Minimisation du sur-coût des communications dans la parallélisation des algorithmes numériques

Christophe Calvin

Thèse

présentée par

Christophe CALVIN

pour obtenir le grade de Docteur

de l'Institut National Polytechnique de Grenoble

(arrêté ministériel du 30 mars 1992)

Spécialité : Mathématiques Appliquées

Minimisation du sur-coût des communications dans la parallélisation des algorithmes numériques

Date de soutenance : 7 Juillet 1995

Composition du jury

Président :	Jean-Claude BERMOND
Rapporteurs :	Michel COSNARD
	Joseph PETERS
Examinateurs :	Brigitte PLATEAU
	Patrick WITOMSKI
	Denis TRYSTRAM

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul
(Institut de Mathématiques Appliquées de Grenoble)

Je tiens, tout d'abord, à remercier les membres du jury :

M. Jean-Claude Bermond pour l'honneur qu'il m'a fait en acceptant de présider mon jury de thèse,

M. Michel Cosnard d'avoir touver le temps, dans son emploi du temps extrêmenent chargé, de rapporter mon travail,

M. Joseph Peters pour sa relecture extrêment minutieuse de la thèse ainsi que pour ses nombreuses remarques sur le fond et la forme de mon document en français et sur les articles en anglais,

Mme. Brigitte Plateau pour sa présence à mon jury et son accueil au sein du projet APACHE,

M. Patrick Witomski d'avoir accepté d'être membre du jury, mais aussi de m'avoir accueilli au sein du LMC,

enfin M. Denis Trystram d'avoir été mon directeur de thèse pendant ces trois années.

Je voudrais également remercier tous les personnes du LMC pour leur accueil au sein du laboratoire, plus particulièrement les membres de l'équipe Calcul Parallèle et Calcul Formel. Un merci tout particulier à Jean-Marc pour ses nombreux conseils éclairés en ce qui concerne mes enseignements ainsi que sur la rédaction de ma thèse.

J'aimerai également exprimer ma gratitude, à tous les thésards de l'équipe, les anciens (Phil, Pascal, Lolo) et les présents (Cécile, Nathalie, Yannick, Fred, ...). Un merci tout particulier à mes deux relectrices de choc, Cécile et Nathalie.

Je voudrais également saluer tous les gens avec qui j'ai eu l'occasion de travailler, Tof, Fred, et les Niçois. Encore merci à papa Joe pour son accueil extraodinaire à Vancouver et pour les six semaines que j'ai passé là bas.

Table des matières

Table des figures

Introduction

Le calcul scientifique a connu un développement considérable ces vingt dernières années. Il est, depuis son apparition, un gros consommateur de puissance de calcul et de taille mémoire pour pouvoir traiter des modèles se rapprochant le plus près possible de la réalité. Il nécessite des chercheurs, non seulement capables d'appréhender les phénomènes physiques du problème à résoudre, de comprendre et maîtriser les modèles mathématiques, mais également aptes à concevoir et programmer efficacement les algorithmes de résolution.

Pour répondre aux besoins en puissance de calcul, il a fallu concevoir des machines de plus en plus performantes. Les machines et l'algorithmique numérique parallèles constituent la réponse actuelle à ces besoins. Le parallélisme n'est pas aussi récent en informatique que nous pourrions le penser. Ses premières formes ont vu le jour dans la conception même des microprocesseurs et des opérateurs pipelines de calcul.

Les premières machines parallèles sont plus récentes, mais souvent leurs utilisateurs ne savent pas qu'ils travaillent sur des multi-processeurs. Ces machines sont des super-calculateurs à base d'un petit nombre de processeurs vectoriels très puissants qui se partagent une mémoire commune. La gestion du parallélisme est totalement transparente et assurée par le compilateur et le système d'exploitation.

Ces machines donnant de bonnes performances, leurs concepteurs ont voulu augmenter le parallélisme et ajouter des processeurs. Le problème suivant s'est alors posé : comment gérer un accès concurrent d'un nombre important de processeurs à une mémoire commune? Quelques solutions ont été apportées, mais une réponse à été de détourner le problème en proposant de distribuer physiquement la mémoire. Ce sont les premières machines massivement parallèles à mémoire distribuée. Une des premières est la DAP (*Distributed Array Processing*) au début des années 80, mais les premières études remontent aux années 70 à l'université d'Illinois avec la machine Illiac IV [96, 117].

À partir de ce moment, deux types de machines parallèles ont coexisté pour le calcul scientifique : celles à mémoire partagée et celles à mémoire distribuée, appelées « machines massivement parallèles » (ou MPP pour *Massively Parallel Processing*).

Les principaux avantages de ce type d'architecture sont tout d'abord une façon « simple » d'augmenter la puissance de calcul en ajoutant des processeurs (pour peu que l'on sache les connecter efficacement entre eux) et la possibilité de traiter des problèmes beaucoup plus gros en augmentant la taille mémoire. C'est en effet une des limites des machines à mémoire partagée car l'espace d'adressage est limité. Cette limite n'existe pas (ou existe dans une mesure nettement moins importante) pour les machines à mémoire distribuée, car il est tout à fait possible de disposer d'une mémoire locale de 100 Méga octets (10^8 octets) sur chaque

processeur, et donc d'avoir une mémoire totale de 100 Giga octets (10^{11} octets) si la machine parallèle possède une centaine de processeurs, ce qui est inimaginable pour une machine à mémoire partagée. Ces deux avantages peuvent se résumer sous le terme de « conservation des facteurs d'échelles » (plus connu sous le terme de *scalability* en anglais).

En revanche le très gros inconvénient de ce type de machine est de rendre explicite le parallélisme. L'utilisateur doit exprimer ses programmes de façon parallèle, et/ou d'effectuer les échanges de messages entre les différents processeurs. De ce fait, le parallélisme sur machines à mémoire distribuée n'a pas eu le développement et le succès escomptés au niveau de l'industrie[1]. Ce phénomène a fait récemment l'objet d'une discussion lors d'une conférence internationale à Taiwan intitulée : « Le calcul parallèle : où nous sommes nous trompés ? ». En effet, bien que les plus gros constructeurs de super-calculateurs (comme Cray Research, IBM, Digital Equipment Corporation, Fujitsu, Convex etc.) se soient lancés dans la construction de MPP, on peut être quelque peu déçu du faible développement de codes parallèles au niveau industriel.

À cela, les participants à cette discussion ont trouvé plusieurs raisons[2]. La principale est qu'il n'est pas possible, à l'heure actuelle, à investissement d'effort de développement équivalent, d'égaler les performances d'une machine vectorielle avec une machine parallèle à mémoire distribuée. Certains industriels vont même jusqu'à dire que le calcul parallèle (sur MPP) est un échec.

Ce constat est certainement exagéré. Cependant, il est vrai que la technologie a évolué beaucoup plus rapidement que le logiciel. Nous nous trouvons aujourd'hui avec des machines capables de plusieurs GigaFlops[3] mais avec pratiquement aucune application qui les atteint, faute d'environnement de programmation adéquat. Le calcul massivement parallèle ne se développera véritablement que lorsque nous serons capables de fournir à l'utilisateur les moyens de porter rapidement son code sur une machine massivement parallèle et que ce code, bien évidemment, lui procure de meilleurs performances qu'avec des machines vectorielles classiques.

Il y a dans la phase précédente trois termes clefs : **portabilité, rapidité** et **efficacité**. Cela ne fait pas très longtemps que l'on commence à écrire des programmes parallèles portables, ceci grâce à l'émergence de standards, que cela soit au niveau langage (comme par exemples HPF *High Performance Fortran* [56]) ou au niveau bibliothèque de communication (comme MPI *Message Passing Interface* [95] ou PVM *Parallel Virtual Machine* [11][4]). On peut maintenant écrire des programmes qui peuvent s'exécuter sur la plupart des machines parallèles avec un minimum de changements dans le code[5].

La « rapidité de développement et de codage » est encore de nos jours à l'opposé de la notion d'efficacité du code parallèle produit. Il est faux de penser que l'on peut écrire

1. On commence à peine à entendre parler de portage ou d'écriture de codes de production sur machines parallèles à mémoire distribuée.

2. Résumées par Ken Kennedy dans [85].

3. De l'ordre du milliard d'opérations flottantes par seconde.

4. Nous reviendrons plus en détail sur ces bibliothèques dans la partie I, chapitre 2.

5. Il est cependant évident que pour avoir un code réellement efficace sur une machine, il faudra tenir compte des spécificités de cette machine et de ses composants. Mais ce constat existait avant l'apparition des machines parallèles à mémoire distribuée.

rapidement un code parallèle performant sur MPP. Certaines personnes en déduisent alors que le parallélisme n'est efficace que pour quelques « niches » du calcul scientifique. Ces personnes ont tendance à confondre ce qu'il est possible de faire actuellement en parallélisation « rapide » et les performances que l'on peut atteindre en s'investissant dans la parallélisation du code. Il est vrai que les compilateurs, souvent encore à l'état de prototype de laboratoire, ne produisent pas de codes parallèles véritablement efficaces sauf dans certains cas. Mais ce n'est pas parce que les compilateurs échouent qu'un programmeur ne pourra pas faire mieux.

On peut distinguer deux voies possibles dans le développement d'environnement de programmation parallèle. La première est à long terme : il s'agit de fournir des environnements complets offrant des compilateurs-paralléliseurs, des systèmes d'équilibrage de charge, d'évaluation de performances, etc [105]. Cependant, il faut être conscient que le développement de compilateur-vectoriseur a pris plus de dix ans (et ce n'est pas fini), et le problème de la parallélisation automatique est certainement plus compliqué. La deuxième voie, qui n'est pas à l'opposé de la première, est plutôt une approche bibliothèque : elle consiste à fournir à l'utilisateur le maximum de procédures « prêtes à l'emploi » qui ont été parallélisées de la manière la plus efficace possible[6].

Après un fort développement de bibliothèques séquentielles de calcul numérique (comme les bibliothèques BLAS [89], puis LINPACK [17] et LAPACK [5], pour ne citer qu'elles), sont apparues depuis quelques années les premières bibliothèques parallèles [33] : comme par exemple les bibliothèques ScaLAPACK [32] ou PETSc [66] (et il en existe bien d'autres). Cependant, ces bibliothèques sont toujours constituées de deux entités assez distinctes qui sont, d'un côté une partie calcul pur (souvent basée sur les bibliothèques séquentielles existantes) et une partie communication (plus ou moins optimisée mais nécessairement portable).

En même temps que se développaient ces bibliothèques numériques, sont apparues des bibliothèques de communication que l'on peut regrouper en trois grandes catégories :

1. les bibliothèques de fonctions de **communication optimisées** : elles sont alors non portables. Nous trouvons dans cette catégorie toutes les bibliothèques des constructeurs : NX (Intel), MPL (IBM), ShMem (Cray), etc ou encore le développement de bibliothèques étoffées sur une base existante (comme par exemple la bibliothèque InterCom [9]).

2. les bibliothèques de **communication portables** : PVM [11], MPI [95], P4 [19], Chameleon [67], etc

3. les **versions optimisées des bibliothèques portables** : il s'agit d'implémentations spécifiques efficaces des précédentes bibliothèques sur des machines parallèles, comme par exemple les différentes implémentations de PVM (SP1/SP2 [77], T3D [41]) ou de MPI (sur SP1/2 [61]).

Que cela soit dans l'une ou l'autre des deux approches précédemment citées, il est évident que pour obtenir une parallélisation efficace d'un algorithme, et donc *a fortiori* d'une

6. Pour plus de détails sur les bibliothèques existantes ayant une utilité dans le domaine du calcul parallèle, le lecteur pourra se reporter à deux articles de synthèse : [4, 124].

application, il est nécessaire de minimiser le temps de « non-calcul » induit par la parallélisation. Nous entendons par temps de « non-calcul » tout simplement le temps durant lequel les processeurs ne calculent pas. En effet, si un calcul est effectué de façon concurrente par plusieurs processeurs, il est inévitable qu'à un moment ou à un autre les processeurs vont avoir à communiquer (et ceci quel que soit le mode de communication entre les processeurs : par échanges de messages ou *via* une variable commune dans une mémoire partagée). Par conséquent, le temps d'exécution parallèle n'est pas le temps d'exécution séquentiel divisé par le nombre de processeurs utilisés, mais le temps de calcul, plus un temps de « non-calcul » comprenant le temps de communication, le temps de gestion des messages communiqués ainsi que le temps d'inactivité (comme par exemple le temps d'attente d'une donnée non locale nécessaire à la poursuite de l'exécution.). Nous appellerons dans la suite, par abus de langage, temps de communication, tout ce qui n'est pas du temps utilisé par le processeur pour calculer.

