# Parallelisation d'applications pour des reseaux de processeurs homogenes ou heterogenes

Laurent Colombet

# THESE

présentée par

**Laurent COLOMBET**

pour obtenir le titre de DOCTEUR

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Informatique**

==========

**Parallélisation d'applications**

**pour des réseaux de processeurs**

**homogènes ou hétérogènes**

==========

Date de soutenance : 7 octobre 1994

Composition du jury :

| | | |
|---|---|---|
| Président | Brigitte | PLATEAU |
| Rapporteurs | Bernard | PHILIPPE |
| | Bernard | TOURANCHEAU |
| Examinateurs | Laurent | DESBAT |
| | Denis | TRYSTRAM |

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul

# THESE

présentée par

**Laurent COLOMBET**

pour obtenir le titre de DOCTEUR

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Informatique**

==========

**Parallélisation d'applications**

**pour des réseaux de processeurs**

**homogènes ou hétérogènes**

==========

Date de soutenance : 7 octobre 1994

Composition du jury :

| | | |
|---|---|---|
| Président | Brigitte | PLATEAU |
| Rapporteurs | Bernard | PHILIPPE |
| | Bernard | TOURANCHEAU |
| Examinateurs | Laurent | DESBAT |
| | Denis | TRYSTRAM |

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul

# Remerciements

Je tiens à exprimer ici mes remerciements aux membres du jury :

A Brigitte Plateau pour l'honneur qu'elle m'a fait en présidant ce jury.

A Bernard Philippe et Bernard Tourancheau pour avoir accepté de juger ce travail malgré les contraintes temporelles. Qu'ils trouvent ici la marque de ma gratitude pour leurs précieux conseils.

A Laurent Desbat de m'avoir choisi pour son premier jury ainsi que pour le travail que nous avons fait ensemble et sans qui une grande partie de ces travaux n'auraient sans doute pas vu le jour.

Et surtout celui qui est à l'origine de ce travail, Denis Trystram, mon directeur de thèse. Je le remercie pour ces trois années sous sa direction Ô combien éclairée.

Je tiens à remercier les membres du LMC et plus particulièrement Gilles Villard et Jean-Louis Roch pour leurs conseils et leur amitié.

Un grand merci a toute la bande des thésards : Alain, Cécile, Didier, Éric, Françoise, Hervé, Nathalie, Michel, Titou, Yannick. Mais surtout, je remercie, Christophe, Frédéric, Pascal et Philippe pour leur grande amitié et leur aide qu'ils m'ont montre au cours de ces trois superbes années qui m'ont semblées bien courtes.

Je remercie également Pierre Chavy, Jean-Claude Rousseau et Jean Potier du CEA pour m'avoir facilité la réalisation matérielle de cette thèse durant les premiers mois de mon travail au CENG.

Enfin un grand merci à mes deux relecteurs : Christophe et Nathalie.

*Putain trois ans ... Inouï!*
*Le guignol Jacques Chirac (Canal+)*

# Table des matières

# Chapitre 1

# Introduction

*Les modèles mathématiques, physiques ou biologiques utilisés dans les universités ou dans l'industrie deviennent de plus en plus complexes. Si nous ajoutons à cela les prodigieux progrès réalisés dans le domaine de l'électronique pour l'acquisition de données de toutes sortes, il en résulte des modélisations de phénomènes physiques qui nécessitent une puissance de calcul très importante et toujours croissante. Par exemple l'évolution du climat sur une longue période ou l'échauffement d'une aile d'avion dans le cadre de la dynamique des fluides sont des applications très gourmandes en temps de calcul et en stockage mémoire. Une solution actuelle à ce problème est l'utilisation du parallélisme.*

## 1.1 Organisation du document

Le développement, dans le domaine du parallélisme, des bibliothèques de fonctions de communication ou de fonctions numériques, des outils et des méthodes permettent aux utilisateurs d'obtenir des codes performants et portables sur des machines à mémoire distribuée. Par exemple, il est possible de développer un code sur un réseau de stations en utilisant les méthodes de recouvrement des communications par le calcul, puis d'en calculer l'efficacité et de l'exécuter sur des machines massivement parallèles une fois la mise au point terminée.

L'objet de notre travail est d'étudier les outils et les méthodes de programmation pour les machines parallèles à mémoire distribuée. Nous porterons une attention plus particulière à une nouvelle sous-classe de machines à mémoire distribuée : les réseaux de processeurs hétérogènes. Par exemple, un réseau de stations de travail formé d'une station IBM RS6000, de deux stations DEC Alpha et d'une station SUN 4 est considéré comme une machine parallèle à processeurs hétérogènes et à mémoire distribuée.

Afin d'aider à la programmation de telles machines, l'utilisateur dispose de bibliothèques de communication par échanges de messages ainsi que de bibliothèques mathématiques. De plus, il est nécessaire de fournir des moyens d'éva-

luation de performances pour les programmes qui s'exécuteront sur les machines parallèles formées de processeurs hétérogènes. C'est ce que nous nous attacherons à mettre en évidence dans ce document dans les cinq chapitres suivant :

- **Chapitre 2** : Etude de deux bibliothèques de communication : PVM [BDG+92, BDG+94] et MPI [Mes93, The93], avec expérimentations sur différentes machines parallèles.
- **Chapitre 3** : Etude de l'accélération et de l'efficacité sur réseaux hétérogènes, avec une validation ces nouveaux outils sur différents réseaux.
- **Chapitre 4** : Etude de l'évolution d'une bibliothèque de fonctions mathématiques : LAPACK.
- **Chapitre 5** : Etude des méthodes de recouvrement du temps de communication par du temps de calcul appliquées à une fonction de base de l'algèbre linéaire : le produit matrice-vecteur.
- **Chapitre 6** : Présentation d'applications scientifiques parallèles mettant en œuvre les outils et les méthodes définis précédemment.

La suite de l'introduction est consacrée à des définitions de concepts qui seront utilisés tout au long de ce document. Bien évidemment, cette introduction est plus particulièrement orientée vers les notions qui seront développées et mises en œuvre dans ce rapport.

## 1.2 L'évolution des machines parallèles

Depuis le début des années quatre-vingts, le parallélisme s'est beaucoup développé ; en effet, beaucoup pensaient que les techniques d'intégration des transistors ne pouvaient plus évoluer de manière significative. Une des principales conséquences était que l'augmentation de la puissance de calcul des processeurs séquentiels ne pourrait pas dépasser une borne jugée trop faible. Le calcul vectoriel et le parallélisme sont donc devenus des moyens efficaces pour augmenter la puissance de calcul des ordinateurs.

Jusqu'à ces dernières années, beaucoup de machines parallèles très différentes ont été construites. L'évolution récente suivie par la plupart des constructeurs privilégie les machines à processeurs très puissants et à mémoire distribuée. Ce choix technologique repose sur plusieurs raisons.

La première est l'apparition de nouvelles méthodes d'intégration comme le VLSI[1] et le développement de la conception de circuits [MC83], qui ont permis d'accroître de manière importante la puissance des unités de calcul au-delà des prévisions initiales. Cependant, le coût de développement d'un processeur devient

[1] Very Large Scale Integration

beaucoup trop important, même pour les grands constructeurs comme IBM et Intel qui vont donc utiliser les processeurs conçus pour leurs machines séquentielles comme nœuds de calcul des ordinateurs parallèles.

Une autre raison est la nécessité, pour ces machines, d'avoir une zone mémoire très étendue afin de pouvoir y exécuter des programmes manipulant des volumes de données très importants. La mémoire partagée par plusieurs processeurs peut être une bonne réponse à ce problème, car elle est facile à mettre en œuvre et à programmer efficacement tant que le nombre de processeurs reste faible. Malheureusement cette mémoire est limitée en taille par le système d'adressage et par la difficulté d'utilisation avec un nombre important de processeurs. La mémoire distribuée est donc actuellement le seul moyen d'avoir une très grande capacité qui soit utilisable par un très grand nombre de processeurs. Ce type d'architecture permet de construire potentiellement des machines de très grande taille.

Enfin, les énormes progrès réalisés au niveau des systèmes d'exploitation des ordinateurs [TvR85], des langages et des outils parallèles entraînent une utilisation beaucoup plus facile et plus efficace de ces machines. Par exemple, la mémoire virtuelle partagée *(adressage indirect étendu avec une circulation des données assurée par le système d'exploitation)* a permis le développement d'outils et de langages de parallélisation automatique.

#### Rappel sur la classification des ordinateurs

Les ordinateurs peuvent être classifiés par la multiplicité des flots d'instructions et de données. La classification la plus connue et la plus utilisée est celle de Flynn [Fly79] :

- **SIMD** (Single Instruction stream Multiple Data stream) : il n'y a qu'un seul flot d'instructions mais plusieurs flots de données, autrement dit la même instruction est exécutée sur des données différentes.

- **MIMD** (Multiple Instruction stream Multiple Data stream) : plusieurs instructions peuvent être exécutées en parallèle sur des données différentes.

## 1.3 Les modèles de parallélisme

### 1.3.1 La base : les processus communicants

L'idée du calcul parallèle est simple : différentes parties indépendantes d'un algorithme peuvent être exécutées simultanément, pourvu que l'on dispose des données nécessaires. Nous appellerons ces différentes parties de programme des tâches. Lorsqu'une tâche s'exécute sur un processeur de la machine parallèle, nous dirons que cette tâche devient un processus. Dans la littérature, il arrive souvent

que l'on confonde la notion de tâche et de processus. En effet, la notion de processus est plus récente et est un héritage d'UNIX qui devient le système d'exploitation des processeurs de la plupart des machines parallèles. Un programme parallèle est vu alors comme un ensemble de processus coopérant à l'exécution d'un travail commun.

Le modèle de programmation par processus communicants se définit de la manière suivante : les données d'un processus sont privées et la coopération entre processus ne peut s'exprimer que par le biais de communications explicites. Si au cours de l'exécution d'un processus, son calcul nécessite l'emploi de données non locales au processus, une requête doit être adressée au processus détenant les données désirées.

Le modèle de programmation par processus communicants peut donc être vu comme des groupes d'instructions s'exécutant de façon concurrente sur des processeurs différents et s'échangeant, en cours d'exécution, des données via des messages

Les incarnations du modèle de processus communicant varient selon le degré de dynamicité ou de structuration de l'ensemble des processus coopérants ainsi que des modes de communications entre processus. Il en est de même pour l'implantation sur une machine parallèle à mémoire distribuée selon le mode d'attribution des processus aux processeurs.

Nous allons brièvement répertorier les caractéristiques les plus importantes de ce modèle.

### Les communications point à point

Les deux primitives de base de ce modèle sont celles qui vont permettre à un processus d'émettre et de recevoir des messages. Elles sont nommées traditionnellement *send* pour l'émission d'un message et *receive* pour la réception. Quels sont les paramètres à préciser pour réaliser une communication? Ils sont principalement au nombre de deux :

1. un identificateur du récepteur ou de l'émetteur du message. Il peut s'agir :
   - dans le cas d'une communication directe, du numéro du processus.
   - dans le cas d'une communication indirecte, d'une boîte aux lettres, correspondant généralement à un numéro de port, ou d'un numéro de canal (on parlera de mode connecté).
2. un tampon contenant le message à envoyer, ou à recevoir.

### Les communications globales

Des opérations de communication plus complexes que les communications point à point sont souvent nécessaires.

Par exemple, un processus peut avoir besoin de diffuser une information à tous les autres processus, c'est ce que l'on appelle une « diffusion » (ou *broadcast*). Cette diffusion peut n'être que partielle, on la nomme alors « diffusion partielle » (ou *multi cast*).

L'opération inverse de la diffusion est le « regroupement » , c'est-à-dire que tous ou plusieurs processus regroupent une information vers un processus particulier. Cette opération est appelée *gathering*.

D'autres opérations de communications s'avèrent encore nécessaires. Il s'agit de généralisations des précédentes, comme par exemple une diffusion effectuée par tous les processeurs appelée « échange total » (ou *all-to-all*).

Enfin, la dernière classe d'opérations de communications globales est formée des versions personnalisées des précédentes. On entend par « communication personnalisée » le cas où le contenu d'un message envoyé dépend du processus destinataire. Ces opérations sont appelées, respectivement :

- « distribution » dans le cas d'une diffusion personnalisée. Ce cas se présente lorsque par exemple, un processus veut distribuer les éléments d'un vecteur aux autres processus.

- « multi-distribution » (ou *personalized all-to-all*) dans le cas d'un échange total personnalisé. Une des applications les plus typiques de ce schéma est la transposition d'une matrice distribuée par lignes (ou par colonnes).

La dernière opération que l'on retrouve assez fréquemment est la « synchronisation » . Cette opération est quelque peu différente des précédentes car elle n'est pas utilisée pour échanger des informations, mais pour que tous les processus s'arrêtent à un même point dans l'exécution de leur tâche.

### La notion de groupe : structuration de l'ensemble des processus

Une dernière notion importante dans le modèle de programmation par processus communicants est la notion de groupe. En effet, il est souvent pratique de pouvoir regrouper certains processus lorsqu'ils possèdent une caractéristique commune, par exemple si quelques processus sont chargés d'effectuer un même calcul, alors que les autres processus sont occupés à un autre travail. Un groupe est communément représenté par un nom unique.

Ainsi, avec la notion de groupe, il est possible d'effectuer des « diffusions implicites » , c'est-à-dire que l'on ne diffuse plus un message à une liste de processus, mais à un nom de groupe. On peut aussi très facilement synchroniser tous les processus d'un groupe par une « barrière de synchronisation » s'appliquant uniquement à ce groupe.

### 1.3.2 Parallélisme de données

L'approche du parallélisme de données consiste à découper les données et à les répartir dans les mémoires des processeurs si l'exécution a lieu sur une machine à mémoire distribuée. Dans cette optique, un code de traitement séquentiel peut être transformé de manière automatique ou avec l'aide de directives spéciales en code parallèle. Le code parallèle obtenu après compilation peut être un code utilisant le modèle à processus communicants. Comme nous le montrerons dans le chapitre 6.4, cette approche du parallélisme est bien adaptée aux calculs sur des structures de données régulières et peut être utilisée pour une parallélisation rapide de codes de calcul existants.

Ce modèle est assez bien adapté à la parallélisation automatique d'applications ; les travaux en parallélisation automatique se développent afin de permettre le portage de gros codes séquentiels existants sur des machines parallèles en un minimum de temps.

### 1.3.3 Parallélisme de contrôle

L'approche du parallélisme de contrôle consiste à isoler, sous forme de fonctions, tous les calculs qui peuvent être effectués en parallèle. Dans cette classe se trouvent les programmes obtenus sous forme procédurale (fonctionnels ou objets) qui caractérisent le mieux cette approche. La forme la plus connue de ces derniers est la forme clients-serveurs [LL94].

**Remarque :**

Dans une sous-classification des modèles parallèle, nous trouvons deux classes de modèles de programmation qui sont les suivantes :

- **SPMD** (Single Program Multiple Data) qui signifie que le même code est placé sur tous les processeurs.
- **MPMD** (Multiple Program Multiple Data) qui implique que des codes différents peuvent être chargés sur les processeurs.

## 1.4 Langages parallèles, bibliothèques et outils pour le programmeur

Actuellement, il n'existe pas de langage parallèle standard. En général, tous les constructeurs proposent une programmation basée sur un langage standard comme C ou Fortran, augmenté d'un ensemble de fonctions de communication. La programmation d'applications avec ce type de langage nécessite l'intervention d'un spécialiste de la programmation parallèle, mais les performances obtenues

sont bien meilleures que celles atteintes par l'exécution de programmes générés par un outil de parallélisation automatique.

Basés sur les acquis des méthodes de vectorisation et du parallélisme de données, des langages comme HPF[2] [HPF93] utilisent pour la plupart une mémoire virtuelle partagée ou bien génèrent un code en langage séquentiel comprenant de plus des fonctions de gestion des données partagées entre les différents processeurs. Généralement, aucune instruction de parallélisme n'est explicitement écrite par le programmeur.

De nouveaux concepts comme le parallélisme de contrôle font leur apparition sous la forme de langages fonctionnels ou à base de RPC[3] [ATK92]. Ces langages permettent une utilisation et une mise en œuvre beaucoup plus faciles des outils de placement, de régulation dynamique de charge et de réexécution [LL94].

### 1.4.1 Bibliothèques de communication

Un effort important a été réalisé au niveau des bibliothèques de fonctions de communication. Des constructeurs et des universitaires se sont réunis afin d'établir un standard appelé MPI[4] [MPIF94]. Cette nouvelle bibliothèque de communication doit garantir à l'utilisateur une totale portabilité de son application, par exemple d'une machine IBM à une machine Cray T3D. Elle n'est pas encore disponible sur toutes les machines, mais ce que l'on peut considérer comme une pré-version, PVM [BDG+92, Man94], est disponible sur toutes les architectures parallèles et distribuées. Le futur utilisateur de machines parallèles peut donc investir des moyens humains pour une parallélisation fine de son application sans avoir à tout concevoir à nouveau quelques années plus tard pour une nouvelle génération de machines. Ces deux bibliothèques sont décrites et étudiées dans le chapitre 2.15.

### 1.4.2 Bibliothèques de calcul

Pour faciliter la programmation parallèle, des noyaux de calcul numérique ont été programmés pour tirer parti des particularités des différentes machines parallèles, comme par exemple le réseau de communication. Ces fonctions sont aussi simples à mettre en œuvre que le sont les fonctions de la bibliothèque NAG[5]. La première bibliothèque, appelée LINPACK, avait été développée pour les ordinateurs vectoriels.

[2] High Performance Fortran
[3] Remote Procedure Call
[4] Message Passing Interface
[5] Numerical Algorithms Group

### Du vectoriel au parallélisme

Le projet LINPACK avait plusieurs objectifs. Le premier était la recherche de mécanismes de production de logiciels, technique qui n'était pas encore très développée dans les années 70. Un deuxième but était de fournir une méthode permettant de mesurer les performances des applications mathématiques et de comparer les ordinateurs sur lesquels ces programmes étaient implémentés. De cette étude est née une série de jeux d'essai (*benchmark*) destinés, lors de leur conception, à aider les utilisateurs de la bibliothèque à évaluer les performances de leurs algorithmes [Don88]. Le troisième objectif était de produire une bibliothèque utilisable facilement, que chacun pourrait modifier ou étendre au gré de ses besoins.

Tous ces objectifs imposent des contraintes sur la production du code et sa documentation. De fait, il a été nécessaire d'écrire un code qui soit à la fois indépendant des machines utilisées et performant. Les routines ont été écrites en Fortran car en 1976 il s'agissait du langage scientifique le plus répandu aux Etats-Unis, il l'est encore actuellement avec la version 90. Afin de rendre les programmes plus lisibles, des conventions d'écriture ont été utilisées. C'est aussi à cette occasion qu'a été utilisée pour l'une des premières fois la programmation modulaire : les sous-programmes ont été écrits sur plusieurs sites puis rassemblés au Argonne National Laboratory pour former la bibliothèque finale.

Une autre bibliothèque de routines Fortran, appelée EISPACK, a été développée en parallèle pour résoudre les problèmes de valeurs et de vecteurs propres [AD90]. Cette bibliothèque utilise, elle aussi, les noyaux d'exécution de type BLAS *(Basic Linear Algebra Subroutines)*.

La bibliothèque la plus développée est SCALAPACK (pour *Scalable Linear Algebra PACKage*) qui propose des fonctions d'algèbre linéaire basées d'une part sur l'utilisation des BLAS pour les calculs internes à un nœud et d'autre part sur les BLACS *(Basic Linear Algebra Communication Subroutines)* pour les calculs distribués. Ces bibliothèques ainsi que les méthodes utilisées sont décrites et développées dans le chapitre 4.7.

### L'avenir : le recouvrement des communications par les calculs

Pour les programmes implémentés sur un ordinateur parallèle à mémoire distribuée, une partie importante du temps d'exécution est due au temps de communication des données non locales aux processeurs. Pour augmenter les performances, il faut minimiser le temps des communications et recouvrir le temps de communication par du temps de calcul. Cette dernière idée, *a priori* très simple, est en fait assez difficile à mettre en œuvre. Dans le chapitre 5.5, nous appliquons ce principe à un noyau de calcul d'algèbre linéaire, le *produit matrice-vecteur*. Nous avons participé à la création et à la mise en œuvre d'une bibliothèque de fonctions qui facilite l'utilisation des méthodes de recouvrement du

calcul par les communications. Cette bibliothèque, en cours d'élaboration, est appelée LOCCS (pour *Low Overhead Computation Communication Subroutine)* [DT93, CCD+94a].

### 1.4.3 Outils

Des outils de placement de tâches permettent de bien répartir les tâches sur les processeurs afin d'éviter le plus possible l'inactivité d'un processeur. En effet, pour des applications importantes et si le système d'exploitation l'autorise, il est fréquent d'avoir plusieurs tâches sur un même nœud de calcul. Il est important de bien placer ces tâches, mais aussi de pouvoir les déplacer ou les créer grâce à une *répartition de charge dynamique* pour ne pas augmenter le coût des communications ou l'inactivité des processeurs [LL94]. Malheureusement, la plupart de ces outils relèvent encore du domaine de la recherche.

L'utilisateur doit donc gérer le placement de ses différentes tâches sur les processeurs de la machine parallèle. Pour permettre la validation ou l'amélioration d'un placement, il faut que l'utilisateur récupère des renseignements concernant les échanges de messages et l'activité des processeurs.

## 1.5 Performances sur des réseaux de processeurs hétérogènes

Avec l'apparition de bibliothèques de communication par échange de messages comme PVM, les programmes développés sur une architecture comportant des processeurs hétérogènes sont de plus en plus nombreux. En effet, ces bibliothèques permettent, par exemple, d'utiliser un réseau de stations de travail comme une machine parallèle. Les programmes développés sur ces plates-formes parallèles sont de plus en plus nombreux, mais il n'existe que très peu de mesures permettant de quantifier leurs performances. Nous avons donc étendu les notions d'accélération et d'efficacité pour des réseaux de processeurs hétérogènes. Les résultats obtenus ainsi que leur validation sont présentés dans le chapitre 3.6.

## 1.6 Parallélisation d'applications

La méthode la plus simple pour paralléliser un code séquentiel existant est d'utiliser un langage dont le compilateur génère du code parallèle, comme par exemple le dernier langage défini et bientôt disponible sur les nouvelles machines, qui est HPF.

Une des méthodes les plus efficaces en terme de performances consiste à paralléliser les différents modules d'une application. Cela demande souvent un travail très long et fait appel aux dernières découvertes et outils du parallélisme. Il est

parfois nécessaire de changer de méthode numérique ou même de modéliser de nouveau le phénomène physique, mais cela permet d'obtenir de meilleures performances.

Dans le chapitre 6.4, nous présentons la parallélisation de plusieurs applications scientifiques. Nous les avons parallélisées afin d'appliquer et de valider les méthodes et les outils que nous avons développés durant cette thèse, mais aussi pour montrer que le parallélisme peut être appliqué à des modélisations importantes.

## 1.7 Conclusion

Une première partie de ce rapport est consacrée à l'étude des performances sur des réseaux de processeurs hétérogènes. Nous avons développé des méthodes assez simples pour évaluer le facteur d'accélération et l'efficacité sur ces réseaux. Une personne qui développe sur un réseau de stations hétérogènes pourra ainsi évaluer les performances de manière beaucoup plus significative.

Une seconde partie de ce rapport présente des méthodes de recouvrement du temps de communication par du temps de calcul, permettant d'améliorer les performances des algorithmes de calcul numérique. Ces méthodes peuvent être mises en œuvre dans des bibliothèques de calcul, comme SCALAPACK, afin d'en augmenter les performances sur des machines à mémoire distribuée. La création de ces bibliothèques permettra aux utilisateurs non spécialistes d'avoir accès à des machines performantes avec des codes parallèles simples et proches de l'écriture séquentielle.

Des applications développées au cours de ces trois dernières années nous ont permis de tester nos méthodes et de montrer leur intérêt sur des codes importants.

# Chapitre 2

# Outils de programmation de machine parallèle

*Nous présentons dans ce chapitre une introduction à deux bibliothèques de communication par échanges de messages (message-passing) qui représentent les standards actuels et futurs : PVM et MPI. Nous décrivons dans un premier temps les principaux concepts de PVM, qui représente un bon résumé des principales fonctionnalités des autres bibliothèques de message-passing, puis ses particularités propres. Nous avons effectué de nombreuses mesures dans le but de comparer les performances d'une même bibliothèque de communication sur différentes machines. Dans la partie suivante, nous présentons les principales caractéristiques de MPI, en insistant plus particulièrement sur les nouveaux concepts qu'elle apporte. Nous avons aussi comparé ses performances par rapport à PVM.*

## 2.1 Introduction générale

Le concept de l'informatique parallèle sur machines à mémoire distribuée s'est particulièrement développé ces dernières années. La mise en œuvre d'environnements de programmation permettant de considérer un réseau d'ordinateurs hétérogènes comme une machine parallèle est apparue nécessaire. En effet, un réseau d'ordinateurs est une modélisation comparable, à de nombreux points de vue, aux dernières machines parallèles conçues par les grands constructeurs comme Cray Research, IBM, Intel et Thinking Machines Corporation et permet de concevoir des applications parallèles à moindre coût. Les principaux enjeux de tels environnements sont l'utilisation d'un réseau de machines distribuées sous les aspects d'une machine parallèle. Ceci permet de développer une application parallèle tout en se passant d'une machine coûteuse. Bien sûr, il faut que celle-ci soit ensuite directement portable sur une machine parallèle quelconque (modulo une recompilation des programmes). Dans cette optique, des universités et des constructeurs ont alors développé des outils logiciels de programmation sur réseaux de machines

hétérogènes (parallèles, vectorielles, RISC...) appelés « Bibliothèques de Communication par Echanges de Messages » . Parmi les plus utilisées, on trouve PVM (Parallel Virtual Machine) [BDG+92], PICL (Portable Instrumented Communication Library) [GHPW90], P4 [BL92], Parmacs [CHHWar] et les BLACS que nous étudions dans le chapitre 4.7. On peut encore citer celle qui est en passe devenir le standard des bibliothèques de communication par échange de messages : MPI (Message Passing Interface) [Mes93].

Nous allons présenter et comparer dans ce chapitre les deux bibliothèques les plus représentatives, en commençant tout d'abord par le « standard » actuel à savoir PVM, puis dans un deuxième temps, peut-être le standard de demain : MPI.

# Première partie : PVM

## 2.2 Introduction à PVM

Nous allons présenter un environnement de programmation appelé PVM (≪ Parallel Virtual Machine ≫ ), qui est le plus utilisé actuellement. Développé par des chercheurs de l'université du Tennessee et du Oak Ridge National Laboratory, il est facile à mettre en œuvre et bien maintenu par ses auteurs. Nous allons détailler sa construction et son fonctionnement afin de pouvoir le comparer au futur standard : MPI.

### Basé sur un modèle de programmation par processus communicants

Le modèle de programmation permettant de modéliser des applications parallèles est la décomposition en graphes de tâches. Un programme parallèle peut donc être défini comme un ensemble d'unités élémentaires s'exécutant sur différentes ressources de calcul et s'échangeant des données. Le programmeur doit donc déterminer les différents travaux ou fonctions à effectuer en parallèle, le placement des données ainsi que les échanges de messages. Ces unités élémentaires sont appelées tâches (ou processus). Bien évidemment, les notions très classiques de groupes de processus, de synchronisation, de placement et d'ordonnancement sont utilisées par ce modèle de programmation.

PVM n'est donc pas seulement une bibliothèque de fonctions de communication par échange de messages. Une part importante de ce logiciel est consacrée à la gestion des processus et des éventuels signaux du système. Cette gestion est assurée en partie par un processus lancé lors de l'initialisation de la machine virtuelle, appelé ≪ démon ≫ . Toutes ces fonctions de gestion donnent à l'utilisateur la possibilité de maîtriser totalement la programmation de la machine virtuelle, comme par exemple la gestion des pannes, l'utilisation dynamique des groupes de tâches ou la mise en œuvre de ses propres fonctions de placement de tâches.

### Pourquoi un tel choix?

PVM présente différentes caractéristiques qui nous sont apparues indispensables pour nos applications. Le langage parallèle est constitué du langage C et de fonctions de communications. Ces dernières sont également accessibles de manière transparente en Fortran et en $C++$. PVM propose donc des primitives de communication, de synchronisation par barrière et de gestion des processus.

De plus, le placement et l'activation de processus sont explicites ou gérés par le système, ce qui permet une plus grande souplesse d'utilisation pour la programmation de grands réseaux ou de réseaux à architectures dédiées.

Si l'application est portée sur un ensemble hétérogène de machines, un format de données indépendant des machines XDR [RS91] est utilisé et une traduction spécifique est effectuée sur chaque nœud. Ceci autorise une compatibilité de PVM avec un grand nombre de machines (CRAY, stations HP, IBM, SGI, SUN(s), ...) y compris les machines parallèles les plus utilisées aujourd'hui (CM5 [MC92], SP1 [dR94], CS2 [dR94], ...).

### L'environnement de programmation

Un ensemble d'outils permet aux utilisateurs de PVM de corriger facilement les erreurs de programmation et d'augmenter les performances de leurs applications. Par exemple, PVM incorpore un mode d'exécution interactif à partir duquel le déroulement d'applications peut être surveillé : on peut voir des événements comme l'envoi ou l'attente de messages, les barrières de synchronisation, l'activation ou la terminaison de processus.

D'autres outils de développement sont aussi disponibles :

**HeNCE** est un environnement graphique de spécification et de contrôle pour les programmes PVM.

**Xab** est une bibliothèque de fonctions modifiant le comportement de PVM afin que les applications produisent des traces en cours d'exécution. C'est aussi un outil de visualisation et d'animation de traces. Le format de traces Xab est convertible au format plus répandu : PICL.

**XPVM** est une interface graphique permettant d'effectuer la prise et l'analyse de traces de programmes PVM. XPVM fournit plusieurs types de visualisation afin d'analyser le déroulement du programme. Ces vues mettent en évidence les interactions entre les différentes tâches au cours de l'exécution. Nous présentons dans la figure 2.1 un exemple d'une visualisation d'un programme PVM avec 4 tâches s'exécutant sur 4 machines. Il s'agit d'un diagramme espace/temps sur lequel on peut voir les phases d'activité (*Computing*), d'attente (*Waiting*) ou de pertes de temps dues aux communications (*Overhead*). Ce diagramme met également en évidence les communications (*Message*) entre les différentes tâches.

Enfin, comme nous le verrons dans ce rapport, il y a de très grandes similitudes avec le futur standard de communications par échanges de messages : MPI. Celles-

ci permettent d'envisager un portage assez aisé d'applications PVM sous MPI.

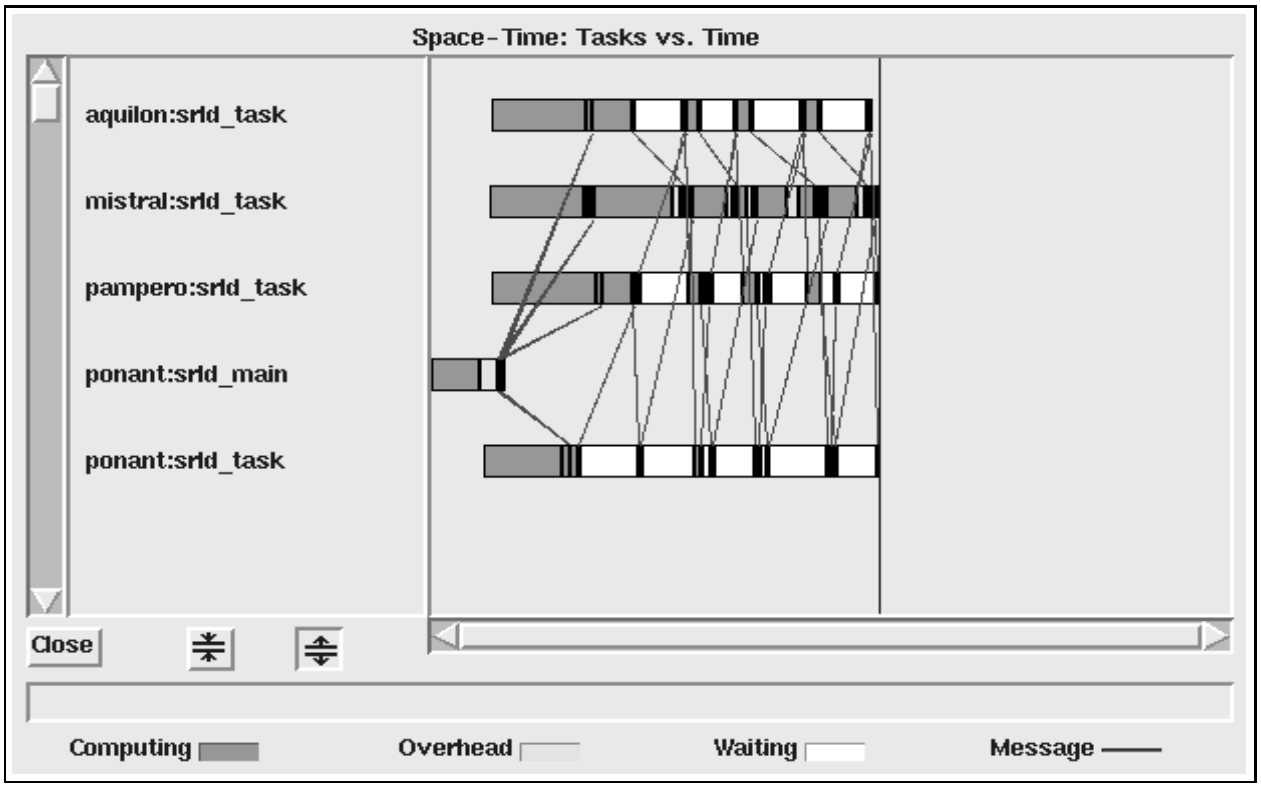



FIG. 2.1 - : *Exemple d'une visualisation de traces par XPVM.*

## 2.3 Une vision globale de PVM

Nous allons détailler maintenant les caractéristiques et possibilités de PVM.

### 2.3.1 L'identification des processus

Tous les processus enregistrés comme utilisant PVM sont représentés par un entier, appelé *tid*, qui les identifie de manière unique dans la machine parallèle virtuelle. Ces identificateurs sont donnés par le « démon » et ne peuvent pas être choisis par les utilisateurs. Ainsi, si les programmeurs ont besoin d'une numérotation particulière des différents processus, ils doivent construire une table de correspondance.

L'utilisateur définit des groupes de processus et nomme chacun d'eux. Un processus peut alors être identifié par un nom de groupe et un numéro représentant son rang dans le groupe. Cette autre possibilité d'adressage des processus peut être utilisée pour faciliter l'écriture et diminuer les risques d'erreur de conception de certaines communications globales.

De plus, PVM contient plusieurs fonctions qui permettent d'utiliser ou d'obtenir le *tid*. Cela permet aux applications d'identifier leurs différents processus afin d'éviter de nombreux problèmes lors des échanges de messages. Ces fonctions sont par exemple : `pvm_mytid` et `pvm_gettid` qui indiquent au processus courant respectivement son identificateur et ceux d'un groupe donné de processus.

### 2.3.2 Contrôle des processus

Le modèle de parallélisme mis en œuvre par PVM implique que la gestion des processus doit être assurée par le programmeur.

La fonction `pvm_spawn` permet le lancement d'autres processus sur les nœuds de calcul de la machine virtuelle et leur donne un identificateur (*tid*). Elle permet aussi un placement précis des nouveaux processus en offrant le choix du type ou même du nom de la machine cible.

Les fonctions `pvm_mytid` et `pvm_parent` retournent respectivement l'identificateur du processus courant et celui du processus qui a déclenché son exécution. Cela permet d'identifier des processus afin de leur appliquer des instructions particulières, comme un arrêt prématuré grâce à la fonction `pvm_kill` .

### 2.3.3 Tolérance aux pannes

Si l'un des ordinateurs de la machine virtuelle s'arrête ou cesse d'être accessible, PVM le détecte automatiquement et le supprime de la liste des machines utilisées. L'état d'une machine peut être obtenu par une application. Il est de la responsabilité du programmeur de rendre son application tolérante aux pannes. Il doit par exemple ne pas attendre de communications qui viendraient d'une tâche exécutée sur une machine défectueuse, car ces dernières ne seront jamais reçues et vont bloquer le programme.

### 2.3.4 Gestion dynamique des groupes

Une gestion dynamique des groupes de processus est possible avec PVM. En effet, lors de l'exécution d'un programme, des processus peuvent changer de groupe ou faire partie de plusieurs groupes afin de bénéficier d'informations diffusées à l'intérieur même d'un groupe. Les groupes sont la base des barrières de synchronisation, qui ne peuvent se faire qu'entre des tâches appartenant à un même groupe.

### 2.3.5 Les routines de communication

La bibliothèque de fonctions de communication par échange de messages a évolué de façon à se rapprocher des besoins des utilisateurs et des études menées par divers groupes de travail et universités.

Le modèle choisi est l'envoi d'une mémoire tampon appelée *buffer*, de taille théoriquement illimitée, à un ou plusieurs processus PVM. L'envoi des messages est considéré comme étant non bloquant et asynchrone, par exemple avec la fonction `pvm_send`, alors que la réception, effectuée par la fonction `pvm_recv`, est asynchrone bloquante. Elle peut être rendue non-bloquante : la fonction de réception des données retourne un identificateur indiquant si les données ont été reçues ou non. Toutefois, la gestion de l'asynchronisme est laissée au programmeur, c'est-à-dire que si les données ne sont pas disponibles, le programme doit revenir vérifier régulièrement si les données attendues sont arrivées.

En plus de la communication point à point classique, deux autres types de communication sont implémentées : la diffusion avec `pvm_mcast` et la diffusion réduite à un groupe avec `pvm_bcast`.

### 2.3.6 L'utilisation de machines parallèles

Les concepteurs de PVM ainsi que les constructeurs de machines parallèles ont développé différentes implémentations de PVM sur la plupart des machines parallèles existantes. Ceci a pour première conséquence qu'un code développé en PVM peut être directement exécuté sur des machines comme le SP1 d'IBM, la CS-2 de Meiko et la CM5 de Thinking Machines Corporation. Les messages échangés par deux nœuds d'une de ces machines utilisent directement le réseau d'interconnexion à haut débit de l'architecture. Il n'y a plus de « démon » chargé de gérer les transferts de données, le système spécialisé de la machine cible est directement mis en œuvre. De plus, un nœud d'une machine parallèle peut lui aussi faire partie d'une machine virtuelle au même titre qu'une station SUN ou qu'un CRAY C-90. Dans ce cas, un nœud de la machine virtuelle correspond soit à un processeur d'une machine parallèle soit à une machine complète.

## 2.4 Utiliser la machine parallèle virtuelle

Cette partie décrit l'interface utilisateur de PVM et donne toutes les informations nécessaires pour créer une machine parallèle virtuelle.

Pour lancer l'interface de contrôle PVM (appelée « console » ), il suffit d'exécuter la commande `pvm` qui initialise la machine virtuelle. Le premier nœud du réseau PVM est donc l'ordinateur sur lequel a eu lieu l'initialisation.

La console va permettre une gestion des différents nœud du réseau PVM, de plus elle donne la possibilité à l'utilisateur de contrôler les processus PVM et d'en vérifier le bon déroulement.

### 2.4.1 Gestion et programmation de la machine virtuelle

**Les fonctions de gestion**

- Informations :

  Les quatre fonctions `pvm_pstat`, `pvm_mstat`, `pvm_config` et `pvm_tasks` fournissent les informations les plus importantes par le programmeur désireux de rendre son application tolérante aux pannes ou de mettre en œuvre un mécanisme de répartition dynamique de la charge.

  Les deux premières fonctions donnent l'état des processus et des machines formant la machine virtuelle. Un programme peut donc être informé dynamiquement de l'arrêt d'une machine, savoir si une tâche est en attente *i.e.* bloquée en réception, s'apercevoir qu'une tâche ne s'exécute plus et contrôler l'inactivité des ordinateurs du réseau PVM.

- Configuration dynamique :

  Les fonctions `pvm_addhosts` et `pvm_delhosts` sont utilisées pour changer la configuration de la machine virtuelle (ajouter ou supprimer des nœuds) en cours d'exécution.

- Signaux :

  Grâce aux fonctions `pvm_sendsig` et `pvm_notify`, il est possible d'envoyer des signaux Unix directement aux processus PVM qui sont aussi des processus Unix, ainsi que des signaux propres à PVM. Ces derniers ont été définis par les concepteurs de PVM pour compléter les possibilités offertes par le système d'exploitation.

- Messages d'erreurs :

  La gestion des erreurs doit être entièrement assurée par le programmeur. En fait, toutes les fonctions PVM sont des fonctions qui exécutent l'instruction dont elles ont la charge sans véritable contrôle du résultat.

## 2.5 Les communications en PVM

Les communications entre les différentes tâches PVM se font par échange de messages. Le programmeur doit donc construire les messages qui vont circuler entre les tâches. Comme ceux-ci contiennent des valeurs ayant des types très différents, l'échange des données s'effectue par envoi et réception d'espaces mémoires tampons appelés buffers. Les avantages de ce mode de communication par échange de buffers sont très nombreux : par exemple, les messages peuvent être facilement découpés pour permettre la mise en œuvre des mécanismes physiques de communication rapide, ils peuvent être codés pour circuler d'une architecture

à une autre si le réseau est hétérogène et ils facilitent l'échange de type de données structurées grâce à la création par le programmeur de fonction d'empaquetage de ses données. Une communication sera toujours construite de la manière suivante :

- Envoi :
  - Allocation du buffer d'envoi
  - Empaquetage du message
  - Envoi du message
- Réception :
  - Réception du message
  - Dépaquetage du message

Les fonctions d'empaquetage et de dépaquetage permettent de remplir et vider les mémoires tampons. Il est à noter que ces dernières ne sont pas limitées en taille par PVM, mais il est important de surveiller l'espace mémoire qu'elles occupent, car il se peut qu'une des machines du réseau soit incapable physiquement de recevoir des messages trop longs.

### Buffers pour les Messages

Les fonctions `pvm_initsend`, `pvm_mkbuf` et `pvm_freebuf` permettent de gérer des buffers utilisés lors de l'exécution d'un programme. Avec l'environnement PVM, un seul buffer peut être « actif » , c'est-à-dire disponible pour recevoir des données. Cependant il est possible d'en créer plusieurs, de les remplir avec des données différentes et de choisir lequel sera actif avec la fonction `pvm_getsbuf`. Dans la plupart des cas, la fonction utilisée est `pvm_initsend`. Elle crée (ou réinitialise) et active le « buffer » d'envoi par défaut. Les fonctions `pvm_mkbuf` (resp. `pvm_freebuf`) servent à créer (resp. libérer) des « buffers » supplémentaires. De plus, c'est lors de la création de ces derniers que le programmeur a le choix d'un éventuel codage des données. Ce codage sera nécessaire si le réseau utilisé est hétérogène.

### Type de routage

La fonction `pvm_advise` permet d'optimiser le coût des communications en évitant d'utiliser le passage des données par le démon PVM. Si le volume des communications n'est pas trop important et si l'architecture de tous les nœuds de la machine virtuelle est la même (même format d'écriture mémoire), alors l'activation d'un lien direct entre les tâches n'entraîne pas une perte de la fiabilité des communications.

### Empaquetage - Dépaquetage

Les fonctions `pvm_pkbyte`, `pvm_pkfloat`, `pvm_pklong` et `pvm_pkstr` servent à remplir les « buffers » avec les types de données les plus classiques (réels, entiers, chaînes de caractères). Pour extraire les données il suffit d'utiliser dans le même ordre ces fonctions mais en remplaçant simplement `_pk` par `_upk`.

### Création de buffers complexes

PVM offre la possibilité de composer des messages de types différents. Tous les différents types de C ou Fortran sont composables.

**Exemple :**

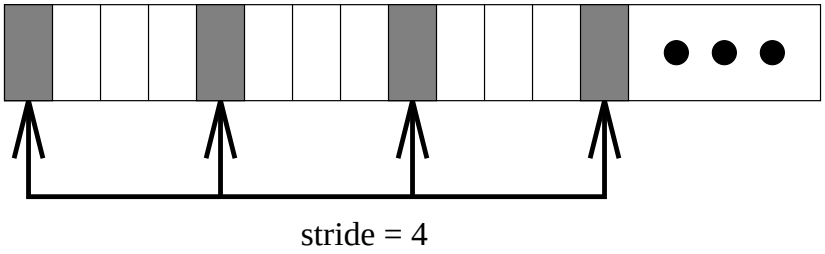
```
send_job(worker, name, wd, ht, coeffs)
         int worker;
         char *name;
         int wd, ht;
         double *coeffs;
{
         pvm_initsend(PvmDataDefault);
         pvm_pkstr(name);
         pvm_pkint(&wd, 1, 1 );
         pvm_pkint(&ht, 1, 1 );
         pvm_pkdouble(coeffs, 6);
         pvm_send(worker, 12);
}
```

Dans l'exemple précédent, la fonction `send_job` a été créée pour envoyer un message composé d'une chaîne de caractère, de deux entiers et d'un tableau de 6 réels. Pour retrouver les différentes valeurs stockées dans le buffer de réception, il suffit d'utiliser les fonctions « unpack » . Les données doivent être dépaquetées en respectant l'ordre d'empaquetage.

```
         pvm_recv(-1, 12);
         pvm_upkstr(name);
         pvm_upkint(&wd, 1, 1 );
         pvm_upkint(&ht, 1, 1 );
         pvm_upkdouble(coeffs, 6);
```

### Données non contiguës en mémoire

PVM permet de préciser, dans ces primitives d'empaquetage, le « saut » (`stride`) à effectuer entre chaque donnée.




### Communications point à point

Pour envoyer un buffer, il faut utiliser la fonction `pvm_send` et préciser quel est le numéro d'identification de la tâche cible *tid*, mais aussi quelle est l'étiquette associée au message. Cette étiquette sert à identifier les différents messages qui proviennent d'un même processeur.

La fonction `pvm_recv` est la fonction de réception bloquante. Tant que le message n'est pas arrivé, la tâche reste bloquée en attente. Pour éviter cela et donc gagner en efficacité, il faut utiliser les fonctions `pvm_nrecv` et `pvm_probe` qui prennent le premier message arrivé ou espionnent l'arrivée de messages.

### Les communications globales

Un message composé peut être :

envoyé à une liste de tâches
diffusé à toutes les tâches
diffusé à un groupe (si la bibliothèque `group` est utilisée).

La fonction `pvm_mcast` diffuse le « buffer » courant à toutes les tâches PVM dont l'identificateur *tid* se trouve dans le tableau `tids`. La fonction `pvm_bcast` diffuse le « buffer » à toutes les tâches du groupe `worker`.

De plus, toutes les autres communications globales sont facilement programmables.

## 2.6 Gestion des erreurs et des pannes

### Gestion des erreurs

- Les fonctions C de PVM retournent toutes une valeur sur l'état ou « statut » d'exécution.

- Généralement, un statut strictement négatif indique une erreur.

- En Fortran cette variable est comprise dans la liste des paramètres.

- Quand des erreurs se produisent sur des processus distribuées, « libpvm » affiche automatiquement sur la console le numéro du processus concerné, la fonction et l'erreur.

- Le rapport automatique des erreurs peut être stoppé avec la fonction : `pvm_serror`.

#### Gestion des pannes

Nous présentons ci-dessous les principales causes de panne de PVM sur un réseau de stations de travail.

- L'arrêt d'un nœud est définitif.
  *Il doit y avoir une demande de relance de l'utilisateur*

- Il y a un « TimeOut » pour les fonctions PVM.

