1796-1799, March 1997.

[74] H. J. KIM, H. S. KIM, K. CHOI, H. LEE, H. K. JUNG, S. HALN, "Cost-Effective Parallel Preconditioner for Network-Based Computing", *IEEE trans. on Mag*, vol. 33, n° 2, pp. 1800-1803, March 1997.

[75] J. SHEN, T. HYBLER, A. KOST, "Preconditioned Iterative Solvers for Complex and Unsymmetric Systems of Equations Resulting from the Hybrid FE-BE method", *IEEE trans. on Mag*, vol. 33, n° 2, pp. 1764-1767, March 1997.

[76] H. LEE, H. JUNG, S. HAHN, "On the Convergence of the ICCG Solver on the FE Mesh", *IEEE trans. on Mag*, vol. 33, n° 2, pp. 1760-1763, March 1997.

[77] J. L. YAO BI, L. NICOLAS, A. NICOLAS, "Vector Absorbing Boundary Conditions for Nodal or Mixed Finite Elements", *IEEE trans. on Mag*, vol. 32, n° 3, pp. 848-853, May 1996.

[78] L. NICOLAS "An Integral-Type approach for the Far Field Radiated by Microwave Devices", *IEEE trans. on Mag*, vol. 30, n° 5, pp. 3124-3127, September 1994.

[79] R. VALKENBERG, G. WARZEE, P. SAINT-GEORGES, R. BEAUVENS, Y. NOTAY, "Parallel Preconditioning for FE Analysis", *Numerical Methods in Engineering' 96, Wiley & Sons Ltd.*

[80] Y. ILIASH, Y. KOUZNETSOV, Y. VASSILEVSKI, "Efficient Parallel Solving of the Potential Flow Problems on Nonmatching Grids", *Numerical Methods in Engineering' 96, Wiley & Sons Ltd.*

[81] D. DUREISSEIX, D. LADEVEZE, "Parallel and Multi-Level Stategies for Structural Analysis", *Numerical Methods in Engineering' 96, Wiley & Sons Ltd.*

[82] C. VOLLAIRE, L. NICOLAS, "Implementation of a Finite Element - Absorbing Boundary Conditions Package on a Parallel Shared Memory Computer", accepté pour COMPUMAG 97, Rio (Brasil), 7-11 Novembre 1997.

[83] C. VOLLAIRE, L. NICOLAS, "Parallel Iterative Solvers For Large Sparse Linear Systems of Equations on a Distributed Memory Computer", accepté pour COMPUMAG 97, Rio (Brasil), 7-11 Novembre 1997.

[84] S. N. VLASOV, I. M. ORLOVA, "Quasi-Optical Transformer which transforms the Waves in a Circular Waveguide to a Highly Directional Beam of Waves", *Radiophysics quantum Electronics*, n° 17, pp. 148-154, 1974.

Annexe 1: PVM

PVM est un logiciel composé d'un *deamon* et d'un ensemble de bibliothèques [43]. Il est utilisable en FORTRAN, C, C++. PVM n'est pas un langage, c'est un ensemble d'utilitaires et de bibliothèques qui offrent au programmeur des primitives avec lesquelles celui-ci pourra développer des mécanismes appropriés. Ses principales caractéristiques sont les suivantes:

pvmd3: il coordonne les différentes machines UNIX composant la machine virtuelle. Un démon est attribué par utilisateur et par machine. Le *deamon* local active automatiquement les *deamons* sur les machines distantes. Ces *deamons* remplissent toutes les fonctions utilisateur de communications et de contrôle de processus. La machine virtuelle est configurée via un fichier (*hostfile*) lu par pvmd3.

Protocoles de communications: les communications inter-*deamons* (pvmd3) utilisent UDP (mode *datagramme*): la taille du message est limitée (paquet), les communications ne sont pas sécurisées et aucun lien n'est établi pour le transit de l'information. Les communications tâches vers *deamon* utilisent TCP (mode flux): la taille des messages est illimitée, les communications sont sécurisées car un lien est établi. Les communications inter-tâches utilisent TCP. Par défaut, la communication des données transite par les *deamons* ce qui est plus rapide. Néanmoins une fonction PVM permet de modifier ce mode (fig. 1).