Comment peut-on alors minimiser le temps de communication d'un algorithme parallèle ? Nous pouvons distinguer plusieurs méthodes, qui ne s'excluent pas mutuellement : une première consiste à optimiser l'algorithme de communication lui-même, tandis que les autres tentent de minimiser la part totale du temps de communication par un travail au niveau de l'algorithme parallèle dans son ensemble. Pour cela on peut optimiser le placement des calculs sur les différents processeurs, trouver un « bon ordonnancement » des calculs locaux afin de minimiser les temps d'inactivité, ou encore tenter, de masquer les temps de communication en effectuant des calculs parallèlement.

Le sujet de ce mémoire porte sur la réduction du temps de « non-calcul » dans la parallélisation d'algorithmes numériques. Pour cela nous étudions les moyens possibles offerts à un utilisateur de machine parallèle pour améliorer l'efficacité de la parallélisation de son application numérique. Nous allons donc développer les deux méthodes citées précédemment : d'une part l'optimisation des algorithmes de communication globale, et d'autre part le recouvrement des communications par du calcul.

Avant tout, il nous semble important de préciser le cadre de cette étude. C'est pourquoi nous définissons dans une première partie les modèles utilisés :

- le modèle de machine : nous nous intéressons essentiellement dans ce mémoire aux machines parallèles asynchrones à mémoire distribuée. Nous nous attardons plus particulièrement sur le modèle de communication utilisé par ces dernières car il a bien sûr une grosse influence sur les méthodes mises en œuvre pour réduire le temps de communication;

- le modèle de programmation : nous décrivons plus particulièrement le modèle de programmation par processus communicants, car c'est celui qui est à la base des machines que nous considérons, et de plus, c'est le modèle que nous avons utilisé pour programmer tous les algorithmes.

Dans une deuxième partie nous nous intéressons à la première méthode citée pour minimiser le temps de communication par l'optimisation de l'algorithme de communication lui-même. Il est maintenant reconnu que la parallélisation des algorithmes numériques entraîne

des mouvements de données plus ou moins réguliers, ces mouvements étant réalisés par des schémas de communication impliquant tout, ou une partie des processeurs de la machine [36, 55, 63, 108, 113]. Nous commençons, tout d'abord, par une description des outils les plus utilisés pour optimiser les algorithmes de communication, ceux-ci sont illustrés par quelques exemples d'algorithmes de communication globale. Nous décrivons ensuite, dans les deux chapitres suivants, deux problèmes de communication globale que nous avons étudiés et qui sont : le problème de la transposition de matrices allouées par blocs, et l'échange total en commutation de circuit et *wormhole* dans les réseaux d'interconnexion dont la topologie est un tore à deux dimensions. Nous proposons pour chacun de ces problèmes des algorithmes efficaces que nous comparons aux bornes inférieures et à d'autres algorithmes.

Enfin, dans une troisième partie, nous étudions la deuxième approche pour minimiser le temps de communication qui consiste à masquer ce dernier. Dans un premier chapitre nous faisons une introduction au problème et nous décrivons l'approche adoptée. Celle-ci consiste à identifier des schémas et à utiliser des techniques algorithmiques afin de recouvrir au mieux le temps de communication dans un algorithme numérique parallèle. Afin qu'au moment de l'implémentation de l'algorithme, le masquage des communications se fasse le mieux possible, il est impératif d'avoir une bonne estimation des temps respectifs de calcul et de communication de l'algorithme. Nous décrivons donc, dans le chapitre suivant, une série de tests qui nous ont permis d'évaluer puis de modéliser les performances des communications globales du Cray T3D.

L'utilisation de ces techniques est illustrée dans le chapitre suivant avec l'étude de nouveaux algorithmes de calcul de transformée de Fourier avec recouvrement des communications et utilisation maximale des liens de communication. Ces algorithmes ont été implémentés sur de nombreuses machines parallèles et montrent l'intérêt pratique des méthodes avec masquage des communications.

L'étude et l'emploi de ces différentes méthodes nécessitent en premier lieu une bonne connaissance des algorithmes numériques cibles, mais aussi des compétences en informatique et notamment en parallélisme, afin de mieux appréhender les problèmes (et bien sûr donner des solutions) qui se posent lors de la parallélisation des-dits algorithmes.

Partie I

Modèles

Nous décrivons dans cette partie les différents modèles de machines et de programmation utilisés en parallélisme. Nous nous intéressons plus particulièrement à ceux que nous avons utilisés, à savoir, le modèle de machine parallèle à mémoire distribuée et fonctionnement asynchrone et le modèle de programmation par échange de messages.

Chapitre 1

Modèles de machines

Dans ce chapitre nous présentons le modèle de machine utilisé dans ce mémoire. Il s'agit de machine parallèle à mémoire distribuée à fonctionnement asynchrone. Nous nous attardons sur les modèles de communication de ces machines.

1.1 Classification des machines parallèles

Depuis l'apparition des premiers ordinateurs, la course à la puissance de calcul est devenue rapidement un des objectifs principaux. Ces **super-calculateurs** ont d'abord été des machines à processeurs vectoriels, puis sont nés les premiers ordinateurs à plusieurs processeurs (moins d'une dizaine) toujours vectoriels. C'est au début des années 1980 que l'on a commencé à parler des **machines massivement parallèles à mémoire distribuée**, le nombre d'unités de traitement fonctionnant en parallèle pouvant atteindre plusieurs milliers.

Plusieurs critères existent pour différencier ces machines [76]. La figure 1.1 résume les principaux et donne des exemples de machines appartenant à ces différentes classes.

Le premier critère de différenciation porte sur la mémoire qui peut être soit distribuée, soit partagée. Celui-ci a une conséquence immédiate sur le mode de communication entre des processus distants : elle va se faire soit *via* des variables communes dans la mémoire partagée, soit par échange de messages (*cf* partie I chapitre 2) dans le cas d'une mémoire physiquement distribuée.

Si la mémoire est non partagée, on distingue ensuite le mode de contrôle, qui est soit synchrone, c'est le cas de toutes les machines SIMD [1], soit asynchrone : c'est la classe des machines MIMD [2]. Enfin, le dernier critère que l'on peut considérer, est le mode d'adressage qui est soit global soit local.

1. SIMD : *Single Instruction stream, Multiple Data streams.*
2. MIMD : *Multiple Instruction streams, Multiple Data streams.*

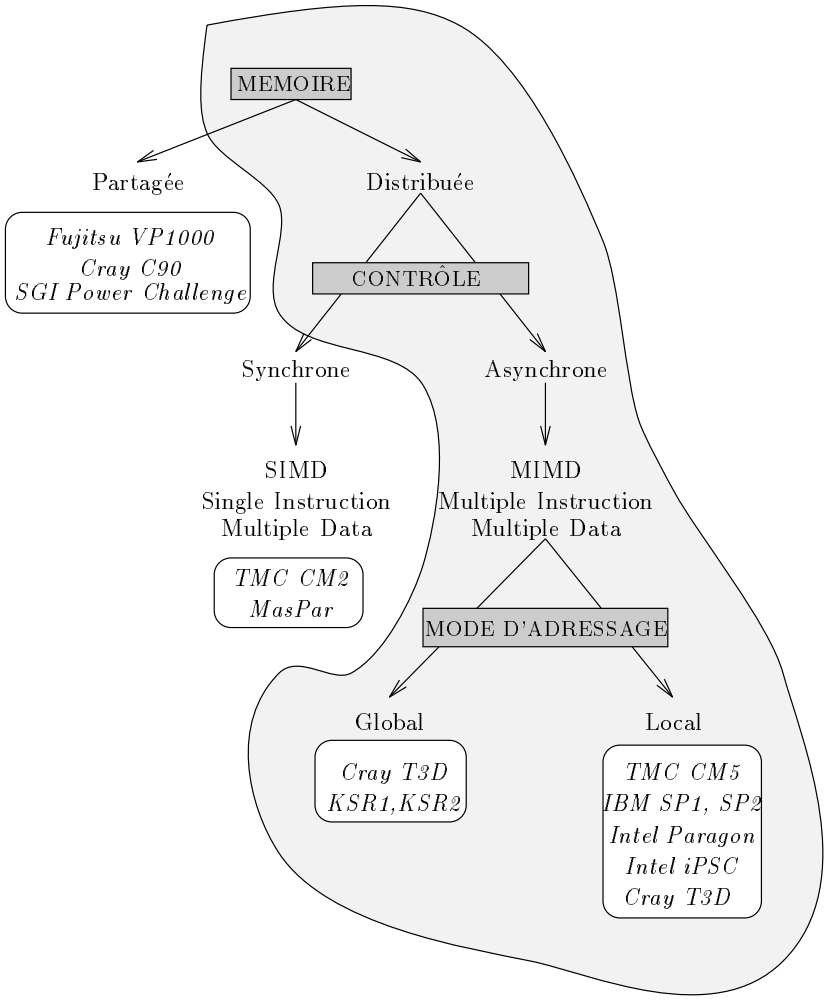


FIG. 1.1 - *Classification des super-calculateurs*

Nous nous intéressons dans ce document aux machines MIMD à mémoire distribuée (patatoïde grisé sur la figure 1.1), c'est-à-dire des machines à plusieurs unités de traitement et de contrôle, possédant chacune une mémoire locale et reliée *via* un réseau d'interconnexion (voir figure 1.2). Ce modèle correspond en effet aux architectures des dernières machines parallèles.

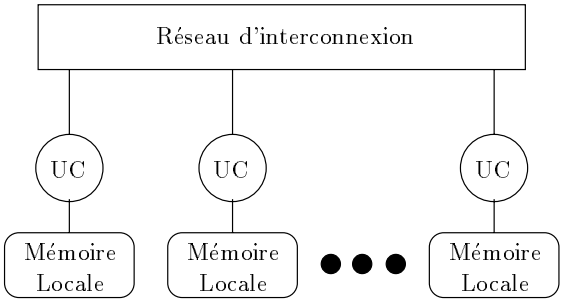


FIG. 1.2 - *Modélisation d'une machine MIMD à mémoire distribuée*

Les critères donnés précédemment sont d'ordre architecturaux. Il faut aussi distinguer les **modèles de programmation** de ces machines qui sont essentiellement au nombre de deux :

Le parallélisme de données : Cela consiste à effectuer des actions successives sur des données parallèles multi-dimensionnelles. Le programme exécuté est soit unique, soit le même pour tous les processeurs. Le séquencement se fait soit au niveau de l'instruction et ce sont alors des programmes de type **SIMD** (c'est-à-dire qu'à chaque top d'horloge, tous les processeurs exécutent la même instruction), soit au niveau du code, et nous avons un programme de type **SPMD** (*Single Program Multiple Data*, c'est-à-dire que le même code est exécuté par tous les processeurs). Dans ce cas le parallélisme est guidé par la **distribution des données**.

Le parallélisme de tâches : Il s'agit de décrire une application en terme de tâches indépendantes (c'est-à-dire qu'elles peuvent s'exécuter de manière concurrente). Ce type de parallélisme est appelé également **parallélisme de contrôle** [105]. Nous nous trouvons en présence ici d'un mode de fonctionnement totalement asynchrone, où plusieurs processus exécutent des codes différents (qui correspondent aux tâches indépendantes) et sont répartis sur tout ou une partie des processeurs de la machine. Il est alors fréquent que plusieurs processus s'exécutent sur un même processeur. Dans ce modèle, le parallélisme est guidé par l'**allocation des tâches aux processeurs**.

Nous nous intéressons, dans notre étude, plutôt au parallélisme de données de type SPMD. C'est en effet celui le plus couramment utilisé en calcul scientifique.