- Un démon tué nécessite d'effacer le fichier pvmd.*id* correspondant.

- Il n'y a pas d'accusé de réception pour les messages envoyés.
  *Tâche bloquée en attente*

- Le nombre d'utilisateurs est limité sur un nœud par les contraintes du système hôte (comme par exemple sur les stations ULTRIX de DEC).

## 2.7 Tests de communications globales

Nous avons pu tester les performances de certaines des communications globales écrites avec PVM sur différentes architectures de machines parallèles. L'intérêt de ces mesures est de montrer comment l'implémentation de cette bibliothèque a été optimisée par les constructeurs et d'étudier les performances des communications les plus utilisées. Nous avons donc écrit une série de jeux d'essais nous permettant d'observer les performances pour les communications suivantes : le point à point, la diffusion et l'échange total.

### 2.7.1 Description rapide des machines

Une description plus précise est donnée dans le livre [dR94].

#### La Connection Machine 5

C'est la dernière-née de Thinking Machines Corporation (TMC) [MC92]. Elle est prévue pour interconnecter de 32 à 2048 processeurs à mémoire distribuée. Dans sa configuration actuelle, elle a une puissance maximale de 262 GFlops

pour 1024 processeurs. La machine que nous avons utilisée est composée de 32 processeurs Viking dont la puissance de crête est de 60 Mflops.

Une des principales originalités de la CM-5 réside dans l'existence de trois réseaux. Un réseau de données (*Data Network*) gère les communications point à point, un réseau de contrôle (*Control Network*) permet des opérations globales (telles que les diffusions, réductions, synchronisations, etc.) et enfin un réseau de diagnostic (*Diagnostic Network*) transmet les messages d'erreur.

La topologie adoptée pour le réseau de données est un *fat-tree* [Lei92] défini par Leiserson dans [Lei85]. Les feuilles sont les processeurs élémentaires, et les nœuds intermédiaires sont des contrôleurs-routeurs [Lei92] .

### La CS2 de Meiko

Il s'agit d'une machine MIMD à mémoire distribuée. Le réseau de communication adopté est un réseau oméga [dR94]. Les commutateurs qui constituent le réseau sont des *crossbars* $8 \times 8$, appelés Elite. Un Elite est couplé avec quatre nœuds. Chacun d'eux est constitué de quatre processeurs : deux processeurs vectoriels Fujitsu développant 200 MFlops chacun, un processeur Viking dont la puissance de crête est de 60 MFlops et le dernier processeur qui est un composant de routage, se nommant Elan. Ce processeur a une architecture RISC. Il possède le même jeu d'instructions que le Viking. Les communications se font en mode *wormhole*. Entre l'Elan et l'Elite le débit est de 50 Mo/s. La machine utilisée comprend 16 nœuds de calcul, sans compter les processeurs vectoriels.

### La SP1 d'IBM

La machine SP1 que nous avons utilisée est composée de 16 processeurs IBM Power1 (ou RS 6000) dont la puissance est de 125 Mflops pour une horloge cadencée à 62.5 Mhz. Cela donne une puissance de crête totale de 2 Gflops pour la machine utilisée. Ces processeurs sont reliés entre eux par plusieurs réseaux, deux pour l'administration du système et deux autres pour l'utilisateur.

Les deux réseaux dédiés à l'administration sont :

- Un bus RS232 chargé de collecter toutes les informations matérielles du processeur RS 6000.

- Un Ethernet qui récupère toutes les informations système, donne des accès au système d'exploitation et est utilisé par tous les fichiers nécessaires à l'initialisation de la machine.

Les deux autres réseaux partagés par les utilisateurs de la machine sont :

- Un réseau multi-étages [dR94] avec une bande passante de 20 Mo/s.

– Un Ethernet pouvant servir à l'accélération des accès disque et à transmettre des informations sur les communications passant par le réseau multi-étages.

**Tableau récapitulatif**

| Machine | Réseau | #P | Bande Passante (*crête*) |
|---|---|---|---|
| TMC CM5 | fat-tree | 32 | 20 $Mo/s$ |
| Meiko CS2 | multi-étages | 16 | 50 $Mo/s$ |
| IBM SP1 | multi-étages | 16 | 20 $Mo/s$ |

Malheureusement, seules les mesures effectuées sur 8 processeurs étaient complètes et nous permettaient de réaliser des comparaisons significatives. Toutes les courbes représentent donc des échanges de messages sur 8 processeurs. De plus, nous avons répété plus de 100 fois toutes les mesures afin de limiter le plus possible les irrégularités qui sont généralement dues au système d'exploitation (les machines étant reservées pour nos tests).

### 2.7.2 Les communications point à point

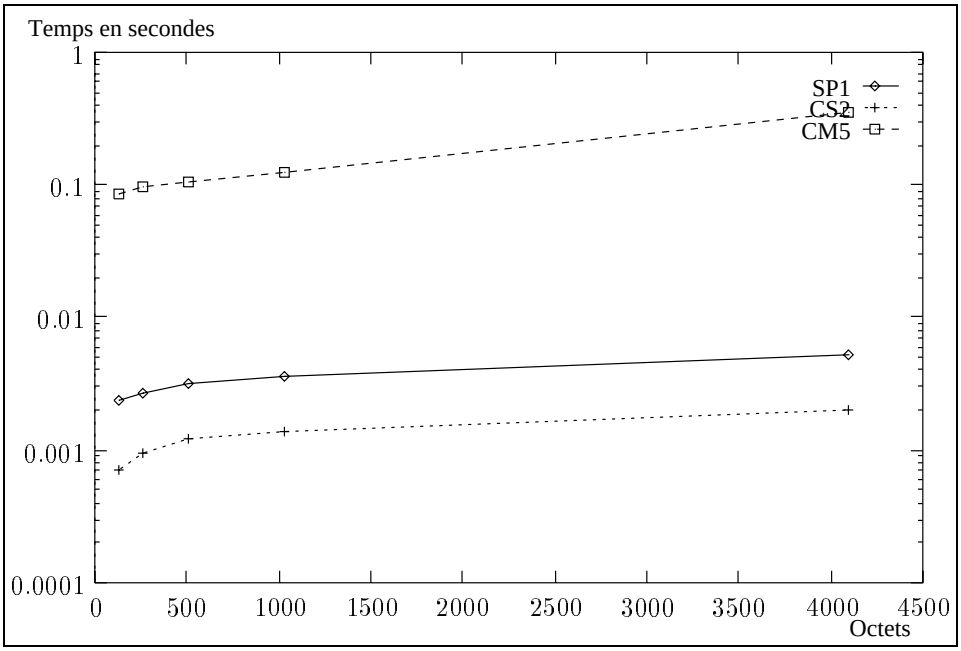



FIG. *2.2 - : Échange entre deux processeurs.*

On remarque que le temps d'une communication point à point est beaucoup plus élevé sur la Connection Machine 5 (CM5) que sur les deux autres machines. La cause de cet écart est la mauvaise implémentation de PVM sur la CM5. Cette

hypothèse est confirmée par les graphiques 2.3 et 2.4. La gestion des communications sur la CM5 est très particulière et elle n'est pas bien adaptée au modèle d'échange de messages proposé par PVM. Mais une nouvelle version beaucoup plus performante est disponible depuis peu. Plus de détails sur l'architecture et la gestion des communications sont donnés dans le livre [dR94].

On peut remarquer que grâce à son réseau beaucoup plus performant, la CS2 a un temps de communication meilleur que celui de l'IBM SP1. De plus, le rapport de la bande passante mesurée de la CS2 sur celle de la SP1 est le même que le rapport obtenu avec les valeurs théoriques, ce qui montre que pour les communications point à point les deux implémentations sont équivalentes.

### 2.7.3 La diffusion

Nous avons testé la diffusion, très souvent utilisée et qui a tendance à charger de façon importante le réseau de communication des machines parallèles, et il nous semble important d'en connaître la qualité d'implémentation.

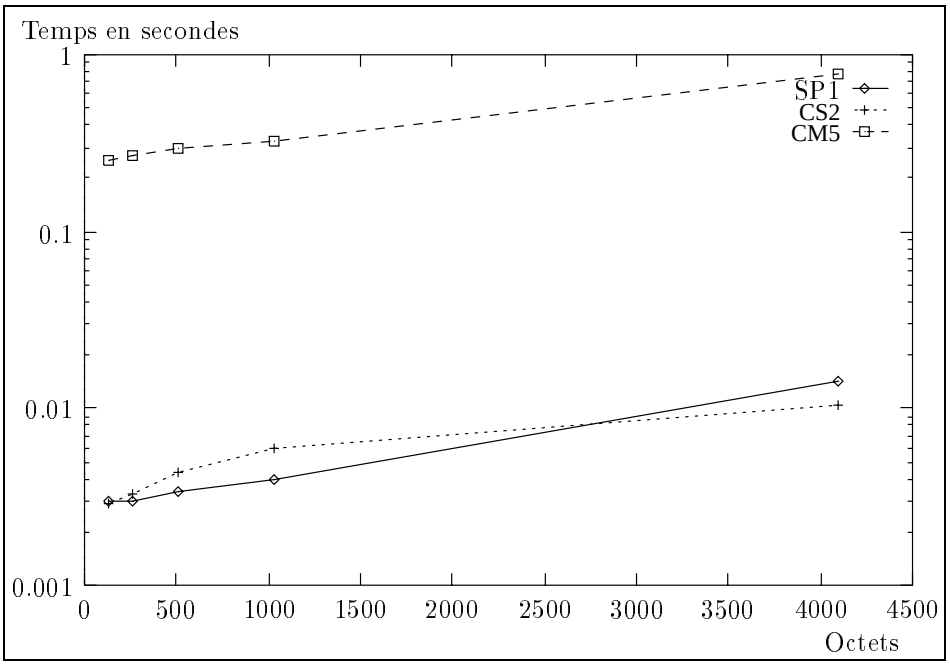



FIG. 2.3 - : *Diffusions sur des machines à 8 processeurs.*

Si le coût des communications est toujours aussi important pour la CM5, on observe que l'écart entre les deux autres machines a diminué. On remarque que le temps d'une diffusion effectuée sur des données de petite taille est plus faible sur la SP1, mais, dès que la taille des données devient importante, la CS2 communique le plus rapidement car c'est la bande passante du réseau qui induit

l'essentiel du temps de la diffusion. On peut en déduire que l'implémentation de la diffusion sur la SP1 utilise un algorithme plus performant mais que cette dernière est handicapée par les faibles débits de son réseau.

**Remarque** : Pour étudier le temps de communication d'un algorithme parallèle, il faut connaître la taille des données à diffuser. En effet, pour des données de petite taille, le temps de la diffusion est très fortement influencé par la qualité de l'implémentation de PVM. Un même algorithme s'exécutera plus rapidement sur la même machine avec une bibliothèque plus performante. Par contre, si la taille des données est importante alors l'évolution de la bibliothèque aura très peu d'influence sur le temps d'exécution.

### 2.7.4 L'échange total

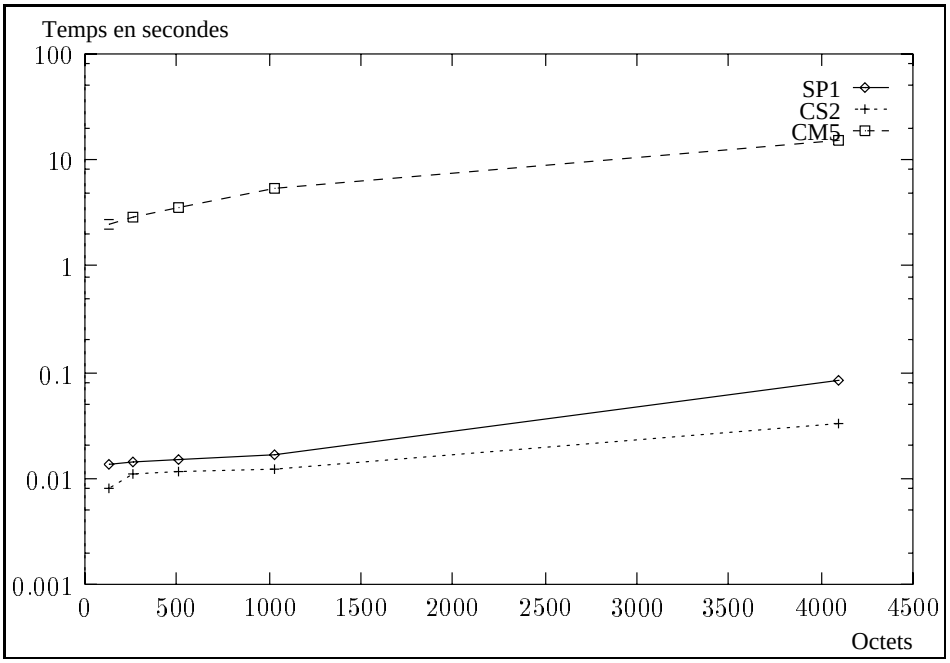



FIG. *2.4 - : Échange total sur des machines à 8 processeurs.*

Malgré une bonne implémentation de l'échange total, l'IBM ne peut pas rivaliser avec la CS2 pour cette communication qui utilise fortement les possibilités de la bande passante du réseau d'interconnexion. Pour l'échange total (figure 2.4) le gain obtenu par une bonne implémentation ne suffit plus à masquer la faiblesse du réseau. Même si pour des petites tailles la différence entre l'IBM et la CS2 est faible, celle-ci croît très rapidement avec l'augmentation de la taille des données échangées.

### 2.7.5 Comparaison de plusieurs bibliothèques sur SP1

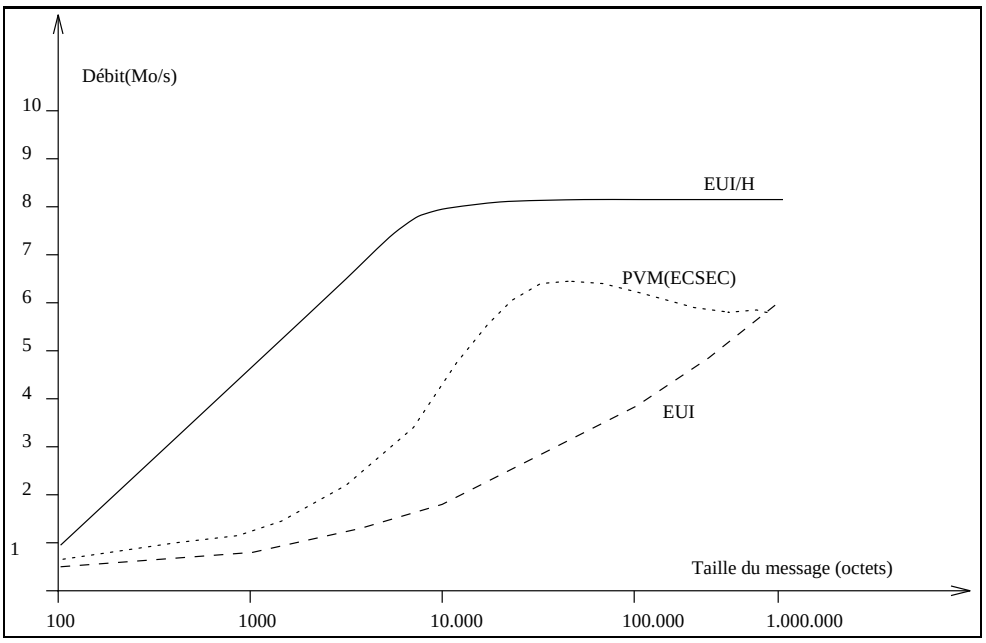



FIG. *2.5 - : Comparaison des trois principales bibliothèques de communication.*

La bibliothèque EUI a été conçue par IBM en prenant modèle sur les spécifications retenues pour la bibliothèque MPI. De plus, elle utilise aussi bien le protocole de communication `lsp` d'IBM que le standard `tcp/ip` sur le réseau multi-étages de la machine. La version hautes performances appelée EUI/H utilise la couche de communication la plus basse. Comme on le voit sur la figure 2.5, cette dernière est très efficace, malheureusement elle ne permet pas l'utilisation de la machine SP1 en mode multi-utilisateurs. Cette figure nous confirme aussi la bonne implémentation de PVM.

### 2.7.6 Mémoire virtuellement partagée avec PVM : le Cray T3D

Le Cray T3D a une architecture MIMD, avec une mémoire distribuée, mais accessible globalement par tous les processeurs. Chaque nœud du réseau est constitué de deux processeurs Alpha. Le réseau de communication est une grille tridimensionnelle. Pour une description plus détaillée on peut consulter la documentation Cray [CR94]

Grâce à l'adressage global de la mémoire distribuée du T3D, la version de PVM proposé par Cray est très rapide. En effet, l'envoi des données se fait en deux étapes : un envoi classique utilisant le mode de communication par échange

de message, suivi d'une lecture directe dans la mémoire du processeur source pour le reste des données. Un tel échange est schématisé dans la figure 2.6.

**Remarque** : La lecture de données en mémoire distante se fait par paquet de 4096 octets au maximum. Cette taille correspond à la capacité maximale du « cache » interne du processeur Alpha.

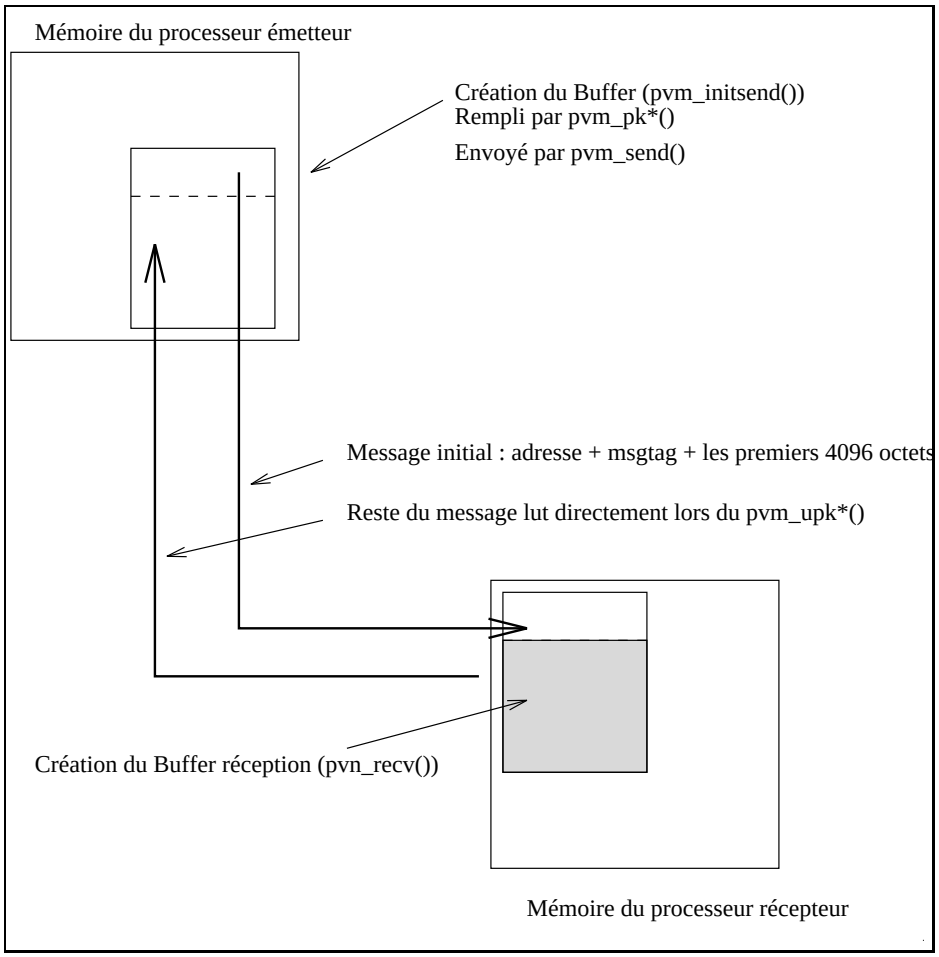



FIG. *2.6 - : Implémentation de l'envoi de message PVM sur le Cray T3D.*

Les latences mesurées pour un *pvm_send* classique entre deux processeurs sont les suivantes :

| taille du message | latence en $\mu s$ |
|---|---|
| $4096 \leq$ | 150 |
| $> 4096$ | 245 |

De plus, la latence de la lecture du reste du message par paquets de taille 4096 est de 6.2 $\mu s$. Ceci montre bien l'intérêt d'utiliser l'adressage direct de la

mémoire sur cette machine. Enfin, l'utilisateur peut, grâce à une bibliothèque appelée *Shmem*, gérer l'écriture et la lecture dans des mémoires distantes.

Nous avons voulu comparer les performances des bibliothèques PVM et Shmem du Cray T3D avec les différentes versions de PVM implémentées sur l'IBM SP1 pour un échange de message entre deux processeurs.

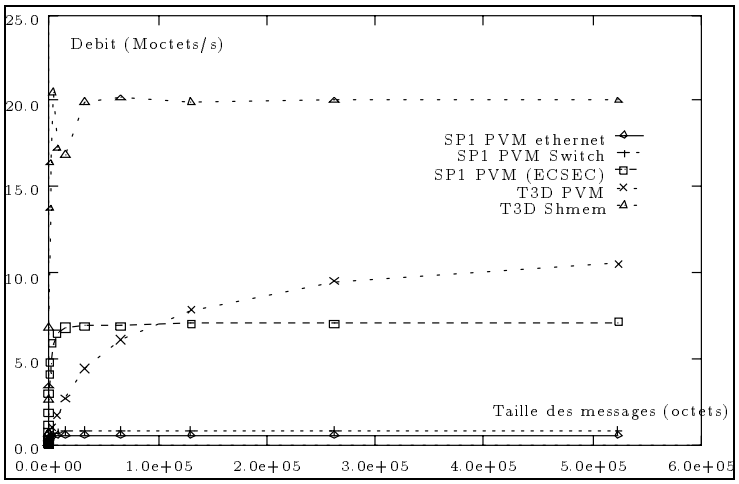



FIG. *2.7 - : Performances de l'échange de données entre deux processeurs.*

Comme nous pouvons le constater sur la figure 2.7, les meilleures performances sont obtenues par les bibliothèques Shmem et PVM du Cray T3D. Cependant, en comparant les débits mesurés et les débits théoriques, nous constatons que PVM(ECSEC) est le plus efficace avec un rapport de 7.5 Moctets/s sur 20 Moctets/s comparé au rapport 20 Moctets/s sur 150 Moctets/s pour la bibliothèque Shmem. Nous pouvons aussi remarquer que pour ces deux bibliothèques le maximum de performance est obtenu pour des petites tailles de messages.

# Deuxième partie : MPI

## 2.8 Introduction

**M.P.I.** (Message Passing Interface) est une proposition de standard de système de communication par échange de messages. Elaboré par des universitaires et par la plupart des constructeurs de machines parallèles, il représente un consensus sur ce qui est nécessaire pour qu'un tel système soit portable sur toutes les architectures parallèles. La forme actuelle de ce futur standard est une bibliothèque de communication par échange de messages (« Message Passing » ), rassemblant les meilleurs éléments des systèmes déjà existants.

### Les objectifs de MPI

Les objectifs visés par la mise en place du standard MPI sont très nombreux et ambitieux. Nous allons en détailler quelques-uns afin de montrer les possibilités de ce système.

L'objectif principal est d'obtenir des communications efficaces et portables, grâce à la mise en œuvre de mécanismes de « bufferisation » des messages échangés. Cela permet un codage des données et donc une prise en compte de l'éventuelle hétérogénéité des ressources de calcul, mais aussi de minimiser les recopies de données, de permettre le recouvrement du calcul et des communications et de déléguer au processeur spécialisé (s'il existe) le traitement et la gestion des communications.

De plus, MPI doit permettre l'utilisation de langages classiques comme C, Fortran ou C++ et doit constituer une interface de communication fiable, l'objectif final étant de concevoir une interface pour la programmation des applications.

### Quelques définitions propres à MPI

- Les notions de **groupes de processus** et de **communicateurs** sont introduites pour permettre de structurer l'ensemble des processus et l'espace de communication d'une tâche et plus généralement d'une application formée de plusieurs tâches. Un **groupe** est un ensemble ordonné de processus et un processus est toujours identifié par son rang dans le groupe. Un **communicateur** détermine le cadre des communications au sein d'un groupe ou entre plusieurs groupes.

- Une procédure de communication est dite **non-bloquante** si elle se termine avant que l'opération de communication associée ne soit finie, et avant que le programmeur ne soit autorisé à réutiliser les buffers invoqués lors de l'appel.

- Une procédure de communication est dite **localement bloquante** si le retour de cette procédure indique si le programmeur est autorisé, ou non, à réutiliser les ressources spécifiées lors de l'appel.

- Une procédure de communication est dite **globalement bloquante** si le retour de cette procédure indique la terminaison de l'opération de communication associée.

- Un **programme MPI** consiste en un ensemble de processus autonomes, exécutant leur propre code. Les codes exécutés par chaque processus n'ont pas à être identiques. Les processus communiquent via les primitives de communications spécifiées par MPI. MPI ne spécifie pas le modèle d'exécution de chaque processus, ni l'interaction avec des signaux dans un environnement UNIX, ou d'autres événements n'ayant pas de relations avec les communications MPI.

L'apport de MPI par rapport aux bibliothèques actuelles de communication par échange de messages se résume en différentes généralisations :

- généralisation de la description des buffers, c'est-à-dire la possibilité de construire des types de données décrivant des structures complexes, ou envoyer des données non contiguës en mémoire (exemple typique d'une matrice)

- généralisation de la notion d' « étiquette » associée à un message par l'utilisation d'un contexte lié à une famille de messages

- généralisation de l'identification de processus par la notion de groupe et de communicateur.

## 2.9 Les communications avec MPI

### 2.9.1 Les communications point à point

Les opérations de base sont l'émission et la réception d'un message. Les informations nécessaire à ces opérations sont pratiquement les mêmes que pour les fonctions PVM. La première information à transmettre est l'adresse du buffer à envoyer, puis le type des éléments des données et le nombre d'éléments. On remarque que le type des données est beaucoup plus évolué que celui fourni par PVM. Il décrit l'organisation des données en mémoire et permet d'avoir des structures plus complexes définies par l'utilisateur. Les autres informations nécessaires sont l'identification des processus source et destination. Celles-ci sont obtenues par l'intermédiaire d'un communicateur, spécifiant à la fois le groupe, le contexte et le rang des processus dans leur groupe. En outre, il est nécessaire,

comme avec les communications de type PVM, de spécifier une étiquette qui peut être considérée comme une sécurité supplémentaire ou comme le moyen le plus simple de lever une ambiguïté pour certains échanges. Il est à noter que l'opération de réception comporte un argument de plus permettant de récupérer l'état de l'opération, à savoir la longueur du message effectivement reçu, la source et la destination finale.

Les versions bloquantes des opérations d'émission et de réception ont la forme suivante :

- `MPI_SEND`*(buf,cnt,type,dest,tag,comm)*
- `MPI_RECV`*(buf,cnt,type,src,tag,comm,status)*

La fonctions `MPI_SEND` est localement bloquante, autrement dit elle se termine dès que le processus appelant peut réutiliser son buffer. Il existe deux autres modes d'émission, correspondant à des variantes des opérations précédentes. Le mode *prêt*, dans lequel l'envoi peut se faire que si la commande de réception correspondante a été lancée, et le mode *synchrone* dans lequel l'envoi d'un message ne se terminera que lorsque le message aura été entièrement reçu. Comme pour PVM, les diverses implémentations de MPI doivent garantir le fait que des messages émis par la même source vers une même destination dans le même contexte sont reçus dans l'ordre où ils ont été émis. De plus, si un envoi et une réception sont initiés par deux processus différents, l'une au moins de ces opérations s'achèvera et cela quel que soit le contexte. Par contre, l'équité n'est pas garantie. Ainsi, un message peut ne jamais être reçu car les réceptions correspondantes auront été satisfaites à chaque fois par un autre message.

Les versions non-bloquantes sont :

- `MPI_ISEND`*(handle,buf,cnt,type,dest,tag,comm)*
- `MPI_IRECV`*(handle,buf,cnt,type,src,tag,comm)*

Les opérations non bloquantes permettent de séparer l'initialisation de la communication de son achèvement. Il est alors possible de masquer du calcul par des communications.

### 2.9.2 Les communications globales ou collectives

Les communications globales doivent être exécutées par chacun des processus du groupe concerné. Elles sont considérées comme une des caractéristiques les plus importantes de MPI pour l'implémentation d'applications portables et efficaces. Liée aux communications globales, on retrouve notamment la notion de barrières de synchronisation avec `MPI_BARRIER`. Cette fonction bloque le processus appelant jusqu'à ce que tous les processus du groupe l'aient appelée.

Les opérations de communications globales fournies par MPI sont les suivantes :

1. **Broadcast** : la diffusion d'un membre vers tous les autres (OTA : One-To-All) est réalisée par la fonction `MPI_BCAST`*(buffer,count,datatype,root,comm)*. La fonction est appelée avec les mêmes valeurs pour les paramètres *root* et *comm*, mais elle aura un comportement différent selon que le rang du processus est égal à *root* (émetteur) ou non (récepteur).

2. **Gather** : le regroupement de tous les membres du groupe vers un élément particulier (ATO : All-To-One) est obtenu grâce à la fonction `MPI_GATHER`.

3. **Scatter** : la diffusion personnalisée (POTA : Personalized-One-To-All) est possible à la fonction `MPI_SCATTER`.

4. **Reduce**, **Scan** : les opérations globales sur tous les membres du groupe comme max, min, somme, etc ... sont activées par la fonction `MPI_REDUCE`.

5. **All_broadcast** : l'échange total ou diffusion depuis tous les processeurs (ATA : All-To-All) est fournie par `MPI_ALLTOALL`

6. **All_gather** : l'échange total personnalisé (PATA : Personalized-All-To-All) est possible avec la fonction `MPI_ALLGATHER`.

### 2.9.3 Les opérations globales

Les opérations globales consistent à effectuer une opération sur les données fournies par chacun des processus d'un groupe, le résultat étant envoyé à l'un des processus, ou redistribué à tous les processus. L'opération de base est la réduction :

`MPI_REDUCE`*(sendbuf,recvbuf,count,datatype,op,root,comm)*

Il existe différentes opérations qui peuvent être associée à la réduction [Mes93, The93]. Le résultat de la réduction peut être transmis à tous les processus du groupe en utilisant la fonction `MPI_ALL_REDUCE`. De plus, une combinaison de la réduction et de la dispersion est disponible. Cette fonction, `MPI_REDUCE_SCATTER` effectue la réduction par composante des vecteurs fournis par chaque processus, puis répartit les composantes du résultat entre les processus.

## 2.10 Groupes, contextes et communicateurs

Les notions de groupe et de contexte forme la caractéristique la plus importante de MPI, en particulier pour l'implémentation de bibliothèques parallèles. Elles permettent de structurer l'espace des processus et l'espace des communications d'une tâche (un groupe de processus) et d'isoler les différents contextes

de communication les uns des autres. En particulier, cela permet d'éviter que les communications d'un programme utilisateur n'interfèrent avec celles d'une bibliothèque ou d'une autre application.

### 2.10.1 Groupe de processus

Un *groupe de processus* est un ensemble ordonné de processus où chacun d'eux est identifié de manière unique par son rang dans le groupe. Pour un groupe de $n$ processus, le rang va de 0 à $n-1$.

Les groupes de processus peuvent être utilisés de deux manières différentes, la première pour spécifier les processus qui sont concernés par une communication globale, comme une diffusion, la deuxième pour introduire la notion de parallélisme de tâches dans une application, c'est-à-dire un changement de granularité. A un groupe de processus est associée une tâche ; si les codes exécutés dans chaque groupe sont différents, alors nous aurons affaire à un parallélisme de tâches de type MPMD [1]. Par contre, si toutes les tâches exécutent le même code, alors nous identifierons un parallélisme de type SPMD [2].

Au lancement d'une tâche, il existe un groupe prédéfini `MPI_GROUP_ALL` composé de l'ensemble des processus de toutes les tâches. A partir de ce moment, on peut créer d'autres groupes ou sous-groupes. Bien évidemment, il est possible de déterminer la taille d'un groupe, le rang du processus appelant dans un groupe, de réordonner tous les processus d'un groupe et de détruire un groupe et donc une tâche MPI.

Bien que le modèle de processus MPI soit statique, les groupes de processus sont gérés dynamiquement, dans le sens où ils peuvent être créés et détruits et où chaque processus peut appartenir à plusieurs groupes à la fois.

### 2.10.2 Contextes de communication

Les *contextes de communication* ont été initialement proposés afin d'autoriser la création de parties de messages distinctes et séparables, avec un contexte associé à chaque partie de message. Cette notion de contexte permet donc de partitionner l'espace de communication. Autrement dit, un message émis avec un contexte de communication ne peut être reçu qu'avec le même contexte.

Une utilisation commune des contextes est de vérifier si les les messages envoyés lors d'une phase d'une application ne sont pas interceptés par une autre phase. En effet, comme le disent Skjellum, Doss et Bangalore dans [SDB94] :

« *There is no reasonable way for libraries to isolate themselves from the ongoing point-to-point message passing present in a running application* » .

[1] Multiple Programs Multiple Data
[2] Single Program Multiple Data

Les étiquettes associées aux messages ne sont pas suffisantes. Par exemple, plusieurs bibliothèques (ou plusieurs appels à la même bibliothèque) peuvent utiliser les mêmes étiquettes.

Pour résoudre ce problème, l'étiquette d'un message est remplacée par un *contexte* rassemblant les informations suivantes : étiquette (message tag) + numéro du processus dans le groupe + numéro de groupe. Ainsi constitués, les contextes permettent la construction d'espaces indépendants de types de messages.

L'utilisateur n'effectue jamais d'opérations explicites sur les contextes, mais ceux-ci sont maintenus de telle sorte que les messages envoyés via un communicateur ne peuvent être reçus que par le bon communicateur.

### 2.10.3 Objets communicateurs

La portée d'une opération de communication est spécifiée par le contexte de communication utilisé et le, ou les, groupes de processus concernés.

Lors d'une communication globale, ou lors d'une communication point à point entre membres d'un même groupe, seul le groupe concerné a besoin d'être spécifié. La source et la destination de la communication sont données par le rang des processus correspondants dans le groupe considéré. Lorsque la communication se fait entre des processus n'appartenant pas au même groupe, il faut préciser le rang de chacun des deux processus dans leur groupe et les groupes correspondants.

Les objets appelés *communicateurs* servent à définir la portée de l'opération de communication. Un *intra(inter)-communicateur* est un communicateur utilisé pour une communication intra(inter)-groupe. Un intra-communicateur peut être vu comme un objet rassemblant contexte et groupe, tandis qu'un inter-communicateur réunit un contexte et deux groupes.

Les objets communicateurs sont passés en argument des fonctions de communication point à point ou globales pour spécifier le contexte ainsi que le, ou les, groupes concernés par l'opération de communication.

### 2.10.4 Caches associés à un communicateur

La notion de « cache » permet à une application d'attacher des informations arbitraires, nommées *attributs*, à un communicateur. Ceci facilite le passage des informations entre les appels, permet de les retrouver rapidement et de garantir que les informations retrouvées sont à jour. On peut ainsi avoir la garantie que les informations périmées ne seront pas récupérées.

Les attributs sont locaux aux processus et spécifiques aux communicateurs auxquels ils sont attachés. Ils ne sont propagés d'un communicateur à un autre que par l'intermédiaire de la fonction `MPI_COMM_DUP`.

## 2.11 Les types de données dérivés

Les opérations de communications permettent évidemment de transmettre des buffers de données contiguës et de même type. Mais MPI fournit aussi des mécanismes pour spécifier des buffers de messages de type mixtes et non contigus en mémoire, ceci en autorisant l'utilisateur à définir des structures de données, qui consistent en un ensemble de types et d'adresses mémoires, et en utilisant les routines de construction fournies par MPI.

Un *type général de données* est un objet qui spécifie deux choses :

- une séquence de types de base
- une séquence de déplacements entiers.

Les déplacements (*stride*) n'ont pas à être positifs, distincts ou ordonnés, ce qui permet aux données de ne pas être obligatoirement contiguës en mémoire.

L'*étendue* d'un type de données est définie comme étant l'espace occupé depuis le premier octet jusqu'au dernier des éléments du type de données.

Exemple : supposons que *Type* = {*(double 0),(char 8)*} (c'est-à-dire un *double* au déplacement 0, suivi d'un *char* au déplacement 8). Nous supposons de plus que les doubles doivent être strictement alignés à des adresses qui sont multiples de 8. Alors, l'étendue de ce type de données est 16 (le plus petit multiple de 8 supérieur à $(8 + 1) = 9$). Les principales routines de manipulation et de construction de type de données sont explicitées ci-dessous.

### Constructeurs

- *Contigus :* le plus simple des constructeurs est `MPI_TYPE_CONTIGUOUS` qui permet la réplication d'un type de données en des endroits contigus en mémoire.

  `MPI_TYPE_CONTIGUOUS`*(count,oltype,newtype)*

  - **count** : nombre d'éléments à répliquer
  - **oldtype** : ancien type de données
  - **newtype** : nouveau type de données

  **newtype** est obtenu en concaténant **count** copies de **oldtype**.

  Exemple : soit **oldtype** le type de données suivant {*(double 0),(char 8)*}, avec une étendue égale à 16, et soit **count**=3. Le nouveau type de données retourné par un appel à `MPI_TYPE_CONTIGUOUS` est le suivant :

  {*(double 0),(char 8),(double 16),(char 24),(double 32),(char 40)*} (voir figure 2.8).

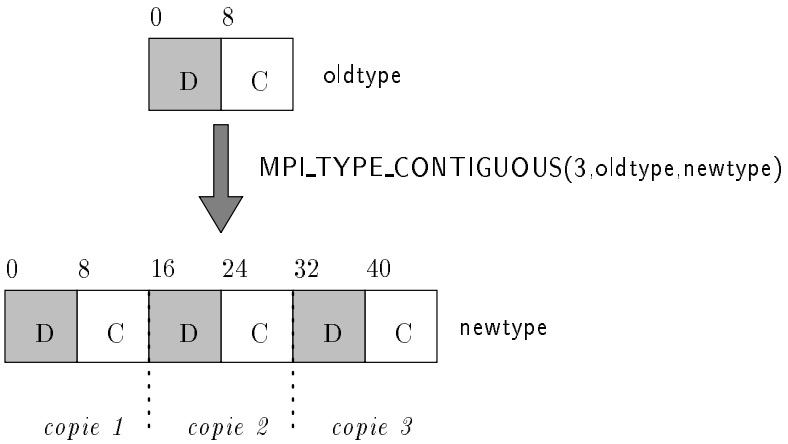



FIG. *2.8 - : Exemple de construction d'un type contigu.*

– *Vecteur*: la fonction `MPI_TYPE_VECTOR` est un constructeur plus général qui autorise la réplication d'un type de données en des endroits équitablement espacés. Chaque bloc est obtenu en concaténant le même nombre de copies de l'ancien type de données. L'espace entre chaque bloc est un multiple de l'étendue de l'ancien type de données.

`MPI_TYPE_VECTOR`*(count,blocklength,stride,oltype,newtype)*

  - **count** : nombre de blocs à répliquer
  - **blocklength** : nombre d'éléments dans chaque bloc
  - **stride** : nombre d'éléments entre le début de chaque bloc
  - **oldtype** : ancien type de données
  - **newtype** : nouveau type de données

Exemple : soit **oldtype** le type de données suivant {*(double 0),(char 8)*}, avec une étendue égale à 16. Le nouveau type de données retourné par un appel à `MPI_TYPE_VECTOR(2,3,4,oldtype,newtype)` est le suivant :

{*(double 0),(char 8),(double 16),(char 24),(double 32),(char 40),(double 64), (char 72), (double 80),(char 88),(double 96),(char 104)*} (voir figure 2.9).

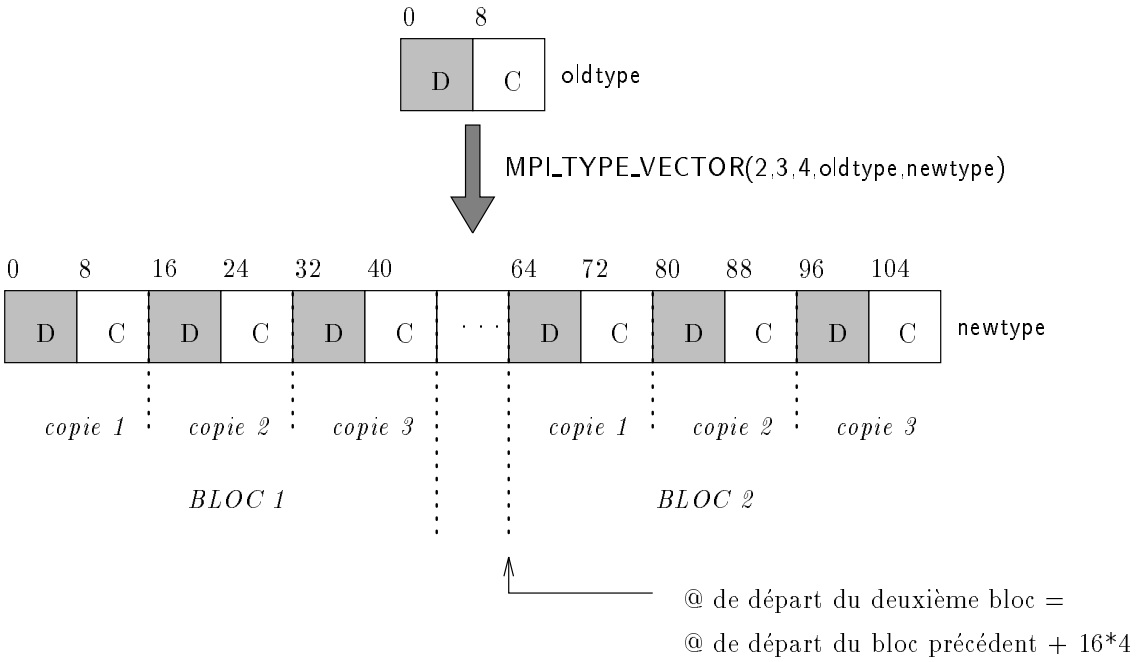



FIG. 2.9 - : *Exemple de l'utilisation du constructeur de vecteur.*

– *Structure* : c'est le constructeur le plus général.

`MPI_TYPE_STRUCT`

*(count,array_of_blocklength,array_of_displacements, array_of_type,newtype)*

- **count** : nombre de blocs, ainsi que le nombre d'éléments des tableaux **array_of_blocklength**, **array_of_displacements**, **array_of_type**
- **array_of_blocklength** : nombre d'éléments dans chaque bloc
- **array_of_displacements** : déplacement en octets dans chaque bloc
- **array_of_type** : type de données des éléments de chaque bloc
- **newtype** : nouveau type de données

Exemple : soit *type1* = {*(double 0),(char 8)*} avec une étendue de 16. Soit *D=(0,16,26)*, *B=(2,1,3)* et *T=(MPI_FLOAT,type1,MPI_CHAR)*.
Alors un appel à `MPI_TYPE_STRUCT(3,B,D,T,newtype)` renverra le type suivant :

{*(float 0),(float 4),(double 16),(char 24),(char 26),(char 27),(char 28)*}.

#### Fonctions de renseignements

MPI fournit des fonctions permettant d'avoir des renseignements sur les types de données construits, ou permettant d'en construire.
Ainsi :

- `MPI_ADDRESS` permet d'avoir l'adresse d'une donnée ;
- `MPI_EXTENT` renvoie l'étendue d'un type de données.

#### Fonctions de création et de libération

Lorsqu'un type est construit par une série de constructeurs, le descripteur de type résultat (un buffer) n'a pas nécessairement la structure optimale pour permettre son utilisation efficace. Avant de pouvoir se servir efficacement des nouveaux types, il faut « compiler » le descripteur de type pour lui donner une représentation plus compacte. Pour cela on utilise la fonction `MPI_COMMIT`.

D'un autre côté, la fonction `MPI_FREE` permet de libérer l'objet. Celui-ci ne sera effectivement libéré que lorsque toutes les communications en cours utilisant des buffers correspondant au type de données en question seront terminées.

## 2.12 Topologie virtuelle de processus

Dans la plupart des applications parallèles, une numérotation linéaire des processus ne correspond pas aux schémas de communication qui interviendront, car ceux-ci dépendent de la géométrie du problème traité et de l'algorithme utilisé. C'est pour cette raison que MPI offre la possibilité de construire et manipuler des topologies virtuelles de processus, celles-ci étant décrites, soit à l'aide d'un graphe de voisinage, soit, comme dans de nombreuses applications, en un pavage régulier (grilles multi-dimensionnelles).

Il faut bien différencier la topologie virtuelle des processus et la topologie physique des processeurs. La topologie virtuelle dépend de l'application, alors que la façon dont cette topologie va être plongée dans le réseau physique ne dépend pas de MPI, mais de l'implémentation qui en sera faite. Il est à noter que de bons plongements ne pourront qu'améliorer les performances des communications.

### 2.12.1 Les constructeurs

MPI fournit deux fonctions de construction de topologie virtuelle, suivant que celle-ci est un graphe quelconque ou possède une structure cartésienne.

### Graphe quelconque

`MPI_MAKE_GRAPH`*(comm_old,nnodes,index,edges,reorder,comm_graph)*

- **comm_old** : communicateur du groupe père
- **nnodes** : nombre de nœuds du graphe
- **index** : tableau indiquant le degré des nœuds
- **edges** : tableau décrivant les arêtes du graphe
- **reorder** : booléen indiquant s'il faut réordonner ou non les processus
- **comm_graph** : nouveau communicateur

Cette fonction retourne un nouveau communicateur auquel l'information de la topologie du graphe est rattachée. Si **reorder = false**, le rang de chaque processus dans le nouveau groupe reste inchangé par rapport à l'ancien groupe. En revanche, si **reorder = true**, la fonction réordonne les processus afin de réaliser un bon plongement de la topologie virtuelle dans le graphe physique des processeurs.