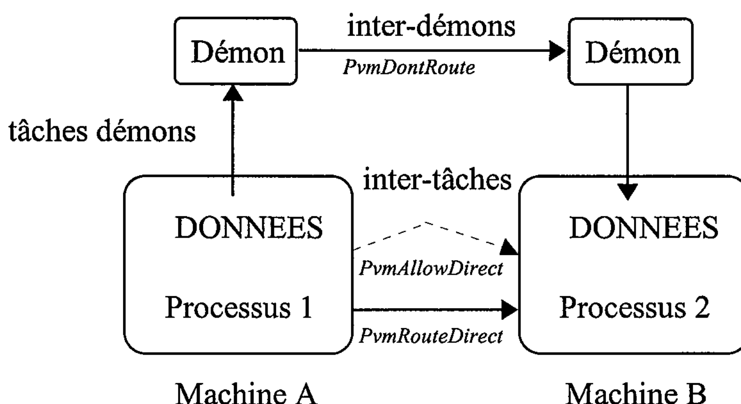


Fig. 1: Les modes de communications PVM.

La bibliothèque libpvm3.a: elle est composée de primitives permettant la gestion des tâches, des communications, des machines, des tampons et des erreurs. Lors de l'échange de messages entre tâches, ceux-ci sont copiés, par défaut, dans une zone tampon (*buffer*) avant d'être envoyés ou reçus. L'envoi d'un message est non bloquant tandis que sa réception peut l'être. Les messages peuvent être sélectionnés par un identificateur (*mstag*) ou par leur expéditeur (*tid*). Entre deux tâches, les messages arrivent dans l'ordre d'émission. L'envoi de messages nécessite l'utilisation de tampons dans lesquels seront codées les données à envoyer si le réseau est hétérogène. Il existe principalement 3 sortes de codages (fig. 2): PVMDATADEFAULT (formatage XDR), PVMDATARAW (pas de formatage XDR) et PVMDATAINPLACE (pas de formatage et les données ne sont pas copiées dans la mémoire tampon). Cette dernière option est utile sur un réseau homogène quand les données sont denses et volumineuses.

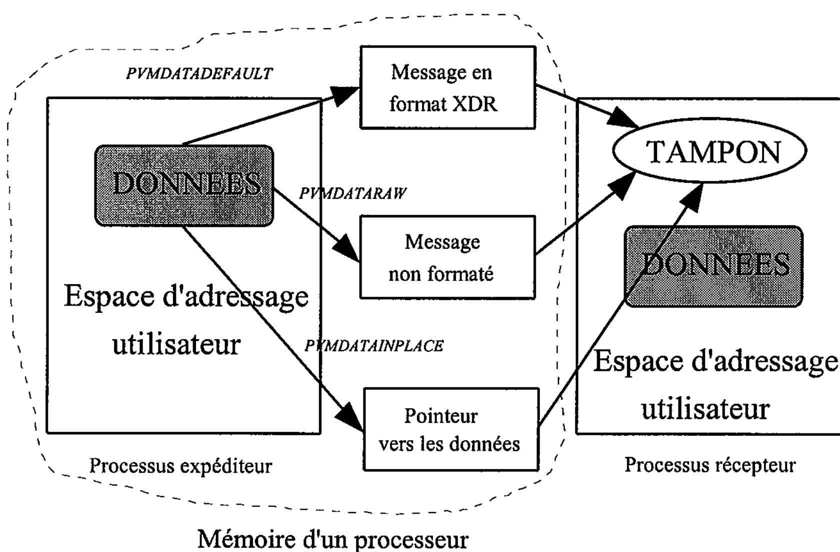


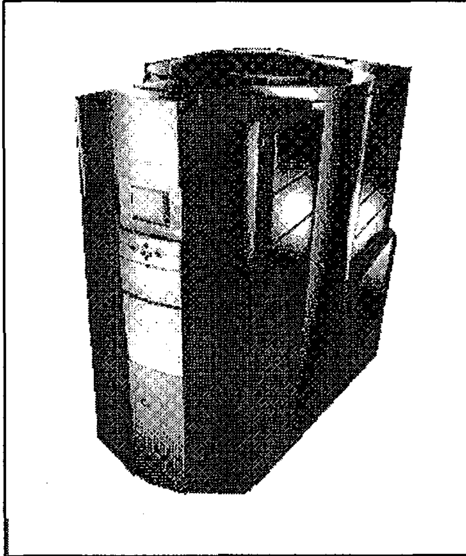
Fig. 2: Mode de formatage PVM.