Du fait qu'ils ne partagent pas physiquement la mémoire, les processeurs s'échangent des messages à travers le réseau d'interconnexion pour communiquer. Nous allons maintenant étudier une modélisation des communications des machines MIMD [102], puis donner les définitions des principaux réseaux d'interconnexion.

1.2 Modélisation des communications sur machines MIMD

La figure 1.3 représente une schématisation d'une machine MIMD, et nous avons représenté sur la figure 1.4 une schématisation d'un nœud.

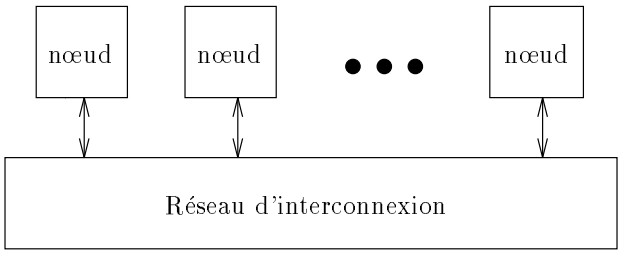


FIG. 1.3 - *Schématisation d'une machine MIMD*

Un nœud d'une machine MIMD est constitué d'une unité de calcul (processeur scalaire, unités vectorielles, ...), d'une mémoire locale et d'un mécanisme de communication. On distingue également les canaux de communication qui relient le routeur au réseau (canaux externes, voir figure 1.4) de ceux qui relient le routeur au processeur et à la mémoire locale (canaux internes, voir figure 1.4).

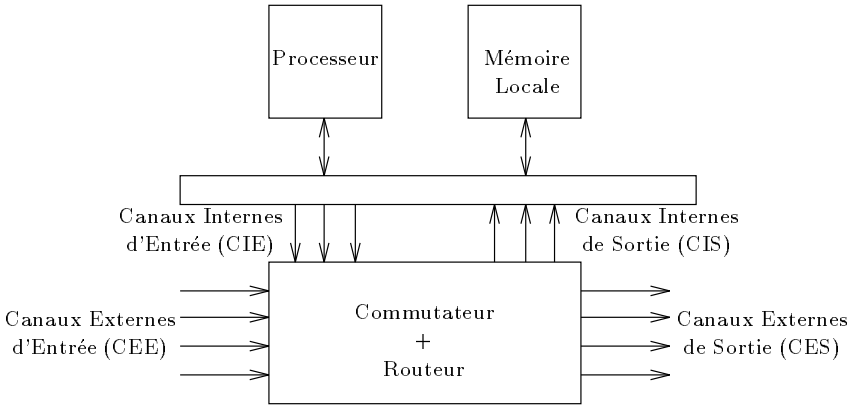


FIG. 1.4 - *Schématisation d'un nœud*

Le mécanisme de communication est généralement constitué d'un **routeur** et d'un système d'interconnexion permettant de relier les canaux entre eux : le **commutateur** (ou *switch*). Le **routeur** est chargé d'acheminer les messages du nœud source au nœud destination. Bien que les fonctionnalités du routeur peuvent être assurées par le processeur lui-même, on trouve dans pratiquement toutes les machines parallèles actuelles, un composant spécifique qui effectue ce travail, permettant ainsi au processeur de se décharger de la communication et d'avoir ainsi des communications véritablement non-bloquantes.

Nous distinguons donc deux entités distinctes qui sont d'un côté le processeur de calcul et la mémoire locale : le **nœud de calcul**, et le mécanisme de communication : le **nœud de communication**.

Nous allons rappeler rapidement les principaux modèles de communication utilisés par la suite dans ce document. Pour plus de précisions, le lecteur pourra se reporter à deux ouvrages [78, 96]. Nous allons, dans un premier temps, citer les principales contraintes dues à la physique des liens et à l'architecture des nœuds, puis nous rappellons quels sont les

principaux modes de commutations utilisés par les machines parallèles pour acheminer les messages.

1.2.1 Les contraintes physiques de communication

Afin de pouvoir concevoir des algorithmes de communication efficaces, il faut tenir compte des contraintes physiques de communication. On considère généralement deux types de restrictions au niveau des canaux externes (*cf* figure 1.4) :

Les contraintes dues aux liens de communication

Lien mono-directionnel Un lien est dit mono-directionnel s'il ne peut être utilisé que dans un seul sens pour communiquer.

Lien bi-directionnel Dans ce cas le lien peut être utilisé dans les deux sens.

Lien *half-duplex* Un seul message peut circuler à la fois sur le lien.

Lien *full-duplex* Deux messages peuvent circuler en même temps sur le même lien, en sens inverse. Dans ce cas, le lien est nécessairement bi-directionnel.

Les contraintes dues au parallélisme des liens

Communication 1-port Parmi les Δ [3] liens existants sur un processeur, un seul peut être utilisé à la fois.

Communication Δ-port Tous les liens existants peuvent être utilisés en même temps.

Communication k-port Seuls k liens parmi les Δ peuvent être utilisés au même moment.

Cependant, il ne suffit pas d'avoir des canaux externes fonctionnant en parallèle pour avoir un modèle Δ-port. En effet, s'il n'existe pas suffisamment de canaux internes alors nous nous retrouvons dans un modèle k-port, voire 1-port (voir figure 1.5) [102]. C'est souvent le cas dans les machines actuelles (hormis les machines à base de transputers qui sont 4-ports et les machines n-Cube), où malgré la présence de plusieurs canaux externes pouvant fonctionner en parallèle, le modèle est 1-port car il n'y a pas assez de canaux internes. Cette restriction est essentiellement due aux problèmes d'accès concurrents à la mémoire locale.

3. Δ représente le degré du nœud, c'est-à-dire le nombre de liens physiques le reliant aux autres nœuds du réseau.

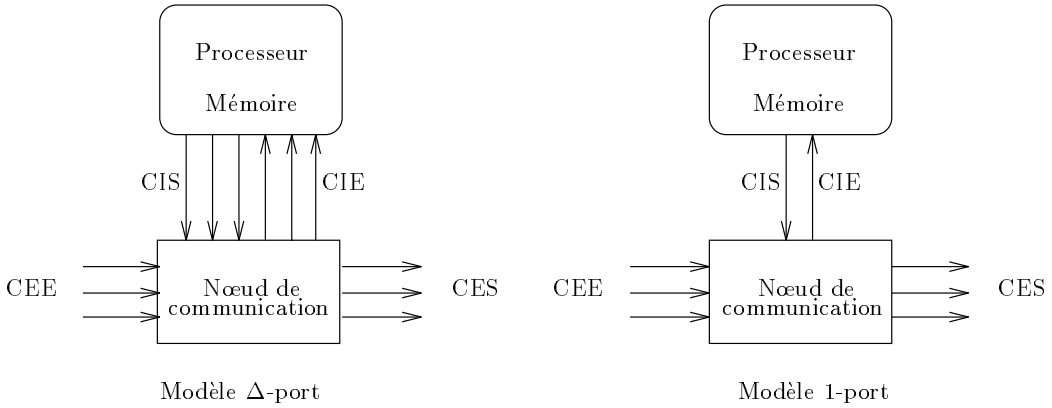


FIG. 1.5 - *Machines 1-port et k-ports*

Le modèle de communication le plus intéressant pour avoir des communications efficaces est un modèle Δ-port, *full-duplex*, c'est-à-dire qu'un processeur peut utiliser tous ses liens en même temps et dans les deux sens.

1.2.2 Les différents modes de commutation

Supposons que l'on désire envoyer un message d'un nœud (appelé émetteur) à un autre (appelé destinataire), plusieurs modes de commutation peuvent être employés [78, 96, 102]:

La commutation de messages (*Store-and-Forward*) Le message est stocké en entier par chaque nœud intermédiaire se trouvant entre les nœuds émetteur et destinataire.

La commutation de circuits (*Circuit-switched*) Dans ce mode, un chemin est tout d'abord établi entre les deux nœuds, ensuite le message est envoyé en une seule fois au nœud destinataire en utilisant le chemin préalablement fixé. Dans ce cas le message n'est pas stocké dans les nœuds intermédiaires. Tant que le message n'est pas arrivé à destination, les liens réservés sont inutilisables pour d'autres communications.

Le *wormhole* [46] Ce mode de routage se rapproche du précédent du fait qu'un chemin est préalablement établi entre les deux nœuds désirant communiquer. Mais dans ce cas le message est découpé en petits paquets, *flits*, qui sont envoyés de façon pipeline sur le chemin fixé. Lorsque le dernier *flit* composant le message a quitté un commutateur le lien est libéré. Si, au cours de son cheminement, un *flit* est bloqué, alors tous les autres paquets restent dans le nœud où ils étaient.

Cette liste des modes de commutation est loin d'être exhaustive, mais elle présente les principaux modes de commutation utilisés par les machines actuelles. Il existe notamment des variantes sur les moyens de régler les conflits d'accès aux liens [96, 102].

1.2.3 Modélisation du temps de communication

Afin d'évaluer le coût d'un algorithme parallèle, il nous faut une modélisation du temps de communication. Nous allons présenter ici le modèle classique du **temps linéaire** qui est

utilisé dans ce document. Il existe d'autres modélisations, comme le modèle temps constant ou des modèles adaptés à certains problèmes [78, 96], mais nous avons préféré utiliser le plus général.

Dans ce modèle, le temps de communication d'un message de longueur L entre deux nœuds directement connectés est le suivant :

$$T = \beta_c + L\tau_c$$

où β_c est le **temps d'initialisation** (*start-up*) correspondant au calcul de la route, à son établissement ainsi qu'à la préparation des registres qui vont être utilisés lors de la communication. τ_c est le **taux de transmission du lien** (c'est-à-dire le temps nécessaire pour faire transiter un octet).
Dans le cas où les processeurs ne sont plus voisins, mais à une distance d, il faut faire une distinction suivant le mode de commutation :

- Commutation de messages : $T = d(\beta_c + L\tau_c)$
- Commutation de circuit et wormhole : $T = \alpha_c + d\delta_c + L\tau_c$
  Ici le temps d'initialisation est divisé en deux, tout d'abord un temps α_c correspondant à l'établissement du protocole, puis un temps δ_c qui est le temps de commutation des commutateurs au niveau de chacun des d nœuds de communication à traverser. Il faut noter que cette modélisation ne tient pas compte des différences qui existent entre le *wormhole* et la commutation de circuits, mais il n'existe pas à l'heure actuelle de modélisation le faisant.

1.3 Les principales topologies

Le réseau d'interconnexion d'une machine parallèle est généralement modélisé par un graphe $G = (V, E)$ où V est l'ensemble des sommets et E l'ensemble des arêtes [78, 90]. Il existe deux grandes classes de réseaux d'interconnexion [76, 102] :

les réseaux directs : les sommets du graphe G correspondent aux nœuds de la machine parallèle.

Les réseaux indirects ou multi-étages : les sommets du graphe ne sont plus les nœuds de la machine mais des commutateurs.

1.3.1 Les réseaux directs

Les principales caractéristiques des réseaux sont le **degré maximum** (maximum sur tous les nœuds du nombre de liens), le **diamètre** (maximum sur tous les chemins de longueur minimale entre toute paire de nœuds) et le nombre d'arêtes. Ces deux paramètres ont une grande importance pour pouvoir ensuite concevoir des algorithmes de communication. Ils mesurent respectivement le nombre de nœuds que l'on peut atteindre à partir d'un sommet donné et le nombre maximum de sommets du graphe à traverser afin d'atteindre un sommet particulier.

Nous supposons que les processeurs sont numérotés de 0 à $P-1$. Nous allons rappeler brièvement les définitions des réseaux considérés dans ce document, à savoir l'hypercube, le tore et le de Bruijn [78, 90, 100, 107]. Nous avons représenté sur la figure 1.6 un exemple de chaque topologie.

Hypercube :

L'hypercube de dimension n (ou n-cube) est un graphe composé de $P = 2^n$ nœuds. Ceux-ci sont représentés par tous les mots binaires de longueur n. Deux nœuds sont reliés suivant la dimension k, si leur représentation diffère seulement selon le k^{ieme} bit $\forall k, 0 \leq k \leq n-1$. Un nœud a donc exactement n voisins [107].