La spécification des entrées est la suivante :

- **index**
  - **index[0]** = degré du nœud 0
  - **index[i]-index[i-1]** = degré du nœud i, pour $i = 1 \ldots nnodes - 1$
- **edges**
  - La liste des voisins du nœud 0 sont stockés dans **edges[j]**, $0 \leq j \leq$**index[0]-1**
  - La liste des voisins du nœud i sont stockés dans **edges[j]**, **edge[i-1]** $\leq j \leq$**index[i]-1**

Exemple : considérons la matrice d'adjacence suivante :

| Processus | Voisins |
|---|---|
| 0 | 1,3 |
| 1 | 0 |
| 2 | 3 |
| 3 | 0,2 |

Les entrées de la fonction sont les suivantes : (voir figure 2.10)

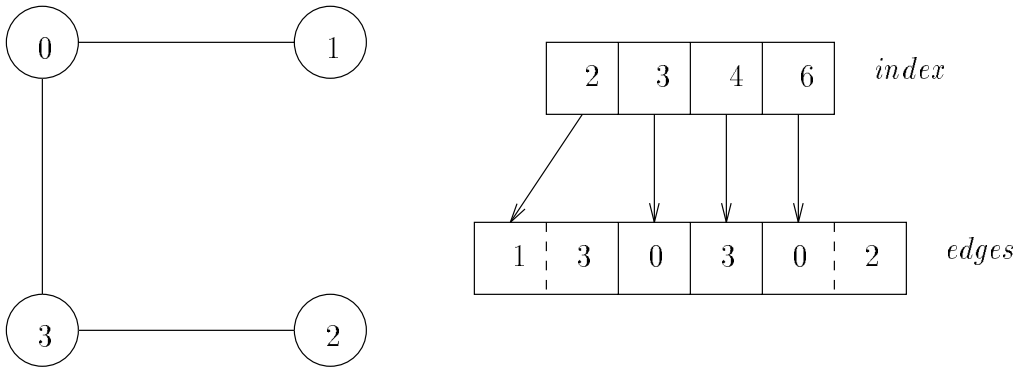



FIG. 2.10 - : *Construction d'une topologie de processus à partir du graphe de voisinage.*

### Graphe cartésien

`MPI_MAKE_CART`*(comm_old,ndims,dims,periods,reorder,comm_cart)*

- **comm_old** : communicateur du groupe père
- **ndims** : nombre de dimensions de la grille
- **dims** : tableau spécifiant le nombre de processus dans chaque dimension
- **periods** : tableau indiquant si la grille est périodique ou non suivant chaque dimension
- **reorder** : booléen indiquant s'il faut réordonner les processus ou non
- **comm_graph** : nouveau communicateur

Le booléen **reorder** a le même effet que précédemment.
Exemple :

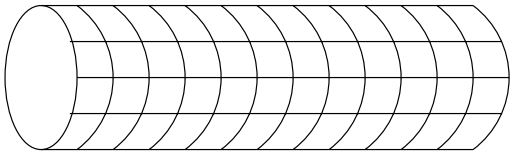


FIG. 2.11 - : *Topologie virtuelle cylindrique.*

Considérons la topologie en cylindre représentée sur la figure 2.11, et supposons que l'on souhaite avoir 12 processus le long des axes horizontaux, et 8 suivant la circonférence. La définition d'une telle topologie est donnée par la figure 2.12.

```
ndims = 2
dims(1) = 12
dims(2) = 8
periods(1) = .false.
periods(2) = .true.
reorder = .true.
call MPI_MAKE_CART(comm_old,ndims,dims,periods,reorder,comm_cart,ierror)
```

FIG. *2.12 - : Programme MPI définissant une topologie cylindrique de 12x8 processus*

### 2.12.2 Les manipulateurs

MPI offre toute une collection d'outils permettant la manipulation de ces topologies virtuelles, qu'il s'agisse de fonctions de renseignement sur la topologie virtuelle construite, ou de fonctions permettant de subdiviser en sous-graphes la topologie définie, ou encore de fonctions de communication de type « shift » ou « translation » (dans le cas de topologies cartésiennes).

## 2.13 Essais sur SP1

Nous avons utilisé la première implémentation de MPI sur le SP1 du CEA de Saclay. Cette version non définitive n'est pas très stable et surtout très pénalisante car elle ne peut cohabiter avec la bibliothèque EUI et PVM de l'ECSEC.

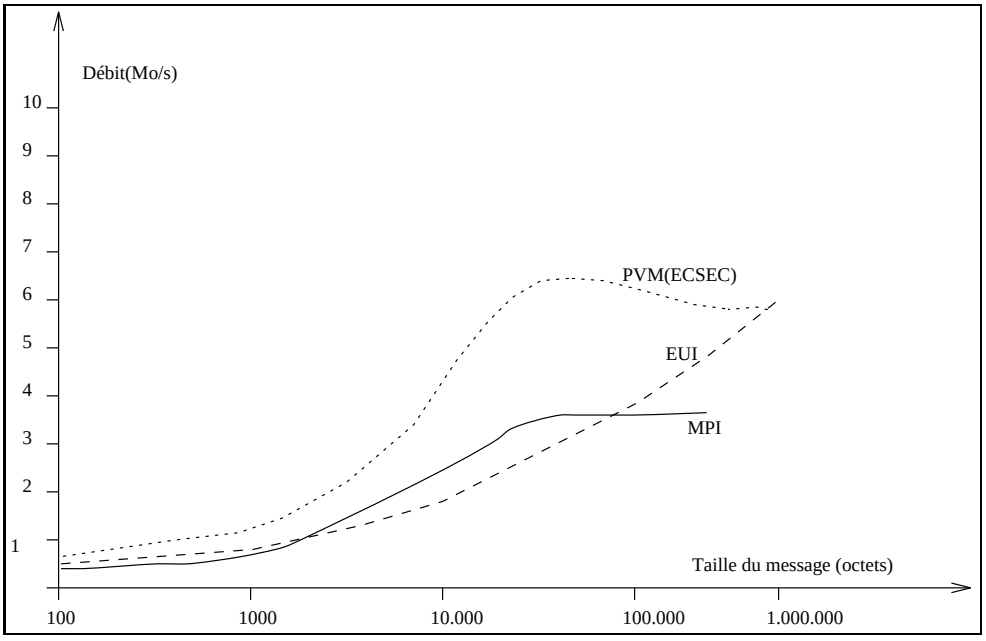



FIG. *2.13 - : Comparaison avec les bibliothèques de communication PVM et EUI.*

A très brève échéance, la version commercialisée devrait apparaître et fournir les mêmes performances que la bibliothèque EUI/H.

## 2.14 Evolutions et conclusions

### 2.14.1 Le futur : MPI-2

MPI version 1.0 n'en est encore qu'au stade de fin de spécification et autres premières implémentations, que les gens du « MPI Forum » pensent déjà à MPI version 2.0. En effet, lors des spécifications de MPI-1, les chercheurs se sont rendus compte qu'il manquerait déjà certaines spécificités :

- MPI-1 ne permet pas encore d'écrire des programmes totalement portables, en effet il manque :
  - la spécification des routines permettant d'effectuer des entrées / sorties en parallèle.
  - le support pour la gestion de tâches
- MPI-2 devrait comporter des opérations « plus sophistiquées » , qui ne sont à l'heure actuelles qu'à l'état de développement, comme :
  - des opérations globales non-bloquantes
  - des opérations nécessitant une utilisation plus poussée du système d'exploitation (messages actifs, exécution distante ...)
  - des outils de construction de programmes
  - des fonctionnalités et des outils de debug
  - le support explicite de «threads »
  - ...

et beaucoup d'autres choses encore.

Les spécifications de MPI-2 devraient commencer en été 94 et devraient suivre le même processus que MPI-1.

### 2.14.2 Conclusions pour MPI

MPI présente un certain nombre de notions qui apparaissent très importantes pour une portabilité et une implémentation efficace des applications parallèles. Citons par exemple les notions de groupe et de communicateur qui permettent de partitionner l'espace des communications, les opérations collectives qui peuvent être optimisées pour les différentes architectures des machines parallèles et enfin

la notion de type dérivé qui répond au souci de permettre une transmission facile et efficace des structures de données complexes de l'utilisateur.

MPI et ses différentes évolutions fournissent des primitives de communications de niveau assez élevé et beaucoup plus sophistiquées que dans les autres bibliothèques de « message passing » . De plus, les chercheurs et constructeurs du groupe MPI sont là pour conseiller et assurer des implémentations efficaces de cette bibliothèque sur les différentes machines parallèles.

## 2.15 Conclusion

Les deux bibliothèques présentées dans ce chapitre ne sont que deux implantations du modèle de programmation par processus communicants. Elles sont finalement assez proches l'une de l'autre dans la solution (ou l'absence de solution) qu'elles apportent aux problèmes qui se posent lorsque l'on veut faire coopérer et communiquer des processus distants. On peut considérer que MPI représente une « généralisation de PVM » , notamment dans les concepts de communications globales, de groupes de processus, de types structurés ou de topologie virtuelle.

Néanmoins, PVM continue à évoluer malgré l'apparition de MPI, et la dernière version PVM3.3 complète la version précédente en fournissant des opérations globales de communication et de calcul, ainsi que des communications point à point véritablement asynchrones [BDG+94, Man94].

# Chapitre 3

# Efficacité sur réseau de processeurs hétérogènes

*Nous présentons, dans ce chapitre, une extension des notions d'accélération et d'efficacité aux réseaux de processeurs hétérogènes [CD94]. Ce travail a été développé en collaboration avec Laurent Desbat, chercheur au laboratoire TIMC de Grenoble.*

## 3.1 Introduction

Le travail présenté dans ce chapitre a débuté par la constatation suivante : les programmes développés sur une architecture comportant des processeurs hétérogènes sont de plus en plus nombreux, mais très peu d'outils sont disponibles pour évaluer leurs performances.

En effet, des environnements de programmation comme CHARM [Kal92], Parform [CS92], P4 [BL92] et PVM [D+91] ont été développés ces trois dernières années. Ils permettent à l'utilisateur de considérer un réseau d'ordinateurs comme étant une machine parallèle virtuelle. Beaucoup plus faciles d'accès, ils sont souvent utilisés comme plate-forme de développement ou même pour effectuer les calculs définitifs lorsque le réseau est dédié à l'application. Les réseaux d'ordinateurs utilisés sont la plupart, du temps, des réseaux de stations de travail différentes. Entre ces ordinateurs, il n'y a pas que la cadence des cycles d'horloge qui change, mais l'architecture même du processeur. Par exemple, une multiplication suivie d'une addition est considérée comme étant une opération élémentaire sur des architectures de type « super-scalaire » alors que sur les architectures « scalaires » ce sont deux opérations bien distinctes. Les utilisateurs de ces architectures appelées hétérogènes ont besoin de connaître quelles performances ils peuvent espérer et s'ils utilisent bien les différentes ressources qui sont mises à leur disposition.

L'analyse des performances est très difficile sur les machines parallèles distribuées, notamment à cause de l'influence des mécanismes d'ordonnancement et

des perturbations induites par les communications. Mais il est nécessaire que les utilisateurs possèdent des moyens simples pour évaluer l'efficacité de l'implémentation parallèle de leur application. Par exemple, ils ont besoin de connaître l'accélération maximale qu'ils peuvent obtenir pour un code parallèle sur un nombre d'unités de calcul donné. Les notions d'accélération et d'efficacité ont été développées pour présenter d'une manière synthétique les performances d'un algorithme s'exécutant sur un réseau de processeurs, comparées à celles obtenues sur un processeur par le même programme. Ces deux notions fournissent des informations élémentaires suffisantes pour juger les gains obtenus sur une machine parallèle. Dans [BL93] une approche plus générale pour l'étude des performances est considérée.

Comme la plupart des machines parallèles existantes ont des architectures basées sur un réseau de processeurs homogènes, c'est-à-dire que tous les processeurs sont identiques, l'accélération et l'efficacité ont été particulièrement étudiées pour ces architectures [FP92, HJ88, LER92, Sto87]. Dans ce cas, pour calculer l'accélération *relative* (3.1), il suffit de comparer le temps d'exécution d'un programme au temps de sa version séquentielle sur l'un des processeurs. L'accélération *relative* est donc définie par :

$$S(P) \stackrel{def}{=} \frac{T(1)}{T(P)} \tag{3.1}$$

où $T(j)$ est le temps d'exécution sur $j$ processeurs et $P$ est le nombre de processeurs utilisés.

L'accélération dite *absolue* est le rapport du temps d'exécution du « meilleur » algorithme séquentiel sur celui de l'algorithme parallèle considéré sur $P$ processeurs. Mais cette dernière notion est difficile à mettre en œuvre dès que les algorithmes utilisés sont nouveaux (nouvelles méthodes numériques) et de taille importante. De plus, des particularités décrites dans [Fis91], comme le surcoût des boucles séquentielles de contrôle, les stockages intermédiaires dans la mémoire, ne sont pas pris en compte pour cette étude. Nous pensons que de tels problèmes ne jouent pas un rôle très important pour une approche macroscopique de la parallélisation, notamment quand on envisage l'étude d'applications de taille importante avec relativement peu de processeurs. Nous avons aussi implicitement supposé que toutes les opérations unitaires ont le même coût en mode séquentiel qu'en mode parallèle sinon il est possible d'obtenir des accélérations « super-linéaires » (voir [ABM+92]).

La définition de l'efficacité [CT93] est donnée par la formule ci-dessous :

$$E(P) \stackrel{def}{=} \frac{S(P)}{P} = \frac{T(1)}{P \times T(P)}. \tag{3.2}$$

Cette définition 3.2 de l'efficacité peut s'interpréter de la façon suivante : il peut être prouvé que $S(P) \leq P$ et que l'égalité est obtenue quand le programme parallèle est parfaitement bien équilibré au niveau de la quantité de calcul affectée

aux différents processeurs et lorsque le coût des communications est complètement masqué.

Lors de l'utilisation d'un réseau de processeurs hétérogènes, la notion d'accélération *relative* doit être généralisée. Donc, dans ce cas, que signifie le temps séquentiel $T_{\text{seq}} \stackrel{def}{=} T(1)$? Ces questions se posent déjà pour le cas d'un réseau homogène, les premières réponses sont fournies par Amdahl et Gustafson [Amd67, Gus88, CT93]. Pour répondre aux problèmes supplémentaires induits par l'utilisation de ressources hétérogènes de calcul, nous proposons une généralisation naturelle des notions d'accélération et d'efficacité. Par la suite, nous étudierons la borne supérieure de l'accélération avec l'introduction de la notion de puissance relative d'une machine. Nous montrons dans ce chapitre que l'efficacité $E(P) = \frac{S(P)}{P}$ peut se généraliser en $E(\varphi) = \frac{S(\varphi,p)}{\#(\boldsymbol{p}^*,p)}$ où $S(\varphi,p)$ est l'accélération calculée sur le réseau hétérogène $\varphi$ et où $\#(\boldsymbol{p}^*,p)$ est le nombre maximal de processeurs de puissance équivalente à la puissance $p$ pour l'algorithme considéré sur un processeur choisi comme référence. $\#(\boldsymbol{p}^*,p)$ se calcule facilement lorsque les processeurs ne diffèrent que par leur fréquence d'horloge. Pour un réseau de processeurs caractérisés par une hétérogénéité plus forte, $\#(\boldsymbol{p}^*,p)$ s'obtient par la résolution d'un simplexe.

Enfin, nous présentons des expérimentations mettant en œuvre les nouveaux outils de mesure de performances.

## 3.2 Accélération et efficacité dans le cas hétérogène

### 3.2.1 Modèle et définitions

Pour les réseaux homogènes, la notion d'accélération est naturellement relative à chacun des processeurs élémentaires. Quand on utilise un réseau d'unités de calcul hétérogènes, le choix d'un processeur particulier comme processeur de référence pour l'évaluation de l'accélération n'est pas évident. En effet, si l'utilisateur choisit le processeur le plus lent en moyenne pour toutes les opérations contenues dans son algorithme, il obtiendra une accélération plutôt optimiste et peut-être même une efficacité supérieure à 100%. Dans ce cas, pourquoi ne pas choisir comme référence un processeur d'une puissance moyenne qui ne ferait pas partie du réseau utilisé par l'application? Nous avons remarqué que cela ne suffit pas et qu'il est possible d'obtenir avec cette référence une efficacité supérieure à 100%. On peut alors se demander ce que signifie une efficacité de 120% par rapport une autre de 138% et quelle est la borne supérieure? On voit donc que les notions d'accélération et d'efficacité ne sont pas formalisées explicitement pour les machines parallèles hétérogènes comme elles le sont dans [BL93] pour

les machines homogènes.

**Définition 1** *On note $W$ le nombre d'opérations de base effectuées par un algorithme. On peut considérer que $W$ est une quantité de travail. On définit $T$ comme étant le temps d'exécution d'une quantité de travail $W$ sur PR (le Processeur de Référence). On peut écrire que :*

$$p \stackrel{def}{=} \frac{W}{T}$$

*est la définition d'une puissance.*

**Remarque 1** *La puissance ainsi définie dépend du travail exécuté sur le processeur. Ce n'est pas une constante qui caractérise le processeur.*

En général, $p$ est donné en Mflops, $W$ représente donc le nombre d'opérations flottantes qui doivent être faites par l'algorithme sur PR. Dans la suite de ce chapitre, nous noterons PR le processeur utilisé pour mesurer le temps de l'algorithme séquentiel, car la valeur de l'accélération est relative à la puissance du processeur sur lequel l'algorithme séquentiel s'exécute. Nous donnons donc la définition suivante pour l'accélération.

**Définition 2** *L'accélération d'un algorithme parallèle exécutant un travail $W$ est définie comme étant une fonction de la puissance du processeur de référence (PR) pour accomplir ce travail :*

$$S(\varphi, p) = \frac{T_{seq}(p)}{T(\varphi)} \text{ où } T_{seq}(p) = \frac{W}{p}$$

*et où $\varphi$ représente la liste des processeurs hétérogènes du réseau, c'est une caractérisation du réseau.*

**Remarque 2** *Si le réseau est constitué de processeurs homogènes alors la notion de processeur de référence n'est plus nécessaire et le nombre de processeurs est suffisant pour caractériser le réseau.*

Quand on utilise une machine parallèle, le travail $W$ est découpé afin d'être réparti sur les $P$ processeurs de la machine.

Nous définissons donc par $W_i$ la partie du travail $W$ exécutée sur le processeur i. Nous pouvons donc écrire que $W = \sum_{i=1}^{P} W_i$. Sur la partie de l'algorithme global qui lui a été affectée, le processeur $i$ a une puissance $p_i \stackrel{def}{=} \frac{W_i}{T_i}$ où $T_i$ est le temps d'exécution séquentielle de $W_i$ sur le processeur $i$. Alors $\sum_{i=1}^{P} p_i$ est la puissance totale du réseau hétérogène pour le découpage $W = \sum_{i=1}^{P} W_i$.

De plus, nous supposons que la quantité de travail $W$ est constituée de types de travaux différents et nous notons par $W^j$ la quantité de travail de type $j$ qui doit être exécutée. Par conséquence, $w_{i,j}$ représente la quantité de travail de type $j$ affectée au processeur $i$ et $c_{i,j} > 0$ le coût unitaire d'une opération de type $j$ sur le processeur $i$.

**Définition 3** *Nous définissons la puissance relative d'un réseau hétérogène pour un travail* $W = \sum_{i=1}^{P} W_i$ *par:*

$$\#(\mathbf{p}, p) = \#(p_1, \ldots, p_P, p) \stackrel{def}{=} \frac{\sum_{i=1}^{P} p_i}{p}.$$

$\#(p_1, \ldots, p_P, p)$ peut être considéré comme étant « le nombre de processeurs équivalent au Processeur de Référence » pour l'algorithme utilisé.

**Proposition 1** *L'accélération est majorée par* $\#(\mathbf{p}, p)$*:*

$$S(\varphi, p) \leq \#(\mathbf{p}, p)$$

*Preuve*:

A cause des coûts supplémentaires dus aux communications et aux éventuelles latences, on peut écrire que :

$$T(\varphi) \geq \max_{i=1}^{P} T_i \text{ où } T_i = \frac{W_i}{p_i}.$$

Donc

$$S(\varphi, p) \leq \frac{W}{p \max_{i=1}^{P} T_i} = \left( \max_{i=1}^{P} \frac{W_i}{W} \frac{p}{p_i} \right)^{-1}. \tag{3.3}$$

Soit $1 \leq i_{max} \leq P$ tel que

$$\frac{W_{i_{max}}}{p_{i_{max}}} = \max_{i=1}^{P} \left( \frac{W_i}{p_i} \right), \text{ alors, } \forall i = 1, \ldots, \frac{W_{i_{max}} \times p_i}{p_{i_{max}}} \geq W_i.$$

Donc

$$\#(\boldsymbol{p}, p) \max_{i=1}^{P} \left( \frac{W_i}{W} \frac{p}{p_i} \right) = \frac{1}{W} \sum_{i=1}^{P} p_i \max_{i=1}^{P} \left( \frac{W_i}{p_i} \right) = \frac{1}{W} \sum_{i=1}^{P} \frac{W_{i_{max}} \times p_i}{p_{i_{max}}} \geq \frac{\sum_{i=1}^{P} W_i}{W} = 1.$$

L'inéquation précédente est utilisée dans (3.3) pour démontrer le résultat final. □

Supposons que les différents processeurs sont à « coût identique pour toutes leurs instructions » , c'est-à-dire que $\forall w \in [1; W]$, $t_i$ le temps d'exécution d'un

travail $w$ sur le processeur $i$ vérifie $w = p_i t_i$ où $p_i$ est *constant*. Autrement dit, le temps d'exécution d'un travail $w$ sur un processeur $i$ est une fonction linéaire par rapport à la quantité de travail. Alors $\#(\boldsymbol{p}, p)$ ne dépend pas du placement du travail $W$ sur les P processeurs, mais, globalement, le réseau peut être hétérogène ($p_i \neq p_j$). Nous proposons la définition suivante pour l'efficacité d'un algorithme sur un réseau hétérogène de processeurs où chacun d'eux a le même coût pour toutes ses opérations internes :

$$E(\varphi) \stackrel{def}{=} \frac{S(\varphi, p)}{\#(\boldsymbol{p}, p)}. \tag{3.4}$$

La borne $\#(\boldsymbol{p}, p)$ est atteinte sous les conditions suivantes :

- premièrement, $T(\varphi) = \max_{i=1}^{P} T_i$, ce qui signifie qu'il n'y a pas de latence et que les communications sont totalement masquées par le calcul ;
- et, $\forall k = 1, \ldots, P$, $t_k = \max_{i=1}^{P} T_i$, ce qui implique que l'algorithme a sa charge de calcul parfaitement répartie (équilibrée).

Dans ce cas, l'efficacité (3.4) est égale à 1.

Grâce à la proposition 1, l'efficacité définie par (3.4) vérifie une propriété classique, c'est-à-dire que $0 \leq E(\varphi) \leq 1$. Ces définitions de l'accélération et de l'efficacité sont des généralisations de celles très connues données pour les réseaux de processeurs homogènes ($\#(\boldsymbol{p}, p) = P$).

**Remarque 3** *$\#(\mathbf{p}, p)$ est égal à $P$ quand la puissance de PR est égale à la puissance moyenne du réseau :*

$$p = \frac{\sum_{i=1}^{P} p_i}{P}.$$

La borne de la proposition 1 et l'efficacité (3.4) sont inutilisables quand les processeurs n'ont pas leurs opérations internes à coût identique, car la puissance $p_i$ dépend alors des différent types de travaux affectés au processeur $i$ et pas seulement de la quantité totale de travail.

### 3.2.2 Illustration grâce à un exemple simple

Nous allons considérer un réseau constitué de deux processeurs $\mathrm{Proc}_1$ et $\mathrm{Proc}_2$. Ceux-ci accomplissent seulement deux types d'opérations, l'addition et la multiplication, avec des coûts d'exécutions très différents. Nous allons supposer qu'une addition coûte une unité de temps $\tau$ sur le processeur $\mathrm{Proc}_1$, respectivement $2\tau$

sur $\text{Proc}_2$, et qu'une multiplication coûte $2\tau$ sur $\text{Proc}_1$, respectivement $\tau$ sur $\text{Proc}_2$. Ces informations sont résumées dans le tableau ci-dessous :

| *coûts* | add | mult |
|---|---|---|
| $\text{Proc}_1$ | $c_{1a} = 1$ | $c_{1m} = 2$ |
| $\text{Proc}_2$ | $c_{2a} = 2$ | $c_{2m} = 1$ |

TAB. *3.1 - : Coût des deux types d'opérations sur les différentes machines.*

Le programme parallèle test essayé sur le réseau formé des deux processeurs $\text{Proc}_1$ et $\text{Proc}_2$ est composé de $W_a = 9 \times 10^6$ additions et de $W_m = 9 \times 10^6$ multiplications, *i.e.* $W = 18 \times 10^6$ opérations flottantes. Nous notons $w_{1a}$ le nombre d'opérations d'additions exécutées par $\text{Proc}_1$, respectivement $w_{2a}$ par $\text{Proc}_2$ et $w_{1m}$ le nombre d'opérations de multiplications exécutées par $\text{Proc}_1$, respectivement $w_{2m}$ par $\text{Proc}_2$. Le temps d'exécution parallèle est minimum pour la répartition du travail suivante (si toutes les opérations sont indépendantes) :

| $\boldsymbol{w}$ | add | mult |
|---|---|---|
| $\text{Proc}_1$ | $w_{1a} = 9 \times 10^6$ | $w_{1m} = 0$ |
| $\text{Proc}_2$ | $w_{2a} = 0$ | $w_{2m} = 9 \times 10^6$ |

TAB. *3.2 - : Placement pour le meilleur temps d'exécution.*

Les éventuels coûts de communication ne sont pas pris en compte pour trouver la borne inférieure du temps d'exécution parallèle. Si on note $p_1^*$ et $p_2^*$ les puissances des deux processeurs pour le placement optimal, alors la répartition du travail global, donnée par le tableau précédent, produit un temps d'exécution de $9 \times 10^6$ et une puissance totale pour le réseau hétérogène de $p_1^* + p_2^* = 2$ opérations par unité de temps.

On peut remarquer que d'autres répartitions équilibrées de la quantité de travail comme :

| $\boldsymbol{w}$ | add | mult |
|---|---|---|
| $\text{Proc}_1$ | $w_{1a} = 6 \times 10^6$ | $w_{1m} = 3 \times 10^6$ |
| $\text{Proc}_2$ | $w_{2a} = 3 \times 10^6$ | $w_{2m} = 6 \times 10^6$ |

TAB. *3.3 - : Répartition équilibrée*

ou

| $\boldsymbol{w}$ | add | mult |
|---|---|---|
| $\text{Proc}_1$ | $w_{1a} = 0$ | $w_{1m} = 9 \times 10^6$ |
| $\text{Proc}_2$ | $w_{2a} = 9 \times 10^6$ | $w_{2m} = 0$ |

TAB. *3.4 - : Répartition extrême*

donnent de moins bonnes performances (respectivement $p_1 + p_2 = 3/2$ et 1).

Il faut donc trouver une autre borne qui peut se calculer quel que soit le découpage du travail et son placement sur les différents processeurs. Cette borne sera la valeur maximale théorique que pourra atteindre l'accélération. L'efficacité sera alors inférieure ou égale à 1, et le pourcentage obtenu donnera le taux d'utilisation des ressources de calcul du réseau. Donc, si $E(\varphi) = 0.8$, nous en déduirons que l'utilisateur n'utilise que 80% des possibilités de traitement du réseau hétérogène.

## 3.3 La borne supérieure de l'accélération

Supposons que la quantité de travail $W$ puisse être décomposée en $J$ types de travaux différents de coûts différents, *i.e.* $W = \sum_{j=1}^{J} W^j$.

Comme $c_{i,j} > 0$ est le coût unitaire d'une opération de type $j$ sur un processeur $i$, alors le temps d'exécution de $w_{i,j}$ opérations de type $j$ sur le processeur $i$ est égal à $c_{i,j}w_{i,j}$. Le vecteur $\boldsymbol{w} \in IN^{JP}$ représente le placement du travail $W$ sur les $P$ processeurs.

Pour pouvoir trouver une borne à l'accélération, il faut minimiser le temps d'exécution parallèle du travail $W$ sur les $P$ processeurs. Ce dernier est la solution de la formule suivante :

$$\min_{\boldsymbol{w} \in IN^{JP}} \max_{i=1}^{P} (\sum_{j=1}^{J} c_{i,j} w_{i,j}). \tag{3.5}$$

Nous considérons des programmes où le nombre d'opérations $W$ est beaucoup plus grand que la taille des vecteurs $\boldsymbol{w}$ qui est de $PJ$. En effet, pour la plupart des programmes développés sur réseau de processeurs, il est très fréquent d'avoir $W > 10^6 PJ$. Nous allons donc pouvoir résoudre (3.5) avec $\boldsymbol{w} \in IR_+^{JP}$. Comme pour l'efficacité, qui est majorée par 1, notre but est de donner une borne supérieure pour l'accélération.

**Remarque 4** *Si il n'y a pas de latence, si toutes les communications sont masquées par du calcul et si la totalité du travail $W$ peut être effectuée en parallèle, alors $T(\varphi)$ est le temps parallèle minimal implique que :*

$$\forall i = 1, \ldots P;\ T_i = T(\varphi). \tag{3.6}$$

Une minoration, notée $T_{//}$, du meilleur temps d'exécution nécessaire pour accomplir le travail $W$ sur $P$ processeurs est obtenue par la résolution du simplexe 3.7. $T_{//}$ est une minoration de $T(\varphi)$ qui va nous permettre de calculer la

borne supérieure de l'accélération pour un réseau hétérogène.

$$T_{//} \stackrel{def}{=} \min_{w_{i,j}} \sum_{j=1}^{J} c_{P,j} w_{P,j},$$
$$\begin{cases} w_{i,j} \geq 0, \\ \sum_{j=1}^{J} c_{i,j} w_{i,j} = \sum_{j=1}^{J} c_{P,j} w_{P,j} \ \forall i = 1, \ldots, P-1, \\ \sum_{i=1}^{P} w_{i,j} = W^j \ \forall j = 1, \ldots, J. \end{cases} \tag{3.7}$$

Quelques précisions peuvent être apportées en ce qui concerne la résolution du simplexe (3.7). Premièrement, on peut remarquer que ce simplexe est toujours réalisé par le vecteur suivant :

$$w^0_{i,j} = \frac{\prod_{k=1 k \neq i}^{P} c_{k,j}}{\sum_{l=1}^{P} \prod_{k=1 k \neq l}^{P} c_{k,j}} W^j. \tag{3.8}$$

**Proposition 2** *Le simplexe (3.7) a toujours une base réalisable de plein rang* $P + J - 1$.

*Preuve*: Soit $\mathbf{c}^i \in IR^J$ le vecteur coût du processeur $i$:

$\mathbf{c}^i{}_j = c_{i,j}, \forall j = 1, \ldots, J$.

Soit $\mathbf{C}^i \in IR^{P \times J}, i = 1, \ldots, P$ les $P$ vecteurs ayant comme éléments :

$(\mathbf{C}^i)_{k,j} = \delta_{i,k} c_{k,j}$ ($\mathbf{C}^1 = (\mathbf{c}^1, 0 \ldots, 0)$, $\mathbf{C}^i = (0, \ldots, 0, \mathbf{c}^i, 0, \ldots, 0)$, $\mathbf{C}^P = (0 \ldots, 0, \mathbf{c}^P)$).

Soit $\mathbf{e}^j \in IR^J$ les $J$ vecteurs de la base canonique ($\mathbf{e}^j{}_i = \delta_{ij}$) et $\mathbf{E}^j \in IR^{P \times J}, j = 1, \ldots, J$ les $J$ vecteurs ayant comme composantes :

$(\mathbf{E}^j)_{i,l} = \delta_{jl}$ ($\mathbf{E}^j = (\mathbf{e}^j, \ldots, \mathbf{e}^j)$).

Nous pouvons maintenant réécrire le problème linéaire (3.7) de la façon suivante :

$$T_{//} \stackrel{def}{=} \min_{w_{i,j}} \langle \mathbf{C}^P, \boldsymbol{w} \rangle_{IR^{PJ}},$$
$$\begin{cases} w_{i,j} \geq 0, \\ A\boldsymbol{w} = \boldsymbol{b}. \end{cases}$$

où la matrice $A$ de taille $(P + J - 1) \times PJ$ représente les contraintes du simplexe (3.7). Elle est composée par les vecteurs lignes $\mathbf{C}^i - \mathbf{C}^P, i = 1, \ldots, P-1$ et

$\mathbf{E}^j, j = 1, \ldots, J$:

$$A = \begin{bmatrix} \mathbf{C}^1 - \mathbf{C}^P \\ \mathbf{C}^2 - \mathbf{C}^P \\ \vdots \\ \mathbf{C}^{P-1} - \mathbf{C}^P \\ \mathbf{E}^1 \\ \mathbf{E}^2 \\ \vdots \\ \mathbf{E}^J \end{bmatrix} = \begin{bmatrix} \mathbf{c}^1 & 0 & \ldots & 0 & -\mathbf{c}^P \\ 0 & \mathbf{c}^2 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \ldots & 0 & \mathbf{c}^{P-1} & -\mathbf{c}^P \\ \mathbf{e}^1 & \mathbf{e}^1 & \ldots & \mathbf{e}^1 & \mathbf{e}^1 \\ \mathbf{e}^2 & \mathbf{e}^2 & \ldots & \mathbf{e}^2 & \mathbf{e}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{e}^J & \mathbf{e}^J & \ldots & \mathbf{e}^J & \mathbf{e}^J \end{bmatrix} \text{ et } \boldsymbol{b} = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ W^1 \\ W^2 \\ \vdots \\ W^J \end{bmatrix}.$$

Si $PJ > 0$ alors $PJ - (P + J - 1) = (P - 1)(J - 1) \geq 0$. Par conséquent, le simplexe (3.7) a toujours une base réalisable de plein rang si les vecteurs lignes $\mathbf{C}^i - \mathbf{C}^P, i = 1, \ldots, P-1$ et $\mathbf{E}^j, j = 1, \ldots, J$ , de la matrice $A$, sont linéairement indépendants (*i.e.* Rang$(A) = (P + J - 1) \iff$ A est surjective).

Soit $\lambda_i$ et $\mu_j$, $(P + J - 1)$ nombres réels tels que:

$$\begin{aligned} & \sum_{i=1}^{P-1} \lambda_i \left( \mathbf{C}^i - \mathbf{C}^P \right) + \sum_{j=1}^{J} \mu_j \mathbf{E}^j = 0 \\ \iff & \begin{cases} \forall (k,l) \in \{1, \ldots, P-1\} \times \{1, \ldots, J\}, \lambda_k c_{k,l} + \mu_l = 0, \\ \forall l \in \{1, \ldots, J\}, -\sum_{i=1}^{P-1} \lambda_i c_{P,l} + \mu_l = 0. \end{cases} \end{aligned} \tag{3.9}$$

Avec la première ligne de (3.9) (et $c_{i,j} > 0$), nous pouvons conclure que si les vecteurs sont linéairement dépendants alors:

$$\forall (k,l) \in \{1, \ldots, P-1\} \times \{1, \ldots, J\}, c_{k,l} = -\frac{\mu_l}{\lambda_k} \text{ et donc } \mu_l \lambda_k < 0.$$

Et grâce à la seconde ligne de (3.9), nous pouvons écrire que:

$$\forall l \in \{1, \ldots, J\}, c_{P,l} = \frac{\mu_l}{\sum_{i=1}^{P-1} \lambda_i}.$$

Or la dernière équation est impossible car $c_{P,l} > 0, \forall l \in \{1, \ldots, J\}$. Par conséquent, Rang$(A) = (P + J - 1)$.

□

Un cas important de notre étude consiste à trouver la ou les configurations du réseau de processeurs pour lesquelles tous les vecteurs réalisables donnés par (3.7) sont optimaux, c'est-à-dire lorsque le simplexe est totalement dégénéré.

**Proposition 3** *Le simplexe (3.7) est totalement dégénéré si et seulement si les vecteurs* $\boldsymbol{c}^i$, $\forall i = 1, \ldots, P$ *sont deux à deux linéairement dépendants.*

*Preuve*: Nous allons considérer $\boldsymbol{w}^0$ le vecteur réalisable (3.8) de (3.7). Tous les points (vecteurs) réalisables $\boldsymbol{w}$ sont dans l'espace $\boldsymbol{w}^0 + \mathrm{Ker}(A) \cap \{\boldsymbol{w}, w_{i,j} \geq 0\}$. Comme tous les éléments de $\boldsymbol{w}^0$ sont strictement positifs, $\forall \boldsymbol{w} \in \mathrm{Ker}(A)(= \mathrm{Im}(A^*)^\perp), \exists \epsilon > 0$, $(\boldsymbol{w}^0 + \epsilon \boldsymbol{w}) \in \boldsymbol{w}^0 + \mathrm{Ker}(A) \cap \{\boldsymbol{w}, w_{i,j} \geq 0\} (= \mathrm{Im}(A^*)^\perp \cap \{\boldsymbol{w}, w_{i,j} \geq 0\})$, *i.e.* $\forall \boldsymbol{w} \in \mathrm{Im}(A^*)^\perp, \exists \epsilon > 0$, $\boldsymbol{w}^0 + \epsilon \boldsymbol{w}$ est réalisable.

Si nous supposons que le simplexe (3.7) est totalement dégénéré alors :

$$\forall \boldsymbol{w} \in \mathrm{Im}(A^*)^\perp, \exists \epsilon > 0, \langle \mathbf{C}^P, \boldsymbol{w}^0 + \epsilon \boldsymbol{w} \rangle_{IR^{PJ}} = \langle \mathbf{C}^P, \boldsymbol{w}^0 \rangle_{IR^{PJ}} \Rightarrow \langle \mathbf{C}^P, \boldsymbol{w} \rangle_{IR^{PJ}} = 0.$$

Donc $\mathbf{C}^P \in \mathrm{Im}(A^*)$, et nous pouvons conclure que les vecteurs $\mathbf{C}^1, \mathbf{C}^2, \ldots, \mathbf{C}^P$, $\mathbf{E}^1, \mathbf{E}^2, \ldots, \mathbf{E}^J$ sont linéairement dépendants. En utilisant le même argument que celui utilisé dans la démonstration de la proposition 2, nous pouvons déduire que :

$$\exists \lambda_i,\ i = 1, \ldots, P,\ \exists \mu_j,\ j = 1, \ldots, J,\ \left( \frac{\mu_j}{\lambda_i} < 0 \right),\ c_{i,j} = -\frac{\mu_j}{\lambda_i}.$$

Donc $\forall i = 1, \ldots, P$, $\mathbf{c}^i = -1/\lambda_i \, (\mu_1, \ldots, \mu_J)$.

La réciproque est maintenant simple :

Si les vecteurs coût sont deux à deux linéairement dépendants, alors $\exists \lambda_i > 0$, $i = 1, \ldots, P$, $\exists \mu_j > 0$, $j = 1, \ldots, J$, $c_{i,j} = \frac{\mu_j}{\lambda_i}$. Ceci implique que les vecteurs $\mathbf{C}^1, \mathbf{C}^2, \ldots, \mathbf{C}^P, \mathbf{E}^1, \mathbf{E}^2, \ldots, \mathbf{E}^J$, ainsi que les vecteurs $\mathbf{C}^1 - \mathbf{C}^P, \mathbf{C}^2 - \mathbf{C}^P, \ldots, \mathbf{C}^P, \mathbf{E}^1, \mathbf{E}^2, \ldots, \mathbf{E}^J$ sont linéairement dépendants. Or les vecteurs $\mathbf{C}^1 - \mathbf{C}^P, \mathbf{C}^2 - \mathbf{C}^P, \ldots, \mathbf{C}^{P-1} - \mathbf{C}^P$, $\mathbf{E}^1, \mathbf{E}^2, \ldots, \mathbf{E}^J$ sont linéairement indépendants. Nous pouvons donc conclure que $\mathbf{C}^P \in \mathrm{Im}(A^*)$, ce qui implique que le simplexe (3.7) est totalement dégénéré. □

Nous pouvons maintenant donner la définition et la proposition suivantes :

**Définition 4** *Un réseau de processeurs sera dit faiblement hétérogène pour le travail $W = \sum_{j=1}^{J} W^j$ si les $P$ vecteurs coût $\boldsymbol{c}^1, \boldsymbol{c}^2, \ldots, \boldsymbol{c}^P$ des différents processeurs du réseau sont deux à deux linéairement dépendants. S'ils sont tous égaux, alors le réseau est homogène.*

On peut remarquer que lorsque un réseau est faiblement hétérogène, alors $\forall \boldsymbol{w} \in IR^{PJ}$ vecteur réalisable de (3.7), $\#(\boldsymbol{p}, p) = \frac{\sum_{i=1}^{P} W_i}{{}^pT_{//}} = \frac{W}{{}^pT_{//}}$ est constant.

**Proposition 4** *La définition suivante de l'efficacité est une extension de la définition couramment utilisée pour le cas des réseaux homogènes ($0 < E(P) \leq 1$, $E=1$ implique qu'il n'y a pas de latence, que le travail est réparti de manière optimale et qu'il y a un recouvrement total des communications par le calcul).*

- *Le problème (3.7) est solvable simplement par l'utilisation d'un algorithme de type simplexe. Soit $w^*_{i,j}$ une solution optimale, on pose $W^*_i = \sum_{j=1}^{J} w^*_{i,j}$*

*et* $p_i^* = W_i^*/T_{//}$*, on peut alors écrire que :*

$$S(\varphi, p) \leq \frac{Tseq(p)}{T_{//}} = \#(p_1^*, \ldots, p_P^*, p) = \#(\mathbf{p}^*, p).$$

*Dans ce cas, l'efficacité est définie par :*

$$E(\varphi) = \frac{S(\varphi, p)}{\#(p_1^*, \ldots, p_P^*, p)}. \tag{3.10}$$

– *Dans le cas où le réseau est faiblement hétérogène, nous supposons pour simplifier que* $c_{i,j} = c_j/\lambda_i$ *où* $\mathbf{c} = (c_1, c_2, \ldots, c_J)$ *est le vecteur coût du processeur de référence pour les* $J$ *types de travaux différents. On a :*

$$E(\varphi) = \frac{S(\varphi, p)}{\sum_{i=1}^{P} \lambda_i}.$$

Le cas des réseaux homogènes correspond au fait que $\lambda_i = 1, \forall i = 1, \ldots, P$. La borne très connue, $P$, est donc un point particulier du théorème traitant le cas des réseaux faiblement hétérogènes.

## 3.4 Un autre outil : le « Size up »

Nous allons définir $W_{seq}$ et $W_{//}$ comme étant deux quantités de travail exécutées en séquentiel (respectivement en parallèle) durant le même temps $T$. Nous avons donc

$$W_{seq} = \sum_{j=1}^{J} W^j(W_{seq}),$$

car c'est $W_{seq}$ qui détermine la quantité de travail $W^j$. De même

$$W_{//} = \sum_{j=1}^{J} \sum_{i=1}^{P} w_{i,j}(W_{//}) = \sum_{j=1}^{J} W^j(W_{//}).$$

Le « Size up » [SG91] est défini comme suit :

$$Sizeup = \frac{W_{//}}{W_{seq}}.$$

Nous pouvons caractériser la durée de l'exécution $T$ par :

$$T = \sum_{j=1}^{J} c_j W^j = \max_{i=1}^{P} \left( \sum_{j=1}^{J} c_{i,j} w_{i,j} + T_{com(i)} \right),$$

où $T_{com(i)}$ est le temps durant lequel le processeur $i$ ne calcule pas.

Notre but est de trouver une borne supérieure pour le « Size up » , c'est pourquoi nous allons considérer que les quantités de travail $w_{i,j}$ sont réelles (et non entières). Si tous les processeurs calculent durant toute la durée $T$ et si toutes les communications sont masquées par du calcul, alors il est évident que la quantité de travail calculée sera plus importante. Donc la solution $W^*_{//}$ du problème d'optimisation 3.11 sera une borne supérieure optimiste pour la quantité de travail parallèle (maximale) qui peut être effectuée durant le temps $T$.

$$\begin{cases} \max_{w_{i,j}} \sum_{i=1}^{P} \sum_{j=1}^{J} w_{i,j}, \\ w_{i,j} \geq 0, \\ \sum_{j=1}^{J} c_{i,j} w_{i,j} = T, \ \forall i = 1, \ldots, P, \\ \sum_{i=1}^{P} w_{i,j} = W^j(W_{//}), \ \forall j = 1, \ldots, J. \end{cases} \tag{3.11}$$

Bien évidemment, cette borne ne peut être atteinte que si les communications sont totalement masquées et si le travail a été parfaitement réparti sur les processeurs. La détermination de $W^*_{//}$ n'est pas évidente, même pour le cas des réseaux homogènes, ceci est dû aux dépendances de $W^j$ avec la quantité de travail globale. En effet, ces dépendances seront souvent non linéaires.

Si on peut faire l'hypothèse que $W^j$ dépend linéairement de $W_{//}$, ou plus précisément que $\exists 0 < \alpha_j < 1, j = 1, ..., J, \ et \ \sum_{j=1}^{J} \alpha_j = 1,$ tel que $W^j = \alpha_j W_{//}$, alors le problème (3.11) est un simplexe et il a une solution réalisable que nous allons calculer.

Si on considère $w_{i,j} = \alpha_j W_i$, comme on sait que $\sum_{j=1}^{J} c_{i,j} w_{i,j} = T$, alors on peut écrire que :

$$W_i = \frac{T}{\sum_{j=1}^{J} c_{i,j} \alpha_j} \text{ et donc que } w_{i,j} = \frac{\alpha_j T}{\sum_{j=1}^{J} c_{i,j} \alpha_j}.$$

Dans ce cas particulier, on peut donc calculer assez facilement la borne supérieure $W^*_{//}$.

Cette notion peut aussi être utilisée pour un réseau de processeurs hétérogènes en choisissant un processeur de référence. Mais le calcul d'une « bonne » borne supérieure pour le « Size up » n'est pas très simple. En effet la relation entre la taille des différents types de travaux et la quantité globale de travail n'est que très rarement linéaire. Le « Size up » est une bonne mesure de comparaison de performances. Il est souvent facile de mesurer des « Size up » et de les comparer, mais il est moins simple de dire d'un « Size up » qu'il est bon car on ne sait pas facilement le borner.

## 3.5 Expérimentations utilisant PVM

### 3.5.1 Modélisation d'étoiles en utilisant des techniques de type Monte Carlo

La première application choisie pour illustrer les notions introduites précédemment est un programme de type Monte Carlo de modélisation d'étoiles. Il a été implémenté en PVM sur un petit réseau de stations de travail dédié (*cf.* [BM88, BM90] pour la partie astrophysique théorique et le chapitre 6.4 pour la partie algorithmique). Le très haut degré de parallélisme contenu dans de telles applications permet d'obtenir d'excellentes performances.

Dans un premier temps, nous avons calculé les trajectoires de deux millions de photons sur des stations de travail IBM RS6000 (modèle 320). Dans le tableau suivant, nous présentons l'accélération classique $S(P)$, $P$ étant le nombre de stations du réseau.

| P | Time | S(P) |
|---|---|---|
| 1 | 5999 | |
| 2 | 3125 | 1.92 |
| 3 | 2101 | 2.86 |
| 4 | 1509 | 3.97 |
| 5 | 1213 | 4.94 |

TAB. *3.5 - : Evolution du facteur d'accélération.*

Ensuite, nous avons effectué les calculs pour cinq millions de photons sur trois IBM RS6000 modèle 320, un modèle 550 et un modèle 560. La station de travail modèle 320 est choisie comme processeur de référence (PR) pour le calcul des de performances sur le réseau hétérogène. On peut considérer que ce réseau est faiblement hétérogène, car les vecteurs coût sont deux à deux linéairement liés. Ceci se justifie par le fait que les processeurs RISC utilisés sont les mêmes. La différence de puissance entre les machines est due essentiellement à des fréquences d'horloge interne différentes.