La bibliothèque libgpvm3.a: elle est constituée principalement de primitives relatives à la gestion des groupes. Les groupes sont un ensemble de tâches. Leur utilisation permet des envois et des synchronisations collectifs.

L'identificateur de tâche (*tid*): à tout processus en exécution, PVM attribue un entier codé sur 32 bits appelé *tid* (analogie avec le PID de UNIX). Le *tid* est géré par PVM. Un certain nombre de fonctions PVM font appel à lui (envois, réceptions bloquantes ou non, ...).

Annexe 2: Configuration matérielle des machines

CRAY C98:



Type MIMD,

8 processeurs vectoriels,

8 registres vectoriels de 128 Mots/processeur,

Cadencement: 250 MHz,

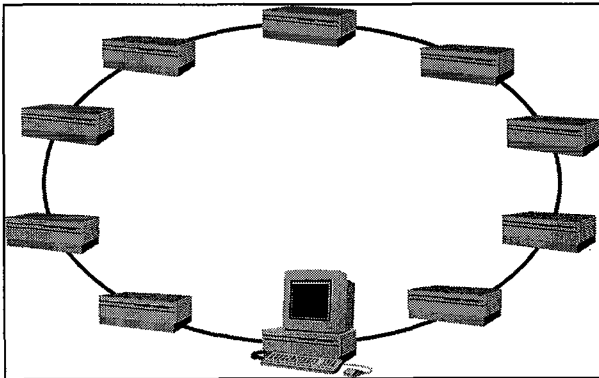
1 Gflops/s par processeur,

Mémoire partagée par tous les processeurs,

4 Go de RAM et 60 Go de *swap,*

Bande passante mémoire de 122.9 Go/s.

Ferme de stations:



Type MIMD,

10 stations de travail DEC ALPHA 300 X,

Cadencement: 175 Mhz,

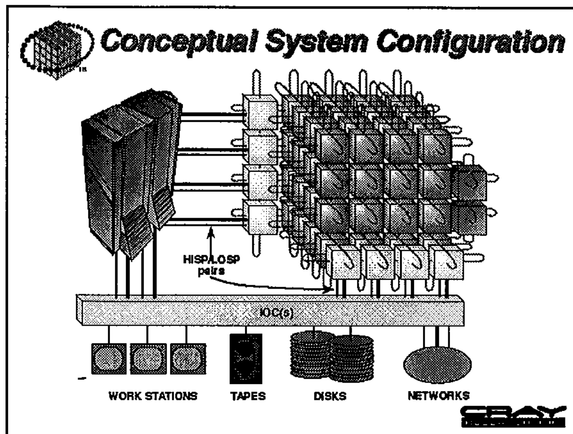
64 Mflops/s (LINPACK) par processeur,

Mémoire distribuée,

64 Mo de RAM et 160 Mo de *swap* par processeur,

Réseau de communications: anneau fibres optiques (FDDI).

CRAY T3E:



Type MIMD,

256 processeurs (DEC ALPHA EV5),

Cadencement: 300 Mhz,

600 Mflops/s par processeur,

Mémoire distribuée,

128 Mo de RAM par processeur,

Réseau de communications: tore 3D à faible temps de latence (adressage global).

Annexe 3: Algorithmes des différents solveurs

Gradiant Conjugué préconditionné:

Soit un système matriciel A x = b à résoudre. Les données sont les suivantes: matrice A et C (issue du préconditionnement) vecteurs b, x et x_0 (fixé à 0) et SEUIL (fixé à 10e-7). r, y, p et z sont des vecteurs complexes de la taille de la matrice et $\alpha, \beta, \delta_0, \delta_1$ *et* δ_{init} sont des complexes.