Grille et tore bi-dimensionnels [4] :

Nous considérons des grilles et des tores bi-dimensionnels de taille $P = P_1 \times P_2$. Un nœud est représenté par un couple d'entiers (q_1, q_2), $0 \leq q_1 \leq P_1$, $0 \leq q_2 \leq P_2$. Les arêtes du tore connectent les nœuds qui diffèrent de $(1 \bmod P_1)$ (respectivement $(1 \bmod P_2)$ suivant la coordonnée q_1 (respectivement q_2). Par conséquent, chaque nœud a exactement 4 voisins. Sur la grille, les nœuds du bord $(q_1 = 0, q_1 = P_1 - 1, q_2 = 0, q_2 = P_2 - 1)$ ne sont reliés qu'avec les nœuds internes de la grille.

De Bruijn :

Le réseau de de Bruijn binaire de dimension n est composé des, $P = 2^n$ nœuds représentés par les mots binaires de longueur n. Chaque nœud $(i_0 \ldots i_{n-1})$ est relié par deux arcs, un vers le nœud $(i_1 \ldots i_{n-1}0)$, le second vers le nœud $(i_1 \ldots i_{n-1}1)$ [13, 62]. Pour obtenir le de Bruijn non orienté, il suffit de transformer chaque arc en une arête, et d'ôter les boucles.

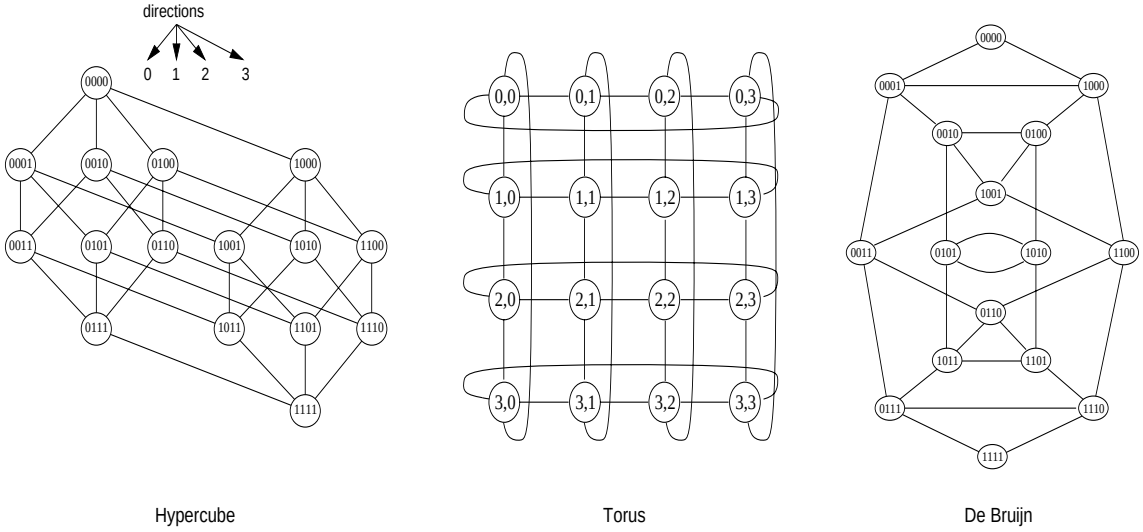


FIG. 1.6 - *Réseaux étudiés avec $P = 16$.*

4. Les définitions données pour des réseaux à deux dimensions, s'étendent naturellement au cas multi-dimensionnels

Nous résumons dans la table 1.1 les principales caractéristiques des précédents réseaux pour P processeurs.

	Degré (max)	Diamètre	Nombre d'arêtes
Grille	4	$(P_1 - 1) + (P_2 - 1)$	$2P_1P_2 - P_1 - P_2$
Tore	4	$\left\lceil \frac{P_1}{2} \right\rceil + \left\lceil \frac{P_2}{2} \right\rceil$	$2P_1P_2$
Hypercube	$\log_2(P)$	$\log_2(P)$	$\frac{P \log_2(P)}{2}$
De Bruijn binaire	4	$\log_2(P)$	$2(P-1)$

TAB. 1.1 - *Principales caractéristiques des réseaux considérés.*

La première génération de machines parallèles utilisait, pour beaucoup, une topologie en hypercube (comme le T40 de FPS, la CM2 de Connection Machine, l'iPSC d'Intel). L'atout principal de cette topologie réside en son diamètre qui est en logarithme du nombre de sommets. Cela signifie que la distance entre les nœuds les plus éloignés est assez faible. En revanche, le degré n'est pas constant, ce qui implique qu'il devient difficile d'interconnecter un nombre important de processeurs.

La notion d'architecture *scalable* s'est imposée, c'est à dire la possibilité d'augmenter le nombre de processeurs simplement en rajoutant des cartes, sans revoir complètement la connexion entre les cartes. Ceci implique un degré constant. C'est pourquoi les topologies de type grille et tore ont connu un grand succès dans la deuxième génération de machines (comme la Paragon, l'AP1000 de Fujistu, ou le T3D de Cray). L'inconvénient de ce type de topologie est son diamètre important. Cependant, les nouveaux mode de commutation, comme le *wormhole* et la commutation de circuits, diminuant le facteur de distance dans le temps de communication, cet inconvénient est nettement moins important qu'en commutation de messages.

On peut alors se demander pourquoi une topologie de type de Bruijn ne connaît pas plus de succès? En effet, elle offre le double avantage d'avoir un degré constant avec un diamètre faible [13].

1.3.2 Les réseaux multi-étages

Les briques de base du réseau sont constituées par des commutateurs configurables permettant de modifier dynamiquement le schéma de connexion. Les commutateurs sont organisés en étages, eux-mêmes juxtaposés. Les sorties des commutateurs d'un étage sont connectées aux entrées des commutateurs de l'étage suivant. Ce sont les « règles » de connexion entre les étages qui discriminent les différents réseaux multi-étages. Nous donnons dans la figure 1.7 un exemple de réseau à trois étages permettant de connecter 8 nœuds.

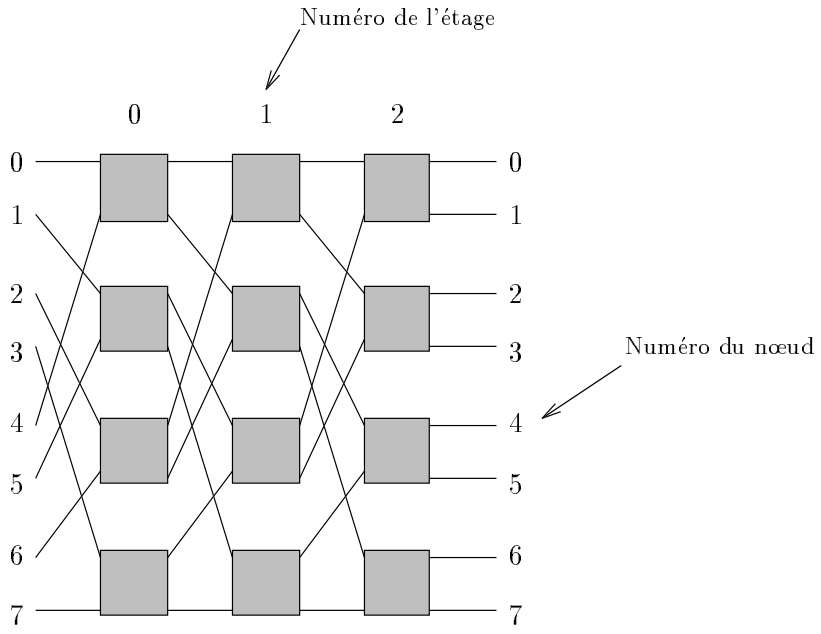


FIG. 1.7 - *Réseau multi-étages oméga de dimension 2*

Les problèmes considérés dans ce type de réseau sont d'ordre de la gestion des contentions sur les liens et de la commande du réseau, c'est-à-dire les protocoles de communication [78].

1.4 Description de quelques machines parallèles

Nous allons décrire rapidement les différentes machines qui ont été utilisées lors des diverses expérimentations présentées tout au long de ce document. Pour plus de précisions sur ces machines ou d'autres, le lecteur pourra se reporter à la thèse de P. Michallon [96] ou au livre de l'école RUMEUR [78]. Les « indices » de performances généralement utilisés pour caractériser les machines sont :

- la puissance de calcul, exprimée en MFlops, c'est-à-dire le nombre de millions d'opérations sur des flottants que le processeur peut réaliser en une seconde.
- la bande passante qui mesure le débit d'un lien, exprimée en Mo/s, c'est-à-dire le nombre de millions d'octets transmissibles sur un lien en une seconde.

Ces deux indices sont donnés par les constructeurs et expriment des puissances crêtes qui correspondent à la limite physique des composants. Ces performances ne sont généralement pas atteintes.

Intel iPSC/860 Cette machine fait suite aux premiers hypercubes d'Intel qui utilisaient des processeurs moins performants et un mode de commutation par paquets. Les nœuds de calculs de l'iPSC/860 sont des processeurs i860 d'Intel d'une puissance crête de 75 MFlops reliés en hypercube, la bande passante est de 20 Mo/s et le mode de commutation est le *wormhole*.

Intel Paragon Comme pour l'iPSC/860, les nœuds de calcul sont des processeurs i860. En revanche à chaque nœud de calcul est associé un nœud chargé du routage. Le réseau d'interconnexion est une grille bi-dimensionnelle utilisant un mode de commutation *wormhole*. La bande passante de 400 Mo/s en bi-directionnel.

IBM SP1 Cette machine est basée sur les processeurs RS6000 d'IBM d'une puissance de crête de 125 MFlops. Les processeurs sont reliés entre eux via un réseau multi-étages, la largeur de bande au niveau de chaque nœud étant de 80 Mo/s en bi-directionnel. La dernière version, le SP2, reprend la même architecture que le SP1, mais les nœuds de calcul sont des Power 2 (250 Mflops) eux-mêmes couplés à un i860 chargé des communications[5]. Le mode de commutation adopté pour les machines d'IBM est le *buffered-wormhole*[6].

Cray T3D C'est la première machine massivement parallèle de Cray. Elle est basée sur le processeur Alpha de DEC (150 Mflops). A chaque nœud de calcul est couplé un processeur chargé du routage connecté suivant un tore à 3 dimensions. La bande passante au niveau de chaque nœud est de 300 Mo/s en bi-directionnel. Nous reviendrons plus en détail sur cette machine dans le chapitre 2 de la partie III.

Nous résumons dans la table 1.2 les caractéristiques des machines citées *supra*.

Machine	Processeur	Performance (MFlops)	Réseau	Bande passante (Mo/s)	Mode de Commutation
iPSC/860	i860	75	Hypercube	20	*wormhole*
Paragon	i860	75	Grille 2D	400	*wormhole*
SP1	Power 1	125	Multi-étages	80	*wormhole*
T3D	Alpha EV4	150	Tore 3D	300	*wormhole*

TAB. 1.2 - *Résumé des caractéristiques des machines utilisées.*

1.5 Conclusion

Ce chapitre nous a permis de préciser le modèle de machine parallèle qui nous intéresse pour la conception des algorithmes parallèles. Il faut noter que ce modèle reste très proche des machines réelles, ce qui nous permet ensuite d'avoir de bonnes correspondances entre les coûts théoriques des algorithmes et leur temps d'exécution sur machine.

Nous nous sommes plus particulièrement intéressés à la modélisation des communications et à l'influence des caractéristiques physiques sur ce modèle. Ceci va s'avérer très important à prendre en compte soit pour concevoir des algorithmes efficaces de communication globale, mais aussi pour recouvrir les communications par du calcul.

5. Ce choix de l'i860 n'est certainement pas le plus efficace mais il provient de l'héritage des projets de recherche d'IBM.

6. *buffered-wormhole* est un mode de commutation permettant de ramener tout, ou une partie, des paquets se trouvant à la suite d'un *flit* bloqué sur un nœud, sur ce même nœud. Ce modèle se rapproche du *virtual-cut-through* [86].