$\#(\boldsymbol{p}^*, p)$ représente le nombre de processeurs équivalents au processeur modèle 320 dans notre réseau. Nous avons déterminé le temps de calcul séquentiel pour traiter un million de photons (ce temps d'exécution séquentielle croît linéairement

avec le nombre de photons).

| modèle | $T_{\text{seq}}(modle)$ | $T_{\text{seq}}(320)/T_{\text{seq}}(modle)$ |
|---|---|---|
| 320 | 3023 s | 1 |
| 550 | 1450 s | 2.1 |
| 560 | 1206 s | 2.5 |

TAB. *3.6 - : Calcul des $\lambda_i$.*

Comme nous avons supposé que $\mathbf{c}^{560}/\lambda_{560} = \mathbf{c}^{550}/\lambda_{550} = \mathbf{c}^{320}$, on a

$$\lambda_{560} = T_{\text{seq}}(320)/T_{\text{seq}}(560) \sim 2.5 \text{ et } \lambda_{550} = T_{\text{seq}}(320)/T_{\text{seq}}(550) \sim 2.1.$$

Nous avons alors calculé $\#(\boldsymbol{p}^*, p)$. Dans le tableau suivant, nous avons par exemple pour $P = 4$, trois modèle 320 et un modèle 560, $\#(\boldsymbol{p}^*, p) = 3 + 2.5 = 5.5$. Nous verrons par la suite que $S(\varphi, p) \leq \#(\boldsymbol{p}^*, p)\#(\boldsymbol{p}^*, p)$.

| P | Time | $S(\varphi, p)$ | $\#(\boldsymbol{p}^*, p)$ |
|---|---|---|---|
| 1: un 320 | 14995 | 1.00 | 1 |
| 2: deux 320 | 7648 | 1.96 | 2 |
| 3: trois 320 | 5111 | 2.93 | 3 |
| 4: trois 320 et un 560 | 2775 | 5.40 | 5.5 |
| 5: trois 320, un 560 et un 550 | 2100 | 7.14 | 7.6 |

TAB. *3.7 - : Accélération sur le réseau hétérogène.*

Les expériences réalisées pour des nombres de photons allant jusqu'à 80 millions, effectuées sur un réseau dont la performance de crête est de 1 Gflops (20 IBM RS6000 : 15 modèles 320, un modèle 520, un modèle 530, deux modèles 550 et un modèle 560) sont récapitulées dans le tableau suivant :

| Exp. | $N$ | $\#(\boldsymbol{p}^*, p)$ | $T_{par}$ | $T_{seq}$ | $S(\varphi, p)$ | $E(\varphi)$ |
|---|---|---|---|---|---|---|
| 1 | $5\ 10^7$ | 23.75 | 8.5 | 144 | 16.9 | .71 |
| 2 | $8\ 10^7$ | 23.75 | 12.7 | 235 | 18.5 | .78 |
| 3 | $8\ 10^7$ | 23.75 | 11.9 | 228 | 19.2 | .81 |
| 4 | $8\ 10^7$ | 23.75 | 12.3 | 228 | 18.5 | .78 |

TAB. *3.8 - : Résultats des expérimentations sur le réseau hétérogène.*

Ici $T_{seq}$ représente le temps d'exécution séquentielle (en $10^3$ secondes) sur un processeur de référence ( ici PR est un RS6000-320), $S(\varphi, p)$ représente l'accélération et $E(\varphi)$ l'efficacité. Comme dans le paragraphe 3.3, nous supposons que les

vecteurs coût des processeurs sont linéairement dépendants deux à deux, et de la même façon nous mesurons $\lambda_{520} = 1., \lambda_{530} = 1.25$. Nous pouvons alors utiliser ces valeurs pour calculer de façon très simple la valeur de $\#(\boldsymbol{p}^*, p)$. Nous remarquons alors que dans ce cas l'efficacité est de l'ordre de 80%. Ceci est dû au fait que l'on ne peut pas empêcher l'accès à la machine parallèle virtuelle pendant nos calculs (d'autres calculs sont exécutés sur certains nœuds du réseau). 80% est donc une sous-estimation de l'efficacité réelle. Nous avons vérifié, grâce à l'expérience utilisant un réseau dédié de trois 320, un 550 et un 560, que l'efficacité est proche de 100% quand le réseau est totalement réservé pour notre application. Nous voudrions simplement finir ce premier exemple par la remarque suivante au sujet de l'erreur commise quand on considère qu'un réseau est faiblement hétérogène alors qu'il est fortement hétérogène.

Dans un tel cas de figure, on utilise pour calculer $\#(\boldsymbol{p}^*, p)$ la somme $\sum_{i=1}^{P} \lambda_i$ avec $\lambda_i = T_{\text{seq}}(p)/T_{\text{seq}}(i)$, où $T_{\text{seq}}(i)$ est le temps de calcul séquentiel sur le processeur $i$ pour effectuer la quantité de travail $W$. Alors $\#(\boldsymbol{p}^*, p)$ vaut $T_{\text{seq}}(p) \sum_{i=1}^{P} 1/T_{\text{seq}}(i)$.

Soit

$$\bar{w}_{i,j} = \frac{\bar{W}_i W^j}{W}, \tag{3.12}$$

alors, $T_i = \sum_{j=1}^{J} c_{i,j} \bar{w}_{i,j} = \frac{\bar{W}_i}{W} \sum_{j=1}^{J} c_{i,j} W^j = \frac{\bar{W}_i T_{\text{seq}}(i)}{W}$. $\bar{w}_{i,j}$ est une solution réalisable de (3.7) si et seulement si

$$\forall i = 1, \ldots, P,\ T_i = \bar{T}_{//} \text{ et } \forall j = 1, \ldots, J, \sum_{i=1}^{P} \bar{w}_{i,j} = W^j,$$

– $$\forall i = 1, \ldots, P,\ \bar{W}_i = W \frac{\bar{T}_{//}}{T_{\text{seq}}(i)} \text{ et } \forall j = 1, \ldots, J, \sum_{i=1}^{P} \bar{W}_i = W,$$

– $$\forall i = 1, \ldots, P,\ \bar{W}_i = W \frac{\bar{T}_{//}}{T_{\text{seq}}(i)} \text{ et } \bar{T}_{//} \sum_{i=1}^{P} 1/T_{\text{seq}}(i) = 1.$$

Si nous choisissons pour $\bar{T}_{//} = 1/\sum_{i=1}^{P} 1/T_{\text{seq}}(i)$ et $\forall i = 1, \ldots, P,\ \bar{W}_i = W \frac{\bar{T}_{//}}{T_{\text{seq}}(i)}$, alors $\bar{w}_{i,j}$ défini en (3.12) est une solution réalisable de (3.7). Il est évident que $\bar{T}_{//} \geq T_{//}$. Nous avons $\bar{p}_i = \dfrac{W}{T_i} = \dfrac{\bar{W}_i}{\bar{T}_{//}}$ et nous pouvons établir que :

$$\#(\bar{\boldsymbol{p}}, p) = \#(\bar{p}_1, ..., \bar{p}_p, p) \stackrel{def}{=} \frac{\sum_{i=1}^{P} \bar{W}_i/\bar{T}_{//}}{p} = \frac{W}{p\bar{T}_{//}} = \frac{T_{\text{seq}}(p)}{\bar{T}_{//}} \leq \frac{T_{\text{seq}}(p)}{T_{//}} = \#(\boldsymbol{p}^*, p).$$

Calculer l'efficacité en utilisant $\#(\bar{\boldsymbol{p}}, p)$ comme pour l'application précédente (*i.e.* en supposant que le réseau est faiblement hétérogène) fournit une valeur optimiste de l'efficacité dans le cas où le réseau n'est pas faiblement hétérogène.

### 3.5.2 Décomposition LU

Pour le second exemple, nous allons utiliser un programme qui calcule une factorisation $LU$. Nous avons développé un programme parallèle en prenant modèle sur l'algorithme, appelé SGETRF, utilisé par la bibliothèque LAPACK [A+92]. Pour plus de détails sur cette bibliothèque mathématique d'algèbre linéaire, le lecteur pourra se reporter au chapitre 4.7. Le résultat est un programme Fortran utilisant PVM pour les échanges de messages [CDG+93].

Dans un premier temps, nous avons lié notre code avec des fonctions BLAS[1] non optimisées. Ces fonctions sont utilisées pour obtenir un code séquentiel local aux processeurs qui soit très performant. Les différents constructeurs proposent maintenant des versions de ces fonctions spécialement optimisées pour l'architecture de leurs processeurs. Celles-ci ont montrées leur influence sur les performances globales des algorithmes. Dans la figure 3.1, nous présentons des mesures de l'accélération pour les deux programmes (BLAS optimisées ou non) sur un réseau de processeurs homogènes et sur un réseau de processeurs faiblement hétérogènes.

Comme on peut le voir sur la figure 3.1, l'accélération que nous utilisons sur le réseau hétérogène a la même propriété que l'accélération sur le réseau homogène. En effet, le code non optimisé a de meilleures accélération et efficacité. Une étude plus détaillée se trouve dans l'article de X.H. Sun et J.L. Gustafson, *Toward a better parallel performance metric* [SG91]. Il en résulte que l'accélération et l'efficacité ne peuvent être utilisées pour comparer des algorithmes parallèles. Mais elles peuvent donner de bonnes mesures sur la qualité du parallélisme obtenue et sur les performances globales d'un algorithme sur un réseau de processeurs donné.

[1] Basic Linear Algebra Subroutines

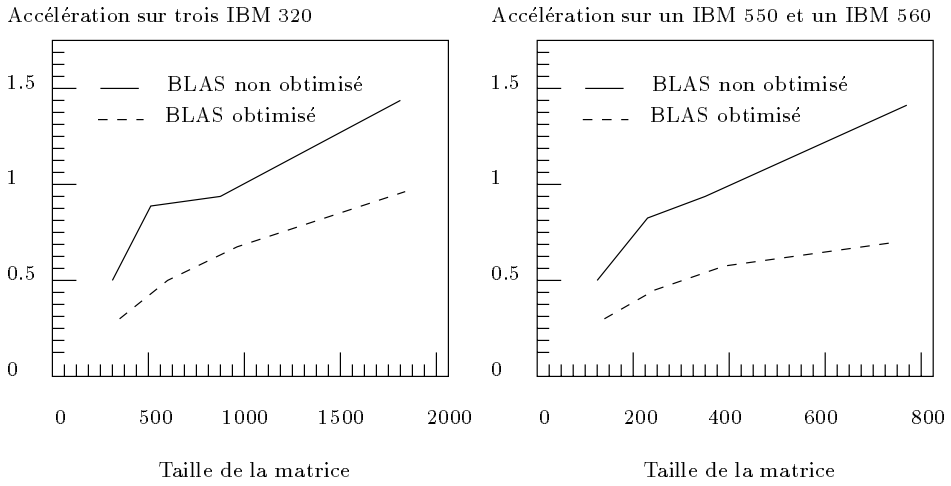



FIG. 3.1 - : Accélération de l'algorithme de factorisation $LU$ en fonction de la taille des matrices traitées. Nous présentons les résultats pour les algorithmes utilisant les BLAS optimisées et non optimisées. A droite : l'accélération sur un réseau homogène. A gauche : l'accélération sur un réseau hétérogène constitué d'un IBM modèle 550 et d'un modèle 560. Le processeur de référence choisi est un IBM modèle 550.

### 3.5.3 Expérimentations avec un réseau fortement hétérogène

Dans l'exemple suivant, nous utiliserons un réseau de cinq stations de travail de type DEC-Alpha qui seront considérées comme une seule ressource de calcul et une machine parallèle de type MasPar (MP1-8192 processeurs). Notre machine virtuelle comporte donc deux nœuds de calcul très hétérogènes.

Nous proposons d'implémenter un programme spécialement construit pour cette architecture hétérogène. Il contient deux types de travaux très différents : un produit scalaire sur des vecteurs contenant des réels double précision et un produit matrice-vecteur de type Laplacien utilisant des entiers. Le second type de travail est bien mieux exécuté sur la machine MasPar car le programme n'utilise que des communications entre processeurs voisins. Sur ce type de machine SIMD à grain fin, plus les communications se font entre les processeurs voisins et plus le programme est efficace [DDT93]. De même, le produit scalaire de deux vecteurs est très bien conçu pour des processeurs ayant une architecture « super-scalaire » comme c'est le cas des stations DEC-Alpha.

Nous allons avec cet exemple étudier les performances d'un programme de grande taille sur un réseau fortement hétérogène. Pour cela nous considérons que le programme est constitué de 14000 produits matrice-vecteur de type Laplacien et de 20000 produits scalaires. Dans le tableau suivant, nous présentons les temps d'exécution des deux différentes parties de l'algorithme sur les deux ressources de

calcul. Pour implémenter les programmes sur le réseau constitué des cinq stations DEC-Alpha, nous avons utilisé PVM.

| Programme \ ressource | $T_{MasPar}$ | $T_{5\ Alpha}$ |
|---|---|---|
| 14000 Laplaciens | 4.99s | 16.37s |
| 20000 Prod. Scal. | 9.91s | 4.29s |
| 14000 Laplaciens + 20000 Prod. Scal. | 15.23s | 21.10s |

TAB. *3.9 - : Temps d'exécution de l'algorithme.*

Pour pouvoir utiliser la formule de l'efficacité obtenue dans le paragraphe 3.3, nous devons définir quels sont les différents types de travaux utilisés par le programme. Nous pouvons utiliser les opérations de base $(+, -, \times, /)$ appliquées aux entiers et aux réels, mais cela ne correspond pas à la granularité de la parallélisation. Pour cet exemple, nous préférons donc définir seulement deux types de travaux. Le premier est le produit matrice-vecteur Laplacien de dimension 8192 utilisant des entiers. Le second est un produit scalaire (Dot) appliqué à des vecteurs de dimension 8192, utilisant des réels double précision.

| Travail \ ressource | MasPar | 5 Alpha |
|---|---|---|
| Laplacien | $3.5610^{-4}$s | $1.1710^{-3}$s |
| Dot | $4.9610^{-4}$s | $2.1510^{-4}$s |

TAB. *3.10 - : Temps d'exécution pour les deux types de travaux utilisés.*

Nous pouvons résoudre le simplexe (3.7), avec $W^{Laplacien} = 14000, W^{Dot} = 20000$, et $c_{MasPar,Laplacien}$, $c_{MasPar,Dot}$, $c_{5\ Alpha,Laplacien}$, $c_{5\ Alpha,Dot}$ dont les valeurs sont données dans le tableau 3.10.

On obtient $T_{//} = 4.82s$, avec le placement suivant :

- $w_{MasPar,Laplacien} = 13551.77$
- $w_{MasPar,Dot} = 0$
- $w_{5\ Alpha,Laplacien} = 448.23$
- $w_{5\ Alpha,Dot} = 20000$

Nous avons donc $\#(\boldsymbol{p}^*, p_{MasPar}) = 3.15$ si le processeur de référence est la MasPar, et $\#(\boldsymbol{p}^*, p_{5\ Alpha}) = 4.37$ si le processeur de référence est l'ensemble des 5 stations Alpha.

Nous considérons maintenant deux placements extrêmes du programme sur les deux ressources de calcul.

| | Mapping | | | |
|---|---|---|---|---|
| Tâche | 5 *Alpha* | *MasPar* | 5 *Alpha* | *MasPar* |
| Laplacien | 14000 | 0 | 0 | 14000 |
| Prod. Scal. | 0 | 20000 | 20000 | 0 |
| Temps | 17.7s | | 6s | |
| $S(\varphi, p_{MasPar})$ | .86 | | 2.54 | |
| $S(\varphi, p_{5\ Alpha})$ | 1.19 | | 3.52 | |
| $E(\varphi)$ | .27 | | .80 | |

TAB. *3.11 - : Deux placements extrêmes du programme.*

Nous pouvons remarquer que, sur un réseau fortement hétérogène, le temps d'exécution du programme avec le second placement multiplié par le nombre de ressources de calcul (ici deux) est plus petit que le temps d'exécution du programme complet sur chacune des deux ressources : $2 \times 6\text{s} \leq 15.23\text{s}$ et $\leq 21.10\text{s}$. Ceci ne peut pas se produire sur un réseau homogène ou même faiblement hétérogène. On peut donc déduire du tableau 3.11 qu'avec le second placement nous obtenons une relativement bonne utilisation de l'hétérogénéité du réseau.

Nous pouvons aussi remarquer, avec cet exemple, que la formule du paragraphe (3.3) peut s'étendre facilement à un réseau dans lequel on trouve comme ressource de calcul une machine parallèle. En fait, il suffit de remplacer l'expression « temps séquentiel » par l'expression « temps d'exécution » .

## 3.6 Conclusion

L'accélération et l'efficacité peuvent être relativement bien généralisées pour les réseaux hétérogènes, bien que, lorsque les processeurs sont très différents, un modèle doive être trouvé afin de caractériser les différents types de travaux pour évaluer la borne supérieure de l'accélération. Cette étude peut être l'un des points de départ pour une formalisation plus précise des moyens d'évaluation de performance des algorithmes s'exécutant sur des réseaux de processeurs hétérogènes.

# Chapitre 4

# Bibliothèques vectorielle et parallèles d'algèbre linéaire

*Pour faciliter la programmation parallèle, des noyaux de calcul algébrique ont été programmés en tenant compte des particularités des différentes machines parallèles à mémoire partagée ou distribuée. Nous allons présenter, dans ce chapitre, la conception et l'évolution des bibliothèques d'algèbre linéaire LINPACK, EISPACK, LAPACK et ScaLAPACK destinées aux ordinateurs à hautes performances.*

## 4.1 Introduction

L'un des problèmes du programmeur d'applications parallèles est d'obtenir un programme qui soit efficace et qui ne dépende pas trop de l'architecture de la machine parallèle cible. Pour réaliser un tel programme, le plus simple est de disposer des bibliothèques de fonctions numériques accessibles sur la plupart des machines parallèles.

### Conception

LINPACK est une collection de programmes Fortran qui résout des systèmes linéaires. Cette bibliothèque est organisée autour de quatre factorisations : LU, CHOLESKY, QR et la décomposition en valeurs singulières.

La conception générale de LINPACK a été fortement influencée par TAMPR et par la notion de BLAS *(Basic Linear Algebra Subroutines)*. TAMPR est un système de développement créé en 1974 [BD74] qui permet de manipuler et de restructurer des programmes en Fortran pour clarifier leur structure. La version principale de tous les programmes de LINPACK utilise l'arithmétique des réels ou des complexes produite par TAMPR. Un utilisateur peut passer d'une arithmétique à une autre en changeant simplement la première lettre du nom du sous-programme. En conséquence, n'importe quelle personne lisant le code

source d'un sous-programme de LINPACK retrouvera facilement les boucles et les structures logiques clairement délimitées par l'indentation de TAMPR.

Les BLAS *(Basic Linear Algebra Subprograms)* forment le noyau de base de tout programme algébrique. Elles ont été imaginées en 1978 [LHKK79]. Elles contribuent à l'efficacité, à la modularité ainsi qu'à la clarté des sous-programmes. LINPACK est diffusée avec des versions de BLAS écrites en Fortran standard qui donnent à l'utilisateur la meilleure efficacité dans le plus grand nombre d'environnements.

LINPACK tient compte du fait que Fortran stocke les tableaux par colonnes. Cela implique quelques modifications des algorithmes conventionnels mais permet une augmentation significative des performances. Cette amélioration est d'autant plus sensible si les machines ont une mémoire hiérarchique. Une technique souvent utilisée est le déroulement des boucles (en anglais : *loop unrolling*). Enfin, on peut noter que la plupart des programmes de LINPACK ne peuvent pas utiliser d'autres programmes de la bibliothèque. Ils peuvent par contre utiliser une ou plusieurs BLAS. Pour faciliter la compréhension, le code source de chaque programme contient la liste des BLAS et des fonctions Fortran utilisées.

### Vers les machines parallèles

Le but du projet LAPACK (*Linear Algebra PACKage*) est de réaliser une bibliothèque d'algèbre linéaire portable utilisant de nouvelles BLAS. Cela permet aux compilateurs de machines parallèles à mémoire partagée de produire une parallélisation efficace des algorithmes d'algèbre linéaire. LAPACK est basé sur les bibliothèques LINPACK et EISPACK qui ont prouvé leur efficacité sur les ordinateurs séquentiels [BDD+88]. De plus, les algorithmes de LAPACK sont restructurés afin de faire appel à un nombre limité de BLAS qui seront optimisées sur chaque machine, alors que les algorithmes numériques, quant à eux, sont portables.

Les échanges de données qui étaient « transparents » pour les machines à mémoire partagée ne le sont plus sur les machines à mémoire distribuée. Pour résoudre ce problème, il a été nécessaire d'étendre les BLAS en leur ajoutant des BLACS (*Basic Linear Algebra Communication Subroutines*). Ces dernières tiennent compte des structures de données utilisées dans les algorithmes d'algèbre linéaire et doivent faciliter leur implantation. Les BLACS ne forment donc pas une bibliothèque de communication complète, mais elles peuvent utiliser des bibliothèques standard de communication par échange de messages comme PVM ou MPI. Afin d'être optimisé, ce nouvel ensemble de routines devra être adapté à chaque architecture. En effet, il existe, en fonction de la topologie du réseau de communication, des algorithmes, comme la diffusion ou l'échange total très utilisés en algèbre distribuée [JH89, dR94], qui minimisent le coût des communications. Le développement de cette bibliothèque permettra de conserver le caractère portable des BLAS sur les calculateurs parallèles ainsi que de garantir

une bonne efficacité et une modularité de haut niveau. Une nouvelle bibliothèque appelée ScaLAPACK a donc été créée pour mettre en œuvre de nouveaux algorithmes d'algèbre linéaire utilisant les BLACS et de nouvelles notions, comme le placement des données, qui n'existaient pas pour les machines vectorielles ou parallèles à mémoire partagée.

**Plan du chapitre**

Après une présentation détaillée des noyaux de calcul BLAS, nous décrirons l'importance et l'utilisation de ces derniers dans les bibliothèques EISPACK, LINPACK et LAPACK. Le paragraphe suivant présente ScaLAPACK et l'évolution de ces bibliothèques pour des machines à mémoire distribuée. Il sera suivi de deux paragraphes présentant de façon approfondie les procédures de communication BLACS et les BLAS distribuées.

## 4.2 L'une des clés de la portabilité : les BLAS

Actuellement, trois facteurs affectent les performances des codes Fortran.

1. **La vectorisation**

   Les algorithmes d'algèbre linéaire peuvent très facilement approcher la puissance crête de la plupart des machines vectorielles. La principale raison est que la puissance maximum dépend du chaînage d'opérations vectorielles d'addition et de multiplication, cette séquence formant le noyau d'exécution de base de tous ces algorithmes. Bien qu'elles soient bien programmées, le plus souvent en Fortran 77, les performances aussi bonnes soient-elles sont décevantes. Ceci est dû à la vectorisation faite par le compilateur qui n'arrive pas à minimiser les accès mémoire.

2. **Le transfert de données**

   Ce qui limite actuellement les performances des processeurs vectoriels, scalaires ou super-scalaires est le taux de transfert des données entre les différents niveaux de la mémoire. Par exemple, les transferts entre les registres vectoriels et la mémoire, ou les échanges entre la mémoire principale et la mémoire « cache » du processeur pour des machines scalaires sont souvent source d'inactivité pour le processeur et donc de perte de performance.

3. **Le parallélisme**

   La structure en boucles imbriquées, souvent appelée « nid de boucles » , de la plupart des algorithmes d'algèbre donne la possibilité d'extraire assez facilement des parties de codes s'exécutant plusieurs fois sans dépendances entre les itérations. Ce type de parallélisme est utilisé dans LAPACK et

ScaLAPACK. De plus, sur les machines parallèles à mémoire partagée, il est automatiquement généré par le compilateur. Pour les ordinateurs à mémoire distribuée, les données doivent pouvoir circuler entre les processeurs. Il est alors nécessaire d'ajouter explicitement des extensions au code Fortran ou C sous la forme de procédures de communication.

Il existe plusieurs niveaux de BLAS, ils sont décrits ci-dessous avec les notations suivantes : $\alpha$ représente un scalaire, $x$ et $y$ sont des vecteurs et $A$, $B$ et $C$ sont des matrices.

▷ **BLAS de niveau 1 :** « opérations vecteur-vecteur » ,

exemple : $x \leftarrow \alpha * x + y$

▷ **BLAS de niveau 2 :** « opérations matrice-vecteur » ,

exemple : $x \leftarrow \alpha * A * x + y$

▷ **BLAS de niveau 3 :** « opérations matrice-matrice » ,

exemple : $C \leftarrow \alpha * A * B$

Les BLAS de niveau 1 introduites dans LINPACK sont efficaces sur les machines scalaires mais sont de granularité trop faible pour être intéressantes sur les ordinateurs vectoriels ou parallèles.

Pour comprendre pourquoi les BLAS de plus grosse granularité sont plus efficaces, il faut étudier la hiérarchie des mémoires d'ordinateurs. Toutes les architectures d'ordinateurs comportent plusieurs niveaux de mémoire, ainsi on peut trouver des registres, puis des caches, puis la mémoire principale et enfin le stockage sur disque. Plus une mémoire se situe haut dans la hiérarchie et plus elle est rapide d'accès, mais elle est aussi plus chère et donc limitée en capacité de stockage. Les traitements ne peuvent s'effectuer que sur les données se trouvant dans la zone la plus haute, d'où un goulot d'étranglement. Les données doivent monter dans toute la hiérarchie pour être traitées puis les résultats doivent redescendre pour être stockés, ce qui provoque une perte de temps.

Il est donc clair qu'un algorithme qui minimise le nombre de montées et de descentes des données sera le plus rapide. Une manière intéressante de mesurer ceci est le rapport du nombre d'opérations en virgule flottante sur le nombre de références à la mémoire d'un algorithme. Plus ce rapport est élevé, plus une donnée est gardée longtemps en mémoire élevée et meilleur est l'algorithme. Par exemple, ce rapport est 2/3 pour un AXPY, 2 pour un produit matrice-vecteur et $n/2$ pour un produit matrice-matrice de taille $n \times n$[ABB+90, Don92]. Le tableau

ci-dessous illustre ces résultats :

| BLAS/Ordinateur | Alliant FX/8 | IBM 3090 | Cray 2S |
|---|---|---|---|
| BLAS1 | 14 | 26 | 121 |
| BLAS2 | 26 | 60 | 350 |
| BLAS3 | 43 | 80 | 437 |
| Puissance Crête | 94 | 108 | 488 |

TAB. *4.1 - : Exemples de vitesse des BLAS en Mflops*

### 4.2.1 Les BLAS de niveau 1

Les BLAS de niveau 1 sont des opérations qui manipulent des vecteurs. Il en existe différentes sortes selon le type des données et des résultats (vecteur $\times$ vecteur $\rightarrow$ vecteur, scalaire $\times$ vecteur $\rightarrow$ vecteur, vecteur $\times$ vecteur $\rightarrow$ scalaire). Le paragraphe suivant donne deux exemples simples de ce type d'opérations.

#### AXPY et DOT

Le noyau AXPY, pour « AX Plus Y » , est la combinaison de deux opérations vectorielles : scalaire $\times$ vecteur $\rightarrow$ vecteur et vecteur $\times$ vecteur $\rightarrow$ vecteur. Le noyau AXPY est utilisé pour multiplier un vecteur $x$ par un scalaire $\alpha$ et additionner ce résultat au vecteur $y$. Le noyau de la fonction AXPY est donc le suivant :

$$\begin{array}{l} Pour\ \ i \leftarrow 1\ \ jusqu'à\ \ n\ \ faire \\ \quad y(i) \leftarrow \alpha * x(i) + y(i) \end{array}$$

FIG. *4.1 - : Algorithme AXPY.*

La somme avec accumulation (DOT) manipule des vecteurs et renvoie un scalaire, c'est une opération de type réduction (produit scalaire). Le noyau d'exécution est le suivant :

$$\begin{array}{l} Pour\ \ i \leftarrow 1\ \ jusqu'à\ \ n\ \ faire \\ \quad s \leftarrow s + x(i) * y(i) \end{array}$$

FIG. *4.2 - : Algorithme DOT.*

### 4.2.2 Les BLAS de niveau 2 et 3

Sur les ordinateurs à processeurs vectoriels, l'ensemble des opérations « matrice-vecteur » a donné lieu à une extension des BLAS, voir [ABD+91b, ABD+91a,

ABD+90], qui semble beaucoup mieux adaptée à l'architecture particulière de ces machines. La bibliothèque originale LINPACK étant basée sur des BLAS de niveau 1, il est alors possible de remplacer les nombreuses boucles qui contiennent des appels aux BLAS, de forme AXPY ou DOT, par un appel à un seul sous-programme BLAS de niveau 2.

Les BLAS de niveau 2 ont été incluses au fur et à mesure de leur mise au point dans les versions successives de LINPACK. Elles ont permis d'atteindre les performances crêtes de plusieurs machines vectorielles mono-processeurs comme les Cray X-MP, Y-MP ou Convex C-2. Par contre, sur d'autres machines telles que le Cray-2 ou l'IBM 30990 VF qui ont plusieurs unités vectorielles, les performances des BLAS de niveau 2 sont limitées par le taux des données circulant entre les différents niveaux de mémoire.

Les BLAS de niveau 3 éliminent en partie ce problème. Ainsi ce troisième niveau de BLAS améliore les performances des algorithmes ayant un coût en $O(n^3)$ pour les opérations flottantes sur $O(n^2)$ données, alors que le deuxième niveau de BLAS n'est efficace que pour des algorithmes ayant un coût en $O(n^2)$ sur des données de taille $O(n^2)$.

Nous allons prendre l'exemple du produit matriciel (BLAS 3) pour exposer la conception de ces algorithmes. La manière la plus simple d'interpréter un produit matrice-matrice est de considérer que chaque coefficient de la matrice résultat est le produit scalaire d'une ligne de la matrice $A$ (de taille $n \times m$) par une colonne de la matrice $B$ (de taille $m \times q$). Cela donne la forme dite *ijk* de l'algorithme (figure 4.4).

```
pour i ← 1 jusqu'à n
    pour j ← 1 jusqu'à m
        C(i,j) ← 0.0
        pour k ← 1 jusqu'à q
            C(i,j) ← C(i,j) + A(i,k) * B(k,j)
```

FIG. *4.3 - : Algorithme de la forme ijk.*

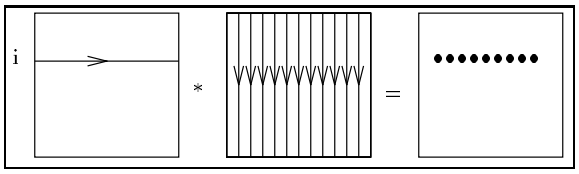



FIG. *4.4 - : Forme ijk du produit matriciel ($n = m = q$).*

On obtient une autre forme de l'algorithme en permutant les indices des boucles. Si on met la boucle sur $k$ à l'extérieur, cela donne la forme *kij*. Une

boucle sur $i$ de cet algorithme est une suite de AXPY sur les colonnes de $A$ et fournit une composante pour chaque coefficient de $C$ (figure 4.6).

```
pour i ← 1 jusqu'à n
    pour j ← 1 jusqu'à m
        C(i,j) ← 0.0
pour k ← 1 jusqu'à q
    pour i ← 1 jusqu'à n
        pour j ← 1 jusqu'à m
            C(i,j) ← C(i,j) + A(i,k) * B(k,j)
```

FIG. *4.5 - : Algorithme de la forme kij.*

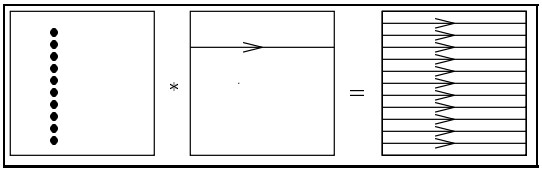



FIG. *4.6 - : Forme kij du produit matriciel.*

En permutant à nouveau les indices des boucles, on obtient une autre interprétation possible du produit matriciel qui revient à considérer $m$ produits matrice-vecteur de A par les colonnes de B et conduit à la forme suivante dite *ikj* (figure 4.8).

```
pour i ← 1 jusqu'à n
    pour j ← 1 jusqu'à m
        C(i,j) ← 0.0
pour j ← 1 jusqu'à m
    pour k ← 1jusqu'à q
        pour i ← 1 jusqu'à n
            C(i,j) ← C(i,j) + A(i,k) * B(k,j)
```

FIG. *4.7 - : Algorithme de la forme ikj.*

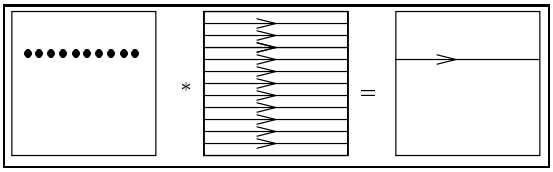



FIG. *4.8 - : Forme ikj du produit matriciel.*

Les formes $ikj$ et $jki$ utilisent GAXPY comme opération de base : les multiples d'un ensemble de vecteurs sont accumulés dans un même vecteur avant d'être stockés en mémoire. La version par colonnes ($jki$) est par conséquent la plus performante des six possibles. Elle fonctionne en mode super-vectoriel (voir [CT93]).

### Tableaux de résultats

Les constructeurs se réfèrent souvent à la performance de crête pour décrire leur machine. Cette performance est calculée généralement en comptant le nombre de multiplications et d'additions flottantes exécutées durant un cycle machine. Pour plus de résultats, on peut consulter [Don88, Phi91].

Nous avons testé les procédures de décomposition LU (_GETRF) et QR (_GEQRF) pour des matrices de taille $512 \times 512$ sur deux machines : la première est une station IBM RS 6000 modèle 320 avec 80 Mo de RAM et la seconde est une station IBM RS 6000 modèle 550 avec 384 Mo de mémoire principale. Nous avons également essayé de compiler les BLAS de LAPACK avec deux options différentes pour le compilateur Fortran XLF d'IBM. La colonne BLAS A a été obtenue avec les options -O -u et la colonne BLAS B avec les options -O -P -u. Cela donne les tableaux suivants :

| BLAS | BLAS A | BLAS B | BLAS IBM |
|---|---|---|---|
| SGETRF | 8,39 | 7,69 | 23,76 |
| DGETRF | 8,21 | 7,49 | 21,43 |
| SGEQRF | 7,94 | 7,81 | 23,65 |
| DGEQRF | 8,73 | 7,32 | 19,47 |

TAB. *4.2 - : Vitesse de quelques BLAS en Mflops sur 320*

| BLAS | BLAS A | BLAS B | BLAS IBM |
|---|---|---|---|
| SGETRF | 17,80 | 18,27 | 52,25 |
| DGETRF | 18,69 | 18,05 | 49,09 |
| SGEQRF | 16,42 | 18,00 | 51,58 |
| DGEQRF | 21,52 | 18,74 | 47,74 |

TAB. *4.3 - : Vitesse de quelques BLAS en Mflops sur 550*

### Remarques

On peut remarquer premièrement que le fait de précalculer les BLAS à la compilation n'a que peu d'influence sur la vitesse de calcul. On peut en déduire que les BLAS ont été très bien écrites et que le compilateur n'apporte rien de

plus. Deuxièmement, on constate que les BLAS du constructeur sont beaucoup plus rapides (on gagne environ un facteur 2,5) que les BLAS Fortran fournies avec LAPACK. Il est donc essentiel qu'à l'avenir tous les constructeurs de machines fournissent aux utilisateurs des BLAS micro-codées. Ceci sera encore plus sensible sur les ordinateurs à mémoire distribuée.

## 4.3 LINPACK, EISPACK et LAPACK

### 4.3.1 Introduction

LINPACK est une bibliothèque de sous-programmes Fortran qui résout des systèmes variés d'équations algébriques linéaires. Les différents sous-programmes qui la composent ont été conçus de façon à être totalement indépendants de l'ordinateur utilisé et à s'approcher le plus possible de l'efficacité optimum pour un maximum de calculs. Les noms des sous-programmes sont significatifs, ils sont formés de cinq lettres : TXXYY. La première lettre T indique le type de données avec lesquelles on travaille. Si cette première lettre est un S cela signifie que les nombres utilisés sont des réels en Simple précision. Il existe trois autres types de données D (Double précision), C (Complexe) et Z (complexe en double précision). Les deux lettres suivantes XX indiquent la forme de la matrice ou sa décomposition, par exemple GE signifie que la matrice est pleine et de forme standard (non symétrique par exemple) alors que PB annonce une matrice bande définie positive. Les deux dernières lettres YY indiquent le type de calcul fait par le sous-programme, en exemple on peut citer SL pour SoLve, DC pour DeCompose ou encore UD pour UpDate (mise à jour)...

Attention, toutes les combinaisons ne sont pas possibles ! Par exemple : STRFA, SCTFA... n'existent pas. L'appel aux sous-programmes de la bibliothèque LINPACK est identique à celui des autres fonctions Fortran. Exemple :

```
CALL SGEFA(A,LDA,N,...)
```

où A est le nom du tableau de dimension 2, LDA est la dimension du tableau et N est la dimension de la matrice stockée dans le tableau. Si la matrice est rectangulaire, alors N indique le nombre de lignes et un autre paramètre M indique le nombre de colonnes. Les autres arguments des fonctions sont généralement liés à l'opération demandée.

EISPACK est une bibliothèque de procédures Fortran permettant le calcul des valeurs et des vecteurs propres. Cette bibliothèque utilise aussi les noyaux d'exécution de type BLAS ainsi que des routines d'algèbre linéaire de LINPACK. Avec la création des BLAS de niveaux 2 et 3 et leur mise en œuvre dans les procédures de LINPACK, la bibliothèque EISPACK a dû elle aussi être mise à jour afin de bénéficier de la meilleure efficacité possible.

LAPACK est la synthèse des bibliothèques LINPACK et EISPACK. Cette fusion a pour avantage d'être plus compacte car beaucoup de BLAS communes étaient contenues dans ces bibliothèques, et d'être plus aisément maintenable car elle est centralisée. Pour les BLAS, on ne peut pas faire totalement abstraction de l'architecture de la machine. Avec l'apparition des ordinateurs parallèles, une nouvelle philosophie de programmation est apparue. Les ordinateurs, qu'ils soient à mémoire distribuée ou partagée, sont de plus en plus difficiles à programmer et il n'existe pas actuellement de méthode de programmation systématique. La bibliothèque LAPACK, qui est développée sur la base de LINPACK, est destinée aux architectures parallèles à mémoire partagée. Elle doit donc utiliser des BLAS étendues (niveau 2 ou 3). Certains programmes BLAS ont été modifiés pour utiliser au mieux les possibilités des ordinateurs modernes et cela a conduit aux algorithmes de calcul par blocs qui utilisent les BLAS de niveau 3. Mais LAPACK doit être encore améliorée (ajout de procédures de communication par échange de message) pour fonctionner sur les ordinateurs à mémoire distribuée.

Pour la résolution des systèmes linéaires, LAPACK permet de calculer des factorisations triangulaires (LU) et de les résoudre par substitution. Il est possible de manipuler des matrices générales, symétriques et symétriques définies denses ou bandes. Les matrices bandes et triangulaires sont stockées en tenant compte de leur structure particulière. Pour résoudre les problèmes de moindres carrés, LAPACK utilise la décomposition QR des matrices.

Pour tous les problèmes de valeurs propres, LAPACK permet de déterminer les valeurs propres, la forme de Schur de la matrice et les vecteurs propres. LAPACK peut aussi être utilisé pour calculer une décomposition selon les valeurs singulières.

LAPACK comprend également des sous-programmes pour le calcul du conditionnement des systèmes linéaires, des problèmes aux valeurs propres, vecteurs propres et sous-espaces invariants.

Pour les problèmes non symétriques, les algorithmes concernant les valeurs et vecteurs propres sont basés sur la décomposition QR, suivie de la résolution d'un problème tridiagonal, qui, pour être optimale, doit être adaptée à chaque architecture.

### 4.3.2 Les algorithmes par blocs

#### Factorisation LU

Tous les algorithmes utilisés ont été écrits en utilisant des calculs par blocs pour utiliser au maximum les BLAS de niveaux 2 et surtout 3. Voyons par exemple ce que cela donne pour la factorisation LU, qui sert dans l'algorithme d'élimination de Gauss, en commençant par utiliser les BLAS 2.

Niveau 2 :

On considère une matrice $A$ régulière que l'on décompose sous la forme $PLU$ où $P$ est une matrice de permutation (qui représente le choix des pivots), $L$ une matrice triangulaire inférieure unitaire et $U$ une matrice triangulaire supérieure. A partir de cette factorisation, résoudre le système $Ax = b$ revient à résoudre deux systèmes triangulaires, en utilisant deux appels de BLAS 2. Cet algorithme procède comme suit:

Au pas $j$, on a déjà calculé les $j-1$ premières colonnes de $L$ et de $U$. On a donc :

$$\begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & L_{23} \end{bmatrix} * \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

où $L_{11}$ et $U_{11}$ sont de taille $(j-1)*(j-1)$ et $L_{22}$, $U_{12}$ et $U_{22}$ sont des colonnes simples. $L_{11}$, $L_{21}$ et $U_{11}$ sont connues, on veut calculer $L_{22}$, $U_{12}$ et $U_{22}$. En identifiant le produit par blocs, on obtient :

1. $L_{11} * U_{12} = A_{12}$ d'où on tire $U_{12}$ par résolution d'un système triangulaire (BLAS 2).

2. $L_{21} * U_{12} + L_{22} * U_{22} = A_{22}$. On commence par mettre à jour $A_{22} \leftarrow A_{22} - L_{21} * U_{12}$ ; c'est un produit matrice-vecteur local (BLAS 2). Ensuite, on permute les $n-j+1$ dernières lignes de $A$ pour que le pivot en tête de $A_{22}$ soit le plus grand (mise à jour de $P$). $U_{22}$ est constituée de ce pivot puis de zéros, $L_{22}$ contient les coefficients de $A_{22}$ divisés par ce pivot.

Niveau 3 :

Pour écrire cet algorithme en utilisant les BLAS de niveau 3, on utilise la décomposition par blocs, mais on traite $n_b$ colonnes à chaque pas. La taille de bloc qui minimise les échanges entre les différents niveaux de mémoire dépend de l'architecture. Au pas $j$, on a déjà calculé $n_b(j-1)$ colonnes de $L$ et de $U$ et on va calculer les $n_b$ colonnes suivantes ($L_{22}$, $U_{12}$ *et* $U_{22}$). Cela donne :

1. $L_{11} * U_{12} = A_{12}$ d'où on tire $U_{12}$ par la résolution d'un système triangulaire avec $n_b$ colonnes en second membre (BLAS 3).

2. $L_{21} * U_{12} + L_{22} * U_{22} = A_{22}$. On commence par mettre à jour $A_{22} \leftarrow A_{22} - L_{21} * U_{12}$ ; c'est un produit matrice-matrice (BLAS 3). Ensuite, on permute les $n-j+1$ dernières lignes de $A$ pour que le pivot en tête de $A_{22}$ soit le plus grand (mise à jour de $P$). On factorise $A_{22} = P^t * L_{22} * U_{22}$ en utilisant l'algorithme qui fonctionne avec les BLAS 2.

Il existe d'autres manières de réécrire l'algorithme du pivot de Gauss : il y a en effet six possibilités d'ordonner les trois boucles de l'algorithme original [CT93]. L'ordre optimal dépend de l'architecture considérée et de la façon dont sont stockées les données. Le but de LAPACK étant d'atteindre la meilleure performance possible avec un code portable, c'est l'algorithme utilisant les BLAS 3 qui est utilisé.

### Factorisation QR

On peut aussi écrire par blocs l'algorithme de factorisation d'une matrice $A$ sous la forme $QR$ où $Q$ est une matrice orthogonale et $R$ une matrice triangulaire supérieure.
Soit $A$ une matrice de taille $m*n$. L'algorithme de décomposition $QR$ est itératif. Dans la méthode classique une itération consiste à effectuer le produit $H*A$ où $H = I - 2*u.u^t$ avec $\|u\| = 1$. Cela se décompose en deux opérations qui sont des BLAS de niveau 2 :

$$\begin{array}{|l} z \leftarrow A^t * u \\ A \leftarrow A - 2 * u * z^t \end{array}$$

FIG. *4.9 - : Décomposition en BLAS de niveau 2.*

La matrice qui correspond à $b$ étapes de cette itération est $H_1 * \cdots * H_b$. Bischof et Van Loan [Bis88, GL89] ont montré que cette matrice peut s'écrire $W * Y^t$ où $W$ et $Y$ sont des matrices de taille $m * n$. Schreiber et Van Loan ont montré ensuite que l'on peut écrire $W = Y * T$ où $T$ est une matrice triangulaire supérieure de taille $b * b$. On a donc $Q = I + Y * T * Y^t$. On appelle cette forme de $Q$ la forme compacte. Cette représentation prend un peu plus de place en mémoire et le produit $Q * A$ demande quelques opérations supplémentaires, mais celles-ci s'expriment beaucoup mieux en termes de BLAS que la forme précédente, d'où son intérêt. En effet, cette méthode utilise deux produits de matrices pour calculer $Z = A^t * Y * T$ et une mise à jour de rang $b$. On utilise donc des BLAS 3. La méthode standard, quant à elle, requiert $b$ produits matrice-vecteur et autant de mises à jour de rang 1 qui sont des opérations correspondant à des BLAS de niveau 2, donc moins rapides. Le nouvel algorithme se décompose en deux étapes:

1. $[Y, T] \leftarrow calcyt(A)$ : calcul de la forme compacte de $Q$ qui donne $Y$ et $T$ vérifiant $Q = I + Y * T * Y^t$.

2. $A \leftarrow applyt(Y, T, A)$ : calcul de la nouvelle valeur de $A$.

Pour réaliser ces calculs, on découpe $A$ en $M * N$ blocs de taille $b * b$ (on suppose que $m$ et $n$ sont des multiples de $b$ pour exposer la méthode). Dans la suite $A(i, j)$ représente le bloc de taille $b * b$ situé en $i^{ème}$ ligne et $j^{ème}$ colonne de $A$. On utilisera aussi la notation $A(i : j, .)$ qui représente l'ensemble des blocs $A(i, .)$ à $A(j, .)$. On décompose l'algorithme précédent de la manière suivante pour obtenir un calcul par blocs :

$$
\begin{array}{l}
Pour\ \ i \leftarrow 1\ \ jusqu'à\ \ n\ \ faire \\
\quad [Y,T] \leftarrow calcyt(A(i:M,i)) \\
\quad A(i:M,i:N) \leftarrow applyt(Y,T,A(i:M,i:N))
\end{array}
$$

FIG. *4.10 - : Calcul par blocs.*

Cet algorithme illustre une propriété très importante des algorithmes par blocs : c'est un algorithme qui nécessite plus d'opérations en virgule flottante que son homologue standard mais à vitesse presque optimum car il n'est pas ralenti par les mouvements de données. Cela est valable tant que la taille des blocs est raisonnable, sinon ils ne rentrent plus dans la mémoire cache. La taille optimum des blocs dépend de la nature du problème (car il faut des blocs les plus indépendants possible) et de l'architecture de la machine.

Cet algorithme calcule un bloc de $A$ puis met à jour la sous-matrice qui reste, on dit que c'est un algorithme *right looking*. La mise à jour se fait sur une matrice de taille $(m-(i-1)*b+1)*(n-(i-1)*b+1)$. On peut réduire les transferts de données avec l'algorithme suivant :

$$
\begin{array}{l}
Pour\ \ i \leftarrow 1\ \ jusqu'à\ \ n\ \ faire \\
\quad Pour\ \ j \leftarrow 1\ \ jusqu'à\ \ i-1\ \ faire \\
\quad\quad A(j:M,i) \leftarrow applyt(Yj,Tj,A(j:M,i)) \\
\quad\quad [Yi,Ti] \leftarrow calcyt(A(i:M,i))
\end{array}
$$

FIG. *4.11 - : Algorithme avec réduction des transferts de données.*

Cet algorithme est dit *left looking* car on met à jour les blocs de la matrice juste avant de les utiliser (voir figure 4.12).

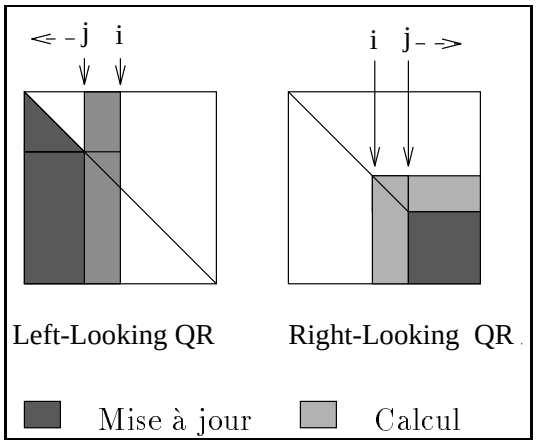



FIG. *4.12 - : Deux variantes de la décomposition QR*

L'intérêt de cette nouvelle méthode est que les écritures en mémoire sont plus localisées, il y a donc moins d'accès à la mémoire de bas niveau tout en utilisant les mêmes BLAS. On réutilise au maximum les données qui se trouvent dans la mémoire cache avant d'en charger d'autres.