$r = Ax - b$

$g = C^{-1}r \Leftarrow$ Préconditionnement

$p = -g$

$\delta_0 = r^t g$

$\delta_1 = \delta_0$

$\delta_{init} = \delta_1$

Tant que $abs(\delta_1 / \delta_{init}) \geq$ **SEUIL**

{

$z = Ap$

$\alpha = \delta_0 / P^t z$

$x = x + \alpha p$

$r = r + \alpha z$

$g = C^{-1} r \Leftarrow$ Préconditionnement

$\delta_1 = r^t g$

si $abs(\delta_1 / \delta_{init}) \geq$ **SEUIL**

{

$\beta = \delta_1 / \delta_0$

$\delta_0 = \delta_1$

$p = -g + \beta p$

}

}

Les algorithmes utilisés pour les phases où intervient l'inversion de la matrice C dépendent du type de préconditionnement utilisé. Pour le préconditionnement diagonal, la matrice C étant constituée des termes diagonaux de la matrice EF, le calcul de $\mathbf{C^{-1} r}$ se résume à une multiplication vecteur-vecteur. Pour le préconditionnement de Cholesky incomplet, la matrice C est remplacée par les 2 matrices L et L^t, issues du préconditionnement de Cholesky incomplet, qui sont respectivement triangulaire inférieure et triangulaire supérieure, et la résolution de $\mathbf{C^{-1} r}$ est réalisée par une descente et une remontée.

QMR préconditionné:

Soit un système matriciel A x = b à résoudre. Les données sont les suivantes: matrice A et C (issue du préconditionnement) vecteurs b, x et x_0 (fixé à 0) et SEUIL (fixé à 10e-4). v, v1, $\tilde{v}$, $\tilde{v}1$, titi_1, titi_2 sont des vecteurs complexes de longueur égale à la taille de la matrice, g, p, p1, p2, x_tmp, h, res sont des vecteurs complexes de longueur égale au nombre de lignes stockées sur le processeur, S, toto, abs_toto, $\tilde{t}oto$, $\alpha, \alpha 1, \beta, \beta 1, \tilde{r}$, s, s1, τ, θ sont des complexes et c, c1, $\omega, \omega 1, \omega 2$, norme_h, norme_r, norme_r0 et S_carré sont des doubles.

Initialisation: v1 = 0, $\tilde{v}1 = 0$, p = 0, p1 = 0, p2 = 0, x_tmp = 0, c = 1, c1 = 1, $\theta = 0$, s = 0, s1 = 0, $\omega = \omega 1 = \omega 2 = 0$, $\tilde{r} = 0$, $\alpha = \alpha 1 = \alpha 2 = 0$, $\beta = 0$, $\beta 1 = (1 + j0)$, $\tau = 0$, toto = 0, $\tilde{t}oto = 0$, S = (1 + j0), norme_h = 0, norme-r = 10, norme_r0 = 1.

x_ tmp = x_ tmp + x (adition partielle car les vecteurs ne sont pas de la même longueur)
res = Ax
$\tilde{v}$ = b - res
g = $C^{-1}\,\tilde{v}$ ⇐ Préconditionnement
h = g
$\beta = \sqrt{\tilde{v} \cdot g}$

$$\omega = \frac{\sqrt{(\mathrm{Re}\,\tilde{v})^t\,(\mathrm{Re}\,\tilde{v}) + (\mathrm{Im}\,\tilde{v})^t\,(\mathrm{Im}\,\tilde{v})}}{|\beta|}$$

$\tilde{r} = \beta\,\omega$
v1 = v
v = g / β
res = Av
α = v.res (multiplication partielle, envoi aux autres processeurs et concaténation)

Tant que ((norme_ r / norme_ r0 ≥ **SEUIL**)
{
titi1 = ($\tilde{v}$ / β) α
titi2 = ($\tilde{v}$1 / β1) β
$\tilde{v}$1 = $\tilde{v}$
$\tilde{v}$ = res - (titi1 + titi2)
g = $C^{-1}\,\tilde{v}$ ⇐ Préconditionnement
β1 = β
$\beta = \sqrt{\tilde{v}\, g}$
ω2 = ω1
ω1 = ω

$$\omega = \frac{\sqrt{(\mathrm{Re}\,\tilde{v})^t\,(\mathrm{Re}\,\tilde{v}) + (\mathrm{Im}\,\tilde{v})^t\,(\mathrm{Im}\,\tilde{v})}}{|\beta|}$$