Chapitre 2

Le modèle de programmation par processus communicants

Le but de ce chapitre est de définir, dans un premier temps le modèle de programmation utilisé par la plupart des machines parallèles à mémoire distribuée, puis d'en décrire deux implantations. Ce travail a été effectué avec la collaboration de L. Colombet[a] [24].
Nous tenons à remercier tout particulièrement J. Briat pour ces nombreuses relectures et ces remarques qui ont fait considérablement progressé le contenu de ce chapitre.

[a] Ingénieur de recherches au Centre d'Études Nucléaires de Grenoble

Ce chapitre est une introduction au **modèle de programmation par processus communicants** et à deux implantations de ce modèle qui sont (plus ou moins) les standards actuels. L'unique moyen de faire communiquer des processus distants, c'est-à-dire situés sur des processeurs différents lorsque la mémoire est physiquement distribuée, est **l'échange de messages** (que cet échange soit explicite ou non[1]). De plus, à l'heure actuelle, ni la compilation de langages à parallélisme de données de type HPF, efficace seulement pour certains types de problèmes, ni la parallélisation automatique, en dépit de ses énormes progrès, ne constituent une réponse satisfaisante pour obtenir des applications efficaces. Le passage à « l'assembleur » du parallélisme s'avère donc nécessaire afin d'obtenir de bonnes performances.

Une autre raison qui motive l'utilisation de ce modèle, tient au développement du concept d'informatique distribuée. La création d'environnements de programmation dans le but de faire fonctionner un réseau d'ordinateurs hétérogènes comme une machine parallèle est devenue nécessaire. En effet, se servir d'un réseau d'ordinateurs pour simuler les dernières machines parallèles à mémoire distribuée conçues par les grands constructeurs comme Cray Research, IBM, Intel etc, rabaisse considérablement le coût des applications parallèles. Cependant il faut veiller à ce que ces applications soient directement portables sur une machine parallèle quelconque, moyennant une recompilation des programmes. Des universités

1. Nous entendons par échange implicite de messages le fait que, par exemple dans un programme HPF, il n'y a pas d'instruction explicite d'échange. Cependant, le code généré puis exécuté contient lui des opérations d'échanges.

et des constructeurs ont alors développé des outils logiciels de programmation sur réseaux de machines hétérogènes (parallèles, vectorielles, RISC etc) appelés **Bibliothèques de Communication par Echange de Messages** (ou *Message Passing Libraries*). Parmi les plus utilisées on trouve PVM (Parallel Virtual Machine) [11], PICL (Portable Instrumented Communication Library) [64], P4 [18], Parmacs [20] ou encore celle qui veut devenir le standard des bibliothèques de communication par échange de messages : MPI (*Message Passing Interface*) [95].

Nous allons présenter dans ce chapitre deux de ces bibliothèques, en commençant par « le standard » *de facto* actuel : PVM, puis, dans un deuxième temps, peut-être le standard de demain : MPI. Mais avant de les décrire, il nous semble utile de présenter le modèle de programmation par processus communicants qui est à la base de ces bibliothèques.

2.1 Description du modèle

2.1.1 Introduction et définitions

Nous nous plaçons dans le cadre d'une programmation d'une machine parallèle, virtuelle (c'est à dire un réseau de stations) ou réelle, à mémoire physiquement distribuée.

La notion de processus communicant a été introduite par Hoare en 1978 [74] avec le modèle CSP (*Communicating Sequential Processes*). Un **processus** peut être défini comme l'exécution séquentielle d'instructions portant sur des données [115]. Un **programme parallèle** est vu alors, comme un ensemble de processus coopérant à l'exécution d'une tâche commune.

« CSP est fondé sur la notion de processus communicant à l'aide de primitives d'émissions-réceptions utilisées explicitement dans le texte des processus. Une communication a lieu quand un processus émetteur désigne un autre processus comme destinataire d'une valeur que celui-ci désigne le premier comme source d'information. Dans ces conditions, la valeur est transmise de l'émetteur vers le récepteur. De plus, il n'y a aucune possibilité pour l'émetteur (resp. le destinataire) de continuer son calcul tant que la valeur émise n'a pas été transmise à (resp. reçue par) son destinataire. C'est le principe de **rendez-vous**. » [106].

Nous nous plaçons ici à un niveau différent, dans le sens où nous allons définir un modèle moins formel qui est mis en œuvre par la plupart des bibliothèques de commnications par échange de messages et qui est utilisé lorsqu'on conçoit des algorithmes parallèles avec un modèle de programmation par échange de messages.

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 son exécution, un processus emploie des données non locales, 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 (ceci

ne veut pas dire qu'il y a un processus par processeur physique) et s'échangeant, en cours d'exécution, des données *via* des messages (voir figure 2.1).

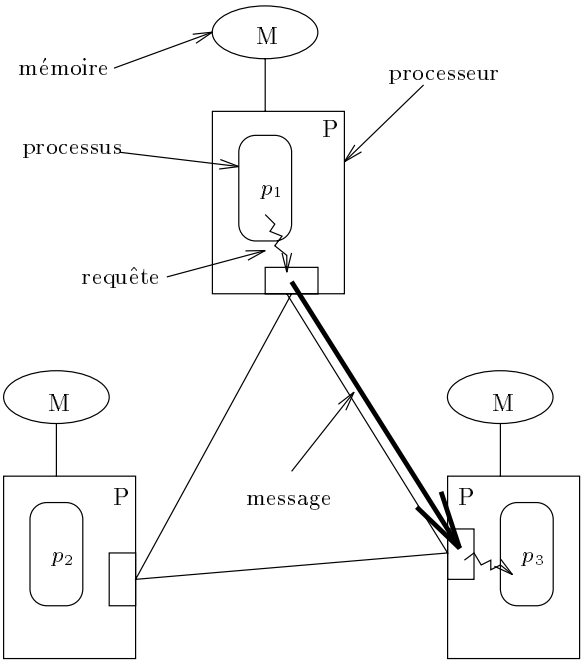


FIG. 2.1 - *Exemple du modèle de programmation par processus communicants*

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

2.1.2 Les communications point-à-point

Les primitives

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` (ou `put`) pour l'émission d'un message et `receive` (ou `get`) pour la réception. Quels sont les paramètres à préciser? Ils sont principalement au nombre de 2 :

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 (*buffer*) ou une adresse d'un espace mémoire contenant le message à envoyer, ou à recevoir

Synchronisation associée à la communication de bout en bout

Une première nuance intervient dès ce niveau : la notion de **communication bloquante** ou **non-bloquante**. En effet, le processus émetteur ne doit pas forcément attendre que le message soit arrivé à destination pour poursuivre l'exécution. On parlera alors d'**émission non-bloquante** (ou ***send* non-bloquant**). De même, le processus récepteur peut, dans certains cas continuer à s'exécuter, sans que le message ne soit arrivé. On parlera alors de **réception non-bloquante** (ou ***receive* non-bloquant**). Dans ce dernier cas, il est nécessaire que ce processus puisse tester (`probe`) ou attendre (`wait`) l'arrivée du message lorsqu'il va en avoir effectivement besoin (voir figure 2.2).

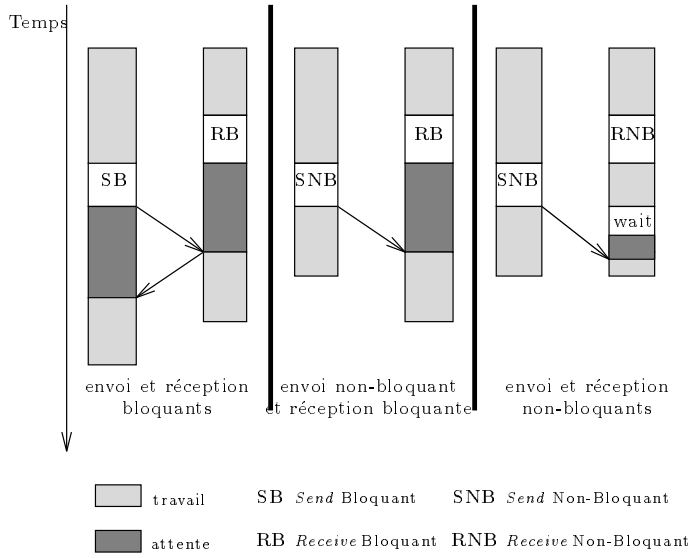


FIG. 2.2 - *Différences entre communications bloquantes et non-bloquantes entre deux processeurs.*

Notons au passage que le modèle CSP implique lui des communications synchrones totalement bloquantes, puisque basées sur un système de rendez-vous.

Le contrôle de flux

La nature asynchrone de l'envoi non bloquant implique qu'un émetteur peut envoyer une série de messages. Que peut-il alors se passer ? Ceci met en évidence une première caractéristique de ce modèle, il s'agit du **contrôle de flux**. Qu'entendons-nous par ce terme ?

Si la fréquence d'émission est inférieure au cycle de réception, il n'y a pas de problème car, soit le récepteur est toujours en attente de message, soit il a fourni un tampon pour recevoir de façon asynchrone chaque message (cas de l'envoi et la réception non-bloquants de la figure 2.2).

Dans le cas contraire, le récepteur doit asservir la vitesse de réception sinon des messages seront perdus : c'est le **contrôle de flux**. Deux situations peuvent alors se présenter.

Soit le modèle spécifie que l'envoi non-bloquant est sans perte, c'est à dire que des tampons sont gérés de façon transparente à l'utilisateur ainsi que le contrôle du flux d'émission.

Dans ce cas l'envoi non bloquant ne l'est que dans la limite du nombre de tampons. On dit que l'émission peut être anticipée sur la réception, d'un nombre fini de pas (qui est une constante du modèle).

Soit le modèle spécifie que la communication est sans contrôle de flux et par conséquent que des messages sont perdus si le délai d'émission est trop élevé pour la capacité de réception. Le programmeur doit lui-même assurer son propre contrôle de flux pour lequel il existe des techniques classiques [110].

Le déséquencement

Comme on peut s'en rendre compte assez facilement, ce modèle de programmation est fondamentalement asynchrone. Il se pose naturellement la question de savoir si l'ordre de réception des messages correspond à l'ordre d'émission. C'est une propriété de la communication point-à-point d'un modèle, d'être avec ou sans **déséquencement**. L'existence du déséquencement caractérise une implantation utilisant un réseau fiable par répétition et/ou à routage non déterministe.

Que faire dans un cas de communication avec déséquencement?

Supposons qu'un processus A envoie un message M_1 au processus B, puis un message M_2, toujours au processus B. Il se peut que le message M_2 arrive avant M_1 au processus B. Ce dernier étant en attente d'un message du processus A, il va prendre le premier lui arrivant, c'est à dire M_2, or ce n'est pas le bon !

S'il y a déséquencement, il faut « restaurer l'ordre ». Un moyen pour résoudre ce type de problème est d'associer un numéro de séquence au message. Ainsi, dans le cas précédent, le processus B va attendre un message de A avec un numéro E_i. Le message M_2 portant un numéro E_{i+1}, ne pourra plus être confondu avec M_1.

Nous n'aborderons pas ici les problèmes de congestion dus au routage, car nous supposons que nous travaillons dans un réseau non chargé, c'est à dire avec une machine parallèle qui nous est dédiée.

2.1.3 Les communications globales

La majeure partie des algorithmes parallèles nécessitent des mouvements structurés de données impliquant tout ou une partie des processus [36, 55, 63, 108, 113]. Un moyen efficace d'effectuer ces mouvements consiste à utiliser des communications globales. Nous reviendrons sur ces schémas de communications globales ultérieurement lors de la description de quelques algorithmes les réalisant sur différentes topologies (*c.f.* partie II).

Les opérations de communications globales

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 tout ou plusieurs processus regroupent une information vers un processus particulier. Cette opération est souvent appelée ***gathering***.

Mais d'autres opérations de communication sont souvent utiles. Il s'agit de généralisations des précédentes, comme par exemple une diffusion de tous les processeurs appelée **échange total** (ou ***all-to-all***).

Enfin, la dernière classe d'opérations de communications globales est constituée par les versions personnalisées des précédentes. On entend par **communication personnalisée** le cas où le message envoyé à un processus dépend de ce processus. 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 ***personnalized 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)(*cf* partie II chapitre 2).

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 code.