### 4.3.3 Performances des ordinateurs

Les performances d'un ordinateur ne sont pas faciles à définir car elles dépendent de nombreux paramètres. Ces paramètres sont par exemple le type de l'application, l'algorithme, la taille du problème, le langage utilisé, l'effort qui a été fait par le programmeur afin d'optimiser au mieux son algorithme, l'architecture de la machine, etc.. Les performances sont mesurées en terme de Megaflops, ce qui signifie en français millions d'opérations flottantes par seconde, les opérations flottantes étant au format IEEE (64-bit d'opérande). Voici un exemple des performances mesurées pour les décompositions LU (_GETRF) et QR (_GEQRF) d'une matrice $512 \times 512$ [ABB+90] :

| Procédure | Convex C210 | Cray Y-MP (1 proc) | Cray Y-MP (8 proc) |
|---|---|---|---|
| SGETRF | 36 | 290 | 1039 |
| SGEQRF | 38 | 294 | 1476 |

TAB. *4.4 - : Vitesse de quelques procédures en Mflops*

### 4.3.4 Les *benchmarks* de LINPACK et LAPACK

#### Introduction

Les premiers *benchmarks* de la bibliothèque LINPACK apparaissent en appendice du « LINPACK Users'Guide » [DMBS79] en 1979. Cet appendice comprend des résultats, obtenus sur les ordinateurs les plus courants à cette époque, pour les fonctions les plus largement utilisées de LINPACK, et pour des matrices de taille 100. Ainsi un utilisateur peut estimer le temps nécessaire pour résoudre un problème matriciel. Au fil des années, le nombre d'ordinateurs sur lesquels ont été testés les *benchmarks* est passé à plus de 200 ; parallèlement, les *benchmarks* ont été étendus aux trois niveaux de BLAS avec des tailles de matrices différentes pour chacun. Ainsi pour les BLAS de niveau 1, la taille des matrices est de 100, alors que celle-ci passe à 300 pour les BLAS de niveau 2 et à 1000 pour les calculs avec les BLAS 3. L'augmentation de la taille des matrices est due au fait que le gain en temps ne se fait pas sur le nombre d'opérations arithmétique effectuées, mais sur le transfert des données. La taille plus importante des systèmes linéaires utilisés par les *benchmarks* permet de mettre en valeur le gain obtenu par la modification des flux de données.

### Résultats sur stations de travail

Les processeurs des stations de travail sont pour la plupart utilisés dans les machines parallèles. Il est donc intéressant de connaître leurs performances afin d'optimiser au mieux le code séquentiel qui sera exécuté sur un nœud de ces machines. Le tableau suivant représente l'exécution du *benchmark* « 1000du.f » fourni avec la bibliothèque LAPACK.

| Performance \ Station | IBM 370 | IBM 580 | IBM 590 | DEC 40000 |
|---|---|---|---|---|
| Mflops | 24.2 | 27.3 | 57.3 | 13.2 |
| Relat. | 0.89 | 1 | 2.10 | 0.48 |

où Relat. est le temps est compté en Mflops relatif par rapport à l'IBM 580.

### Utilisation des *benchmarks*

La résolution d'un système d'équations, basée sur la décomposition LU, nécessite $O(n^3)$ opérations flottantes, et plus précisément $2/3n^3+2n^2+O(n)$ additions et multiplications flottantes. Alors le temps requis pour résoudre un système linéaire, avec les BLAS de niveau 1, sur un ordinateur donné s'approche de la façon suivante :

$$time_n \approx time_{100} \times \frac{n^3}{100^3}.$$

Pour les *benchmarks* de LINPACK une taille de matrice égale à 100 est utilisée comme base, car le terme en $O(n^2)$ n'a pas d'effet important sur les temps de calcul des algorithmes pour des valeurs de n supérieures. Pour les *benchmarks* utilisant des BLAS de niveau 2, on approche les performances avec la formule suivante :

$$time_n \approx time_{100} \times \frac{\frac{2}{3}\times n^3+2\times n^2}{\frac{2}{3}^3+2\times 100^2}.$$

## 4.4 L'évolution : ScaLAPACK

La conception de LAPACK a posé plusieurs problèmes qui n'ont pas tous été résolus de manière définitive. Actuellement, la bibliothèque est écrite en Fortran 77 qui est portable, l'équipe qui écrit LAPACK a pour projet d'en faire des versions en Fortran 90 et en C [ABD+90]. L'un des problèmes est l'adaptation des BLAS à la machine sur laquelle on travaille : il est nécessaire d'en connaître quelques caractéristiques comme l'erreur d'arrondi et les seuils de dépassement de capacité de calcul. Le dépassement de ces seuils étant une erreur bloquante sur la plupart des machines, il faut le détecter sans provoquer cette erreur. Dans la version actuelle, c'est la routine SLAMCH qui effectue ce travail lors de l'installation de LAPACK. Dès que la norme IEEE et un langage standard de haut

niveau qui gère ses traitements d'exception seront disponibles, cette procédure, très complexe, ne sera plus nécessaire.

Avec la version actuelle de LAPACK, sur les machines parallèles, à mémoire partagée ou distribuée, c'est le compilateur qui, s'il est bien conçu, se charge de la parallélisation des programmes. Les algorithmes ont été écrits pour que cette parallélisation automatique se passe bien. En effet, toutes les fonctions algébriques utilisées utilisent les BLAS de niveau 3. Cependant, pour implanter LAPACK de manière plus efficace sur les ordinateurs à mémoire distribuée, des sous-programmes de communication appelés BLACS ont été ajoutés. Cette évolution très importante des algorithmes a entraîné la création d'une nouvelle bibliothèque, ScaLAPACK, qui est la bibliothèque LAPACK pour machines parallèles à mémoire distribuée [CDW93c, CDW93a]. Cette bibliothèque contient également la version distribuée des BLAS de niveau 3 présentées sous la forme de bibliothèques PBBLAS pour *Parallel Block BLAS* et PUMMA pour *Parallel Universal Matrix Multiplication Algorithms*, cette dernière étant plus optimisée mais encore en cours de développement. ScaLapack contient également la bibliothèque de communication adaptée à l'algèbre linéaire, BLACS, ainsi que les BLAS séquentielles de niveaux 1, 2 et 3. Les buts principaux lors du portage de ScaLAPACK sont de conserver la portabilité, la flexibilité et la facilité d'utilisation. La «scalabilité» est très importante car elle réfère au comportement d'un algorithme lorsque le nombre de processeurs augmente. Ceci est primordial afin de connaître le comportement futur des algorithmes sur les machines massivement parallèles. Les routines de ScaLAPACK disponibles actuellement sont essentiellement des factorisations de matrices ($LU$, $QR$, $HDR$ et $LL^t$), disponibles sur les machines d'Intel iPSC/860, Delta, Paragon, Thinking Machine Corporation CM-5 et les réseaux d'ordinateurs utilisant PVM. Une des particularités de la bibliothèque est que tous les appels nécessaires à l'utilisation du parallélisme sont cachés à l'intérieur des PBBLAS et des procédures PUMMA.

Plusieurs interfaces d'appels sont envisagées pour ScaLAPACK. La première sera la même que LAPACK originelle, avec des arguments supplémentaires pour spécifier la distribution des matrices sur les processeurs. Une deuxième version n'aura plus les arguments supplémentaires de rangement de matrices mais ces renseignements seront passés grâce à des procédures d'initialisation. Une dernière version concerne le portage pour des langages orientés objets comme C++, la

bibliothèque est alors appelée ScaLAPACK++ [DPW93].

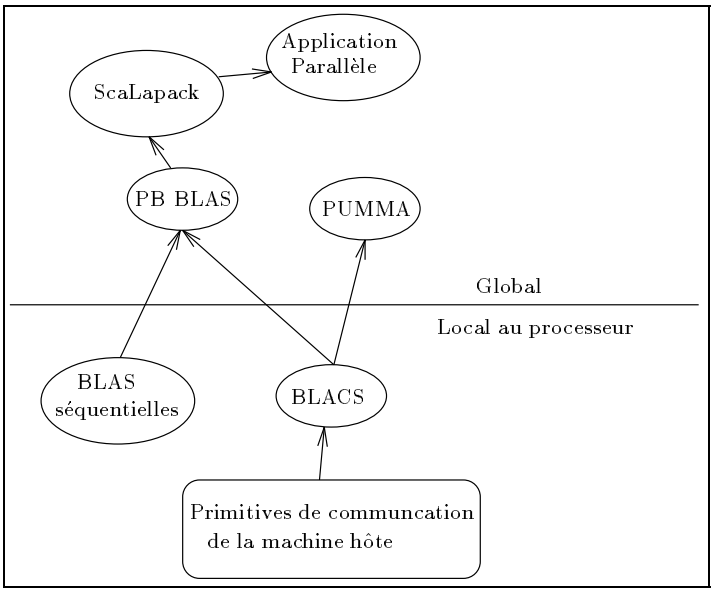



FIG. *4.13 - : Les différents modules de ScaLAPACK*

## 4.5 BLACS

Les BLACS sont un ensemble de procédures Fortran qui sont conçues dans le même esprit que l'avaient été les BLAS. En effet, elles doivent être très faciles d'emploi et utilisables sur toutes les différentes architectures parallèles distribuées. Par conséquent, l'écriture d'un programme algébrique sur ordinateurs MIMD consiste en une succession d'appels à des procédures BLACS pour les communications et BLAS pour les calculs matriciels. Ces procédures étant optimisées pour les opérations algébriques, de bonnes performances peuvent être obtenues avec une relative facilité. De plus, le changement de plate-forme de calcul ne nécessite que des modifications minimales de l'application.

### 4.5.1 Notations et modèle de calcul

Pour toute la suite de ce paragraphe, la notion de matrice sera équivalente à la notion de tableau de dimension 2. Ceci est dû au fait que les matrices sont supposées non creuses. De plus, les processeurs seront considérés comme étant $p$ nœuds connectés en grille de manière logique, la topologie physique pouvant être un anneau, un hypercube ou même un réseau multi-étages. Ce réseau logique est utilisé pour pouvoir connaître la position exacte d'un nœud dans le réseau par ces deux coordonnées $\{i, j\}$, avec $0 \leq i < L$, $0 \leq j < C$ et $L \times C = p$.

La structure en grille permet à de nombreux placements de tirer avantage de la structure régulière des matrices pleines.

### 4.5.2 Convention d'écriture des BLACS

Comme pour les BLAS, les procédures BLACS ont une convention d'écriture bien précise, qui permet à l'utilisateur de connaître la signification exacte d'une procédure grâce à un nom de sept lettres. Sa forme générale est :

vXXYY2D *pour les primitives de communication*
vGZZZ2D *pour des opérations globales*

La lettre v, commune aux deux formes, indique le type des données qui doivent être communiquées. Par exemple, un C signifie que la communication ou l'opération globale porte sur des complexes.

#### Les primitives de communication

Les deux lettres XX de la forme vXXYY2D représentent la structure de données utilisée. Il n'y a que deux possibilités pour cette option :

| | |
|---|---|
| GE | Matrice rectangulaire Générale |
| TR | Matrice Triangulaire |

Les deux lettres suivantes YY indiquent la fonction de la procédure. Les choix possibles sont :

| | |
|---|---|
| SD | Envoi d'un message |
| RV | Réception d'un message |
| BS | Diffusion d'un message |
| BR | Réception d'un message envoyé par une diffusion |

Exemple : CGEBR2D(SCOPE, TOP, M, N, A, LDA, RSRC, CSRC)
où

| | |
|---|---|
| SCOPE | Indique quels sont les processeurs impliqués dans la diffusion ; les valeurs possibles sont « ROW » , « COLUMN » ou « ALL » |
| TOP | Indique la topologie physique de la machine |
| M et N | Indiquent la taille de la matrice envoyée ou reçue |
| A | Matrice envoyée ou reçue |
| LDA | Principale dimension du tableau A |
| RSRC etCSRC | Indice de ligne et de colonne du nœud émetteur |

#### Les opérateurs globaux

Pour un opérateur global, il n'y a que trois options possibles qui sont indiquées par les lettres ZZZ. Ces opérations sont :

| | |
|---|---|
| MAX | Le résultat renvoyé est le maximum d'une variable donnée |
| MIN | Le résultat renvoyé est le minimum d'une variable donnée |
| SUM | Le résultat renvoyé est la somme d'une variable donnée |

Exemple : CGESUM2D(SCOPE, TOP, M, N, A, LDA, RDEST, CDEST)
où

| | |
|---|---|
| RDEST et CDEST | Indice de ligne et de colonne du nœud de stockage du résultat |

### Exemple d'utilisation sur un réseau de stations

Pour illustrer la facilité d'utilisation des BLACS, nous présentons ici un programme calculant un produit matrice-vecteur ($b$= A $x$) par blocs. La figure 4.14 indique le placement des blocs de la matrice A et du vecteur $b$ sur un tore de $2 \times 2$ processeurs. La matrice A va donc être découpée en quatre blocs (notés $A_{11}$, $A_{21}$, $A_{12}$ et $A_{22}$). Les variables LM et LN indiquent respectivement le nombre de lignes et de colonnes du bloc de la matrice A stocké dans la mémoire locale d'un processeur.

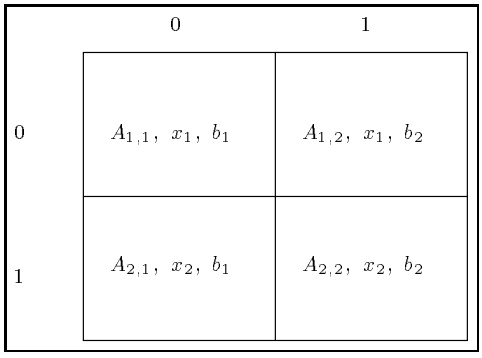



FIG. *4.14 - : Produit matrice-vecteur sur une grille* $2 \times 2$.

Le programme consiste en une diffusion des deux parties du vecteur $x$ suivant les lignes, puis en un produit matrice-vecteur (BLAS 2) sur chaque processeur, enfin en une diffusion sur les colonnes et une addition des résultats partiels du vecteur $b$. Le programme 4.17 représente l'écriture en Fortran de l'algorithme décrit précédemment. Il utilise une procédure BLAS de niveau 2 afin d'optimiser le code séquentiel des processeurs.

Nous avons testé le programme 4.17 sur un réseau de 4 stations de travail SUN. Les résultats sont bons comparés à une version qui utilise directement les fonctions PVM 4.15. Il n'y a donc pas trop de pertes dues au rajout de la bibliothèque BLACS. Nous observons sur la figure 4.16 que l'utilisation des fonctions de calcul BLAS fournies par le constructeur permet un gain de performance important. La

mise en œuvre des bibliothèques BLACS et BLAS permet de construire facilement des programmes efficaces.

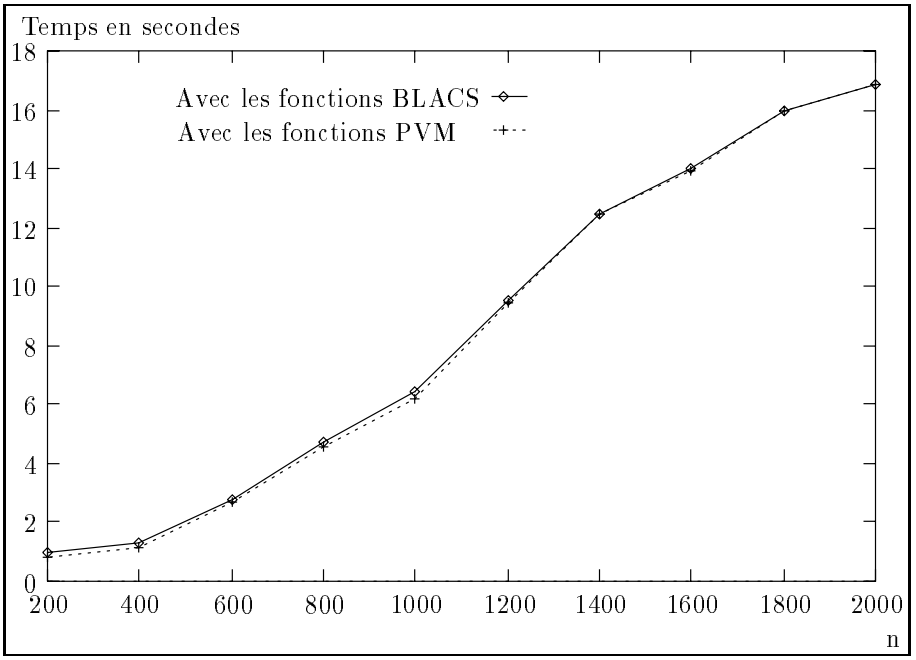



FIG. *4.15 - : Produit matrice-vecteur utilisant des fonctions BLACS et PVM.*

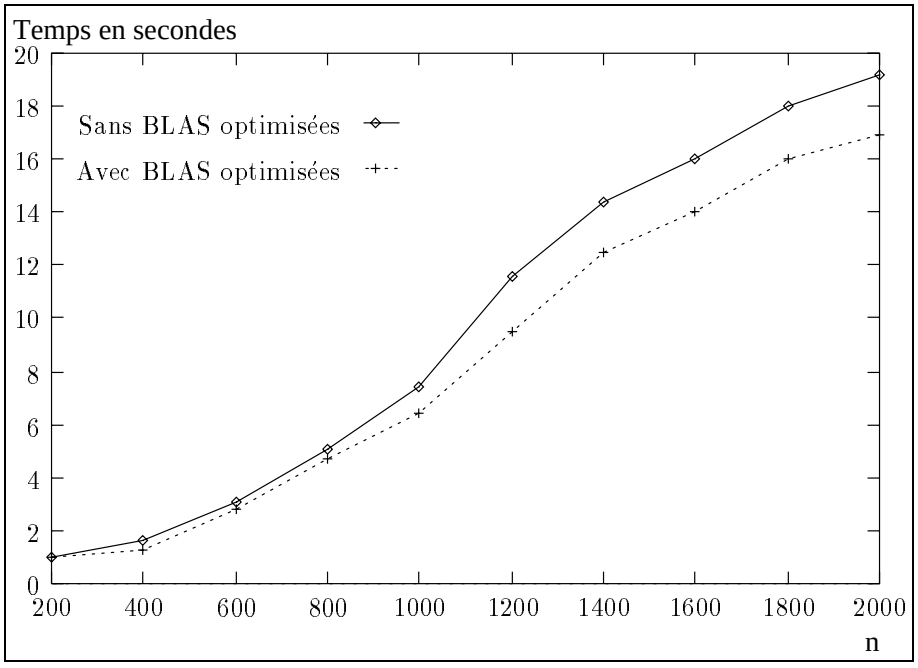



FIG. *4.16 - : Expérimentation avec la version constructeur des BLAS.*

```
      program mvpoc
c     Procedures externes
      external AUXSETUP, BLACSINIT, GRIDINFO, DGMAX2D, DCSUN2D, DGEMV
c     Scalaires
      integer LDA, LM, LN
      parameter (LDA = 50)
      parameter (LM = 50)
      parameter (LN = 50)
      integer IAM, MYCOL, MYROW, NPCOL, NPROW, NNODES, CONTINUE
      integer I, J
c     Tableaux
      double precision A(LM,LN), X(LN), B(LM), B1(LM)
      call INITPINFO(IAM,NNODES)
      if (NNODES .lt. 1) then
         NNODES=4
         call AUXSETUP(IAM,NNODES)
      endif
      NPROW=2
      NPCOL=2
c     Creation de la grille
      call BLACSINIT(NPROW,NPCOL)
      call GRIDINFO(NPROW,NPCOL,MYROW,MYCOL)
      if (MYROW .eq. 0) then
         do J = 1, LN
            X(J) = 2.
         enddo
c     Tous les processeurs lignes ont besoin d'une copie de X
         call DGEBS2D('COLUMN','I',LN, 1,X,LN)
      else
c     Les autres processeurs recoivent le vecteur X
         call DGEBR2D('COLUMN','I',LN, 1,X,LN,0,MYCOL)
      endif
c     Calcul de A*x=b
      call DGEMV('N',LM,LN,1.0D0,A,LDA,X,1,0.0D0,B,1)
c     Addition des résultats partiels de b
      call DGSUM2D('ROW','1',LM,1,B,LM,-1,0)
      if (MYROW .eq. 0) then
         if (MYCOL .eq. 0) then
            call DGERV2D(LM,1,B1,LM,0,1)
         else
            call DGESD2D(LM,1,B,LM,0,0)
         endif
      endif
      call BLACSEXIT(CONTINUE)
1000   format('B = ',G20.15)
      end
```

FIG. *4.17 - : Algorithme parallèle du produit matrice-vecteur par blocs.*

## 4.6 Les BLAS distribuées

Dans ce paragraphe, nous effectuons un tour d'horizon des projets de parallélisation des BLAS. Il faut noter que la plupart des algorithmes parallélisés sont les BLAS de niveau 3. En effet, leur grain étant important, les résultats en terme d'efficacité sont plus intéressants sur les machines à mémoire distribuée qui possèdent en général des processeurs très puissants. De plus, les derniers ordinateurs, parallèles ou non, possèdent des mémoires hiérarchisées. Au minimum, une machine a deux niveaux de mémoire. Le premier, appelé mémoire « cache » , se trouve à l'intérieur même du processeur, il permet aux unités de calcul de ce dernier un accès très rapide aux données. Le second niveau est souvent la mémoire principale de la machine dont l'accès se fait par « bus » . Donc, plus un programme utilisera la mémoire de niveau le plus élevé, plus il gagnera en vitesse et en efficacité. Pour les algorithmes d'algèbre linéaire manipulant des matrices pleines, la solution consiste à utiliser des BLAS de niveau 3 qui manipulent des blocs de matrices.

### 4.6.1 Le placement des données

Il existe essentiellement deux bibliothèques de procédures BLAS 2 et 3 parallèles : *Parallel Universal Matrix Multiplication Algorithms (PUMMA)* et *Parallel Block Basic Linear Algebra Subprograms (PB-BLAS)*. Elles sont basées sur une répartition des données par blocs, dite « dispersée » (ou « blocs cyclique » ) ; elles sont actuellement en cours de développement. La fonction de placement d'une matrice $m \times n$ en bloc cyclique de taille $l \times c$ sur une grille de processeurs $P \times Q$ est donnée par :

Le processeur d'indice $\{q, d\}$ contient dans sa mémoire locale les éléments de la matrice de coordonnées $(p + i * P,\ q + j * Q)$, où $i = 0, ..., \left\lfloor \frac{m-1-p}{P} \right\rfloor$ et $j = 0, ..., \left\lfloor \frac{n-1-p}{Q} \right\rfloor$.

**Remarque :** nous obtenons une décomposition de la matrice en $m_b \times n_b$ (avec $m_b \left\lceil \frac{m}{l} \right\rceil$ et $n_b \left\lceil \frac{n}{c} \right\rceil$) blocs de taille $l \times c$.

Un exemple avec une matrice $6 \times 6$ sur une grille de processeurs $2 \times 2$ est

illustré par les figures 4.18 et 4.19.

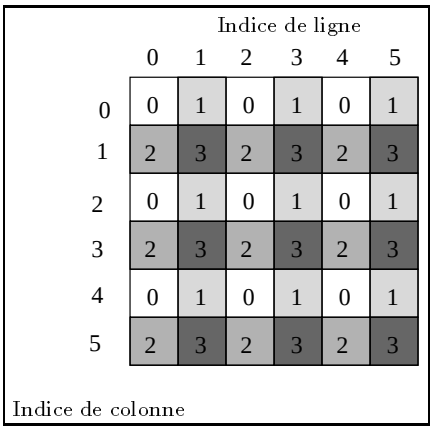



FIG. *4.18 - : Placement des éléments d'une matrice* $6 \times 6$ *sur une grille de* $2 \times 2$ *processeurs.*

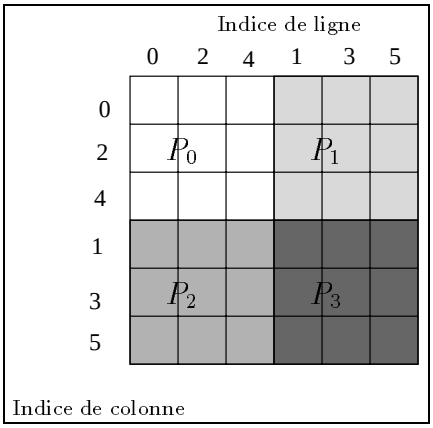



FIG. *4.19 - : Eléments affectés aux différents processeurs..*

Ce placement n'est pas optimal en terme de communication pour un produit matrice-matrice, mais il peut devenir intéressant si ce produit a lieu après d'une décomposition LU qui a besoin d'un tel placement pour être efficace.

### 4.6.2 PB-BLAS

La bibliothèque PB-BLAS (pour *Parallel Block Basic Linear Subroutines*) a été constituée par les premières fonctions BLAS de niveaux 2 et 3 destinées aux machines parallèles à mémoire distribuée. Les études effectuées pour le développement de cette bibliothèque ont permis la mise au point des fonctions BLACS et du placement cyclique par blocs.

### 4.6.3 PUMMA

La bibliothèque PUMMA (pour *Parallel Universal Matrix Multiplication Algorithms*) est une évolution de la bibliothèque PB-BLAS proposée par Choi, Dongarra et Walker [CDW93b]. Le mot *Universal* signifie que les auteurs désirent que les performances des fonctions de cette bibliothèque ne dépendent que très faiblement de la configuration des processeurs de la machine parallèle cible. La répartition des données « dispersées » par blocs est utilisée par toutes les fonctions BLAS 3 de cette bibliothèque.

La bibliothèque PUMMA reprend toutes les fonctions de la bibliothèque PB-BLAS, mais aussi de nouvelles fonctions utilisant des schémas de communications globales, comme par exemple la transposition. PUMMA va donc remplacer à terme la bibliothèque PB-BLAS et permettre aux constructeurs de machines parallèles d'en produire une version optimisée qui pourra être utilisée par la bibliothèque standard ScaLAPACK.

## 4.7 Conclusion

Nous avons présenté dans ce chapitre les méthodesles plus récentes de parallélisation d'algorithmes d'algèbre linéaire sur des machines à mémoire distribuée. La bibliothèque ScaLAPACK a été créée pour obtenir le maximum de performances tout en restant portable sur tous les ordinateurs du marché. La solution adoptée est l'utilisation d'un placement des données « dispersées » par blocs, qui permet de réduire en moyenne le coût des communications.

Cependant cette bibliothèque n'utilise pas les fonctionnalités des dernières machines parallèles, qui permettent de calculer et d'envoyer ou de recevoir des messages en parallèle. Le chapitre suivant est consacré à l'étude sur une BLAS de niveau 2 des possibilités de masquer les temps de communication par du temps de calcul.

# Chapitre 5

# Vers une programmation plus efficace

*Nous présentons, dans ce chapitre, l'étude et la mise en œuvre de méthodes de recouvrement du temps de communication d'un algorithme parallèle par le temps de calcul. Ce travail a été réalisé en collaboration avec Philippe Michallon et Denis Trystram.*

## 5.1 Introduction

Pour la plupart des applications, l'une des façons les plus simples pour obtenir un code parallèle est d'utiliser des bibliothèques de calculs comme ScaLAPACK et de communications comme PVM ou MPI (voir les chapitres 4.7 et 2.15). De plus, si le programmeur décide de paralléliser un code déjà existant, il est intéressant de pouvoir simplement remplacer l'appel à une fonction séquentielle par un appel à une fonction parallélisée calculant le même résultat. Sur les machines à mémoire distribuée, ces parallélisations sont effectuées en décomposant l'application en phases de calculs et de communications bloquantes. Malheureusement, ce type de programmation ne donne pas toujours de très bons résultats. En effet, les processeurs n'effectuent pas de calculs pendant les phases de communications, d'où la nécessité d'utiliser des bibliothèques de communications non-bloquantes. Mais dans certains cas il subsiste des problèmes, notamment quand le processeur émetteur a plus de calculs à effectuer que le processeur récepteur. Dans ce cas, le processeur émetteur peut, dès qu'il a calculé une certaine quantité de données, les envoyer au récepteur qui pourra commencer de nouveaux calculs. Ainsi, en enchaînant les envois, on arrive à recouvrir une partie du temps de communication par du temps de calcul.

Toutefois le choix de la taille des données à envoyer n'est pas simple. Ce choix va dépendre de la taille du problème, de la vitesse de calcul des processeurs et du coût des communications. Nous illustrerons la difficulté du calcul de la meilleure taille de paquets à envoyer à chaque étape sur un exemple simple tel que le produit

matrice vecteur sur un anneau de processeurs.

## 5.2 Pourquoi faire du recouvrement ?

Le temps d'exécution d'un programme sur une machine parallèle à mémoire distribuée est fonction du temps de calcul et du temps de communication. Si ce programme n'utilise pas les méthodes de recouvrement du temps de communication par du temps de calcul, alors le temps total d'exécution de l'algorithme parallèle est égal à la somme des temps de calcul et de communication. Avec la mise en œuvre de méthodes de recouvrement, on obtient un temps global d'exécution plus faible car, même si le temps de calcul n'a pas changé, le coût du temps de communication a diminué. Si nous observons le comportement d'un programme parallèle sans recouvrement avec un outil de visualisation, on constate que les processeurs perdent du temps à attendre des données, d'où l'idée de faire du calcul pendant que les processeurs communiquent, c'est-à-dire faire du recouvrement calcul/communication.

Afin de mieux comprendre la démarche qui a conduit King, Chu et Ni [KCN88] à proposer une solution enchaînée (*pipeline*) des envois de données, nous allons prendre un exemple, et analyser les problèmes qui se posent.

Nous allons considérer deux processeurs $P_1$ et $P_2$ sur lesquels s'exécute un programme parallèle. Chaque processeur doit effectuer deux calculs $T_i^j$, où $i$ est le numéro du processeur et $j$ le numéro du calcul, et une communication. Afin de simplifier le problème, nous considérerons que seul le deuxième calcul de l'un des processeurs a besoin des résultats du premier calcul s'exécutant sur l'autre processeur.

Si l'on considère des communications bloquantes, sur cet exemple nous obtenons le schéma d'exécution représenté figure 5.1.

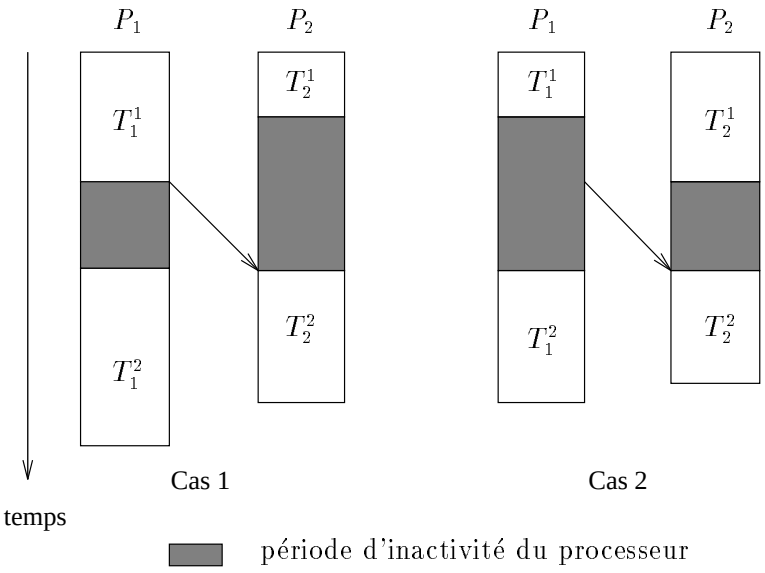



FIG. *5.1 - : Exécution avec des communications bloquantes.*

Dans le cas 1 le processeur $P_2$ a fini son premier calcul avant le processeur $P_1$, mais il est obligé d'attendre la fin de la phase de calcul $T_1^1$ de $P_1$ avant de recevoir des données, puis de recommencer une seconde phase de calcul $T_2^2$. Dans le cas 2 c'est le processeur $P_1$ qui ne peut pas envoyer ses données tant que le processeur $P_2$ n'a pas fini sa phase de calcul. Ceci se produit quand le modèle de communication est avec rendez-vous (*cf.* chapitre 2.15). Dans les deux cas, nous remarquons que l'un des processeurs attend que l'autre arrive dans une phase de communication pour envoyer ou recevoir un message, ce qui crée des périodes d'inactivité du processeur.

Afin de réduire ces périodes et de permettre aux utilisateurs de faire du recouvrement calcul/communication, des bibliothèques de communications non-bloquantes se sont développées ; par exemple PVM et MPI fournissent aussi bien les fonctions de communications bloquantes que non-bloquantes.

L'utilisation de ces bibliothèques permet de supprimer les périodes d'inactivité des processeurs $P_1$ et $P_2$ dans le cas 2. Le processeur $P_1$ peut commencer $T_1^2$ sans attendre la fin de la communication et le processeur $P_2$ peut donc enchaîner ses deux calculs, car il a reçu les données nécessaires au calcul $T_2^2$ pendant le calcul de $T_2^1$ (voir figure 5.2).

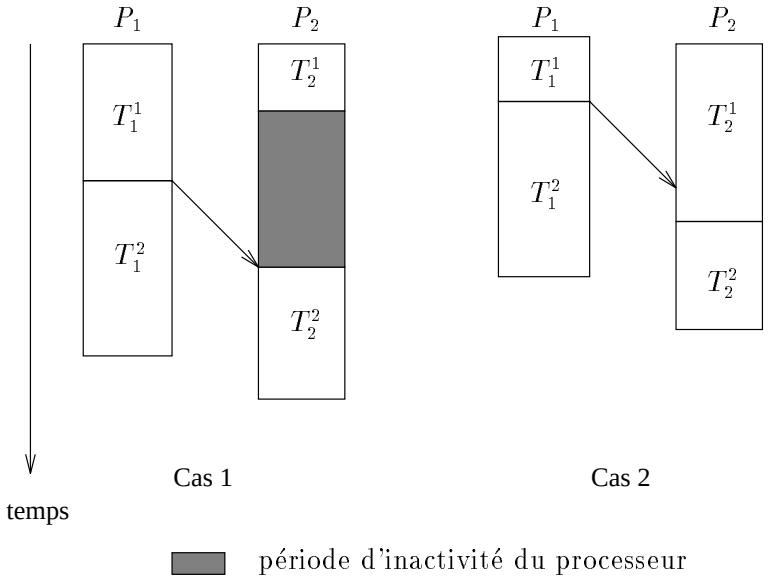



FIG. *5.2 - : Exécution avec des communications non-bloquantes.*

Mais l'utilisation de communications non-bloquantes ne permet pas d'effectuer le type de recouvrement présenté dans le cas 1 de la figure 5.1. En effet, dans ce cas le processeur $P_1$ ne peut pas envoyer son message avant d'avoir fini ses calculs. Mais le deuxième calcul $T_2^2$ a-t-il besoin de toutes les données de $T_1^1$ pour commencer? Si oui, il est clair que l'on ne pourra pas réaliser de recouvrement calcul/communication. Dans le cas contraire, il est possible de découper le calcul $T_1^1$ en $n$ calculs plus petits qui s'exécuteront les uns à la suite des autres. Ainsi, chaque petit calcul peut envoyer à $T_2^2$ les données qu'il vient de calculer et, de cette façon, $T_2^2$ peut commencer ses calculs avec les données qu'il vient de recevoir. Cette méthode de recouvrement est illustrée par la figure 5.3.

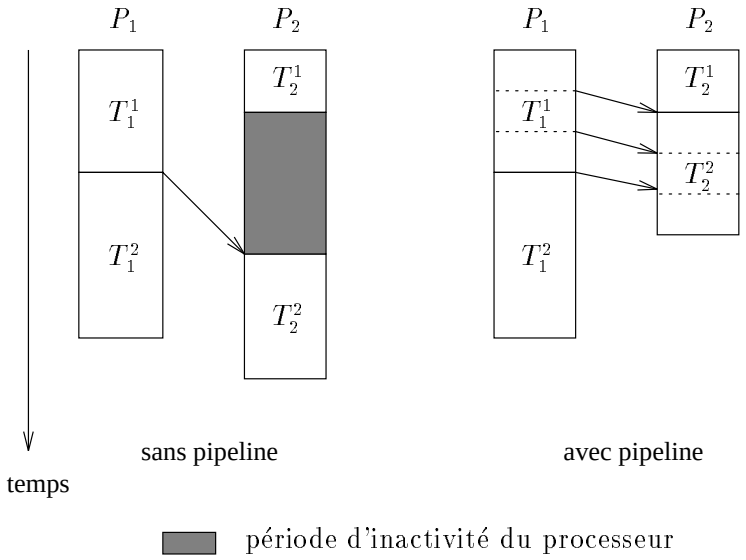



FIG. *5.3 - : Exécution avec des communications non-bloquantes pipelinées.*

De nombreux chercheurs ont étudié le problème du recouvrement calcul/communication, [CD93, DT93, KCN88] pour ne citer qu'eux. Mais personne n'a proposé d'outils faciles à utiliser pour le programmeur. Desprez et Tourancheau [Des94, DT93] travaillent sur une bibliothèque permettant de gérer les pipelines et d'exprimer plus facilement les programmes parallèles. Nous avons collaboré avec eux afin de rendre cet outil performant et facilement utilisable par tout programmeur spécialiste ou non [CCD+94b, CCD+94a]. Mais pour rendre cette bibliothèque plus efficace, il est nécessaire d'avoir un calcul automatique de la taille des messages à envoyer. L'idéal serait que cette taille soit calculée lors de la compilation ou se fasse de manière dynamique au cours de l'exécution de l'algorithme. Le problème qui se pose est de savoir comment découper ces données et quelle doit être la taille des envois afin que les deux processeurs n'aient pas de période d'inactivité. En effet, si les paquets sont trop gros, le recouvrement ne sera pas assez efficace, et au contraire si les paquets sont trop petits il va y avoir beaucoup trop d'initialisations de communications, ce qui nuira aux performances de l'algorithme. Comme nous allons le voir dans la section suivante sur un exemple simple et régulier, ce choix n'est pas si facile.

## 5.3 Le choix de la taille des paquets

L'exemple que nous avons choisi d'étudier pour illustrer le problème de la taille des paquets à échanger est le produit matrice-vecteur, sur une architecture à mémoire distribuée connectée en anneau de processeurs [CMT94]. Nous allons

montrer, en fonction des paramètres de la machine, comment calculer la taille des paquets et ainsi obtenir un algorithme avec un coût de communication le plus faible possible.

Nous nous sommes intéressés au produit matrice-vecteur, car c'est l'une des opérations les plus simples et les plus utilisées de l'algèbre linéaire [BT89, GL89]. Le problème consiste à calculer $v = Ax$ où $A$ est une matrice $(n, n)$ et $x$, $v$ deux vecteurs de dimension $n$. Chaque composante du vecteur $v$ est le résultat du produit scalaire suivant :

$$v[i] = \sum_{j=1}^{n} A[i,j]\ x[j]. \tag{5.1}$$

**Notations et modèle**

Si l'on considère un anneau de $p$ processeurs (numérotés de 1 à $p$), il est possible de réaliser une répartition équilibrée de la matrice $A$ en $p$ blocs de $\lceil \frac{n}{p} \rceil$ ou $\lfloor \frac{n}{p} \rfloor$ lignes. Le type d'allocation par lignes (consécutives, cyclique, *wrap*) choisi n'influe pas sur l'étude qui va suivre [Joh87]. De même, on effectue une répartition équilibrée du vecteur $x$ (chaque processeur possède $\lceil \frac{n}{p} \rceil$ ou $\lfloor \frac{n}{p} \rfloor$ composantes du vecteur). De façon à simplifier les formules, nous allons considérer une allocation de la matrice $A$ par lignes avec $\frac{n}{p}$ entier.

Nous supposons également que les liens de communication sont *half-duplex* [SS89] et que les processeurs sont en mode 2−ports [SS89, dR94]. De plus, un processeur peut réaliser simultanément des communications et du calcul. L'échange de messages entre deux processeurs est réalisé par commutation de messages (ou *store-and-forward*) [JH89, SS89, SW90] ; le temps d'un tel échange est modélisé par : $\beta_c + L\tau_c$.

Le coût d'une opération arithmétique élémentaire (multiplication suivie d'une addition) est donné par : $\tau_a$.

Après une présentation de l'implémentation standard du produit matrice-vecteur, un principe de recouvrement est donné. Nous étudierons alors le coût de ce nouvel algorithme et étendrons le modèle aux liens *full-duplex* [SS89].

### 5.3.1 Analyse du produit matrice-vecteur

Comme nous l'avons présenté dans le chapitre 4.7, la méthode pour paralléliser une application numérique algébrique consiste à identifier le découpage des données (décomposition de matrices par lignes, colonnes ou blocs), puis à organiser les mouvements de données en utilisant des bibliothèques de communications.

### La solution standard

L'équation 5.1 montre que le calcul de chaque composante du vecteur $v$ requiert la connaissance de tous les éléments du vecteur $x$, donc chaque processeur doit envoyer ses $\frac{n}{p}$ composantes du vecteur $x$ à tous les autres processeurs. Ce type de mouvement de données correspond à un échange total, plus souvent appelé ATA (pour *All To All*) [SS89]. L'algorithme du produit matrice-vecteur est le suivant :

**Algorithme 5.1** *produit matrice-vecteur standard*
*ATA de taille* $\frac{n}{p}$ *(partie du vecteur x);*
*Faire en parallèle sur chaque processeur*
*Calcul de* $\frac{n}{p}$ *produits scalaires de taille n (lignes de A et vecteur x);*

L'échange total de données de taille $\frac{n}{p}$ coûte $(p-1)(\beta_c + \frac{n}{p}\tau_c)$ unités de temps [SS89]. Son algorithme sur un anneau de processeurs possédant des liens *half-duplex* est le suivant :

**Algorithme 5.2** *ATA (Algorithme s'exécutant sur le processeur q)*
*Pour* $t \leftarrow 1$ *à* $p-1$ *faire en parallèle*
*Envoyer au processeur (q+1 modulo p) les* $\frac{n}{p}$ *composantes de x nécessaires au calcul suivant;*
*Recevoir de (q-1 modulo p) les nouvelles* $\frac{n}{p}$ *composantes de x;*

Le calcul des $\frac{n}{p}$ produits scalaires de taille $n$ nécessite $\frac{n^2}{p}\tau_a$ unités de temps. Ainsi l'algorithme standard du produit matrice-vecteur sur un anneau possédant des liens *half-duplex* a un coût de :

$$T_{stand} = \frac{n^2}{p}\tau_a + (p-1)\left(\beta_c + \frac{n}{p}\tau_c\right) \tag{5.2}$$

Sans recouvrement calcul/communication, cet algorithme est optimal.

### Recouvrement des communications

Afin de réduire le temps d'exécution, la solution consiste à masquer au maximum les communications par du calcul.

Durant l'échange total de l'algorithme précédent, il est facile de voir que les processeurs perdent du temps, puisque tous les processeurs peuvent commencer les calculs sur leurs données initiales pendant l'échange total. Mais le temps d'un échange total sur un anneau est en général supérieur à $(\frac{n}{p})^2$ opérations flottantes. Une solution consiste à décomposer l'échange total en une série de communications de voisin à voisin, de façon à masquer le plus possible les communications. En effet, à une étape donnée, un processeur peut recevoir de nouvelles données alors qu'il calcule avec les données reçues à l'étape précédente.

### Principe de la méthode

Au début de l'algorithme, chaque processeur possède $\frac{n}{p}$ lignes de la matrice $A$ et $\frac{n}{p}$ composantes du vecteur $x$, il peut donc calculer $\frac{n}{p}$ produits scalaires partiels $v_i$. Ainsi, un processeur peut exécuter $(\frac{n}{p})^2$ opérations flottantes et recevoir simultanément au plus $\frac{n}{p}$ données de ses voisins. En fonction des paramètres de la machine ($p, \beta_c, \tau_c$ and $\tau_a$) et de la taille du problème, nous pouvons observer deux cas :

- si $\beta_c + \frac{n}{p}\tau_c \leq \frac{n^2}{p^2}\tau_a$ alors il y a recouvrement total ;
- si $\beta_c + \frac{n}{p}\tau_c > \frac{n^2}{p^2}\tau_a$ alors il y a recouvrement partiel.

Dans le premier cas, comme le temps de communication des $\frac{n}{p}$ éléments est inférieur au temps d'un calcul local, toutes les communications peuvent être masquées.

Dans le deuxième cas, le temps des calculs locaux correspond à une communication d'un nombre de données plus petit que $\frac{n}{p}$. Soit $\lambda_1$ le plus grand entier tel que $\beta_c + \lambda_1\tau_c \leq \frac{n^2}{p^2}\tau_a$, alors toutes les communications de taille $\lambda_1$ peuvent être masquées ($\lambda_1 < \frac{n}{p}$). Après ces communications, chaque processeur a $\lambda_1$ nouvelles opérations locales à exécuter. A l'étape suivante, on peut calculer une nouvelle taille de message $\lambda_2$ telle que $\beta_c + \lambda_2\tau_c \leq \lambda_1\tau_a$, ce qui permet de masquer des communications de taille $\lambda_2$. On itère ce principe jusqu'à ce qu'il ne soit plus possible de réaliser des communications pendant un calcul. Cela permet de construire une série de tailles de messages $\lambda_t$ ($t = 0, ...N$) de $\lambda_0 = \frac{n}{p}$ à $\lambda_{N+1} = 0$ telles que l'on puisse effectuer un maximum d'opérations locales, tout en masquant au mieux les communications.

## 5.3.2 Description des algorithmes

### Recouvrement total ($\beta_c + \frac{n}{p}\tau_c \leq \frac{n^2}{p^2}\tau_a$)

Dans ce cas toutes les communications peuvent être masquées. L'algorithme parallèle du produit matrice-vecteur devient :

**Algorithme 5.3** *produit matrice-vecteur parallèle avec recouvrement total*
*/* Algorithme du processeur q avec q=1...p */*
*Pour* $t \leftarrow 1$ *à* $p-1$ *faire en parallèle*
*Envoyer à (q+1 modulo p) les* $\frac{n}{p}$ *éléments de x reçus à l'étape précédente;*
*Calculer les* $\frac{n}{p}$ *produits scalaires locaux;*
*Recevoir de (q-1 modulo p) les nouveaux* $\frac{n}{p}$ *éléments de x;*
*Calcul des* $\frac{n}{p}$ *derniers produits scalaires;*

Il y a $(p-1)$ étapes où les communications sont totalement masquées par les calculs locaux d'un coût de $\frac{n^2}{p^2}\tau_a$, puis une dernière étape de calculs locaux sans communication. Cet algorithme a un coût total de $\frac{n^2}{p}\tau_a$.