v1 = v
v = g / β
res = Av
α1 = α
α = v.res (multiplication partielle, envoi aux autres processeurs et concaténation)
θ = conj (s1) ω2 β1
τ = (β1 ω2 c c1) + (conj (s) ω1 α1)
$\tilde{t}$oto = (α1 ω1 c) - (s ω2 β1 c1)
abs_ toto = $\sqrt{((\mathrm{abs}(\beta))^2\ \omega^2) + (\mathrm{abs}\,(\tilde{t}oto))^2}$
c1 = c
si (abs ($\tilde{t}$oto) ≤ **EPSILON**
{
toto = complex (abs_ toto, 0)
c = 0
}
sinon
{
toto = ($\tilde{t}$oto . abs_ toto) / (abs ($\tilde{t}$oto))
c = abs ($\tilde{t}$oto) / abs_ toto
}
$s1 = s$
s = $\beta\,\omega$ / toto
p2 = p1
p1 = p

```
p = (v1 - (p1 / τ) + (p2 θ)) (soustraction partielle car les 2 vecteurs
                                  n'ont pas la même taille)
x_tmp = x_tmp + (p (r̃ c))
r̃ = - (s r̃)
S = S s
S_carré = (abs (S))²
h = h + v (r̃ c) / (ω . S_carré )
norme_h = ||h||
si norme_r = 10
  {
    norme_r0 = S_carré .norme_h
  }
norme_r = s_carré .norme_h
}
```

Concaténation de x à partir des x_tmp sur chaque processeur

QMR requiert donc plus d'opérations et de passages de messages que le GC, et pour cette raison il est moins performant que le GC.

VOLLAIRE christian   29 septembre 1997   Thèse ECL 97-26   Spécialité: Génie Electrique

---

Titre : Modélisation de phénomène électromagnétiques hyperfréquences sur calculateurs parallèles.

Mots clés : électromagnétisme - éléments finis - calcul parallèle.

Résumé

L'objectif de ce travail est le portage efficace, sur différents calculateurs parallèles, d'une formulation permettant la modélisation de phénomènes électromagnétiques hyperfréquences en régime harmonique et en espace libre. Cette démarche découle du besoin accru de modéliser des dispositifs toujours plus grands et plus complexes qui ne peuvent être résolus sur des calculateurs scalaires classiques.

Les équations de Maxwell en régime fréquenciel sont discrétisées en 3 dimensions par la méthode des éléments finis couplée à des conditions aux limites absorbantes.

Les calculateurs parallèles qui servent de support à l'implantation de la formulation sont à mémoire partagée ou distribuée. Le portage sur un CRAY C98 (mémoire partagée) est faite à partir du code séquentiel. Le parallélisme est introduit par l'adjonction manuelle de directives interprétées à la compilation. Les résultats ainsi obtenus tendent à prouver que la parallélisation des algorithmes séquentiels est une granulité adaptée à ce type d'architecture. Néanmoins, le type de stockage du système matriciel est modifié afin d'accroître les performances vectorielles. L'utilisation de calculateurs à mémoire distribuée (ferme de stations, CRAY T3E) demande un modification complète des algorithmes de sorte à exploiter au mieux les ressources de ces calculateurs en distribuant le stockage de la matrice éléments finis et en minimisant le nombre de passages de messages.

L'expérience acquise montre que le calcul parallèle permet la modélisation de géométries réalistes en 3D mais que l'utilisation de tels supports informatiques requière une bonne adéquation algorithmes - architecture cible.

Des exemples de problèmes de diffractions par des objets de grande taille sont exposés. Une application de cette méthode aux problèmes de guide d'ondes ouverts de forte puissance est également décrite. Le calcul du champ lointain calculé à partir du champ proche est en bon accord avec les résultats expérimentaux.

Title: Numerical modelization of microwave electromagnetic problems on parallel computers.

Keywords: electromagnetism - finite elements - parallel computing.

Direction de recherche
Monsieur Laurent NICOLAS, chargé de recherche CNRS
CEntre de Génie Electrique de LYon (CEGELY)
UPRESA - CNRS 5005
Ecole Centrale de Lyon, B.P. 163
69131 Ecully Cedex, France