Les problèmes de synchronisation liés aux communications globales

Du fait de l'asynchronisme des communications, les problèmes rencontrés lors des communications point-à-point peuvent se reproduire lors des communications globales. On les caractérise par les propriétés de **déséquencement** ou **d'atomicité**. Prenons l'exemple simple de la diffusion. Une diffusion est dite sans déséquencement si lors de deux diffusions successives provenant d'un même processus, l'ordre de réception respecte l'ordre d'émission. Elle est dite atomique si l'ordre de réception est le même pour tous les processus récepteurs quand deux diffusions ne proviennent pas du même processus.

Dans la figure 2.3, nous avons représenté le cas d'une diffusion d'un message `msg1` de la part d'un émetteur E1, et d'un message `msg2` de la part d'un émetteur E2. On dit que la diffusion est atomique si R1 et R2 reçoivent les deux messages `msg1` et `msg2` dans le même ordre.

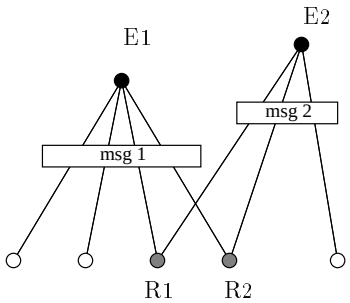


FIG. 2.3 - *Diffusions atomique et non atomique*

Les opérations globales de communication et de calcul

Lors du regroupement, il peut être intéressant de combiner les informations entre elles. Un exemple typique de ce genre d'opérations se présente lorsqu'on désire trouver le maximum de valeurs réparties sur les différents processus, ou encore si on veut effectuer une somme d'éléments. Cette opération est communément appelée **regroupement avec recombinaison** (ou ***global combine***). Ce schéma peut être suivi alors d'une diffusion afin que le résultat de la combinaison soit connue des autres processus.

Là aussi des problèmes de synchronisation interviennent, comme dans le cas de deux opérations d'additions globales successives sur des vecteurs différents. S'il y a désynchronisation, il se peut que le résultat de la somme ne corresponde pas au résultat correct. Lorsque la synchronisation n'est pas faite lors de l'implantation, il est préférable de prévoir une synchronisation des processus entre les deux opérations.

Il est important de noter que généralement les opérateurs utilisés lors de ces recombinaisons ont des propriétés de commutativité (et souvent d'associativité) afin d'éviter d'éventuels problèmes si un déséquencement survient lors de la recombinaison. Cela permet de ne pas imposer un ordre sur les opérations à effectuer. Par exemple, si l'opérateur considéré est `max`, l'ordre dans lequel il est appliqué sur les données n'a aucune importance.

2.1.4 La notion de groupe

Structuration de l'ensemble des processus

La 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 ensemble car ils possèdent une caractéristique commune. Par exemple quelques processus sont chargés d'effectuer un même calcul, alors que les autres processus sont occupés à une autre tâche. 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 à une liste de processus, mais à un groupe. Tous les processus d'un groupe peuvent être très facilement synchronisés par une **barrière de synchronisation** s'appliquant uniquement à ce groupe.

Gestion de l'ensemble des processus

La gestion des groupes peut être **statique**, c'est-à-dire que les groupes sont définis une fois pour toute en début d'exécution des différents processus. La gestion est dite **dynamique** si un processus peut à tout moment rejoindre ou quitter un groupe. Dans ce dernier cas, la question se pose de savoir si la « vision » locale d'un groupe est toujours exacte. En effet, il se peut qu'une diffusion (ou un autre schéma global) atteigne plus de processus que prévus (ou moins) si entre le début et la fin de la communication, le nombre de processus appartenant au groupe considéré a changé. Certaines implantations synchronisent les processus avant toute communication globale à l'intérieur du groupe afin d'éviter ce genre de problèmes. Sinon, il faut le prévenir soi-même.

2.1.5 Les différences dans les implantations

Les variations dans les implantations de ce modèle sont dans la façon d'attribuer l'ensemble des processus aux processeurs. Voici les diverses possibilités que l'on trouve généralement :

- **attribution statique** des processus aux processeurs
  - modèle **mono-programmation** : un processus est associé à chaque processeur.
  - modèle **multi-programmation** : n processus peuvent être associés à un processeur.

  Le problème à résoudre est alors le **placement/ordonnancement** des différents processus. Quelle « méthode » doit-on adopter pour allouer un processus à un processeur ? Sur quel critère grouper plusieurs processus sur un processeur s'il y a multi-programmation ?

- **Attribution dynamique** des processus aux processeurs (et multi-programmation)
  - l'attribution d'un processus à un processeur s'effectue à la création du processus
  - la migration d'un processus en cours d'exécution, vers un autre processeur est autorisée.

  Le problème ici, est la **régulation de la charge**. Où doit-on créer le nouveau processus afin que la charge de travail de chaque processeur soit équilibrée ? Quelles sont les conditions de migration de processus en cours d'exécution afin de répartir la charge totale ?

Il n'existe pas aujourd'hui de langage de programmation conforme à un tel modèle et disponible sur des machines parallèles. C'est pourquoi ce modèle est fourni dans des langages « classiques » (C, Fortran) *via* des **bibliothèques de communication par échange de messages**. Dans la suite nous allons décrire les principales caractéristiques de deux de ces bibliothèques PVM et MPI.

2.2 PVM : *Parallel Virtual Machine*

2.2.1 Introduction

PVM (*Parallel Virtual Machine*) est l'un des environnements de programmation parallèle les plus utilisés actuellement dans le monde universitaire. Développé par des chercheurs de l'université du Tennessee et du Oak Ridge National Laboratory, PVM est facile à mettre en œuvre et en constante évolution. Sa simplicité et sa portabilité sont certainement les deux principales raisons de son succès.

Un environnement de programmation parallèle sur réseau de stations de travail : PVM

Pourquoi un tel choix?

PVM présente différentes caractéristiques qui nous ont paru intéressantes. Ce modèle de langage parallèle est constitué du langage C et de fonctions de communication. Ces dernières sont aussi 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 (*cf* voir la section précédente).

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 de cet outil pour la programmation de grands réseaux ou de réseaux avec des architectures dédiées.

Si l'application est portée sur un ensemble hétérogène d'architectures, un format de données indépendant des architectures (XDR) est utilisé et une traduction spécifique est effectuée sur chaque nœud. Ces propriétés autorisent une compatibilité de PVM avec un grand nombre de machines (CRAY, stations HP, IBM, SGI, SUN, etc.) y compris les machines parallèles les plus utilisées aujourd'hui (CM5, SP1, IPSC, CS2, T3D, Paragon etc.).

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, la visualisation des 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, développé par le Oak Ridge National Laboratory (USA).

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 trace 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 2.4 un exemple d'une visualisation d'un programme PVM avec 4 tâches esclaves (`srld_task`) et une tâche maître (`srld_main`) s'exécutant sur 4 machines. Il s'agit d'un diagramme espace-temps (*space-time diagram*) 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 en évidence également les communications (*Message*) entre les différentes tâches.

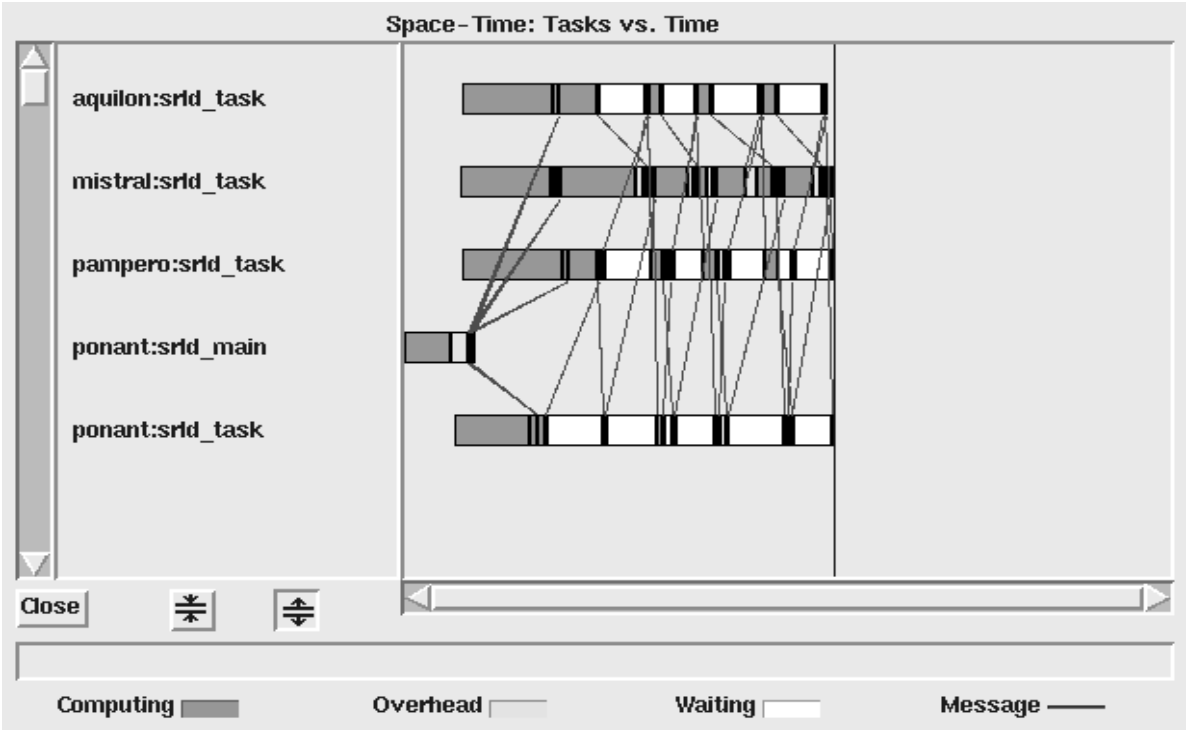


FIG. 2.4 - *Exemple d'une visualisation de traces par XPVM*

Développer des programmes avec PVM : un bon investissement?

Enfin, comme nous le verrons plus loin, il existe de très grandes similitudes avec le futur standard de communication par échange de messages : **MPI** (*Message Passing Interface*). Celles-ci permettent d'envisager un portage assez aisé d'applications PVM vers MPI[2].

2.2.2 Caractéristiques de PVM

PVM n'est 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, des tâches 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 processus, ou la mise en œuvre de ses propres fonctions de placement.

Nous allons détailler maintenant les caractéristiques et les possibilités de PVM.

L'identification des processus

Tous les processus (voir la section 2.1) enregistrés comme utilisant PVM, sont représentés par un entier, appelé `tid`, qui les identifie de manière unique dans la machine parallèle

2. IBM propose par exemple un traducteur PVM-MPI.

virtuelle. Ces identificateurs sont donnés par le démon et ne peuvent pas être choisis par les utilisateurs. Ainsi, si ces derniers ont besoin d'une numérotation particulière de leurs différents processus, ils doivent construire une table de correspondance.

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 de processus donné.

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 est utilisée pour faciliter l'écriture et diminuer les risques d'erreurs de certaines communications globales.

Contrôle des processus

PVM fournit des fonctions qui permettent à un processus utilisateur de devenir un processus PVM ou le contraire. De plus, d'autres fonctions sont disponibles pour permettre une gestion de la machine virtuelle elle-même. L'utilisateur supprime ou ajoute un ordinateur à son réseau, ou envoie des signaux aux différents processus. Une gestion dynamique des processeurs est possible car toutes les commandes utilisées par l'interface utilisateur, nommé « console », peuvent être appelées par un programme durant son exécution.

Tolérance aux pannes

Si un des ordinateurs de la machine virtuelle s'arrête ou n'est plus accessible, PVM le détecte automatiquement et l'enlève de la liste des machines utilisées. L'état d'une machine peut être obtenu par une application, mais 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'un processus exécuté sur une machine défectueuse, car ces dernières ne seraient jamais reçues et bloqueraient le programme.

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 utilisés pour définir la portée d'une barrière ou d'une diffusion. Une barrière ne peut se faire qu'entre des processus appartenant à un même groupe.