**Recouvrement partiel** ($\beta_c + \frac{n}{p}\tau_c > \frac{n^2}{p^2}\tau_a$)

- **Phase I: recouvrement total**

Dans ce cas, à l'étape $t$, un processeur calcule $\lambda_{t-1}\frac{n}{p}$ et reçoit (envoie) $\lambda_t$ données. La valeur de $\lambda_t$ est donnée par la formule suivante:

$$\lambda_t = \left\lfloor \frac{n}{p}\frac{\tau_a}{\tau_c}\lambda_{t-1} - \frac{\beta_c}{\tau_c} \right\rfloor \tag{5.3}$$

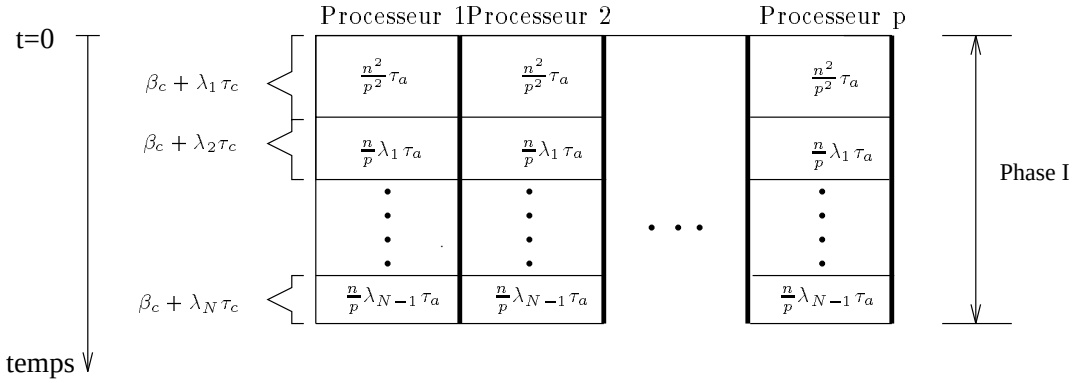



FIG. *5.4 - : Schéma d'exécution de la Phase I.*

**Proposition 5** *La série des $\lambda_t$ est strictement décroissante.*

*Preuve*: par récurrence sur $t$.
Comme $\beta_c + \frac{n}{p}\tau_c > \frac{}{n^2}p^2\tau_a$, il est évident que $\lambda_1 < \frac{n}{p}$. Nous pouvons donc écrire que:

$$\lambda_2 = \lfloor \frac{n}{p}\frac{\tau_a}{\tau_c}\lambda_1 - \frac{\beta_c}{\tau_c} \rfloor$$

alors

$$\lambda_2 \leq \lambda_{over} = \frac{n}{p}\frac{\tau_a}{\tau_c}\lambda_1 - \frac{\beta_c}{\tau_c}.$$

De plus $\frac{n}{p}\frac{\tau_a}{\tau_c} < 1 + \frac{p}{n}\frac{\beta_c}{\tau_c}$ car $\beta_c + \frac{n}{p}\tau_c > \frac{}{n^2}p^2\tau_a$,
d'où

$$\lambda_{over} < (1 + \frac{p}{n}\frac{\beta_c}{\tau_c})\lambda_1 - \frac{\beta_c}{\tau_c}.$$

Donc

$$\lambda_{over} < \lambda_1 + \frac{\beta_c}{\tau_c}(\frac{p}{n}\lambda_1 - 1),$$

mais $\frac{p}{n}\lambda_1 - 1 < 0$ car $\lambda_1 < \frac{n}{p}$, donc $\lambda_2 < \lambda_1$.

La propriété est vraie pour $\lambda_1$ et $\lambda_2$. Nous allons supposer que la série des $\lambda_t$ est strictement décroissante jusqu'au rang $t$ et nous allons prouver que la propriété est toujours vraie au rang $t+1$. Nous avons :

$$\lambda_{t+1} = \lfloor \frac{n}{p}\frac{\tau_a}{\tau_c}\lambda_t - \frac{\beta_c}{\tau_c} \rfloor$$

alors

$$\lambda_{t+1} \leq \lambda_{over} = \frac{n}{p}\frac{\tau_a}{\tau_c}\lambda_t - \frac{\beta_c}{\tau_c},$$

d'où

$$\lambda_{over} < \lambda_t + \frac{\beta_c}{\tau_c}(\frac{p}{n}\lambda_t - 1).$$

Mais $\frac{p}{n}\lambda_t - 1 < 0$ car $\lambda_t < \lambda_{t-1} < ... < \frac{n}{p}$, donc $\lambda_{t+1} < \lambda_t$. □

Pendant les $N$ premières étapes, les communications sont totalement masquées. Ensuite il ne reste plus assez de données locales pour effectuer des calculs permettant de masquer une communication. Par la suite nous nommerons Phase I l'exécution des $N$ premières étapes (voir figure 5.4).

Après avoir calculé au préalable la série des $\lambda_t$, l'algorithme de la Phase I est le suivant :

**Algorithme 5.4** *Phase I*

*/* algorithme du processeur q avec q=1...p */*
$t \leftarrow 1$;
*Tant que* $\lambda[t] \geq 1$ *faire en parallèle*
  *Envoyer au processeur (q+1 modulo p) les* $\lambda[t]$ *éléments de x reçus à l'étape précédente;*
  *Calculer les* $\lambda[t-1]$ *produits scalaires locaux;*
  *Recevoir du processeur (q-1 modulo p) les nouveaux* $\lambda[t]$ *éléments de x;*
  $t \leftarrow t+1$;
*Fin tant que*

- **Le reste des calculs**

Après la Phase I, il reste $\lambda_N$ calcul locaux à effectuer en parallèle sur chaque processeur. Pendant ces calculs, chaque processeur peut communiquer le maximum de données (c'est-à-dire $\frac{n}{p}$) à son voisin. Ainsi, il est possible de redémarrer une nouvelle phase de recouvrement total (voir figure 5.5 où les périodes d'inactivité des processeurs sont représentées en gris).

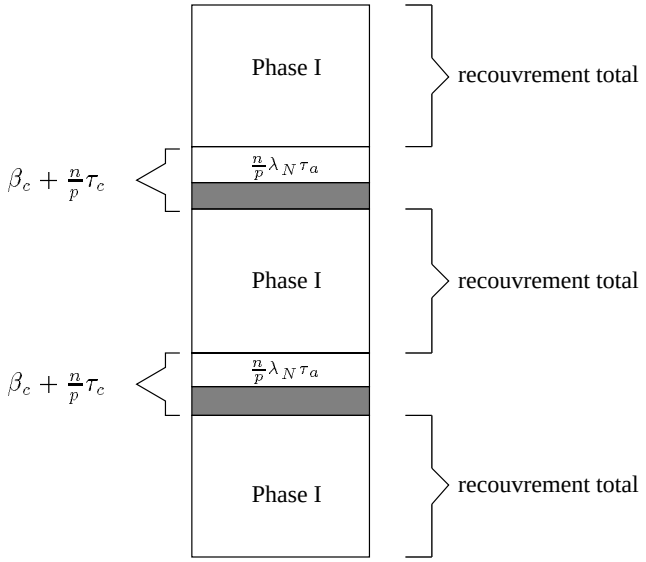



FIG. 5.5 - : *Phases successives.*

Nous répétons $\left\lceil \frac{n}{\sum_{i=0}^{N-1}\lambda_i} \right\rceil$ fois la succession de Phase I.
L'algorithme de recouvrement partiel est donc :

**Algorithme 5.5** *Les phases successives*
*/* algorithme du processeur q avec q=1...p */*
*Exécuter la Phase I;*
*Pour* $k \leftarrow 1$ *à* $\left\lceil \frac{n}{\sum_{i=0}^{N-1}\lambda_i} \right\rceil$
*Faire en parallèle*
*Envoyer au processeur (q+1 modulo p) les* $\frac{n}{p}$ *éléments de x reçus à l'étape précédente;*
*Calculer les* $\lambda[N]$ *produits scalaires locaux;*
*Recevoir du processeur (q-1 modulo p) les nouveaux* $\frac{n}{p}$ *éléments de x;*
*Fin faire en parallèle;*
*Exécuter la Phase I;*
*Fin faire;*

**Remarque 5** *Dans le pire des cas, la dernière exécution de la Phase I se termine avant la* $N^{me}$ *étape. Pour en tenir compte, avant d'exécuter la dernière phase de recouvrement il est possible de trouver un* $\lambda_t$ *plus petit que* $\frac{n}{p}$ *afin de minimiser le temps d'inactivité des processeurs, mais ceci est très difficile à réaliser automatiquement. La solution la plus simple consiste à tester pendant la Phase I le nombre de produits scalaire locaux, et de stopper l'algorithme lorsque tous les produits scalaires ont été calculés. Ceci peut être réalisé en ajoutant un test dans l'algorithme 5.4 :*

$$Tantque\ ((k \sum_{i=0}^{N-1} \lambda_i + \sum_{i=0}^{t-1} \lambda_i < n)\ et\ (\lambda_t \geq 1))\ faire\ en\ parallèle \qquad (5.4)$$

### Evolution de l'allocation des données

Comme à chaque étape tous les processeurs exécutent la même tâche, n'importe quel processeur peut être choisi comme référence. Notons $PR$ ce processeur.

A une étape **t** un processeur exécute $\lambda_{t-1}\frac{n}{p}$ opérations, envoie et reçoit $\lambda_t$ éléments du vecteur $x$.

Nous rappelons que la série des $\lambda_t$ décroît et que la Phase I s'arrête à l'étape $N$. En fait à cette étape $\lambda_{N+1}$ est plus petit que 1. De plus, à cette étape le processeur $PR$ n'a pas encore reçu toutes ses données. Pour déterminer le temps d'exécution de l'algorithme, il est nécessaire de connaître la distance entre $PR$ et le processeur le plus éloigné qui possède encore des données destinées à $PR$, ainsi que le nombre d'éléments contenus dans ce processeur. A chaque étape $t$ de l'algorithme, nous pouvons calculer la distance entre $PR$ et ce processeur. Nous noterons $d_t$ cette distance, et $\lambda_c(t)$ le nombre de composantes du vecteur $x$ nécessaire aux calculs locaux de $PR$ qui se trouvent sur ce processeur (processeur à distance $d_t$ de $PR$). La proposition 6 donne le nombre d'éléments destinés à $PR$ restant sur chaque processeurs.

Dans un échange total, le processeur $PR$ doit recevoir $(p\ -\ 1)$ messages de taille $\frac{n}{p}$, c'est-à-dire un message de taille $\frac{n}{p}$ à chaque étape, et donc $d_{t+1}\ =\ d_t - 1$. Mais dans notre algorithme à chaque étape $PR$ reçoit un message de taille $\lambda_t$, ce qui complique l'expression de $d_t$, qui est donnée par la proposition 7. Cette valeur nous permettra de calculer le temps final de l'algorithme.

**Proposition 6**
*Les processeurs situés à une distance supérieure à $d_t$ ne possèdent pas d'éléments destinés à PR.*
*Le processeur situé à une distance $d_t$ possède $\lambda_c(t)$ éléments destinés à PR.*
*Les processeurs situés à une distance inférieure à $d_t$ possèdent $\frac{n}{p}$ éléments destinés à PR.*

*Preuve*:

On peut facilement vérifier que la proposition est vraie pour l'étape 1, lorsque tous les processeurs envoient $\lambda_1$ éléments.

Supposons que cette proposition soit vraie pour l'étape $t \geq 1$, montrons qu'elle est aussi vraie pour l'étape $t+1$.

Nous avons deux cas à considérer: $\lambda_{t+1}\ <\ \lambda_c(t)$ ou $\lambda_{t+1}\ \geq\ \lambda_c(t)$.

• si $\lambda_{t+1}\ <\ \lambda_c(t)$ alors nous avons $d_{t+1} = d_t$ et $\lambda_c(t+1)\ \leftarrow\ \lambda_c(t)\ -\ \lambda_{t+1}$:

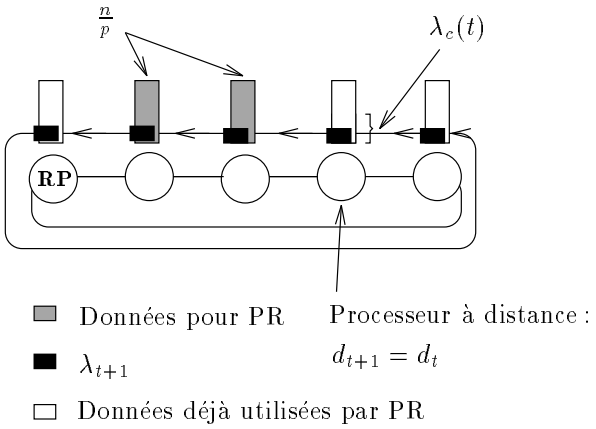



FIG. 5.6 - : *Cas :* $\lambda_{t+1} < \lambda_c(t)$.

Les processeurs à distance plus petite que $d_t$ envoient $\lambda_{t+1}$ et reçoivent $\lambda_{t+1}$ données utiles à $PR$, ainsi ils possèdent toujours $\frac{n}{p}$ données destinées à $PR$. En utilisant le même argument on prouve que les processeurs situés à une distance supérieure à $d_t$ ne possèdent plus de données destinées à $PR$ (voir figure 5.6). Le processeur situé à la distance $d_t$ envoie $\lambda_{t+1}$ données destinées à $PR$ et il ne reçoit pas de données. Donc il reste $\lambda_c(t+1) \leftarrow \lambda_c(t) - \lambda_{t+1}$ données sur ce processeur. Dans ce cas on a $d_{t+1} = d_t$.

- si $\lambda_{t+1} \geq \lambda_c(t)$ alors nous avons $d_{t+1} = d_t - 1$:

Le processeur à distance $d_{t+1}$ reçoit $\lambda_c(t)$ éléments de $x$ nécessaire à $PR$ et $\lambda_{t+1} - \lambda_c(t)$ ont déjà été utilisées par $PR$.

Ce processeur dispose maintenant de $\lambda_c(t+1) = \frac{n}{p} + \lambda_c(t) - \lambda_{t+1}$ données et il est facile de voir d'une part que les processeurs à distance strictement inférieure à $d_{t+1}$ possèdent $\frac{n}{p}$ éléments et d'autre part que ceux à distance strictement supérieure à $d_{t+1}$ ne possèdent plus d'éléments destinés à $PR$ (voir figure 5.7).

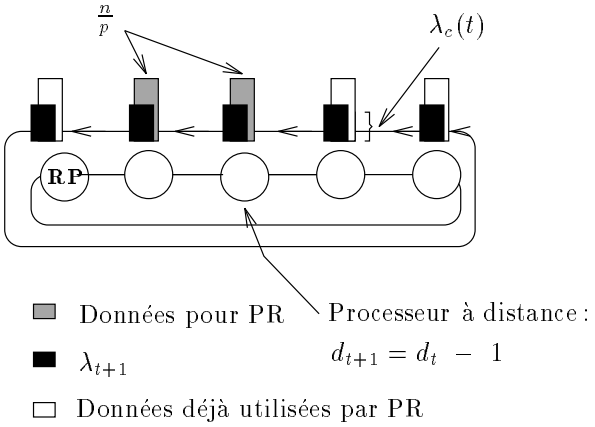



FIG. 5.7 - : *Cas :* $\lambda_{t+1} \geq \lambda_c(t)$.

□

**Proposition 7** *La distance $d_t$ est égale à :*

$$d_t = p - 1 - \left\lfloor \frac{p}{n} \sum_{i=1}^{t} \lambda_i \right\rfloor \tag{5.5}$$

*Preuve*:

Si on considère la première étape, la formule est facile à vérifier. On l'admet jusqu'au rang $t$ et on la montre pour le rang $t+1$.

Quand $PR$ reçoit un message de taille $\lambda_{t+1}$, le nombre de messages de taille $\frac{n}{p}$ déjà reçus augmente et $d_{t+1} = d_t - 1$, ou alors le nombre de messages reçus de taille $\frac{n}{p}$ reste le même et dans ce cas $d_{t+1} = d_t$. Nous devons donc considérer deux cas :

- cas $\lambda_{t+1} < \lambda_c(t)$. Nous avons $\left\lfloor \frac{p}{n} \sum_{i=1}^{t+1} \lambda_i \right\rfloor = \left\lfloor \frac{p}{n} \sum_{i=1}^{t} \lambda_i \right\rfloor$.

  Alors $d_{t+1} = d_t = p - 1 - \left\lfloor \frac{p}{n} \sum_{i=1}^{t} \lambda_i \right\rfloor = p - 1 - \left\lfloor \frac{p}{n} \sum_{i=1}^{t+1} \lambda_i \right\rfloor$.

- cas $\lambda_{t+1} \geq \lambda_c(t)$. Nous avons $\left\lfloor \frac{p}{n} \sum_{i=1}^{t+1} \lambda_i \right\rfloor = 1 + \left\lfloor \frac{p}{n} \sum_{i=1}^{t} \lambda_i \right\rfloor$.

  Alors $d_{t+1} = d_t - 1 = p - 1 - \left\lfloor \frac{p}{n} \sum_{i=1}^{t} \lambda_i \right\rfloor - 1 = p - 1 - \left\lfloor \frac{p}{n} \sum_{i=1}^{t+1} \lambda_i \right\rfloor$.

□

La proposition suivante nous permet de connaître la valeur de $\lambda_c(t)$ à chaque étape $t$.

**Proposition 8** *A une étape* $\boldsymbol{t}$*, la valeur de $\lambda_c(t)$ est égale à $\frac{n}{p} - (\sum_{i=1}^{t} \lambda_i) \bmod \frac{n}{p}$.*

*Preuve*:par récurrence sur t.

Si nous considérons l'étape 1, tous les processeurs envoient $\lambda_1$ éléments de $x$ et masquent le coût $\beta_c + \lambda_1\tau_c$ des communications par des calculs. Le processeur à distance $d_t = d_1$ possède alors $\lambda_c(1) = \frac{n}{p} - \lambda_1$ données destinées à $PR$ ; or $\lambda_1 < \frac{n}{p}$ donc on peut écrire $\lambda_c(1) = \frac{n}{p} - (\lambda_1 \ mod \frac{n}{p})$.

La formule est vérifiée pour l'étape 1, supposons-la vraie à l'étape $t$ et prouvons qu'elle est vraie à l'étape *t+1* quand nous envoyons $\lambda_{t+1}$ éléments.

- Si $\lambda_{t+1} < \lambda_c(t)$,
  nous savons que $d_{t+1} = d_t$ et le nouveau $\lambda_c(t+1)$ vaut $\lambda_c(t) - \lambda_{t+1}$,
  donc
  $$\lambda_c(t+1) = \frac{n}{p} - \left(\sum_{i=1}^{t} \lambda_i\right) \ mod \ \frac{n}{p} - \lambda_{t+1} mod \ \frac{n}{p}.$$
  Alors la valeur de $\lambda_c$ à l'étape *t+1* est :
  $$\lambda_c(t+1) = \frac{n}{p} - \left(\sum_{i=1}^{t+1} \lambda_i\right) \ mod \ \frac{n}{p}$$

- Si $\lambda_{t+1} \geq \lambda_c(t)$,
  nous savons que $d_{t+1} = d_t - 1$ et le nouveau $\lambda_c(t+1)$ vaut $\frac{n}{p} + \lambda_c(t) - \lambda_{t+1}$,
  donc
  $$\lambda_c(t+1) = \frac{n}{p} + \frac{n}{p} - \left(\left(\sum_{i=1}^{t} \lambda_i\right) \ mod \ \frac{n}{p} + \lambda_{t+1}\right).$$
  Ainsi
  $$\lambda_c(t+1) = \frac{n}{p} + \frac{n}{p} - \left(\frac{n}{p} + \left(\sum_{i=1}^{t+1} \lambda_i\right) \ mod \ \frac{n}{p}\right) = \frac{n}{p} - \left(\sum_{i=1}^{t+1} \lambda_i\right) \ mod \ \frac{n}{p}. \quad (5.6)$$
  En utilisant la formule suivante dans l'équation 5.6, on démontre la proposition :
  $$\left(\sum_{n=1}^{t} \lambda_i\right) \ mod \ \frac{n}{p} + \lambda_{t+1} = \frac{n}{p} + \left(\sum_{i=1}^{t+1} \lambda_i\right) \ mod \ \frac{n}{p}.$$

□

### 5.3.3 Calcul du temps d'exécution

Après la Phase I, les processeurs doivent envoyer des données sans recouvrement. Afin de terminer le calcul du vecteur $v$, l'algorithme consiste à répéter tant que cela est nécessaire la phase I (voir 5.3.2), d'où le temps final de :

$$T_{exec} = \frac{n^2}{p}\tau_a + \left\lfloor \frac{n}{\frac{n}{p} + \sum_{i=1}^{N-1} \lambda_i} \right\rfloor (\beta_c + \frac{n}{p}\tau_c - \lambda_N \frac{n}{p}\tau_a). \quad (5.7)$$

où $\left\lfloor \frac{n}{\frac{n}{p}+\sum_{i=1}^{N-1} \lambda_i} \right\rfloor$ représente le nombre de communications $(\beta_c + \frac{n}{p}\tau_c)$ non masquées nécessaires à la réinitialisation des Phases I.

### 5.3.4 Adaptation au lien *full-duplex*

Nous allons maintenant considérer qu'un processeur peut envoyer et recevoir en parallèle sur chacun de ses liens et allons réaliser la même étude que précédemment.

On constate que dans ce cas un processeur reçoit deux fois plus de données : pendant le temps $\frac{n^2}{p^2}\tau_a$ correspondant aux calculs locaux, un processeur peut envoyer $\lambda_1 = \left\lfloor \frac{n^2}{p^2}\frac{\tau_a}{\tau_c} - \frac{\beta_c}{\tau_c} \right\rfloor$ à ses deux voisins et recevoir $2\lambda_1$ éléments du vecteur $x$. Ainsi nous obtenons deux cas:

- $2\lambda_1 \geq \frac{n}{p}$

  A l'étape suivante, chaque processeur a plus de calculs locaux à exécuter, et peut envoyer un nombre d'éléments de $x$ supérieur à $\lambda_1$. A la prochaine étape chaque processeur pourra donc envoyer à son voisin de droite (resp. gauche) les données reçues de son voisin de gauche (resp. droite) à l'étape précédente, ainsi qu'une partie des données initiales restantes. On notera toutefois qu'un processeur ne peut pas envoyer plus de $\frac{n}{p}$ données dans la même direction.

  Le nombre d'éléments de $x$ qui doivent être envoyés à l'étape $\mathbf{t}$ est donné par : $\lambda_t = min\left(\left\lfloor \frac{n}{p}\frac{\tau_a}{\tau_c} 2\lambda_{t-1} - \frac{\beta_c}{\tau_c} \right\rfloor, \frac{n}{p}\right)$. On constate que cette série de $\lambda_i$ croît et atteint la borne supérieure $\frac{n}{p}$. Dans ce cas, toutes les communications sont masquées (voir algorithme 5.6).

  **Algorithme 5.6**

  ** algorithme du processeur q avec q=1...p **
  $\lambda[0] \leftarrow \frac{n}{p}$;
  $\lambda[1] \leftarrow \left\lfloor \frac{n}{p}\frac{\tau_a}{\tau_c}\frac{n}{p} - \frac{\beta_c}{\tau_c} \right\rfloor$;
  $t \leftarrow 1$;
  *Tant que* $(\frac{n}{p} + 2\sum_{i=1}^{t-1} \lambda_i \leq n)$ *faire en parallèle*
  *Envoyer au processeur (q+1 modulo p) les* $\lambda[t]$ *éléments de x utilisés par le processeur (q-1 modulo p);*
  *Envoyer au processeur (q-1 modulo p) les* $\lambda[t]$ *éléments de x utilisés par le processeur (q+1 modulo p);*

*Si (t=1) alors calculer les* $\lambda[t-1]$ *produits scalaires sinon calculer les* $2\lambda[t-1]$ *produits scalaires;*
*Recevoir du processeur (q+1 modulo p) et du processeur (q-1 modulo p)*
*les nouveaux* $\lambda[t]$ *éléments de x;*
$t \leftarrow t+1$;
$\lambda[t] \leftarrow min\left(\left\lfloor \frac{n}{p}\frac{\tau_a}{\tau_c}2\lambda[t-1] \ - \ \frac{\beta_c}{\tau_c}\right\rfloor , \ \frac{n}{p}\right)$;
*Fin tant que;*

Le temps total d'exécution de cet algorithme est :

$$T_{exec} = \frac{n^2}{p}\tau_a.$$

- $2\lambda_1 < \frac{n}{p}$

Dans ce cas la série des $\lambda_t = \left\lfloor \frac{n}{p}\frac{\tau_a}{\tau_c}2\lambda_{t-1} - \frac{\beta_c}{\tau_c}\right\rfloor$ est strictement décroissante. Nous ne pouvons pas avoir un recouvrement total, mais une succession de Phases I entrecoupées de communications d'un coût de $\beta_c + \frac{n}{p}\tau_c$ permettant leur réinitialisation. Après la première réinitialisation, chaque processeur doit effectuer $2\frac{n^2}{p^2}$ calculs locaux (voir figure 5.8). Ainsi il est possible de commencer une nouvelle Phase I avec $\lambda_1^{'} = min\left(\left\lfloor 2\frac{n^2}{p^2}\frac{\tau_a}{\tau_c} \ - \ \frac{\beta_c}{\tau_c}\right\rfloor , \frac{n}{p}\right)$ plus grand que $\lambda_1 = \left\lfloor \frac{n^2}{p^2}\frac{\tau_a}{\tau_c} \ - \ \frac{\beta_c}{\tau_c}\right\rfloor$. La nouvelle série de $\lambda_t^{'}$ est définie par : $\lambda_t^{'} = min\left(\left\lfloor \frac{n}{p}\frac{\tau_a}{\tau_c}2\lambda_{t-1}^{'} \ - \ \frac{\beta_c}{\tau_c}\right\rfloor , \frac{n}{p}\right)$.

Du fait des coûts des communications et des calculs locaux, il faut distinguer deux cas.

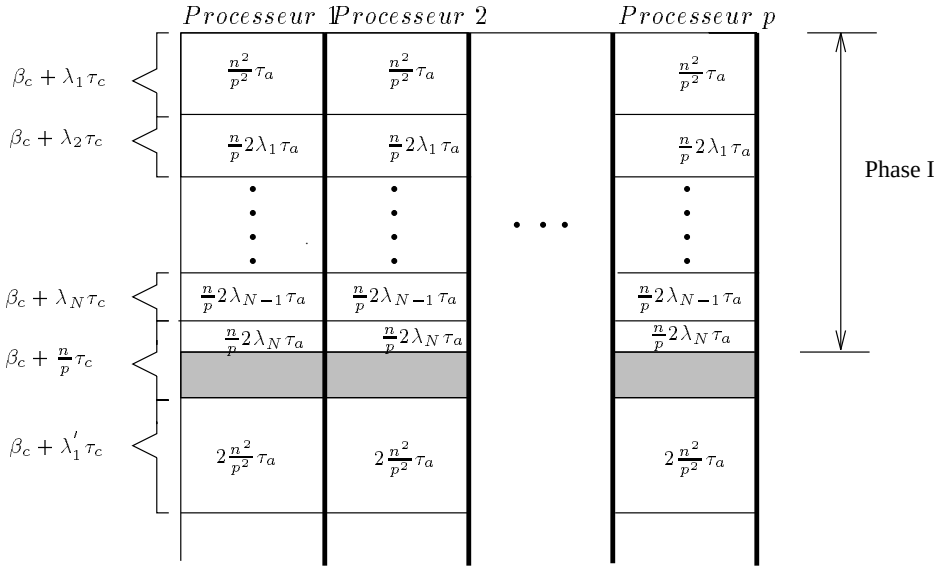



FIG. 5.8 - : *Le nouvel enchaînement des phases.*

- Si $\beta_c + \frac{n}{p}\tau_c \leq 2\frac{n^2}{p^2}\tau_a$ alors à chaque étape un processeur peut calculer $2\frac{n}{p}$ produits scalaires et recevoir $2\frac{n}{p}$ éléments. Dans ce cas le temps total d'exécution est :

$$T_{exec} = \left(\frac{n^2}{p} - \frac{n}{p}\lambda_N\right)\tau_a + \left(\beta_c + \frac{n}{p}\tau_c\right)$$

- Si $\beta_c + \frac{n}{p}\tau_c > 2\frac{n^2}{p^2}\tau_a$ alors la série des $\lambda_i'$ est strictement décroissante, et nous avons une succession de Phase I, entrecoupées de communications ayant un coût de $\beta_c + \frac{n}{p}\tau_c$. Le temps total d'exécution est :

$$T_{exec} = \frac{n^2}{p}\tau_a + \left\lfloor \frac{n - \sum_{i=0}^{N} \lambda_i}{\sum_{i=0}^{N'} \lambda_i'} \right\rfloor \left(\beta_c + \frac{n}{p}\tau_c\right)$$

### 5.3.5 Simulations

Après avoir réalisé une étude théorique, nous allons présenter des résultats obtenus par simulation dans le cas des liens *half-duplex* et *full-duplex*. Les paramètres utilisés correspondent au coût des communications sur les nouvelles machines parallèles avec des processeurs puissants et une faible latence.

Il est évident que le choix de l'algorithme utilisé va dépendre des paramètres de la machine: $P$, $\beta_c$, $\tau_c$ et $\tau_a$, mais aussi de la taille $n$ de la matrice $A$. Pour choisir l'algorithme, il faut simplement calculer le temps d'exécution théorique de chaque algorithme et utiliser l'algorithme correspondant au temps le plus

faible. Par exemple, pour une machine parallèle constituée de 100 processeurs, l'algorithme utilisant l'échange de message de taille $\lambda_i$ sera le plus performant si $\beta_c < 14$, $0.1 < \tau_c < 1.2$ et $\tau_a < 0.05$.

Pour une première simulation, nous considérons un nombre $P$ de processeurs égal à 100, avec $\beta_c = 10\mu s$, $\tau_c = 1\mu s/octet$ et $\tau_a = 0.01\mu s$.

La figure 5.9 compare l'algorithme sans recouvrement avec le nouvel algorithme présenté précédemment sur un anneau en utilisant des liens *half-duplex*.

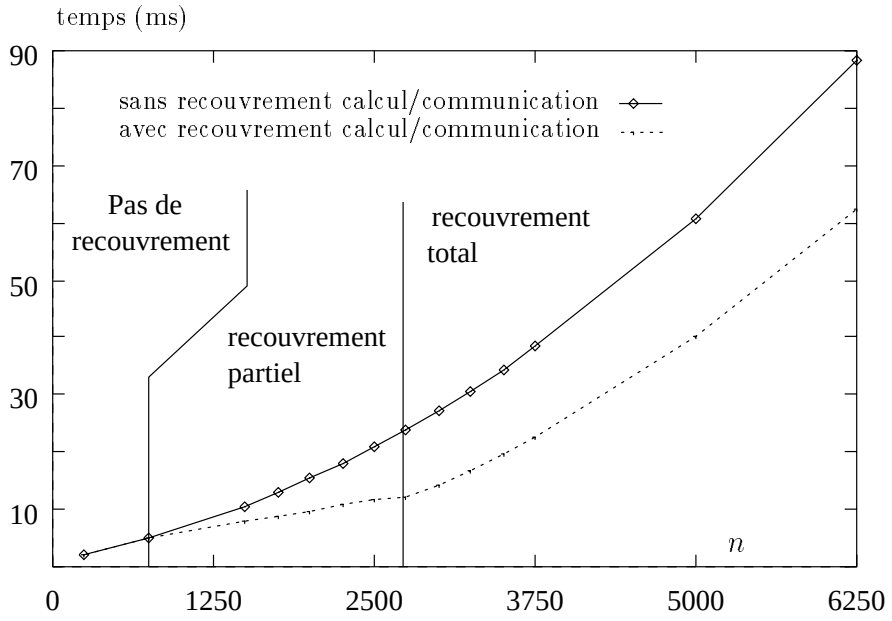



FIG. *5.9 - : Comparaison des algorithmes avec et sans recouvrement (liens half-duplex).*

La figure 5.10 montre les résultats obtenus avec des liens *full-duplex*. Comme les processeurs reçoivent plus de données, nous obtenons un recouvrement pour de plus petites valeurs de $n$.

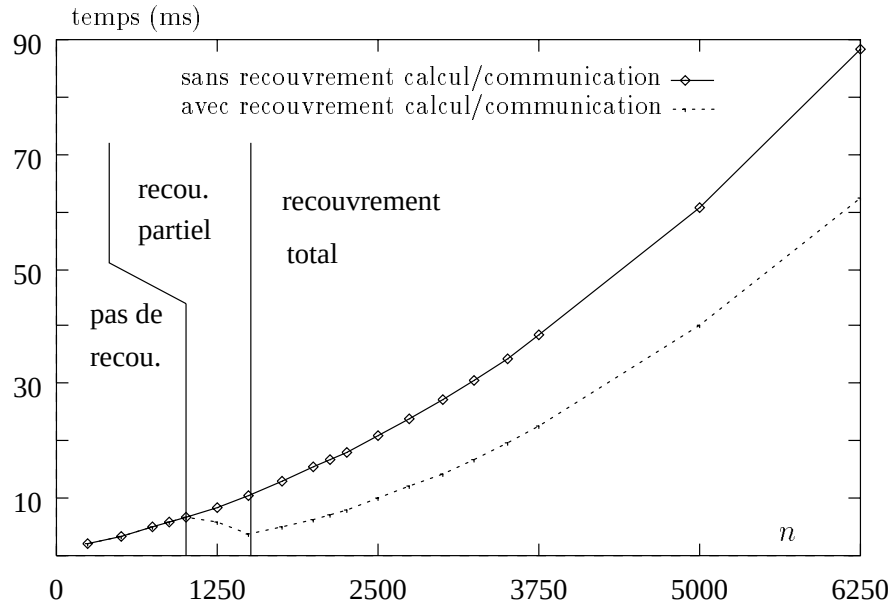



FIG. *5.10 - : Comparaison des algorithmes avec et sans recouvrement (liens full-duplex).*

Nous avons aussi utilisé d'autres paramètres qui sont : $\beta_c = 50\mu s$, $\tau_c = 2.5\mu s/octet$ and $\tau_a = 0.01\mu s$, pour confirmer les résultats des précédentes simulations, ainsi qu'un nombre de processeurs plus petit ($p = 10$). On remarque sur la figure 5.11 que l'on obtient les mêmes résultats.

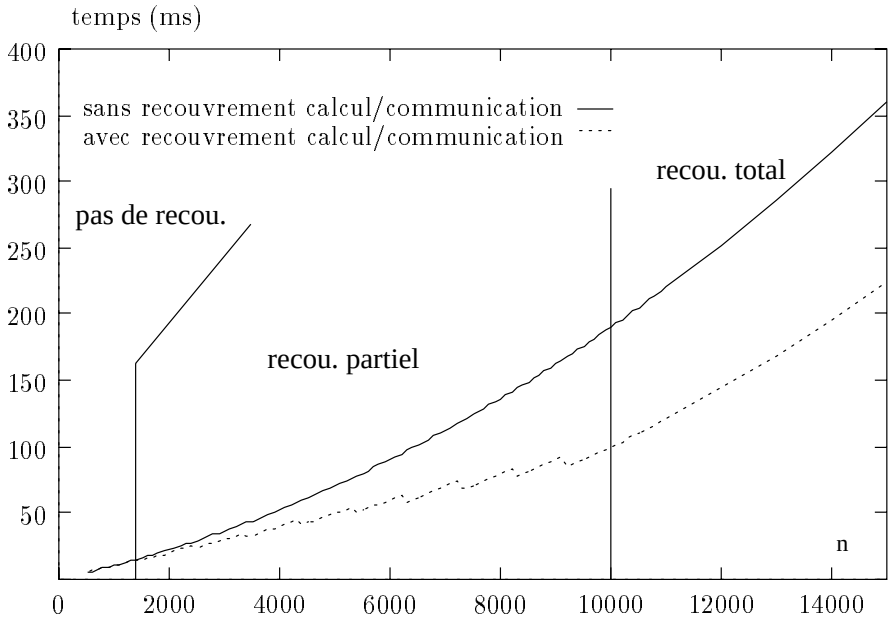



FIG. 5.11 - : *Comparaison avec de nouveaux paramètres des algorithmes avec et sans recouvrement (liens half-duplex).*

Les simulations montrent que, sur les machines actuelles, il est possible d'obtenir des gains importants en effectuant du recouvrement calcul/communication.

On peut remarquer que la stratégie utilisant les envois adaptatifs ($\lambda_i$ ) est en moyenne meilleure que la stratégie avec une communication constante de $\frac{n}{p}$ composantes de $x$ (voir figure 5.12). Les discontinuités de la courbe sont dues à la dernière communication de taille $\frac{n}{p}$ (initialisation) qui est trop importante pour finir le calcul global.

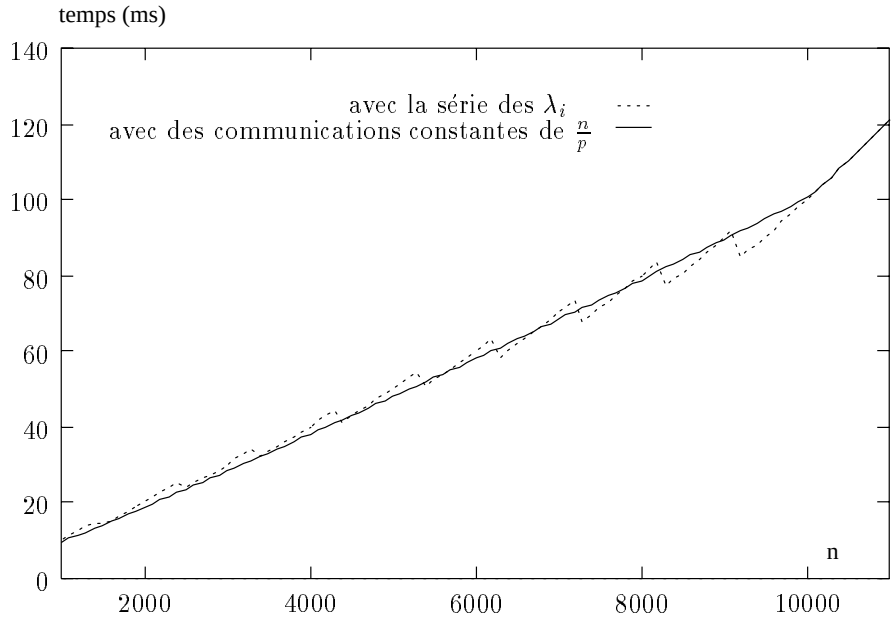



FIG. *5.12 - : Comparaison entre des envois adaptatifs et des envois constants.*

Mais il est possible de calculer exactement la meilleur taille de donnée à envoyer pour la dernière Phase I. En effet, il n'est pas n'ecessaire de faire une initialisation de taille $\frac{n}{p}$ s'il reste moins de $\frac{n}{p} + \sum_{i=1}^{N-1} \lambda_i$ données à recevoir. Dans ce cas, la solution avec les envois adaptatifs est toujours meilleure (figure 5.13).

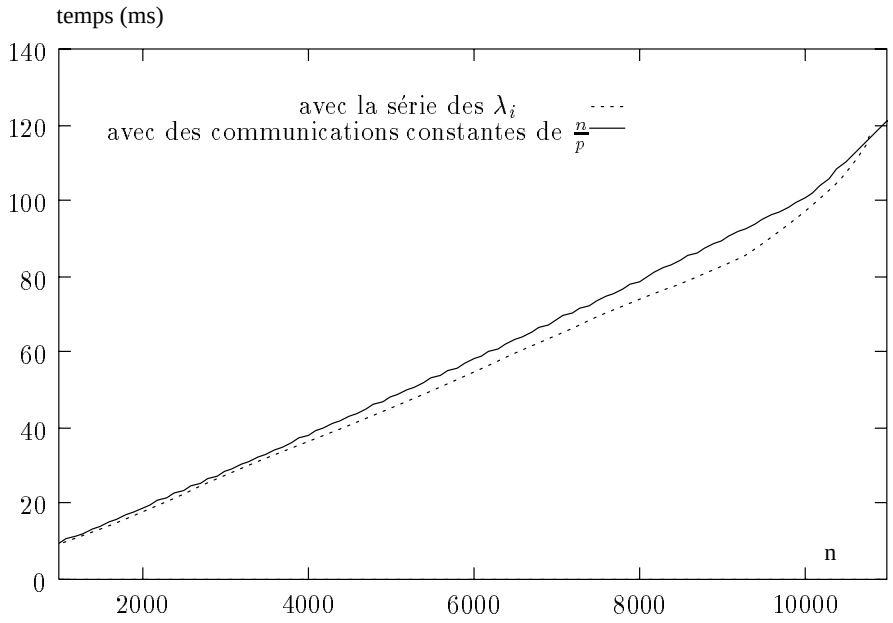
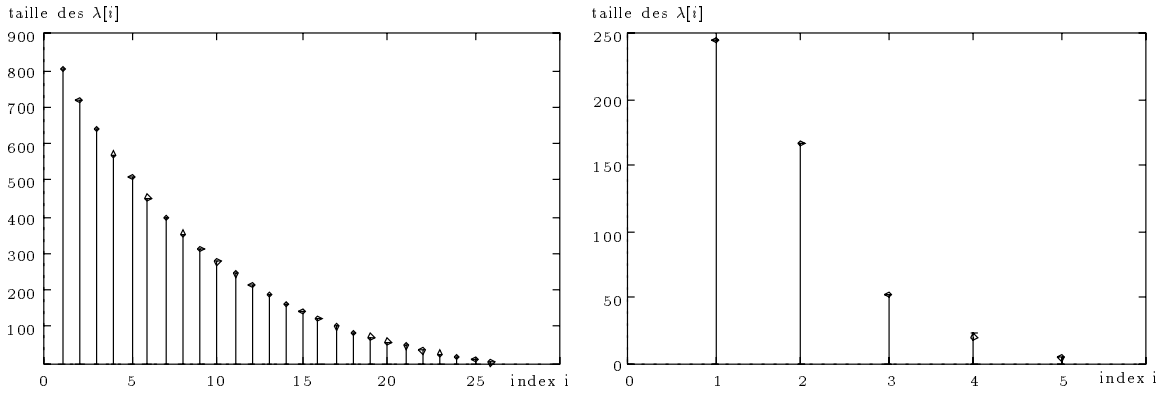



FIG. *5.13 - : Avec amélioration de la dernière phase de l'algorithme.*

Pour de petites tailles de $n$ il n'y a pas de recouvrement car le coût du *start-up* est très élevé. La série des $\lambda[i]$ est dégénérée (réduite à un terme). Pour des tailles moyennes de $n$ nous avons un nombre important de termes dans la série des $\lambda[i]$. Les figures suivantes représentent l'évolution de la série des $\lambda[i]$ pour des tailles égales à 2000 et 2625 réels. Enfin, pour les grandes tailles de $n$, il y a un recouvrement total, car les calculs locaux sont très importants.




(a) n=9000 (b) n=5000

FIG. *5.14 - : Comportement de la série des* $\lambda[t]$.

### 5.3.6 Expérimentations sur Paragon

Dans ce paragraphe, nous allons présenter les résultats obtenus sur la machine Paragon de l'IRISA, en utilisant les algorithmes de recouvrement calcul/communication décrits précédemment.

#### Description de la Paragon de l'IRISA

Depuis juin 1993 l'IRISA dispose d'une machine parallèle Intel : la Paragon. Cette machine est la version commerciale du projet *Delta-Touchstone* mené par Intel.

C'est une machine MIMD à mémoire distribuée [Cor91]. Elle dispose de 56 nœuds de calculs, de 3 nœuds de services sur lesquels on trouve le système OSF/1 avec des services tels que des éditeurs et des compilateurs, et de 2 nœuds d'entrées/sorties permettant l'accès à un système de fichiers parallèle. Ces nœuds sont connectés physiquement en grille bi-dimensionnelle.

Chaque nœud possède une mémoire de 16 Mo et est constitué de plusieurs éléments :

- deux processeurs i860 XP, l'un destiné au calcul et l'autre à la gestion des communications. Le i860 XP possède une horloge cadencée à 50 MHz (20 ns/cycle), et développe une puissance crête de 42 MIPS et de 75 MFlops double-précision (100 MFlops simple-précision). Il dispose de caches d'instructions et de données de taille 16 Ko. La bande passante entre le cache et les unités flottantes est de 800 Mo/s, tandis que la bande passante entre le cache et la mémoire est de 400 Mo/s. Le processeur de communication est chargé de la préparation des messages lors des émissions ou réceptions, ce qui permet de libérer le processeur de calcul de cette tâche ;

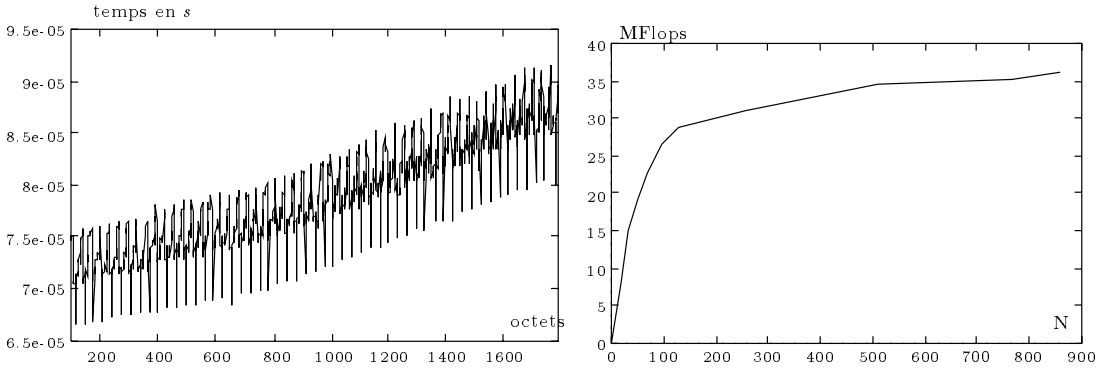
- deux *Data Transfert Engine* travaillant en parallèle qui fonctionnent comme des DMA. Ces *Data Transfert Engine* sont connectés via un contrôleur (*Network Interface Controller*) au réseau ;

- l'analyseur de performance qui permet d'enregistrer, sans perturber, les événements se déroulant sur les processeurs.

Chaque nœud est connecté par deux liens mono-directionnels à un PMRC (*Paragon Mesh Routing Chip*), l'ensemble des PMRC constituant les sommets de la grille. Ces PMRC ont pour rôle de router les informations dans le réseau. Ce routage est effectué en mode *wormhole*. Chaque PMRC possède 4 liens bi-directionnels avec ses 4 voisins et chaque lien a un débit de 200 Mo/s en *full duplex*.

### Modélisation des paramètres de la PARAGON

Afin de calculer correctement la série des $\lambda_i$ il est nécessaire de bien connaître les paramètres $\beta_c$, $\tau_c$ et $\tau_a$ de la machine. Des études [Mic94] ont montré que ces paramètres étaient très difficiles à évaluer.




(a) Communication entre deux voisins (b) Performance du produit scalaire

FIG. *5.15 - : Comportement de la Paragon.*

Comme les échanges de messages s'effectuent en mode *wormhole* [dR94] ( $\alpha + d\delta + L\tau_c$) et que nous effectuons des communications entre voisins, nous pouvons regrouper le terme en $\alpha$ et $\delta$ (puisque d=1) dans un $\beta_c$. Mais, comme le montre la figure 5.15(a), les communications sur la Paragon ne suivent pas un modèle linéaire. C'est pour cela que nous avons choisi un $\tau_c$ moyen. Nous avons procédé de même pour le $\tau_a$, en effet ce paramètre évolue en fonction de la taille des vecteurs à traiter (voir figure 5.15(b)).

Du fait de la puissance de calcul des processeurs, le recouvrement ne sera possible que pour de petites tailles de $n$, c'est pourquoi nous avons choisi un $\tau_c$ correspondant à la transmission de petits messages. Les paramètres que nous avons choisis pour modéliser la Paragon et ainsi calculer la série des $\lambda_i$ sont : $\beta_c = 65\mu s$, $\tau_c = 0.2\mu s/octet$, $\tau_a = 0.033\mu s$.