Envoi de signaux

PVM fournit deux méthodes différentes pour envoyer des signaux aux processus. La première envoie un simple signal Unix à un ou plusieurs processus. La seconde consiste à émettre des notifications sous forme d'un entier avec un marqueur défini par l'utilisateur. Pour le moment, les événements possibles sont l'arrêt d'une machine, l'addition d'une nouvelle machine et l'arrêt d'un processus.

Les fonctions de communication

La bibliothèque de fonctions de communication par échange de messages a évolué avec l'apparition de la version 3 de PVM. Cette bibliothèque se rapproche beaucoup plus des besoins des utilisateurs et des études faites par divers groupes de travail et universités. Le modèle choisi est l'envoi de tampons (*buffers* en anglais), de taille théoriquement illimitée, à un ou plusieurs processus PVM. L'envoi des messages est considéré comme étant non bloquant et asynchrone, alors que la réception est asynchrone bloquante (*cf* section 2.1). Cette dernière peut être rendue non-bloquante : la fonction de réception des données renvoie un identificateur indiquant si les données ont été reçues ou non. Mais la gestion de cette réception est laissée au programmeur, c'est à dire que si les données ne sont pas disponibles, le programme peut 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 et la diffusion réduite à un groupe.

L'utilisation de machines parallèles

Les constructeurs et les chercheurs du « PVM Team » de l'université du Tennessee et du Oak Ridge National Laboratory 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 la plupart des machines massivement parallèles. Les messages échangés par deux nœuds d'une de ces machines utilisent directement le réseau d'interconnexion à haut débit. Il n'y a plus de « démon » chargé de gérer les transferts de données: c'est le système spécialisé de la machine cible qui est directement mis en œuvre (dans le cas d'une implémentation efficace de PVM). De plus, un noeud 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.

2.2.3 Les communications en PVM

Les communications entre les différents processus PVM se font par échange de messages. Le programmeur doit donc construire les messages qui vont circuler entre les processus. Comme ceux-ci contiennent des valeurs ayant des types très variés, 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 tampons sont très nombreux. Par exemple, les messages peuvent être facilement découpés pour 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. 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'ils occupent car il se peut qu'une des machines du réseau soit incapable physiquement de recevoir des messages trop longs.

PVM assure un contrôle du flux (*cf* section 2.1) *via* le démon. C'est à dire que si un message est envoyé alors que le tampon de réception n'est pas alloué, il y aura alors stockage de ce message au niveau du démon jusqu'à ce que l'espace mémoire soit alloué au niveau du processus destinataire.

Le problème du dé-séquencement (*cf* section 2.1) ici est résolu par l'utilisation d'une étiquette associée au message.

Tampons pour les Messages

Avec l'environnement PVM, un seul tampon peut être actif. Mais 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 tampon d'envoi par défaut. Les fonctions `pvm_mkbuf` (resp. `pvm_freebuf`) servent à créer (resp. libérer) des tampons supplémentaires. De plus, c'est lors de la création de ces derniers que le programmeur a le choix d'un éventuel codage des données.

Différents types d'encodage

- `PvmDataDefault` : Codage XDR si le réseau est détecté comme étant hétérogène, sinon il n'y a pas de codage.
- `PvmDataRaw` : Aucun codage
- `PvmDataInPlace` : Seuls les pointeurs et les tailles des données sont empaquetés.

Le codage XDR est nécessaire si le réseau est hétérogène. Sinon il ne fait qu'augmenter le temps des communications.

Type de routage

Le coût des communications est optimisé 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, alors l'activation d'un lien direct entre les processus n'entraîne pas une perte de la fiabilité des communications.

Communication point-à-point

Pour envoyer un tampon, il faut utiliser la fonction `pvm_send( int tid, int mstag)` et préciser quel est le numéro d'identification du processus 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 (C'est la solution PVM au problème du dé-séquencement).

Pour la réception, il y a plus de choix. La fonction `pvm_recv` est la fonction de réception bloquante. Tant que le message n'est pas arrivé, le processus reste bloqué en attente. Pour éviter cela et donc gagner de l'efficacité, il faut utiliser les fonctions `pvm_probe` et `pvm_nrecv` qui prennent le premier message arrivé ou espionnent l'arrivée de messages (réception non-bloquante).

Dans les dernières versions de PVM, des primitives « rapides » d'envoi et de réception sont fournies, qui évitent d'empaqueter et de dépaqueter les données dans le cas de machines homogènes. Il s'agit des primitives `pvm_psend` et `pvm_precv`. Ces primitives apportent un gain de performance dans le cas d'utilisation de PVM sur des machines massivement parallèles (voir partie III chapitre 2).

2.2.4 Concepts avancés en PVM

Création de buffers complexes

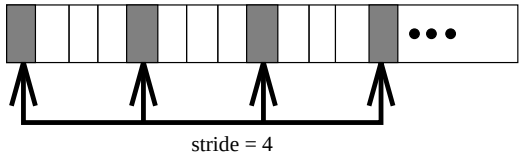
Types différents de données

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

Pour retrouver toutes les valeurs stockées dans le tampon de réception, il suffit d'utiliser les fonctions de dépaquetage en prenant soin de respecter l'ordre d'empaquetage.

Données non contiguës en mémoire

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



Les communications globales

Un message composé, peut être :

Envoyé à une liste de processus
Diffusé à tous les processus
Diffusé à un groupe (si la bibliothèque `group` est utilisée)

En fait, toutes les communications globales sont facilement programmables. Les primitives de diffusion de PVM sont sans dé-séquencement mais non atomiques (*cf* section 2.1)(on parlera aussi de **cohérence faible**). Il existe également une opération globale de communication/calcul agissant sur un groupe : `pvm_reduce` permettant la combinaison de données distribuées sur les processus d'un groupe et la récupération sur un processus désigné du groupe.

2.3 MPI : *Message Passing Interface*

2.3.1 Introduction

MPI (*Message Passing Interface*) est une bibliothèque de communication par échange de messages ayant pour but de rassembler les meilleurs éléments des systèmes déjà existants plutôt que d'en choisir un; le but ultime est de proposer MPI comme le standard des systèmes de *message passing*. Voici les principales caratéristiques de cette bibliothèque :

1. Ce qui est inclus dans MPI
   - Communication point-à-point
   - Opérations collectives
   - Notion de groupes de processus
   - Notion de contextes associés à une communication
   - Topologie virtuelle de processus
2. Ce qui n'est pas inclus dans MPI
   - Des fonctions d'entrées/sorties parallèles.
   - Opérations explicites de mémoire partagée
   - Des opérations nécessitant une utilisation plus que standard du système d'exploitation, comme par exemple les messages actifs [3] ou l'exécution distante
   - Des outils de construction de programmes
   - Des facilités de deboguage
   - Des fonctions auxiliaires comme pour la prise de temps
   - Le support explicite de processus légers (au sens Unix).
   - Le support pour la gestion de tâches

3. Un message actif, est un système permettant d'activer un processus distant qui utilisera le message communiqué comme données d'entrée.

Nous allons définir un peu plus précisément le modèle de communication de MPI.

Un programme MPI est considéré comme un ensemble de processus communicants. Cet ensemble, que l'on appelle **groupe de communication**, est désigné par un **communicateur**. La numérotation des processus est relative à un communicateur donné. La communication peut se faire à l'intérieur d'un même groupe de communication, on parlera alors de communication **intra-communicateur** ou entre différents goupes de communications (communications **inter-communicateurs**). Lors d'une communication, est associée à chaque message une **enveloppe** contenant :

- le processus émetteur / destinataire
- une étiquette (équivalent à l'étiquette de PVM)
- le communicateur

Au départ, tous les processus appartiennent au même groupe de communication `MPI_COMM_WORLD`, cet espace peut ensuite être sub-divisé en sous-espaces de communication, disjoints ou non.

MPI considère deux grandes classes de communication : les communications dites fiables et celles dites non-fiables. Parmi les communications fiables on distingue, le mode **standard** où le contrôle de flux est assuré par MPI, et le mode **synchrone** (***synchronous***) à base de rendez-vous entre les processus communicants (voir figure 2.5).

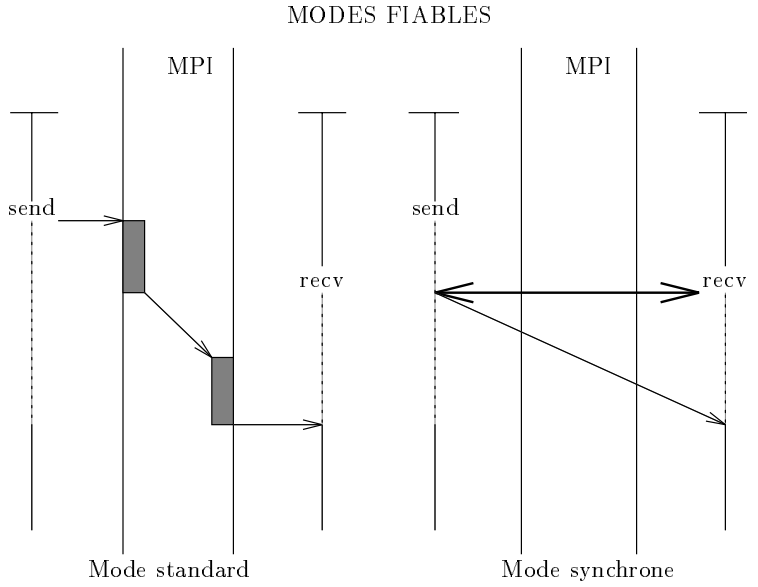


FIG. 2.5 - *Les modes de communications fiables*

Les modes de communication non-fiables sont au nombre de deux (voir figure 2.6): le mode **tamponné** (***buffered***) dans lequel l'utilisateur gère ses propres buffers de communication et le mode **prêt** (***ready***) où l'utilisateur a la charge de garantir qu'il y a une réception prête pour chaque émission. Dans ces deux cas, c'est donc l'utilisateur qui assure le contrôle du flux.

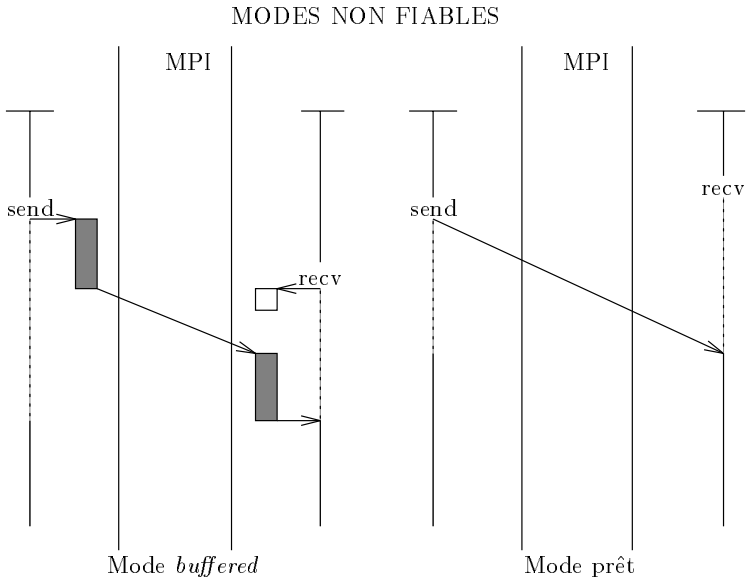


FIG. 2.6 - *Les modes de communications non-fiables*

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 tampons, c'est à dire la possibilité de **construire des types** de données décrivant des structures complexes, ou d'envoyer des données non contiguës en mémoire (exemple typique d'une matrice allouée comme un tableau multi-dimensionnel)
- généralisation de la notion de « type » associé à un message par l'utilisation d'un **contexte** lié à une famille de messages
- généralisation d'identificateur de processus par la notion de **groupe** et de **communicateur**.

Cette section est essentiellement fondée sur deux supports, le premier qui est le document établissant les spécifications du standard MPI [95] et le deuxième, un rapport de recherche de l'université du Tennessee qui présente les principales caractéristiques de MPI [116].

2.3.2 Les concepts de base

En dépit d'un grand nombre des fonctions dans MPI et de nouveaux concepts, son utilisation directe est assez aisée. En effet, seules quelques fonctions sont à connaître, les communications point-à-point et globales étant similaires à celles que l'on trouve généralement.