### Comparaison des différentes stratégies

Dans l'étude théorique nous avons décrit plusieurs méthodes, que nous avons expérimentées sur la Paragon sur laquelle nous simulons un anneau de 16 processeurs. Du fait de la complexité du modèle, nous nous sommes intéressés uniquement au cas *half-duplex*.

La figure 5.16 illustre le comportement des différentes stratégies en fonction de la taille du problème.

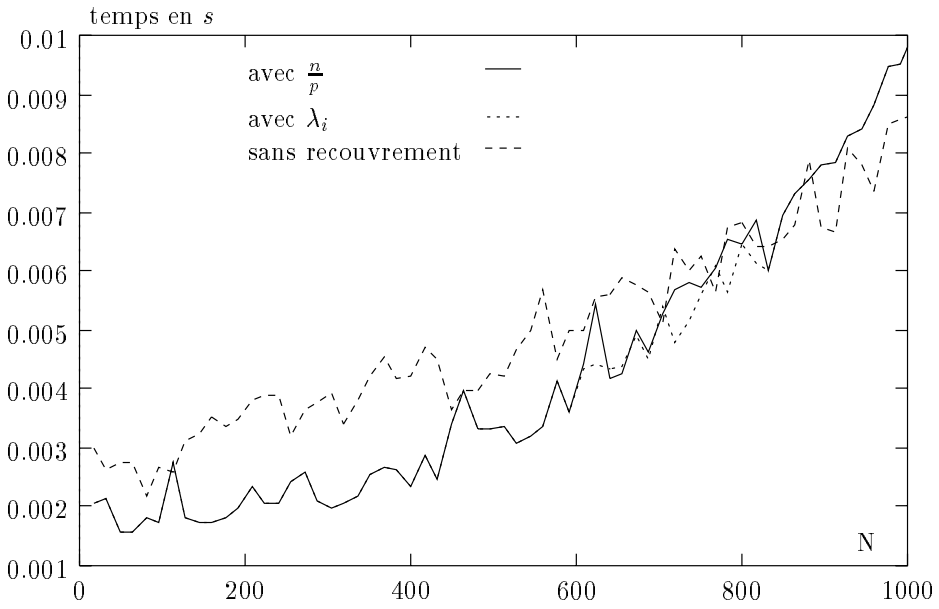



FIG. *5.16 - : Comparaison entre les différentes stratégies.*

Nous observons une série de "dents de scie" qui sont dues au fonctionnement des communications sur la Paragon. Par ailleurs, on constate que pour des tailles supérieure à 832 la méthode sans recouvrement, c'est-à-dire phase de communication (*All To All*) suivie d'une phase de calcul, donne de meilleurs résultats que la méthode avec un $\lambda$ constant. En effet, ceci est dû au fait que lorsque nous utilisons la méthode des $\lambda$ constants les communications se font sur de petits bouts de vecteurs, alors qu'avec l'autre méthode elles s'effectuent sur de grands vecteurs, ce qui permet aux unités pipelines du processeur de travailler au maximum de leur puissance.

La zone de recouvrement existe pour des problèmes dont la taille varie de 608 à 816. La figure 5.17 montre plus en détail cette zone.

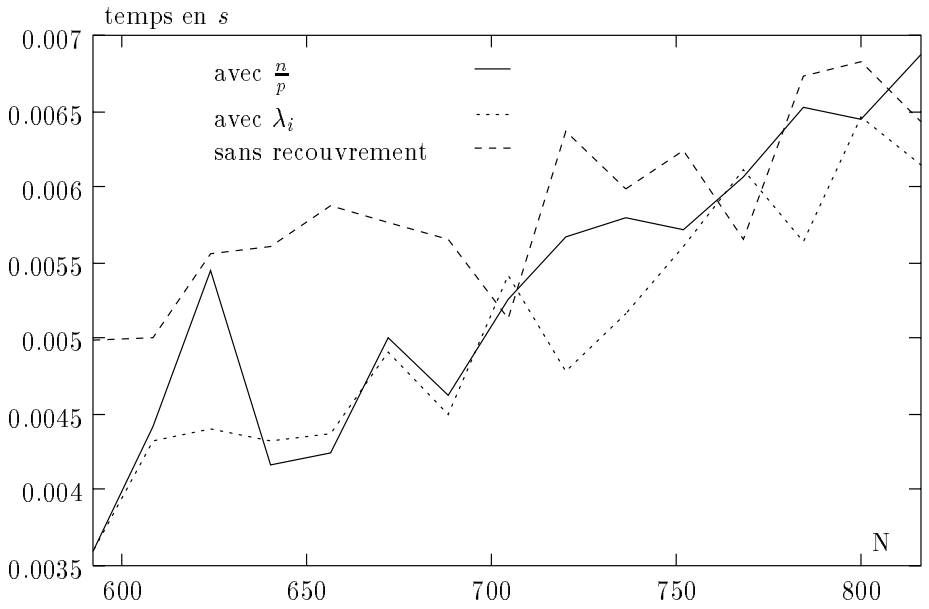



FIG. *5.17 - : Comparaison entre les différentes stratégies dans la zone de recouvrement.*

On peut remarquer que globalement la stratégie des $\lambda_i$ est meilleure que celle des $\lambda$ constants.

Maintenant nous allons introduire dans nos expériences l'amélioration de la dernière phase qui est le calcul et l'envoi de la meilleure taille de données pour initialiser la dernière Phase I (voir figure 5.18).

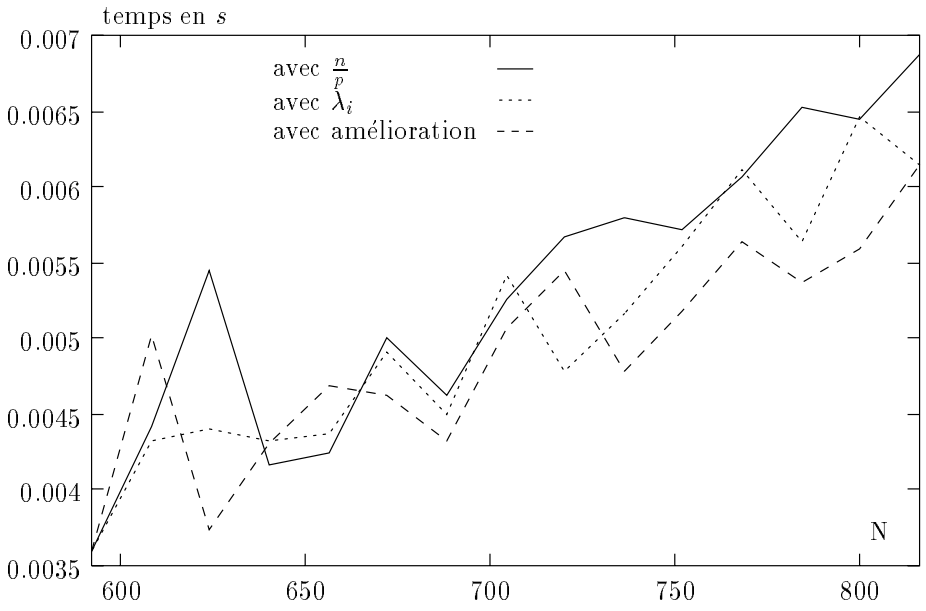



FIG. *5.18 - : Comparaison des algorithmes de recouvrement avec l'algorithme amélioré.*

Cette amélioration permet d'éviter une réinitialisation complète, d'où un gain de temps que l'on retrouve sur la figure 5.18.

## 5.4 Une dernière optimisation ?

Nous avons pu constater que le problème de la méthode précédente est l'augmentation non négligeable du nombre de réinitialisations. Ceci a pour conséquence de rendre inefficace la méthode de recouvrement du temps de communication par le temps de calcul pour de grandes valeurs du *Start-up* $\beta_c$. L'étude suivante est donc valable dans le cas où :

$$\beta_c + \frac{n}{p}\tau_c > \frac{}{n^2}p^2\tau_a.$$

### 5.4.1 Un autre modèle de communication

Pour diminuer au maximum le temps total d'exécution, nous avons eu l'idée d'utiliser une nouvelle possibilité fournie par le système d'exploitation de la machine Cray T3D : l'écriture d'une donnée locale à un processeur dans la mémoire d'un autre processeur, le but étant toujours d'effectuer des communications de taille $\frac{n}{p}$ et d'anticiper au maximum le temps de calcul. Ainsi, un processeur ira lire ou écrire directement en mémoire les données nécessaire à son calcul sans

être bloqué par une attente en réception. On peut donc considérer que la mémoire joue le rôle d'un « pipeline » ; le problème est d'assurer que les données se trouvent dans la mémoire avant d'effectuer le calcul les utilisant. Pour comparer avec les algorithmes et les méthodes précédents, nous allons reprendre les mêmes conditions que celles données dans le paragraphe 5.3. En fait, seul le modèle de communication change.

Afin de modéliser le temps d'exécution d'un algorithme utilisant l'écriture ou la lecture directe sur des mémoires distantes, nous allons considérer que l'écriture de données d'un processeur dans la mémoire d'un processeur voisin peut aussi être modélisée par : $\beta_c + L\tau_c$ où $L$ est le nombre d'octets à transférer. En fait, le mode de routage du Cray T3D est de type *wormhole*, modélisé par $\alpha + d\delta + L\tau_c$ où $d$ représente la distance entre les deux processeurs communiquant [dR94]. Puisque, entre deux processeurs voisins, $d = 1$, les facteurs $\alpha$ et $\delta$ peuvent être inclus dans seul un paramètre, d'où $\alpha + \delta = \beta_c$. De plus, pour la modélisation du coût de l'écriture en mémoire distante, la valeur de $\beta_c$ correspond, à plus de 97%, à la valeur de l'initialisation classique du réseau de communication $\alpha + \delta$. Les 3% restant sont dus au contrôleur de mémoire qui vérifie que la donnée modifiée n'est pas dans le « cache » du processeur de la mémoire cible. Si c'est le cas, il change la valeur en mémoire et invalide la valeur restée dans le « cache » . Toutes ces opérations ne prenant que deux ou trois cycles d'horloge sur le Cray T3D, nous pouvons donc modéliser correctement ce type de communication entre processeurs voisins sous la forme : $\beta_c + L\tau_c$.

### 5.4.2 Un nouvel algorithme

Un processeur ne va plus envoyer des messages de taille $\frac{n}{p}$, mais il va directement écrire les données dans la mémoire du processeur voisin. La figure 5.19 représente l'ordonnancement des opérations de calcul et de communication.

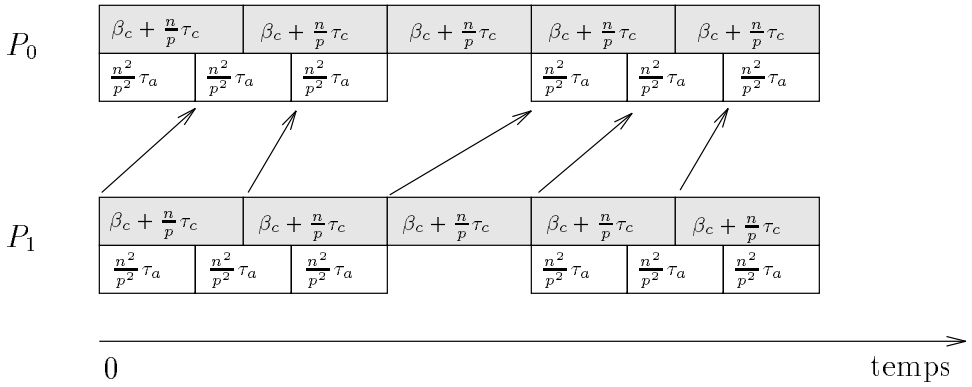



FIG. *5.19 - : Anticipation des calculs pour des communications à coût constant.*

Du fait que les communications sont asynchrones, il y a des réinitialisations de communication qui seront indispensables afin d'attendre l'arrivée en mémoire des données nécessaires pour le calcul local. Ces périodes de réinitialisations sont illustrées dans la figure 5.19.

De plus, durant la phase d'initialisation $\beta_c$ d'une communication, il n'y a aucune donnée écrite en mémoire. Il faut donc en tenir compte et observer un décalage du même ordre pour le temps de calcul. La figure 5.20 montre l'utilité de ce décalage afin d'optimiser au mieux l'arrivée des données.

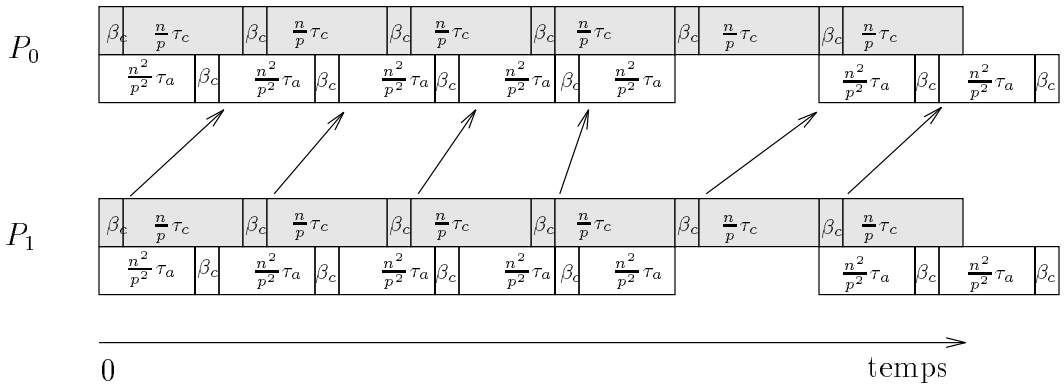



FIG. *5.20 - : Synchronisation avec augmentation du temps de calcul.*

Nous allons calculer la valeur du temps $R$ qui est définie comme étant la différence entre le temps de communication et le temps de calcul (voir figure 5.21).

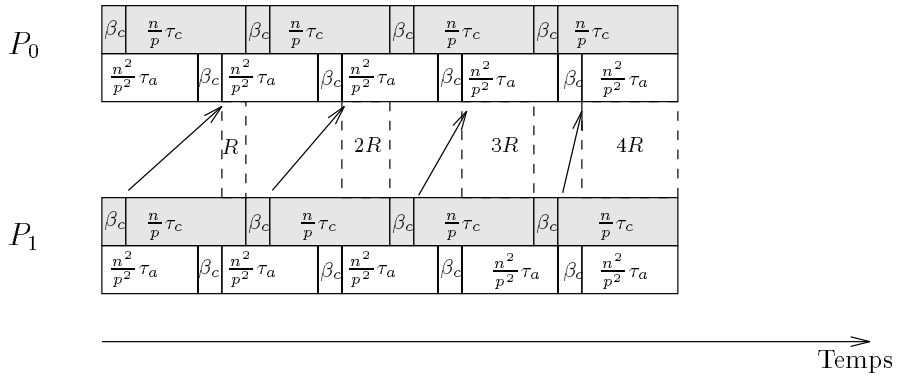



FIG. *5.21 - : Calcul de la valeur de décalage R.*

On remarque qu'à chaque étape de communication, il y a un décalage supplé-

mentaire de coût $R$. La valeur de $R$ est la suivante :

$$R = \beta_c + \frac{n}{p}\tau_c \ - \ \beta_c - \frac{n^2}{p^2}\tau_a = \frac{n}{p}\tau_c \ - \ \frac{n^2}{p^2}\tau_a \tag{5.8}$$

Nous pouvons maintenant calculer le nombre de communications $E$ que l'on peut faire avec recouvrement, la communication suivante ne pouvant être masquée par du calcul par manque de données locales. En fait, l'algorithme devra faire une communication de réinitialisation lorsque le temps de $E$ communications $\beta_c + \frac{n}{p}\tau_c$ sera égal au temps de $E+1$ calculs $\frac{n^2}{p^2}\tau_a$ plus $E$ décalages $\beta_c$ (voir figure 5.21). Nous avons donc :

$E \times (\beta_c + \frac{n}{p}\tau_c) - (E+1) \times (\frac{n^2}{p^2}\tau_a) - E \times \beta_c = 0$

d'où

$E = \frac{\frac{n^2}{p^2}\tau_a}{\frac{n}{p}\tau_c - \frac{n^2}{p^2}\tau_a}$,

donc

$$E = \frac{\frac{n^2}{p^2}\tau_a}{R}. \tag{5.9}$$

Sur les figures 5.20 et 5.21 la valeur de $E$ est de 4.

L'algorithme du produit matrice-vecteur sur un anneau de $p$ processeurs avec écriture en mémoire distante est le suivant :

**Algorithme 5.7**

*/* algorithme du processeur q avec q=1...p */*

*Calculer le nombre d'étapes E sans nouvelle initialisation*
*Pour i=1, mod((p-1),(E+1)) faire*
  *Pour j=1, E+1 faire en parallèle*
    *si (j < E+1) alors*
      *Ecrire les $\frac{n}{p}$ éléments courants dans la mémoire du processeur q+1*
    *finsi*
    *Calculer le produit scalaire avec les $\frac{n}{p}$ éléments courants*
  *Fin Pour*
  */* On attend l'écriture des données en mémoire */*
  *Ecrire les $\frac{n}{p}$ éléments courants dans la mémoire du processeur q+1*
*Fin Pour*
*Pour i=1, reste((p-1)/(E+1)) faire en parallèle*
  *Ecrire les $\frac{n}{p}$ éléments courants dans la mémoire du processeur q+1*
  *Calculer le produit scalaire avec les $\frac{n}{p}$ éléments courants*
*Fin Pour*

### 5.4.3 Calcul du temps d'exécution

**Cas où** $R = \frac{n}{p}\tau_c - \frac{n^2}{p^2}\tau_a > 0$

**Proposition 9** *Le nombre de communications de coût* $\beta_c + \frac{n}{p}\tau_c$ *est de :*

$$NoRec = \left\lfloor \frac{p-1}{E+1} \right\rfloor \tag{5.10}$$

*avec* $E = \frac{\frac{n^2}{p^2}\tau_a}{R}$ *et* $R = \frac{n}{p}\tau_c - \frac{n^2}{p^2}\tau_a$.

*Preuve*: Toutes les $E$ communications, l'algorithme effectue une réinitialisation d'un coût $\beta_c + \frac{n}{p}\tau_c$. De plus, on sait qu'au minimum pour réaliser un ATA (*cf.* paragraphe 5.3.1), il faut exécuter $p-1$ communications avec des messages de taille $\frac{n}{p}$. Sachant que les communications avec recouvrement se font par phases de $E$, on peut en déduire que le nombre de réinitialisations que l'algorithme doit faire afin que chaque processeur obtienne les données attendues, est de $\lfloor \frac{p-1}{E+1} \rfloor$. □

Une conséquence de cette proposition est que le nombre de communications masquées après la dernière réinitialisation est égale à :

$R_{rec} = reste((p-1),(E+1))$.

Nous pouvons en déduire que le temps de calcul après la dernière communication est $\frac{n^2}{p^2}\tau_a - R_{rec} \times R$.

Le temps total de l'algorithme est donc :

$$T_{exec} = (p-1)(\beta_c + \frac{n}{p}\tau_c) + \frac{n^2}{p^2}\tau_a - R_{rec} \times R \tag{5.11}$$

**Cas où** $R = \frac{n}{p}\tau_c - \frac{n^2}{p^2}\tau_a \leq 0$

Nous sommes dans le cas ou $\frac{n}{p}\tau_c \leq \frac{n^2}{p^2}\tau_a$. Cela signifie bien sûr qu'une communication est plus coûteuse que le calcul, essentiellement à cause du temps d'initialisation d'une communication $\beta_c$. Le temps de calcul global, égal à $\frac{n^2}{p}\tau_a$ plus le temps supplémentaire dû à un décalage de $\beta_c$ à chaque étape, devient supérieur au temps total des communications. Dans ce cas, il ne faut pas anticiper les calculs, utiliser des communications bloquantes d'un coût de $\beta_c + \frac{n}{p}\tau_c$ et pendant ce temps calculer des produits scalaires de taille $\frac{n}{p}$ avec les données reçues à l'étape précédente (voir figure 5.22).

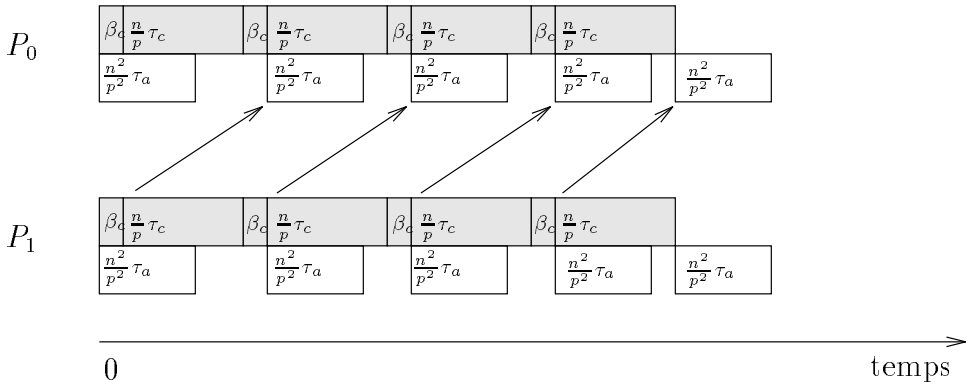



FIG. *5.22 - : Avec des réception bloquantes.*

Le temps total de l'algorithme est donc :

$$T_{exec} = (p-1)(\beta_c + \frac{n}{p}\tau_c) + \frac{n^2}{p^2}\tau_a. \tag{5.12}$$

### 5.4.4 Expérimentations sur Cray T3D

Nous présentons dans le tableau 5.1 les résultats d'exécution sur 16 processeurs du Cray T3D. Dans ce tableau la deuxième colonne représente le temps de calcul séquentiel sur chaque processeur. Nous notons par *sr* l'algorithme sans recouvrement et par *ar* l'algorithme avec recouvrement.
Sur la figure 5.23 nous comparons l'implémentation de l'algorithme sans recouvrement avec PVM et Shmem.

| Taille de la matrice | Temps de calcul | Shmem (ar) | Shmem (sr) | PVM (sr) |
|---|---|---|---|---|
| 256 | 3.41e-6 | 1.57631e-4 | 1.61021e-4 | 1.3904e-3 |
| 512 | 3.43e-6 | 1.94322e-4 | 1.97750e-4 | 1.49e-3 |
| 1024 | 3.48e-6 | 2.4671e-4 | 2.5016e-4 | 1.58e-3 |
| 2048 | 3.5e-6 | 4.16773e-4 | 4.2027e-4 | 1.83e-3 |
| 4096 | 3.6e-6 | 6.39331e-4 | 6.42031e-4 | 2.215e-3 |
| 8192 | 3.8e-6 | 1.27937e-3 | 1.283e-3 | 2.803e-3 |

TAB. 5.1 - : Temps d'exécution (en secondes) du produit matrice-vecteur sur 16 processeurs du Cray T3D.

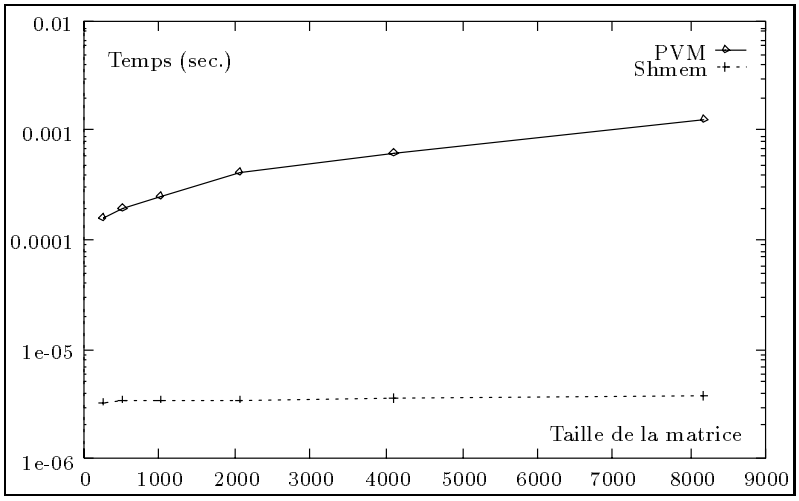



FIG. 5.23 - : Comparaison des temps d'exécution (échelle logarithmique) du produit matrice-vecteur sans recouvrement avec PVM et Shmem.

Tout d'abord, il faut remarquer que le temps de calcul n'évolue pas linéairement en fonction de la taille de la matrice, car les calculs se font sur des entiers et le processeur Alpha du Cray T3D est super-pipeline. De plus, la taille des produits scalaires locaux étant petits, tous les accès aux données ont lieu dans le cache du processeur. Les calculs ont été réalisés sur des entiers car il y a des problèmes de cohérence des données avec l'utilisation des fonctions Shmem sur des réels.
Nous pouvons remarquer également, que le temps de calcul est très petit par rapport au temps de communication (*cf.* tableau 5.1), ce qui implique que le gain apporté par la méthode avec recouvrement communication/calcul n'est pas très important. La seule possibilité de gain est le masquage du temps de calcul, ce qui est presque totalement réalisé par la méthode avec recouvrement. L'algorithme *ar* utilise une taille de paquet constante égale à $\frac{n}{p}$. Comme on peut le constater, une suite décroissante de tailles de paquets n'apporterait aucune amélioration, du fait que les communications sont bien plus coûteuses que le calcul.
D'après le tableau 5.1 et la figure 5.23, il est évident que l'utilisation de la bibliothèque Shmem apporte un gain considérable par rapport à l'utilisation de PVM. Cependant, l'utilisation des fonctions d'accès à la mémoire distante sont à manipuler avec précaution, car l'utilisateur doit gérer lui-même la cohérence des données.

## 5.5 Conclusion

L'utilisation de bibliothèques de communications non-bloquantes permet de réaliser des recouvrements calcul/communication. Ceci est d'autant plus facile sur les machines actuelles que tous les nœuds possèdent un processeur dédié aux

communications. Nous avons cherché à masquer au mieux les communications, pour cela nous avons étudié, sur un algorithme simple, la taille des messages qui doivent être envoyés à chaque communication.

L'expérimentation sur la Paragon a permis de mettre en œuvre la stratégie de recouvrement développée au cours de ce chapitre et de montrer l'amélioration que l'on peut obtenir avec cette méthode. Mais cette expérimentation a permis aussi de voir la difficulté du passage de la théorie à la pratique. En effet, pour calculer nos $\lambda_i$ nous avons fixé les paramètres de la machine, mais ces paramètres ne sont pas constants. $\tau_a$ va dépendre de la longueur du vecteur, il en est de même pour $\tau_c$ qui dépend de la longueur du message. Ceci va influer sur le calcul des $\lambda_i$ et sur la stratégie à choisir, car plus les messages sont longs plus $\tau_c$ va diminuer (de l'ordre de 5 Moctets/s pour quelques milliers d'octets à 88 Moctets/s pour un message de 2 Moctets !!) [Mic94]. De même, $\tau_a$ vaut 15 MFlops pour un vecteur de taille 50 et le double pour un vecteur de taille 300. Ceci a pour conséquence de rendre la méthode sans recouvrement (phase de communication suivie d'une phase de calcul) plus efficace que la méthode où, en parallèle, on effectue des calculs locaux et des communications de taille $\frac{n}{p}$, pour $n$ supérieur à 832 et de ne pas avoir pour chaque taille de $n$ les valeurs optimales pour $\lambda_i$.

L'étude révèle donc que le problème du recouvrement calcul/communication n'est pas simple et qu'il risque de devenir très compliqué pour des problèmes plus complexes.

# Chapitre 6

# Programmation d'applications scientifiques

*Le but de ce chapitre est de montrer qu'il est possible de paralléliser efficacement des applications scientifiques sur des machines parallèles à mémoire distribuée. Nous allons présenter les résultats de la parallélisation de deux applications. La première est la modélisation de jeunes étoiles qui a été effectuée en collaboration avec Laurent Desbat*[a] *et François Ménard*[b] *[CDM93]. La seconde est une modélisation de l'effet Compton (interaction entre un photon et un électron) qui a été développée avec la collaboration de Laurent Desbat et de deux stagiaires [BR93, BBB+94].*

[a] *TIMC-IMAG*
[b] *Observatoire de Grenoble*

## 6.1 Introduction

Les machines parallèles ont réalisé de grands progrès aussi bien dans le domaine de la conception matérielle que dans le domaine des systèmes d'exploitation et des logiciels de programmation. Comme nous l'avons vu dans les chapitres 2.15 et 4.7, des bibliothèques d'aide à la programmation sont disponibles sur la majorité des machines parallèles actuelles. Grâce à toutes ces évolutions, il est possible de concevoir le développement d'applications scientifiques parallèles importantes. Ceci peut aboutir à court terme à l'utilisation des machines parallèles par les industriels et les scientifiques comme machines de production. Le problème qui se pose actuellement est l'absence de démarche pour le développement de grands codes parallèles. En effet, il n'existe pas encore de génie logiciel dans le domaine du parallélisme, ce qui fait que seule l'expérience permet la création de programmes portables, efficaces, avec des possibilités d'évolution et de mise en œuvre des méthodes les plus récentes, comme par exemple le recouvrement calcul/communication (voir chapitre 5.5).

Quand il s'agit de paralléliser une application, il arrive souvent que le code séquentiel existe déjà. Dans ce cas, le chef de projet peut se poser les questions schématisées ci-dessous, afin de savoir si la parallélisation du code séquentiel est possible et avec quelle efficacité.

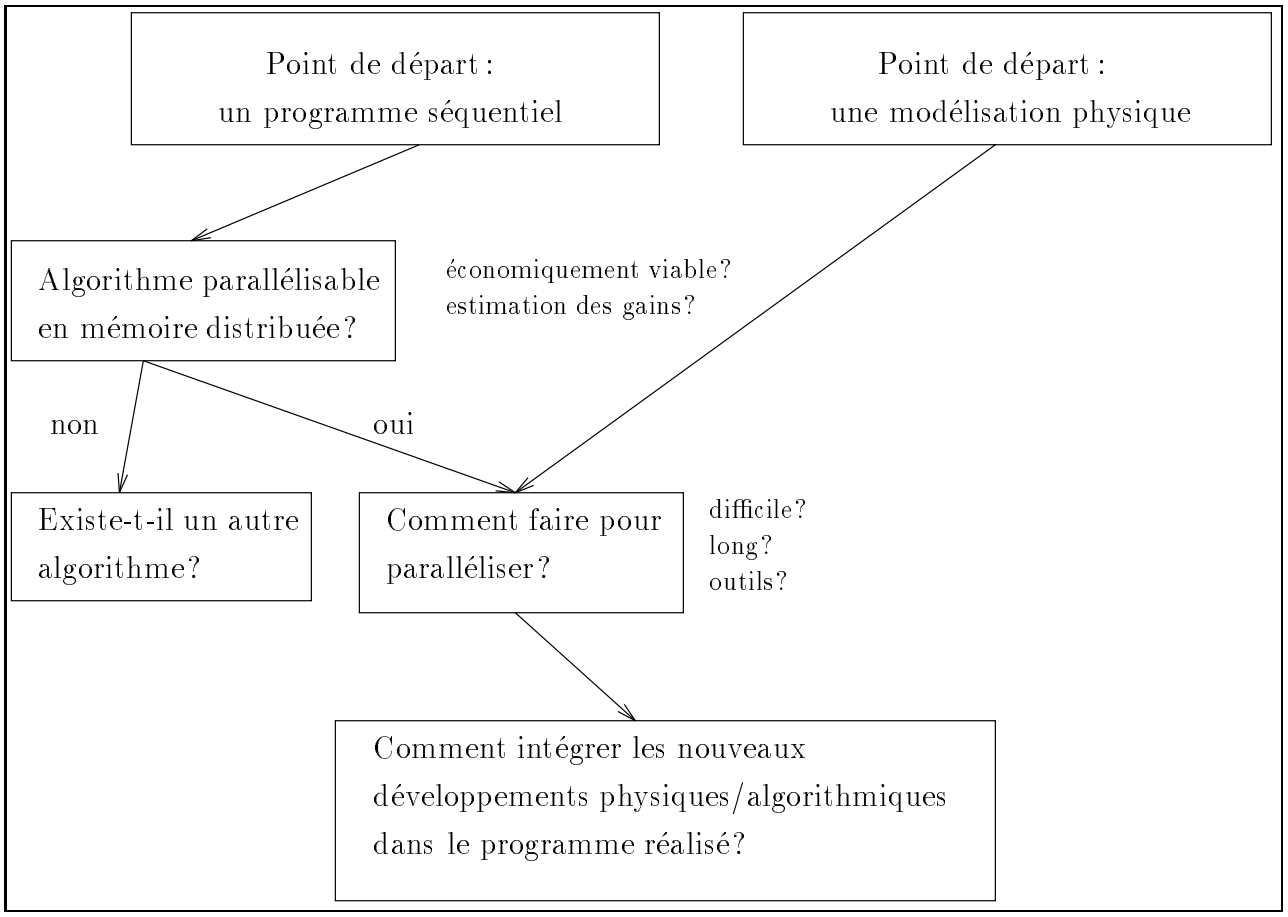



FIG. *6.1 - : Les principales difficultés.*

Cela représente beaucoup de questions et malheureusement il n'y a pas de réponses simples, car tout dépend des programmes étudiés !

Pour pouvoir répondre à ces questions, il faut étudier plus particulièrement les points suivants :

- les parties consommatrices en temps de calcul,
- la structuration des données,
- la localité des données,
- l'évolution du rapport opérations/transferts,

– l'équilibrage entre les tâches.

En fait, il s'agit de comprendre quels sont les points qui peuvent constituer des difficultés pour la parallélisation.

Nous pouvons commencer à élaborer une classification des programmes à paralléliser qui permet de référer plus rapidement à des travaux déjà effectués et donc d'évaluer assez précisément des possibilités offertes par la parallélisation.

### Les programmes parallèles embarrassants

Le terme « programme intrinsèquement parallèle » (« *embarrassingly* » *parallel* en anglais) désigne une catégorie de programmes par essence très parallèles. Le prototype de ce type d'application est la méthode du « lancer de rayons » , où le même traitement est effectué sur un grand nombre de données, les traitements étant complètement indépendants les uns des autres. En général, le volume des échanges est très faible et l'efficacité de la parallélisation est donc maximale. Les problèmes qui restent à traiter sont la répartition de la charge et donc des données ainsi que la tolérance aux pannes pour des applications s'exécutant sur des réseaux d'ordinateurs.

### Les programmes très séquentiels

Les programmes très séquentiels sont, en général, les programmes pour lesquels la parallélisation aboutit à des tâches présentant un rapport calcul/communication défavorable : par exemple, pour une factorisation de matrice pleine, le coût de calcul est en $O(n^3)$ et le coût de communication est en $O(n^2)$, ce qui signifie que pour des petites tailles de matrice le rapport sera trop défavorable et donc la parallélisation ne donnera pas de bons résultats. Dans ce cas, il faut essayer de mettre en œuvre des méthodes de recouvrement du temps de communication par le temps de calcul (voir chapitre 5.5) ou éventuellement utiliser des bibliothèques spécifiques de calcul numérique si elles existent (voir chapitre 4.7).

### Les autres programmes

Pour les autres programmes, une des techniques les plus utilisées pour les paralléliser est de découper les données. Cette technique est très utilisée pour les programmes de mécanique des fluides ou de traitement d'images. Mais, en général, il faut soit se contenter d'utiliser au mieux les outils de parallélisation automatique ou les bibliothèques numériques soit repenser entièrement le programme.

Parmi les applications que nous avons parallélisées, il n'y avait aucun code existant et nous avons donc considéré une nouvelle modélisation du problème afin d'en extraire le maximum de parallélisme.

## 6.2 Modélisation de jeunes étoiles

### 6.2.1 Introduction

En astronomie, l'amélioration permanente de la qualité des sites d'observations, des télescopes et surtout des détecteurs produit des volumes de données de quantité et de qualité croissantes. Cette richesse d'information permet d'envisager des modélisations plus fines, nécessitant bien souvent des méthodes d'analyse de plus en plus sophistiquées. Ainsi, les logiciels de traitement et d'analyse d'images nécessitent aujourd'hui plus de puissance de calcul. En astronomie, les principales raisons de cette demande croissante de puissance sont le nombre ou la dimension toujours croissante des détecteurs mais aussi la complexité des techniques d'observation (*i.e. polarimétrie, interférométrie*). De même, les modèles et donc les codes de calcul issus de théories astrophysiques permettant la simulation des observations sont, eux aussi, de plus en plus sophistiqués et souvent très gros.

Dans ce qui suit, nous décrivons le code numérique et l'environnement informatique que nous avons utilisés pour simuler le transfert de rayonnement dans une enveloppe circumstellaire contenant des grains de poussière. Toute la problématique réside dans l'abondance de ces grains autour des étoiles en formation, ce qui entraîne un grand nombre de diffusions (*i.e. d'interactions grain-photon*) dont l'effet global ne peut pas être évalué de façon analytique. Comme nous le verrons plus loin, la méthode la plus adaptée pour résoudre le problème est la méthode stochastique dite de Monte Carlo.

Cette méthode est très puissante mais pour simuler finement les observations, en incluant la haute résolution angulaire, une très grande quantité d'opérations est nécessaire, rendant inutilisables, sauf pour les cas les plus simples, les ordinateurs de faible puissance ($< 10 - 15$ Mflops[1]).

Parce que la méthode Monte Carlo nous aide à résoudre le problème en suivant littéralement des millions de photons de façon indépendante, il est évident que le calcul réparti, ou l'utilisation de machines massivement parallèles, est la voie la plus prometteuse pour les simulations. L'utilisation de Transputers nous a d'ailleurs permis de vérifier cette affirmation dès 1990. Les possibilités des machines parallèles, ou du moins de celles qui nous sont accessibles à l'heure actuelle, sont cependant assez limitées et peuvent être dépassées par un réseau de stations de travail puissantes disposant d'environnement permettant de paralléliser des applications, comme par exemple la bibliothèque de fonctions de communication PVM.

Après une présentation succinte du contexte et des objectifs astronomiques dans lesquels s'inscrit ce code de calcul, nous décrirons la parallélisation sous PVM de notre code de modélisation de transfert de rayonnement dans une enveloppe circumstellaire sur un réseau de stations de travail IBM-RS6000. Nous

[1] unité désignant un million d'opérations flottantes par seconde

développons ensuite une étude de la performance de notre application parallèle en fonction de la quantité de travail élémentaire attribuée à chaque processeur et en fonction du nombre de processeurs.

### 6.2.2 Transfert de rayonnement dans une enveloppe

#### Le contexte astrophysique général

Le problème qui nous intéresse concerne la formation des étoiles et plus particulièrement la formation des disques de poussière autour des étoiles jeunes de faible masse. L'étude de la formation de ces disques est primordiale pour la compréhension des mécanismes de formation des systèmes solaires comme le nôtre par exemple. En effet, le soleil est une étoile de faible masse et les planètes l'entourant sont toutes situées à peu près dans le même plan orbital. Le but d'études telles que la nôtre est donc de donner des éléments de réponse à des questions comme « le soleil est-il la seule étoile entourée de planètes ? » ou « comment les planètes se forment-elles ? » .

Les processus généraux menant à la formation d'une étoile sont relativement bien connus. Très schématiquement, tout débute par l'effondrement, sous l'effet de sa propre gravité, d'un nuage moléculaire en une concentration centrale de matière (poussière et gaz) qui finalement deviendra l'étoile.

La formation d'un disque entourant l'étoile centrale résulte probablement de la rotation du nuage moléculaire originel. En effet, à cause de la rotation différentielle (différentes vitesses de rotation en fonction de la distance au centre de la Galaxie) les nuages moléculaires tournent sur eux-mêmes, très lentement, mais suffisamment pour forcer l'agglomération de matière dans une région aplatie, le disque, dans le plan de rotation.

Si ce scénario général est relativement bien fondé théoriquement, il n'en demeure pas moins que plusieurs questions importantes restent sans réponse à ce jour. Nous savons par exemple que la Galaxie est traversée par un champ magnétique, mais on comprend encore très mal l'effet de ce champ sur la dynamique de l'effondrement et sur la structure des nuages moléculaires. On ne sait pas avec certitude dans quelle mesure ce champ peut retarder, modifier, voire empêcher la formation d'étoiles. De même, malgré beaucoup d'efforts récents de simulation, les conditions initiales (pression, température, moment cinétique) menant à la formation d'un système binaire plutôt qu'à celle d'une étoile simple plus massive sont elles aussi mal connues.

Pour apporter des connaissances supplémentaires aux modèles d'étoiles et de disques, nous avons mis sur pied un programme de recherche et d'étude de ces disques faisant appel à une technique originale : la polarimétrie.

#### Les buts scientifiques immédiats

L'étape à franchir est donc celle de la modélisation d'objets individuels. Le but premier est d'obtenir, en plus de la connaissance que nous avons déjà de leur existence, les paramètres géométriques précis de plusieurs d'entre eux. Nous espérons ainsi pouvoir connaître des estimations des dimensions et des masses que peuvent avoir ces disques, valeurs qui serviront de tests aux modèles actuels de formation d'étoiles et de point de comparaison avec notre système solaire.

### 6.2.3 Parallélisation du code de calcul de transfert de rayonnement

#### Présentation du site informatique

L'Observatoire de Grenoble dispose d'une grande quantité de stations de travail IBM RS6000 réparties sur le réseau Grenet et peut atteindre d'autres stations de travail toutes aussi puissantes via différents réseaux. Le cumul de toutes les puissances de calcul des stations du réseau utilisées par notre application permet d'obtenir une puissance (1 Gflops[2] crête) comparable à celle des différents supercalculateurs accessibles par l'Observatoire de Grenoble. L'accès aisé à ce réseau de stations et sa puissance de calcul potentiellement très élevée nous ont conforté dans l'idée de son utilisation pour la parallélisation de notre application. Vingt stations IBM RS6000, réparties sur le réseau Grenet (réseau du domaine universitaire Grenoblois) et du Magistère de Physique, ont formé le plus grand réseau disponible pour nos expériences.

Nous n'avons pas utilisé directement la possibilité de calculer sur un réseau de machines très hétérogènes, car nous n'avions pas de machines vectorielles ou parallèles. En fait, notre réseau est faiblement hétérogène dans la mesure où les machines sont *différentes* dans la gamme des stations IBM-RS6000 : 320, 520, 530, 550, 560.

#### Etude du parallélisme de la modélisation

L'évolution des trajectoires des photons diffusés par l'étoile est calculée par une méthode de Monte Carlo. Cette méthode est caractérisée par une grande indépendance des calculs car les trajectoires des photons émis sont supposées indépendantes.

La parallélisation de notre application s'effectue à travers deux types de tâches : une tâche maître appelée *main* donne du travail (c'est-à-dire un certain nombre de trajectoires à calculer) à des tâches esclaves appelées *tile* qui effectuent les calculs.

[2] unité désignant un milliard d'opérations flottantes par seconde.

### Généralités à propos de la répartition dynamique

Une question fondamentale du parallélisme concerne le placement des tâches. En général il faut favoriser les techniques statiques en recherchant *a priori* un placement optimal. Les techniques dynamiques, plus délicates à mettre en œuvre, ne sont utilisées que s'il reste des décisions de répartition qui ne peuvent être prises avant l'exécution. Notre programme parallèle est basé sur une répartition ou allocation automatique (*ou dynamique*) de tâches sur les processeurs ($load - balancing$ en anglais). Dans notre contexte de travail, la répartition automatique de type gestion de file d'attente (rôle joué par la tâche *main*) est simple à mettre en œuvre, donne de bons résultats et donc s'impose. En effet, les processeurs de notre réseau sont ceux des stations de travail IBM RS6000 utilisées en *batch* ou en interactif par plusieurs utilisateurs d'une part, et leurs performances sont différentes d'autre part. La charge de travail pour notre application doit donc être adaptée à chaque processeur. Cette répartition s'effectue simplement en découpant le travail total en tranches élémentaires qui sont allouées dynamiquement aux différents processeurs. Cette répartition dynamique permet de rendre le programme relativement tolérant aux pannes, en effet si un ou plusieurs des ordinateurs de notre réseau tombent en panne, le programme *main* redistribuera le travail non effectué par ces derniers à d'autres stations du réseau.

### Génération des nombres aléatoires

Une méthode Monte Carlo a besoin de nombre aléatoire en grande quantité.

Les périodes des générateurs de nombres « aléatoires » ne sont pas très bien connues mais elles sont près des limites imposées par l'ordinateur [Knu81]. En général, cette limite est fonction du plus grand entier représentable pour un générateur simple et de la multiplication des modules pour des générateurs combinés, qui sont une combinaison d'un générateur pour les nombres du type $A_{i+1} = (a \times A_i + c) modulo\ m$ où $a$ et $c$ sont respectivement le multiplicateur et l'incrément et d'un autre générateur pour effectuer le mélange de ces nombres [Knu81, MÉ89].

Avec un générateur combiné le même nombre peut revenir plusieurs fois, mais grâce au mélange il n'est pas toujours suivi par la même séquence, donc effectivement, la séquence est le produit des séquences individuelles des deux générateurs [Knu81]. En prenant pour hypothèse que la période du générateur égale le module nous obtenons des périodes efficaces de l'ordre de $10^{10}$ voire $10^{11}$ nombres pour un processeur 32 bits. Une étude plus précise été menée par François Ménard [MÉ89]

Une tâche du programme parallèle, pour notre application la tâche appelée *main*, devra fournir aux tâches de calcul des germes ($A_0$) différents et contrôler que les séries des premiers nombres aléatoires de chaque tâche sont bien diffé-

rentes.

### Description du programme parallèle

Comme nous l'avons décrit précédemment, la parallélisation de la modélisation s'effectue selon le schéma maître-esclave. Elle se décompose en deux types de tâches :

- La première tâche, notée *main*, est chargée de distribuer le travail aux différents processeurs, de gérer une éventuelle panne d'un des processeurs, de fournir un nouveau travail au processeur qui vient de finir son calcul et bien évidemment de récupérer et de traiter les résultats intermédiaires et finaux. La tâche *main* est donc constituée d'un algorithme d'allocation dynamique de charge et de traitement des résultats accumulés (voir l'algorithme 6.1).
- Les tâches notées *tile*, sont chargées de communiquer avec la tâche *main* et de lancer un calcul avec les derniers paramètres reçus. La tâche *tile* est décrite par l'algorithme 6.2.

**Algorithme 6.1** *main*

```
N=80000000 /* nombre de trajectoires à calculer */
nbdone = 0 /* nombre de trajectoires calculées */
diffuser(parameters,random(1: P),nb)
nbrandom = P+1 /* nombre de graines utilisées */
tant que (nbdone < N)
    recevoir(results,cet_ordinateur)
    nbdone = nbdone+nb
    /* « nb » est le nombre de trajectoires calculées
    par « cet_ordinateur » */
    si = (nbdone < N) alors
        si verifie(cet_ordinateur) alors
            envoyer(parameters,random(nbrandom),nb,cet_ordinateur)
            nbrandom=nbrandom+1
    else
        enlever_du_reseau(cet_ordinateur)
    finsi
    Traitement_des_donnees (results)
fin tant que
Traite_et_sauve_les_resultats
Fin_pour_toutes_les_taches
stop
```

**Algorithme 6.2** ***tile***

```
recevoir(parameters,random(1:P),nb,main)
newseed=random(myinum)
tant que (il_y_a_du_travail())
      calculer(parameters,nb,newseed,results)
      envoyer(results,main)
      recevoir(parameters,newseed,nb,main)
fin tant que
```

Les principaux paramètres de l'algorithme sont les suivants :

→ $N$ est le nombre total de photons à traiter.

→ $nb$ est la tranche élémentaire de photons à calculer entre les communications de paramètres et de résultats entre le processus maître (*main*) et ses esclaves (*tile*).

→ $P$ est le nombre de processeurs. Les processus *tile* sont numérotés de 1 à $P$.

→ *parameters* est la liste des paramètres physiques de la modélisation.

→ *results* est le tableau des résultats

→ *random* est le tableau d'index des différents générateurs de nombres aléatoires.

On peut remarquer que le programme *main* répartit dynamiquement la charge sur les processeurs - dit esclaves - en découpant le travail en tranches élémentaires (de $nb$ photons) et en alimentant les processeurs dès qu'ils sont disponibles pour notre application. L'algorithme est tolérant aux pannes. En effet, les résultats finaux sont obtenus dans tous les cas sauf si la station qui exécute la tâche *main* tombe en panne (cette station exécute en pratique aussi une tâche *tile*).

### Performances de l'algorithme

Dans cette partie, nous présentons d'une part une étude des performances de notre algorithme parallèle sur un réseau restreint et d'autre part des expérimentations réelles sur un réseau complet. Le réseau restreint de stations qui nous a permis de tester les performances de notre programme parallèle est le réseau *token-ring* de l'Observatoire de Grenoble, composé d'une station IBM-RS6000 560, une station 550 et trois 320. Lors de ces expérimentations, les stations étaient réservées à notre application. La première série de mesures permet d'étudier le temps d'exécution de l'algorithme séquentiel sur une seule machine. La seconde série permet de mesurer l'influence de $nb$ sur le temps total d'exécution. La dernière série permet de visualiser le temps d'exécution de l'algorithme en fonction

du nombre $P$ de processeurs, le réseau étant faiblement hétérogène. Toutes les expériences sont réalisées avec le même jeu de paramètres *parameters*.

Les expériences réelles de modélisation ont été exécutées sur le réseau complet, soit vingt RS6000.

### Algorithme séquentiel

Dans la figure 6.2, nous remarquons que l'augmentation du temps de calcul est linéaire par rapport au nombre de photons à diffuser lorsque le nombre de photons est suffisamment grand. Pour les quantités de photons qui nous intéressent ($N \in [10^5, 10^8]$), les variations statistiques du temps d'exécution sont négligeables : le temps d'exécution du programme est le produit du nombre $N$ de photons à calculer par le temps moyen de calcul d'un photon. Ainsi, à partir du temps d'exécution d'un faible nombre de photons ($10^5$), nous pouvons prédire le temps séquentiel d'exécution de l'algorithme pour les nombres supérieurs de photons ($10^7$ à $10^8$). Nous constatons d'autre part que le programme séquentiel est environ 2.5 fois plus rapide sur un RS6000-560 que sur un RS6000-320, 2 fois plus rapide sur un 550, 1.25 fois plus rapide sur un 530 et prend le même temps sur un 520. Le rapport des puissances des machines sur notre application est en gros le rapport des fréquences d'horloge.

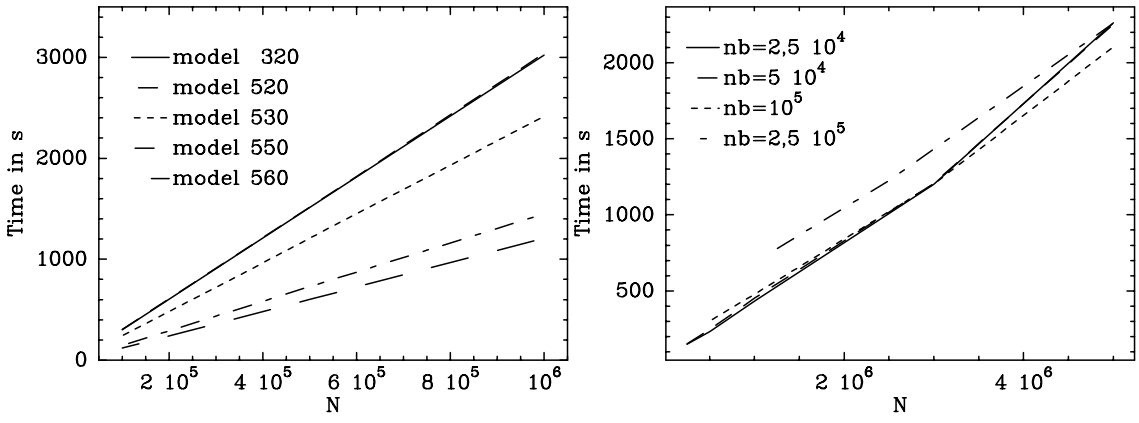



FIG. 6.2 - : A gauche, exécution séquentielle : variation du temps (en secondes) suivant le nombre total $N$ de photons à traiter. A droite, exécution parallèle : variation du temps (en secondes) suivant le nombre de photons calculés par une tâche.

### Algorithme parallèle : influence de la charge élémentaire $nb$

En général un paramètre comme $nb$ ne doit pas être choisi trop grand : il faut $nb \ll N$ pour que la répartition dynamique soit assez fine. Par contre si $nb$ est trop petit, le surcoût dû aux communications devient trop important. Dans la figure 6.2 nous constatons que pour notre application la valeur de $nb$ (choisie suffisamment grande) n'est pas cruciale car notre programme parallèle ne nécessite que très peu de communications.

### Algorithme parallèle : influence du nombre $p$ de processeurs

Dans la figure 6.3, nous montrons que la parallélisation de l'algorithme est excellente puisque le temps d'exécution est quasiment divisé par le nombre de stations sur la figure de gauche et par $\#(\boldsymbol{p}^*, p)$ (voir chapitre 3.6) dans le cas de la figure de droite avec un 320 comme PE ($\#(\boldsymbol{p}^*, p)$ vaut successivement 1, 2, 3, 5.5, 7.5). Il faut de plus souligner qu'une application réelle nécessite plus de $5.10^7$ photons avec des paramètres produisant un temps moyen par photon plus grand. Les conditions réelles sont donc encore plus favorables au parallélisme.

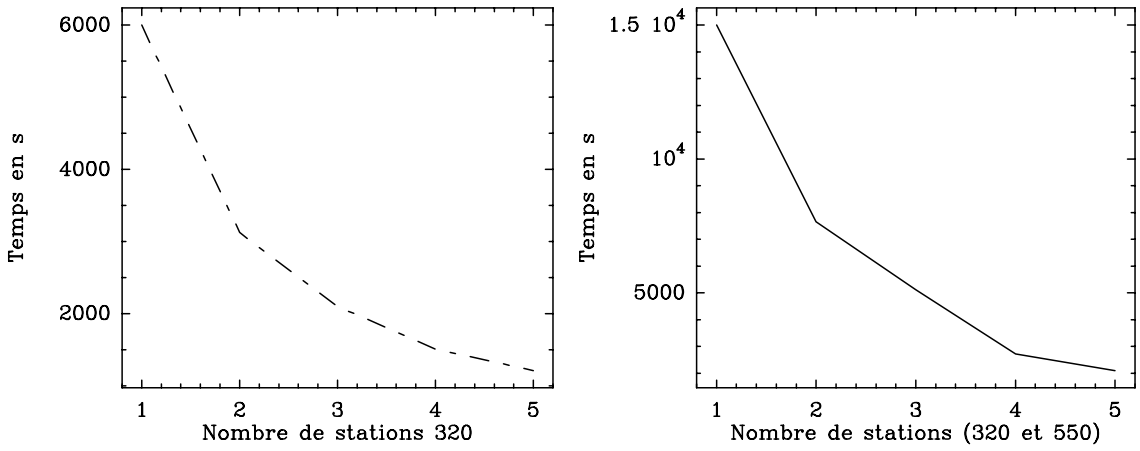



FIG. 6.3 - : A gauche : évolution du temps d'exécution en fonction du nombre de stations RS6000 320 utilisées. A droite : évolution du temps d'exécution pour trois 320 plus un 560 et un 550 dans cet ordre d'apparition.

### Expériences réelles

Nous avons mesuré pour chaque expérience le temps d'exécution sur chaque type de machine du réseau pour le programme séquentiel avec $10^5$ photons. D'après le paragraphe 6.2.3, nous pouvons prédire à partir de ces mesures le temps d'exécution séquentielle de l'expérimentation réelle (50 à 80 millions de

photons) sur chaque machine ainsi que les puissances relatives des différentes machines. Nous pourrons donc aisément donner une accélération relative à chacune des machines du réseau et une efficacité de l'algorithme parallèle en appliquant les résultats du chapitre 3.6.

| Expérience | IBM 320 | IBM 520 | IBM 530 | IBM 550 | IBM 560 |
|---|---|---|---|---|---|
| G-gg 10.10.1 | 288 s | 286 s | 228 s | 136 s | 114 s |
| G-gg 5.5.1 | 294 s | 294 s | 232 s | 138 s | 116 s |
| S-gg 10.10.1 | 285 s | 283 s | 227 s | 134 s | 113 s |
| S-gg 5.5.1 | 285 s | 284 s | 226 s | 134 s | 113 s |

Dans le tableau suivant, nous utilisons le RS6000-320 comme processeur de référence et supposons que les vecteurs coût des différentes architectures sont linéairement dépendants (cette approximation est raisonnable dans la mesure où les architectures des RS6000 sont semblables et toutes super-scalaires). Ainsi la puissance relative de deux processeurs sera le rapport des fréquences d'horloge (comme l'indiquent les mesures des temps séquentiels). La puissance relative du réseau, $\#(\boldsymbol{p}^*, p)$, est la somme des puissances relatives. La mesure de l'efficacité sera ainsi simplement donnée par $S_P(p)/\#(\boldsymbol{p}^*, p)$. Nous présentons enfin quelques résultats sur l'exécution de ce programme pour des modélisations réelles sur un maximum de 20 stations dans la gamme des IBM-RS6000 (quinze 320, un 520, un 530, deux 550 et un 560) représentant une puissance crête de 1 Gflops. L'exécution s'est effectuée de nuit sur des machines non réservées à notre application, mais faiblement chargées. Les valeurs de l'accélération et de l'efficacité sur un réseau dédié auraient été supérieures et plus régulières, néanmoins les résultats sont déjà très satisfaisants.

| Expérience | $N$ | $\#(p)$ | Temps parallèle | $T_{seq}$ estimé | $S_P(p)$ | $E$ |
|---|---|---|---|---|---|---|
| G-gg 10.10.1 | 50 millions | 23.75 | 8465 | 144000 | 17. | .72 |
| G-gg 5.5.1 | 80 millions | 23.75 | 12667 | 235200 | 18.57 | .78 |
| S-gg 10.10.1 | 80 millions | 23.75 | 11818 | 228000 | 19.29 | .81 |
| S-gg 5.5.1 | 80 millions | 23.75 | 12299 | 228000 | 18.54 | .78 |

Ces calculs ont permis d'obtenir les résultats astrophysiques présentés dans la partie suivante.

### 6.2.4 Résultats et perspectives astrophysiques

Nous avons connu ces dernières années un accroissement de puissance très rapide des ordinateurs, en particulier des stations de travail. Ceci nous a permis d'améliorer la qualité de nos simulations. Par exemple, les premières simulations Monte Carlo que nous avons effectuées, sur un CYBER 855, contenaient six

millions de photons répartis sur vingt cartes de dimension $11 \times 11$. A l'époque, ces simulations étaient les seules du genre. Mais rapidement des machines plus rapides, en particulier celles massivement parallèles (comme le *Transputer Super Cluster 128* de Parsytec), nous ont permis de pousser les mêmes modèles jusqu'à des résolutions de $31 \times 31$ pour chacune des vingt cartes, impliquant un nombre total de photons de l'ordre de 100 millions. Les temps requis pour ces simulations, jugés alors raisonnables, sont de l'ordre de 24 heures de temps CPU. Ce type de simulations nous a permis de mettre en évidence la présence des disques mais pas de modéliser des objets précis par exemple, ni d'explorer de manière exhaustive l'effet du disque, en particulier dans l'infrarouge où la réémission thermique des grains est importante.

Mais l'utilisation de notre programme parallèle, simplement avec le réseau de machines disponibles à Grenoble, nous permet déjà de gagner un facteur 10 en performance. Ceci autorise des simulations plus fines qui nous permettent de modéliser adéquatement les observations et ainsi de mieux connaître les objets étudiés, mais aussi de pousser plus loin les expériences numériques et ainsi de mettre à jour de nombreux diagnostics, facilement observables, mais inutilisés jusqu'à présent faute de prédictions théoriques ou d'outils d'interprétation autres que qualitatifs.

### 6.2.5 Conclusion

Nous avons présenté la parallélisation d'un code de modélisation de type Monte Carlo sur un réseau de stations de travail. Dans ce contexte privilégié (indépendance des calculs, peu de communications) l'utilisation de PVM pour l'écriture du code a permis d'obtenir d'excellentes performances. Nous avons mis en œuvre une répartition dynamique de charge rendant l'algorithme performant et très tolérant aux pannes. Lorsque les machines sont réservées à notre application, l'efficacité de l'algorithme est alors quasi optimale. Dans le cas contraire, notre algorithme utilise au mieux la puissance disponible sur le réseau.

Cette démarche du calcul parallèle s'inscrit dans la politique de calcul réparti de l'Observatoire de Grenoble. Le calcul parallèle sur réseau de stations ne permet pas toujours d'atteindre des performances telles que celles que nous obtenons pour notre application dans laquelle le trafic des données sur le réseau est très faible relativement au volume des calculs. Cependant, il existe de nombreuses applications (en géophysique, en mécanique des fluides, en imagerie médicale, etc...) pour lesquelles le calcul sur réseau de stations permet d'accroître les performances. Enfin, ces outils sont relativement simples à mettre en œuvre dans des environnements que nous pratiquons tous les jours (celui de nos stations de travail). Ils nous permettent de nous préparer à l'utilisation des supercalculateurs massivement parallèles qui devraient être opérationnels d'ici un an ou deux et offrir des puissances de calculs de l'ordre du Tera-flops. Nous pourrons alors envisager des simulations beaucoup plus complexes.

L'utilisation de notre programme nous permet d'utiliser l'espace mémoire et la puissance de calcul du réseau de stations de l'Observatoire de Grenoble. Il a contribué de façon significative à notre progression dans la compréhension du phénomène de la formation des étoiles en nous permettant d'obtenir les indications les plus directes à ce jour de la présence des disques entourant les étoiles jeunes et des informations sur leur géométrie. De plus, la modélisation des distributions de polarisation met en évidence l'absence probable de disque entourant un sous-groupe d'étoiles jeunes, celui des étoiles T Tauri à émission faible. Ce résultat suggère que la formation des disques autour des étoiles jeunes n'est pas universelle. Il est alors très tentant de faire l'analogie avec notre propre système solaire où seules quelques planètes possèdent des anneaux (ayant de plus des masses très variables).

## 6.3 Modélisation de l'effet Compton

### 6.3.1 Introduction

Le projet développé a pour objectif la prise en compte, dans la reconstruction d'images médicales tomoscintigraphiques 3D, d'un phénomène physique majoritairement perturbateur : l'effet Compton. La tomoscintigraphie est une technique d'imagerie nucléaire médicale qui permet de visualiser certains aspects fonctionnels de différents organes du corps humain [TDDJ63]. On injecte au patient des éléments radioactifs appelés marqueurs qui se fixent sur certaines cibles prédéfinies. On enregistre ensuite des projections de l'activité nucléaire sur des capteurs. Une modélisation mathématique de la mesure, basée sur l'hypothèse de trajectoire rectiligne des photons émis, permet de fournir à partir des données une estimation de la distribution des marqueurs (étape d'inversion ou de reconstruction) et donc de l'activité des éléments biologiques marqués.

Plusieurs obstacles s'opposent à une diminution des flous d'images médicales tomoscintigraphiques 3D, en particulier l'atténuation (*certains photons sont absorbés par la matière qu'ils traversent*) et l'effet Compton (*certains photons sont déviés lors de leur interaction avec la matière*). C'est ce dernier point qui nous intéresse plus particulièrement.

Les méthodes mathématiques classiques de reconstruction sont bi-dimensionnelles et basées sur l'inversion de la transformée de Radon (méthodes analogues à celles utilisées pour les scanners). Malheureusement, l'effet Compton perturbe la trajectoire des photons qui ont une probabilité non négligeable d'être déviés de leur trajectoire. Ainsi la formulation du problème devient tridimensionnelle. Elle nécessite le calcul de la probabilité qu'un photon émis par un point de l'espace soit enregistré sur un capteur particulier. Comme cette probabilité dépend de la densité des tissus traversés par les photons, il faut utiliser des images de scanner classiques mises en correspondance avec les informations scintigraphiques.

Plusieurs solutions ont été proposées pour prendre en compte l'effet Compton. Nous pouvons soustraire aux images obtenues le résultat d'une simulation de Monte Carlo de la diffusion [LS90]. L'atténuation et la diffusion peuvent être modélisées sous forme d'une convolution, dont on cherchera à corriger l'effet par déconvolution. Enfin, les méthodes de « recombinaison » proposent une analyse statistique (analyse en composantes principales ou analyse factorielle) de toute l'information disponible, qui permet d'identifier les composantes liées au Compton et de les éliminer.

La modélisation qui a été développée consiste à calculer la probabilité qu'un effet Compton se produise grâce à la connaissance de la densité électronique du milieu. Cette modélisation nécessite beaucoup de calculs et l'implémentation de celle-ci est devenue possible sur des machines parallèles.

### 6.3.2 Interactions des photons et de la matière

Dans le domaine considéré en radiothérapie, l'effet Compton est l'interaction la plus fréquemment observée. Ce phénomène physique introduit un bruit dans les images de médecine nucléaire et constitue la première source d'erreurs. Il se produit entre un photon et un électron planétaire une interaction. On admet que tous les électrons du milieu sont des électrons libres et on fait abstraction de l'énergie de liaison entre les électrons et leurs noyaux, l'énergie du photon étant bien supérieure à cette dernière.

Un photon incident d'énergie E entre en collision avec un électron et projette ce dernier en lui abandonnant sous forme d'énergie cinétique une certaine fraction de son énergie (voir figure 6.4). Le photon est, après l'interaction, dévié d'un angle $\theta$ par rapport à sa trajectoire initiale (*photon diffusé*) ; la trajectoire de l'électron projeté fait un angle $\phi$ avec cette même direction.

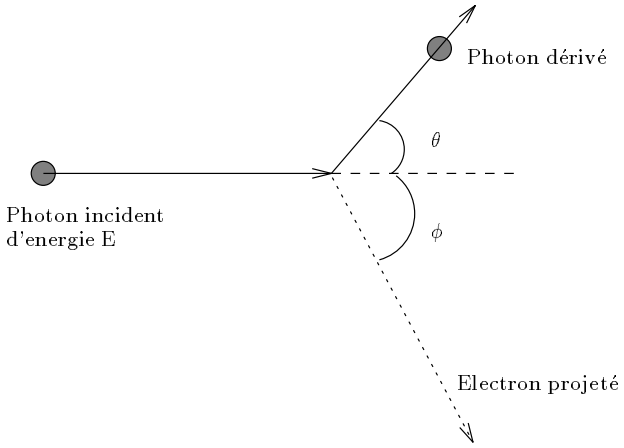



FIG. *6.4 - : Effet Compton*

La modélisation consiste donc à discrétiser le corps à observer en petits éléments de volume appelés « voxels » et on cherche à calculer la trajectoire d'un photon et donc l'ensemble des voxels ayant subi un choc Compton. Il nous faut prendre en compte toutes les directions possibles de propagation pour un photon et, pour chacune d'entre elles, nous devons calculer pour chaque point la probabilité de subir un choc Compton sous tous les angles possibles, ces calculs devant modéliser l'avancée du photon dans le milieu et dans toutes les directions de propagation possibles.

### 6.3.3 La parallélisation

Le coût très élevé des calculs nécessite le recours au parallélisme pour la mise en œuvre informatique. En effet, pour calculer la probabilité qu'un photon émis en

un point $(x, y, z)$ de l'espace soit détecté au pixel $(u, v)$ de la caméra (ou capteur) positionnée à l'angle $\theta$, il faut de l'ordre de $N^7$ évaluations de la formule de Klein Nishina [TDDJ63, Eva85], où $N$ désigne, pour simplifier, le nombre de pas de discrétisation d'espace, le nombre de pixels dans chaque direction du capteur et le nombre de positions de ce capteur.

Considérons n voxels et n positions de capteurs.

L'algorithme va donc être constitué d'une série de boucles imbriquées :

- trois boucles imbriquées pour décrire l'ensemble des voxels (une pour $x$, une pour $y$ et enfin une pour $z$) entraînent $O(n^3)$ calculs ;
- puis, pour chaque voxel, on discrétise selon les directions dans l'espace depuis un voxel, ce qui fait deux boucles imbriquées (une sur $\theta$ et une sur $\phi$, angles dans le système de coordonnées sphériques) qui ont un coût en $O(n^2)$ ;
- et enfin chaque direction est discrétisée, afin de progresser pas par pas, ce qui fait une boucle sur le pas, suivie du calcul de la contribution sur chaque capteur (voir figure 6.5). Cette partie a un coût en $O(n^2)$.

  Nous obtenons donc bien $O(n^3 \times n^2 \times n^2) \; = \; O(n^7)$

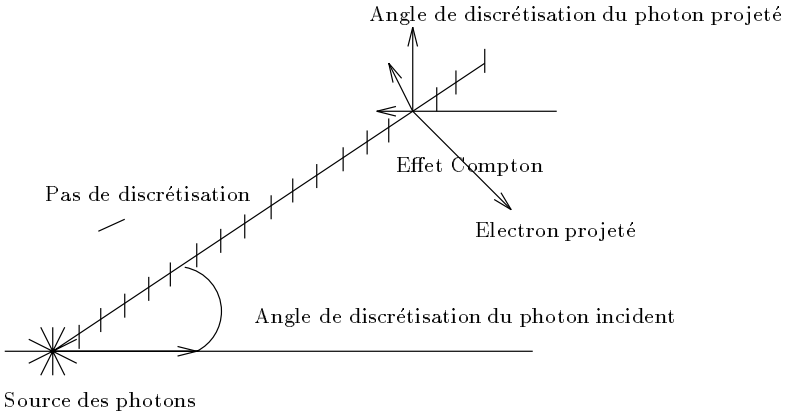



FIG. *6.5 - : Effet Compton.*

Nous obtenons ainsi une matrice reliant les valeur des $N^2$ pixels dans les $N$ positions de la caméra aux $N^3$ voxels d'espace. Pour $N = 64$, 69 Giga-mots sont nécessaires pour stocker la matrice de modélisation du phénomène, et 4 Téra-opérations complexes. Des simplifications du modèle, basées sur des considérations physiques (on ne considère qu'un angle limité de diffusion par exemple) sont envisagées. On peut estimer réduire de l'ordre de 100 la complexité et la taille mémoire nécessaire grâce à ces simplifications.

La parallélisation s'effectue selon le même schéma que l'application précédente. Une tâche maître va distribuer aux tâches de calcul les trajectoires initiales des photons. Les voxels sont répartis et les résultats sont envoyés à la tâche maître qui réalise le traitement final avant la création de l'image. De plus, nous avons bénéficié de l'expérience acquise avec la parallélisation de l'application de modélisation d'étoiles pour mettre en œuvre le même algorithme d'allocation dynamique de la charge et de gestion des pannes du réseau de stations.

### 6.3.4 Résultats

L'algorithme a été implémenté sur trois réseaux différents. Le premier réseau, noté SUN, est constitué de deux SUN4, deux SUN SPARC-ELC et d'un SUN SPARC-10. La performance crête de ce réseau est de 120 Mflops. Le second, noté DEC, est constitué d'une station Alpha-3000 et de 14 stations de travail Digital Ultrix (7 de type 2100, 2 de type 3100 et 5 de type 5000). La puissance de crête de ce réseau est de 300 Mflops. Enfin le dernier réseau, appelé IBM, est constitué de 5 stations de travail IBM RS6000 (1 de type 580, 1 de type 560 et 3 de type 320). Ce dernier réseau a une puissance de crête de 430 Mflops.

| N | Sparc-10 | SUN | Alpha | DEC | 580 | IBM |
|---|---|---|---|---|---|---|
| 4 | 21 s | 11 s | 8 s | 13 s | 10 s | 6 s |
| 6 | 333 s | 143 s | 138 s | 100 s | 169 s | 66 s |
| 8 | 2667 s | 1113 s | 989 s | 640 s | 1290 s | 456 s |

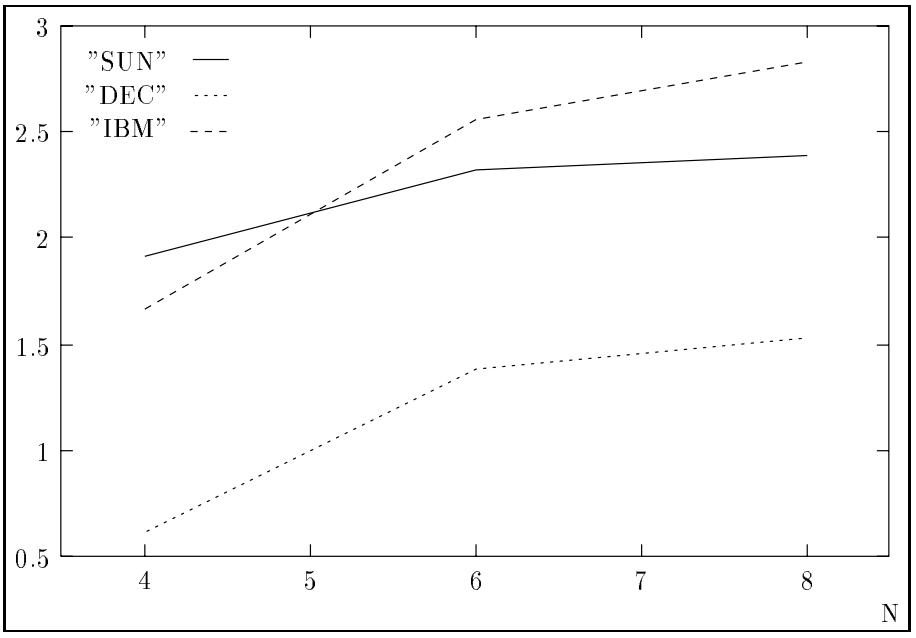



FIG. 6.6 - : *Facteur d'accélération en fonction de N.*

Le coût des communications dans cet algorithme est très important, ce qui explique les mauvaises performances obtenues sur les différents réseaux de stations de travail. En effet ces ordinateurs communiquent grâce à des réseaux à faible débit et déjà chargés.

Une implémentation de cet algorithme a été réalisée sur une machine SIMD à grain fin : la MasPar [Bla91]. La parallélisation s'applique sur les éléments des différentes matrices et sur les boucles qui sont réparties entre les processeurs [BR93]. La plupart des communications se font de voisin à voisin, ce qui permet d'avoir de bons résultats.

| N | MasPar | DEC |
|---|---|---|
| 4 | 7 s | 13 s |
| 6 | 35 s | 100 s |
| 8 | 226 s | 640 |

Nous pensons qu'avec la mise en œuvre des méthodes de recouvrements calcul/communication, l'utilisation des BLAS distribuées et l'implémentation sur machine IBM SP1 ou Cray T3D, nous obtiendrons de bonnes performances et nous pourrons traiter de plus grosses modélisations.

### 6.3.5 Conclusion

Une mise en œuvre informatique sur une machine massivement parallèle MasPar composée de 8192 processeurs ainsi qu'une implémentation sur un réseau de stations sous PVM ont été réalisées pour $N = 8$. Une nouvelle version en développement utilise les fonctions BLAS distribuées de la bibliothèque PUMMA (voir chapitre 4.7) lors de la mise en œuvre de la résolution d'un système linéaire creux par la méthode du gradient conjugué. Avec cette évolution du code parallèle, nous espérons réaliser des expériences avec $N = 16$ sur une machine SP1 à 32 processeurs. De plus, l'accès à une machine telle que le CRAY T3D nous permettra d'envisager des calculs pour $N = 32$, voire $N = 64$ avec des simplifications.

## 6.4 Conlusion

L'expérience acquise nous permet de participer au développement d'une autre application sur la dynamique moléculaire (simulation du mouvement des atomes et des molécules). La détermination des déplacements des atomes nécessite le calcul des forces d'interactions entre les atomes d'une même molécule et les interactions non liées de Van der Waals et de Coulomb. Le coût de ces calculs est réellement important et la parallélisation s'avère indispensable pour obtenir une modélisation significative de la réalité. En effet, sur des machines classiques, les

modèles utilisés ne sont pas assez complets, c'est-à-dire que le nombre de molécules est limité par les temps de calcul trop importants. Ce projet est dirigé par Helmut Heller[3] et Yves Chapron[4].

Nous avons également commencé le développement d'une application dédiée à l'étude du mouvement des liquides à l'intérieur d'un réacteur d'une centrale nucléaire. Ce projet dirigé par Bernard Brund[5] a comme objectif de traiter des problèmes résolus par une méthode d'éléments finis avec dix millions de mailles. Cette application sera développée sur le Cray-T3D car seul ce dernier peut fournir assez de mémoire et de puissance de calcul.

La parallélisation apparaît nécessaire pour la mise en œuvre de modélisations de plus en plus complètes. De plus en plus de chercheurs de la communauté scientifique s'intéressent au parallélisme et il faut donc fournir à ces nouveaux utilisateurs toutes les méthodes et les outils afin qu'ils puissent développer leurs applications le plus facilement possible tout en conservant de bonnes performances.

[3]München University
[4]CEA-Grenoble
[5]CEA-Grenoble

# Chapitre 7

# Conclusion et perspectives

Nous avons présenté dans ce documment une série d'études sur des méthodes et des outils qui ont beaucoup d'importance pour le développement des applications parallèles sur des machines à mémoire distribuée.

Il est évident que de nombreuses études restent encore à faire autour des sujets abordés dans ce documment. Nous allons donc résumer les différents chapitres et proposer des travaux qui peuvent être effectués par la suite.

### Outils de programmation de machine parallèle

Les deux bibliothèques de fonctions de communication présentées ne sont que deux implantations du modèle de programmation par processus communicants. PVM est celle qui nous semble la plus utilisée. En effet, comme nous l'avons vu, elle est disponible sur la plupart des machines parallèles et peut être utilisée sur des réseaux d'ordinateurs, ce qui permet de développer des applications même si l'accès à une machine parallèle est difficile. La bibliothèque MPI est beaucoup plus récente, puisque la première implémentation date du début de l'année 1994. Elle propose des fonctions de communication plus complexes. En fait, nous pouvons considérer que MPI représente une « généralisation de PVM » , notamment dans les concepts de communications globales, de groupes de processus, de types structurés ou de topologies virtuelles.

Néanmoins, PVM continue à évoluer concurremment à l'apparition de MPI et la dernière version, PVM 3.3, complète la version précédente en fournissant des opérations globales de communication et de calcul, ainsi que des communications point à point véritablement asynchrones [BDG+94, Man94].

Malheureusement, il existe encore de nombreuses possibilités qui ne sont pas fournies par ces bibliothèques mais qui sont nécessaires pour le développement des applications sur des machines parallèles à mémoire distribuée. Par exemple, il faudrait des fonctions permettant d'effectuer des entrées/sorties en parallèle ainsi que des fonctions de communications globales non-bloquantes afin d'utiliser plus facilement les méthodes de recouvrement calcul/communication. Actuellement,

nous essayons de développer ces fonctions sur le Cray T3D car ces dernières ne sont pas disponibles, or elles semblent indispensables pour les applications que nous avons à développer au CEA.

### Efficacité sur réseau de processeurs hétérogènes

Une des caractéristiques connues des deux bibliothèques de communication PVM et MPI que nous avons étudiées est qu'elles proposent une portabilité des programmes sur la plus grande majorité des machines parallèles, mais aussi sur des réseaux d'ordinateurs hétérogènes. Ceci nous a amené à adapter le calcul du facteur d'accélération et de l'efficacité afin de pouvoir évaluer les performance d'un algorithme sur un réseau de processeurs hétérogènes. L'utilisation de ces deux outils permet, par exemple, l'étude de la *scalability*, c'est-à-dire de l'évolution des performances d'un algorithme en fonction du nombre de processeurs ou, pour le cas hétérogène, en fonction de la puissance relative du réseau. Il faut maintenant utiliser ces fonctions comme base pour une étude plus précise qui tiendrait compte des problèmes mémoire et de séquentialité des programmes.

### Les bibliothèques vectorielles et parallèles d'algèbre linéaire

Les bibliothèques d'algèbre linéaire commencent à se développer pour les machines parallèles. ScaLAPACK a été créée pour obtenir le maximum de performances tout en restant portable sur tous les ordinateurs du marché. La solution adoptée est l'utilisation d'un placement des données « dispersées » par blocs, qui permet de réduire en moyenne le coût des communications. Des études et implémentations doivent être effectuées au niveau de nouvelles fonctions de résolution de systèmes linéaires creux et de la possibilité de calculer et d'envoyer ou de recevoir des messages en parallèle. Dans le cadre du développement d'un code de simulation des mouvements des fluides d'un réacteur nucléaire, il est absolument nécessaire d'avoir des fonctions de résolution différentes suivant la structure des données et le nombre de processeurs. Il est donc intéressant de développer et d'intégrer ces fonctions dans la bibliothèque ScaLAPACK.

### Vers une programmation plus efficace

L'étude des bibliothèques numériques, comme ScaLAPACK, a amené à développer une méthode assez étudiée de manière théorique, mais peu utilisée en pratique pour augmenter les performances des fonctions de ces bibliothèques : le recouvrement calcul/communication. Cette méthode donne de bons résultats et elle peut être mise en œuvre sur la plupart des machines parallèles car ces dernières possèdent un processeur de calcul et un circuit dédié à la gestion des communications. Nous avons commencé une collaboration avec Frédéric Desprez [1]

[1] LAboratoire Bordelais de Recherche en Informatique

pour développer une bibliothèque portable de fonctions permettant une programmation plus facile de ces méthodes.

### Perspectives

Pour que les machines parallèles puissent entrer dans le monde industriel ou, en d'autres termes, pour qu'elle soient utilisées comme outils de production, il faut que la facilité d'utilisation et les performances soient les deux principales caractéristiques de ces machines. Si les performances sont présentes tant sur le plan de la capacité et de la rapidité d'accès mémoire que de la puissance de calcul, leur programmation n'est pas encore à la portée du programmeur de machines vectorielles qui représentera le principal utilisateur des machines parallèles. Les bibliothèques représentent donc la solution à moyen terme pour obtenir la facilité de programmation et des performances satisfaisantes.

Pour le plus long terme, le compilateur-paralléliseur devrait être l'outil qui permettra la « vulgarisation » de la programmation pour les machines parallèles. C'est ce dernier qui décidera alors de remplacer une partie du code par un appel à des fonctions de bibliothèques spécialisées permettant d'optimiser les communications et la répartition de la charge de calcul.

# Bibliographie

[A+92] E. Anderson et al. *LAPACK Users'Guide*. SIAM Philadelphia, 1992.

[ABB+90] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. Lapack : A portable linear algebra library for high-performance computers. Working note, Argonne National Laboratory, May 1990. LAPACK Working Note 20.

[ABD+90] E. Anderson, C. Bischof, J. Demmel, J.J. Dongarra, J. DuCroz, S. Hammarling, and W. Kahan. Prospectus for an Extension to LAPACK. Working Note ANL-90-118, Argonne National Laboratory, November 1990. LAPACK Working Note 26.

[ABD+91a] E. Anderson, A. Benzoni, J.J. Dongarra, J. DuCroz, B. Tourancheau, and R. Van de Geijn. Basic Linear Algebra Communication Subprograms. In *In Sixth Distributed Memory Computing Conference*, pages 287–290. IEEE Computer Society Press, April 1991.

[ABD+91b] E. Anderson, A. Benzoni, J.J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. Van De Geijn. Lapack for distributed memory architectures. Progress report, Argonne National Laboratory, 1991.

[ABM+92] M.J. Atallah, C.L. Black, D.C. Marinescu, H.J. Siegel, and T.L. Casavant. Models and Algorithms for coscheduling compute-intensive tasks on a network of workstations. *J. Parallel Distrib. Comput.*, 16, 1992.

[AD90] Edward Anderson and Jack Dongarrra. Evaluating block algorithm variants in lapack. Working note, Argonne National Laboratory, April 1990. LAPACK Working Note 19.

[Amd67] G.M. Amdahl. Validity of the single-processor approach to achieving large scal computing capabilities. In D.J. Evans, G.R. Joubert, and H. Liddell, editors, *IAFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS press, 1967.

[ATK92] A.L. Ananda, B.H. Tay, and E.K. Koh. A Survey of Asynchronous Remote Procedure Calls. *ACM Operating Systems Review*, 26(2), April 1992.

[BBB+94] F. Berthommier, V. Bouchard, L. Brunie, L. Colombet, P. Cinquin, and All. Medical imaging and modeling using maspar. In *Parallel Computing: Tends and Applications*, 1994.

[BD74] J. Boyle and K. Dritz. An automated programming system to aid the development of quality mathematical software. In North Holland, editor, *IFIP Proceedings*, pages 542–546, 1974.

[BDD+88] C. Bischof, J. Demmel, J.J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, and D.C. Sorensen. Provisional Contents. Working Note ANL-88-38, Argonne National Laboratory, September 1988. LAPACK Working Note 5.

[BDG+92] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A User's Guide to PVM Parallel Virtual Machine. Technical report, Oak Ridge National Laboratory, September 1992.

[BDG+94] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A User's Guide to PVM Parallel Virtual Machine (version 3). Technical report, Oak Ridge National Laboratory, May 1994.

[Bis88] C. Bischof. Fundamental linear algebra computations on high-performances computers. Technical report, Argonne National Laboratory, 1988.

[BL92] R. Butler and E. Lusk. User's Guide to the P4 Programming System. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.

[BL93] D.K. Bradley and J.L. Larson. A parallelism-based analytic approach to performance evaluation using application programs. *Proc. IEEE*, 81(8):1126–1135, 1993.

[Bla91] T. Blank. The MasPar MP-1. *IEEE Computers*, pages 20–24, 1991.

[BM88] P. Bastien and F. Ménard. On the Interpretation of Polarization Maps of Young Stellar Objects. *Ap. J.*, 326:334–338, 1988.

[BM90] P. Bastien and F. Ménard. Parameters of Disks Around Young Stellar Objects from Polarization Observations. *Ap. J.*, 364:232–241, 1990.

[BR93] V. Bouchard and S. Rouault. Amélioration des images scintigraphiques. Master's thesis, ENSIMAG, 1993.

[BT89] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation - Numerical Methods*. Prentice Hall, 1989.

[CCD+94a] C. Calvin, L. Colombet, F. Desprez, B. Jargot, P. Michallon, B. Tourancheau, and D. Trystram. Towards Mixed Computation/Communication in Parallel Scientific Libraries. In *Proceedinfs of the "Conference on Parallel Computing" (CONPAR'94) - Linz - Austria*, 1994.

[CCD+94b] C. Calvin, L. Colombet, F. Desprez, B. Jargot, P. Michallon, B. Tourancheau, and D. Trystram. Overlapped Communication in Scientific Libraries. In *proceedings of "The 1994 Scalalable High Performance Computing Conference" (SHPCC'94) - Knoxville - Tenessee - USA*, May 1994.

[CD93] C. Calvin and F. Desprez. Minimizing communication overhead using pipeline for multi-dimensional FFT on distributed memory machines. In *Parallel Computing 93*, 1993.

[CD94] L. Colombet and L. Desbat. Speedup and Efficiency of large size applications on Heterogeneous Network. *Integrated Computer-Aided Engineering Journal*, 1994.

[CDG+93] L. Colombet, L. Desbat, L. Gauthier, F. Ménard, Y. Tremolet, and D. Trystram. Scientific and Industrial Experiments Using PVM. Preprint, LMC-IMAG, 46 av. Félix Viallet, 38031 Grenoble Cedex France, January 1993.

[CDM93] L. Colombet, L. Desbat, and F. Ménard. Star Modeling on IBM RS6000 Networks using PVM. In *The 2nd International Symposium on High Performance Distributed Computing*, 1993.

[CDW93a] J. Choi, J. Dongarra, and D. Walker. Parallel Matrix Transpose Algorithms on Distrubuted Memory Concurrent Computers. Technical Report ORNL/TM-12309, Oak Ridge National Laboratory, October 1993.

[CDW93b] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: parallel universal matrix multiplication algorithms on distributed concurrent computers. Technical Report ORNL/TM-12252, Oak Ridge National Laboratory (USA), April 1993.

[CDW93c] J. Choi, J.J. Dongarra, and D.W. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 3–15. Elsevier Science Publishers, 1993.

[CHHWar] R. Calkin, R. Hempel, H-C. Hoppe, and P. Wypior. Portable Programming with the Parmacs Message-Passing Library. *Parallel Computing, special issue on message-passing interfaces*, to appear.

[CMT94] L. Colombet, P. Michallon, and D. Trystram. Parallel Matrice-Vector Product on Ring with a minimum of Communication. Technical report, LMC-INPG, 1994.

[Cor91] Intel Corporation. *PARAGON XP/S, product overview*, 1991.

[CR94] Inc. Cray Research. *Cray T3D*, 1994.

[CS92] C.H. Cap and V. Strumpen. The Parform: A High Performance Platform for Parallel Computing in Distributed Workstation Environment. Technical report, Institut für Informatik, Universität Zürich, 1992.

[CT93] M. Cosnard and D. Trystram. *Algorithmes et Architectures Parallèles*. InterEditions, iia edition, 1993.

[D+91] J.J. Dongarra et al. *A Users' Guide to PVM Parallel Virtual Machine*. Oak Ridge National Laboratory, July 1991.

[DDT93] D. Delesalle, L. Desbat, and D. Trystram. Resolution de grands systemes lineaires creux par methodes it eratives paralleles. *RAIRO-M2AN*, 27(6):651–671, 1993.

[Des94] F. Desprez. *Procédures de base pour le calcul scientifique sur machines paralleèles à mémoire distribuée*. PhD thesis, Institut National Polytechnique de Grenoble, January 1994.

[DMBS79] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK users'guide*. SIAM Philadelphia, 1979.

[Don88] J.J. Dongarra. The LINPACK benchmark: An explanation. In Supercomputing, editor, *1st International Conference Proceedings*, pages 456–474, 1988.

[Don92] J.J. Dongarra. Performance of various computer using standard linear equations software. Technical report, Oak Ridge National Laboratory, e-mail: netlib@ornl.gov, 1992.

[DPW93] J.J. Dongarra, R. Pozo, and D.W. Walker. An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architecture. Technical report, Oak Ridge National Laboratory, 1993.

[dR94] Jean de Rumeur. *Communications dans les réseaux de processeurs*. Collection ERI. Masson, 1994.

[DT93] F. Desprez and B. Tourancheau. LOCCS low overhead communication and computation subroutines. In *High Performance Computing and Networking conference*, 1993.

[Eva85] R.D. Evans. *Encyclopedia of physics (Compton effect)* . S. Flugge, 1985.

[Fis91] Dietrich Fischer. On superlinear speedups. *Parallel Computing*, 17:695–697, 1991.

[Fly79] M.J. Flynn. Some computer organization and their effectiveness. *IEEE Transaction on Computer*, pages 948–960, 1979.

[FP92] T.L. Freeman and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

[GHPW90] G.A. Geist, M.T. Hetah, B.W. Peyton, and P.H. Worley. PICL: A Portable Instrumented Communication Library. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computation.* The John Hopkins University Press, 1989. Second edition.

[Gus88] J.L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31, 1988.

[HJ88] R. Hockney and C. Jesshope. *Parallel Computer 2.* Adam Hilger, Bristol and Philadelphia, 1988.

[HPF93] HPF Forum. *High Performace Fortran Language Specification*, Version 1.0, May 1993.

[JH89] S. L. Johnsson and C.T. Ho. Optimum Broadcasting and Personnalized Communications in Hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1267, September 1989.

[Joh87] S.L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, pages 133–172, 1987.

[Kal92] L.V. Kale. The CHARM (3.2) Programming Language Manual. Technical report, University of Illinois at Urbana Champaign, (e-mail: kale@cs.uiuc.edu), December 1992.

[KCN88] C. T. King, W. H. Chu, and L. M. Ni. Pipelined data parallel algorithms - concept and modeling. In *International Conference on Supercomputing*, pages 385–395, 1988.

[Knu81] D.E. Knuth. *The art of computer Programming*, volume 2. Addison-Wesley, 1981.

[Lei85] C.E. Leiserson. Fat-tree: universal networks for hardware-efficient supercomputing. *IEEE Transaction on Computer*, 34(10):892–901, October 1985.

[Lei92] C.E. Leiserson. The network of the CM-5. In *SPAA 92*, 1992.

[LER92] T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing.* Prentice-Hall, 1992.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Software*, 5(3):308–371, 1979.

[LL94] LGI and LMC. Rapport APACHE. Technical Report 1, LGI-IMAG and LMC-IMAG, 1994.

[LS90] M. Ljungberg and S.E. Strand. Scatter and attenuation correction in S.P.E.C.T. using density maps and Monte Carlo simulated scatter functions. *J. Nucl. Med.*, 31, 1990.

[Mén89] F. Ménard. *Etude de la polarisation causée par des grains dans les enveloppes circumstellaires denses.* PhD thesis, Univ. of Montréal, 1989.

[Man94] Robert J. Manchek. *Design and Iplementation of PVM version 3.* PhD thesis, University of Tennessee, Knoxville, May 1994.

[MC83] C. Mead and L. Conway. *Introduction aux systemes VLSI.* InterEditions, Lavoisier, 1983.

[MC92] Thinking Machines Corporation. *CM-5*, 1992.

[Mes93] Message Passing Interface Forum. Document for a Standard Message-Passing Interface, November 1993.

[Mic94] P. Michallon. Etude des performances du paragon. rapport interne, Etablissement Technique Cental de l'Armement, Site Expérimental en Hyperparallélisme, 1994.

[MPIF94] Message Passing Interface Forum. Document faor a Standard Message-Passing Interface. Technical report, European MPI Workshop, 1994.

[Phi91] C. Phillips. The performance of the blas and lapack on shared memory scalar multiprocessor. *Parallel Computing*, 17:751–761, 1991.

[RS91] W. Richard Stevens. *UNIX Network Programming*. Prentice All, 1991.

[SDB94] A Skjellum, N. Doss, and P Bangalore. Writing Libraies in MPI. Technical report, CSD and NSF ERCCFS - Mississipi State University, Mississipi Sate University PO Box 6176 - Mississipi State, MS 39762, March 1994.

[SG91] X.H. Sun and J.L. Gustafson. Torward a better parallel performance metric. *Parallel Computing*, 17:1093–1110, 1991.

[SS89] Y. Saad and M. Schultz. Data communication in parallel Architectures. *Parallel Computing*, pages 131–150, 1989.

[Sto87] H.S. Stone. *High-Performance Computer Architecture*. Addison&Wesley, 1987.

[SW90] Q. Stout and B. Wagar. Intensive hypercube communication, prearranged communications in link-bound machines. *Journal of Parallel and Distributed Computing*, 10:167–181, 1990.

[TDDJ63] M. Tubiana, J. Dutreix, A. Dutreix, and P. Jockey. *Bases physiques de la radiothérapie et de la radiobiologie*. Masson et Cie., 1963.

[The93] The MPI Forum. MPI: A Message Passing Interface. Technical report, University of Tenessee, Knoxville, 1993.

[TvR85] Andrew S. Tanenbaum and Robbert van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4), December 1985.