2.3.3 Communications point-à-point

Bien que beaucoup de fonctions point-à-point soient disponibles, les procédures d'envoi et de réceptions bloquantes ou non-bloquantes sont d'une utilisation standard.

- Versions bloquantes
  - `MPI_SEND(buf,cnt,type,dest,tag,comm)`
  - `MPI_RECV(buf,cnt,type,src,tag,comm)`
- Versions non-bloquantes
  - `MPI_ISEND(handle,buf,cnt,type,dest,tag,comm)`
  - `MPI_IRECV(handle,buf,cnt,type,src,tag,comm)`
    - `handle` : contrôle qui permet de s'assurer par la suite de la terminaison de la communication.

On voit donc qu'à la différence de PVM, MPI n'assure pas de contrôle de flux, puisque soit la communication est synchrone, soit c'est à l'utilisateur de s'assurer que le processus destinataire est prêt à recevoir. L'avantage de cette option est qu'il n'y a pas de stockage intermédiaire, d'où des communications plus efficaces. Comme pour PVM, le moyen d'éviter le dé-séquencement des messages est résolu par l'étiquette. Cette notion est étendue à l'aide du contexte de communication.

2.3.4 Communications collectives

Les opérations de communications collectives n'ont pas d'étiquette. Une communication collective doit être appelée par tous les membres d'un groupe. Comme pour PVM, ces opérations sont à cohérence faible, c'est à dire sans dé-séquencement mais non atomiques (*cf* section 2.1)[95].

Communications de groupe

C'est une communication qui implique tous les processus d'un même groupe. Le cas le plus simple est le groupe `everybody`. Les opérations fournies sont les suivantes[4] (voir figure 2.7) :

1. `Barrier` : barrière de synchronisation pour tous les membres du groupe
2. `Broadcast` : diffusion d'un membre vers tous les autres (OTA : *One-To-All*)
3. `Gather` : regroupement de tous les membres du groupe, vers un élément particulier (ATO : *All-To-One*)
4. `Scatter` : diffusion personnalisée (POTA : *Personalized-One-To-All*)

4. Nous reviendrons plus en détail sur les définitions de ces opérations globales de communication dans le chapitre 1 de la partie II.

5. `Reduce`, `Scan` : opérations globales sur tous les membres du groupe comme max, min somme, etc ...

6. `All_broadcast` : échange total, diffusion depuis tous les processeurs (ATA : *All-To-All*)

7. `All_gather` : échange total personnalisé (PATA : *Personalized-All-To-All*)

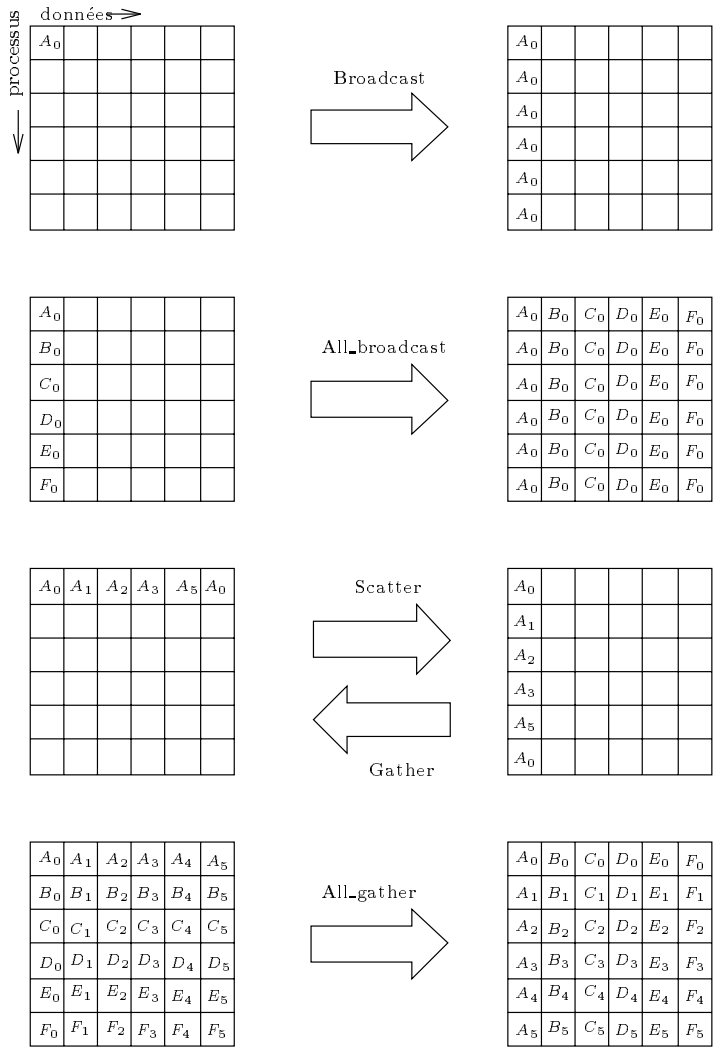


FIG. 2.7 - *Principales fonctions de communication dans un groupe de processus*

Fonctions de réduction

Ce sont les opérations globales de communication et de calcul (*cf* section 2.1). MPI définit neuf variantes, suivant la portée de l'opération de réduction :

One : opération de réduction vers un processus.

All : opération globale à tous les processus du groupe, c'est à dire que tous les processus effectuent l'opération de réduction.

Scatter : réduction composante par composante d'un vecteur de données, suivie de la redistribution du vecteur résultat à tous les processus du groupe (exemple de la somme de deux vecteurs répartis sur les n processus d'un groupe, le vecteur somme résultant étant alors redistribué aux n processus du groupe).

et suivant le type de l'opération associée à la réduction :

set : ensemble d'opérations prédéfinies : `MAX`, `MIN`, `SUM`, `AND`, `OR`, `XOR`, Bitwise `AND`, `OR`, `XOR`,`MAX` et rang du processus contenant le maximum, `MIN` et rang du processus contenant le minimum

user : opération définie par l'utilisateur (opération associative et commutative)

userA : opération définie par l'utilisateur, mais uniquement associative

2.4 Conclusion

Les deux bibliothèques présentées dans ce document ne sont que deux implantations du modèle de programmation par processus communicants. Elles sont finalement assez proches dans la solution (ou non solution) qu'elles apportent aux problèmes qui se posent lorsqu'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 ou de types structurés.

Néanmoins, PVM continue à évoluer malgré l'apparition de MPI, et la dernière version PVM 3.3 complète la précédente en fournissant des opérations globales de communication et de calcul, ainsi que des communications point-à-point véritablement asynchrones [11, 94].

Partie II

Communications Globales

Nous allons présenter dans cette partie une première méthode pour minimiser le temps de communication dans les algorithmes parallèles. Elle consiste en l'optimisation de l'algorithme de communication lui-même. Nous commençons par une introduction dans laquelle nous effectuons un rapide survol des très nombreux travaux réalisés dans ce domaine. Nous insistons sur quelques outils issus de la théorie des graphes permettant de concevoir des algorithmes de communication efficaces en fonction du mode de commutation. Les deux chapitres suivants sont consacrés à la description de nouveaux algorithmes efficaces de communication dans différents modes de commutation.

Chapitre 1

Communications globales dans les réseaux directs

Le but de ce chapitre est de décrire quelques méthodes utilisées pour concevoir des algorithmes de communication dans les réseaux directs. Nous illustrons leurs utilisations sur quelques exemples d'algorithmes de communication globale.

1.1 Les principaux schémas de communication

Comme nous l'avons vu au chapitre 2 de la partie I, nous pouvons distinguer deux classes de schéma de communication : ceux mettant en œuvre au plus deux processeurs (les communications point-à-point) et ceux qui impliquent un nombre plus important (les communications globales ou collectives). Nous en rappelons les définitions [78] :

La communication point-à-point : il s'agit soit d'un envoi simple d'un message entre un nœud source et un nœud destinataire, soit d'un échange entre deux nœuds. On distingue généralement l'échange simultané des messages entre le nœud source et le destinataire, du **ping-pong** où dans ce cas le destinataire attend d'avoir reçu le message avant de le renvoyer à nouveau

La diffusion (*One-To-All* ou *Broadcasting*) : ce schéma consiste à envoyer un message identique à tous les processeurs à partir d'un processeur particulier appelé **racine** de la diffusion. Une variante de ce schéma consiste à ne diffuser la même information qu'à un sous-ensemble de processeurs (on parlera alors de *One-To-Many* ou de *multi-cast*).

Ce schéma de communication se retrouve par exemple dans les algorithmes parallèles de résolution de systèmes linéaires, comme dans Gauss, où il y a diffusion du pivot à chaque étape [39].

L'échange total (*All-To-All* ou *gossiping* ou *total exchange*) : consiste à effectuer une diffusion simultanée de tous les nœuds. Cet schéma est utilisé dans une version de l'algorithme du produit matrice-vecteur [37] ou encore dans l'algorithme du gradient conjugué [39, 96].

La distribution (*Personalized-One-To-All* ou *scattering*) : il s'agit de la version personnalisée de la diffusion, c'est-à-dire que la racine ne diffuse pas le même message à tous les autres, mais un message spécifique à chacun des processeurs. Un exemple de de schéma est la distribution initiale de données. L'opération inverse est appelée **rassemblement** (ou *gathering*). Le rassemblement est généralement utilisé lors de la récupération de résultats locaux détenus par chacun des processeurs.

La multi-distribution (*Personalized-All-To-All* ou *complete exchange*) : comme l'échange total est l'extension à tous les processeurs de la diffusion, la multi-distribution est une distribution simultanée depuis tous les nœuds.

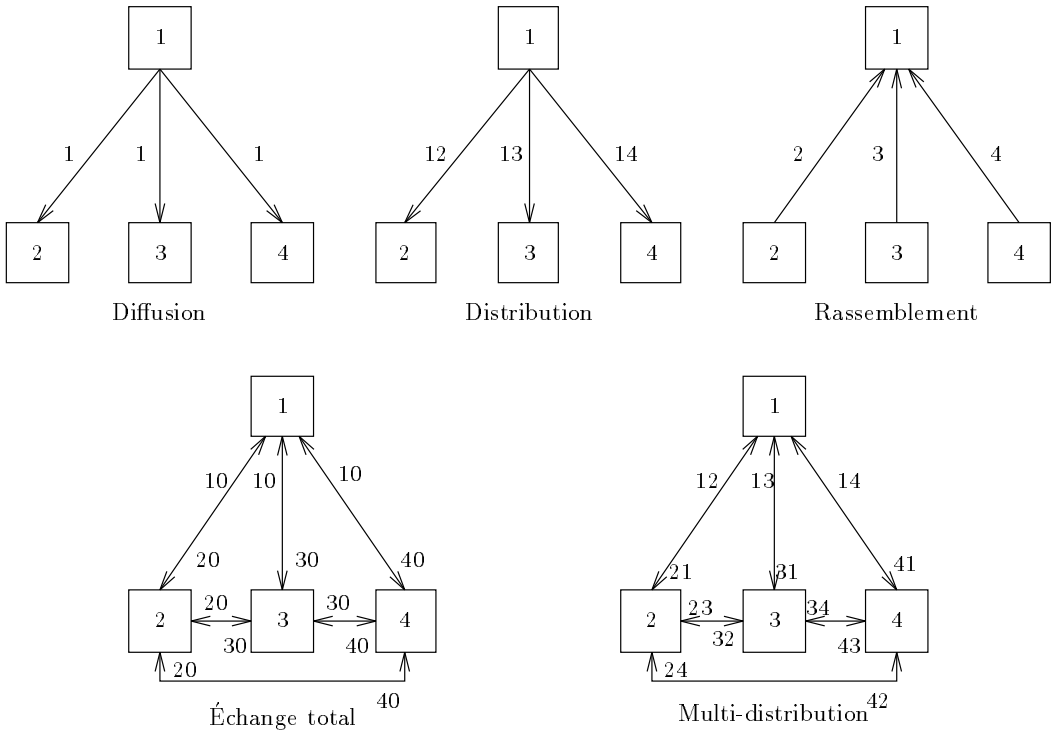


FIG. 1.1 - *Communications globales.*

Il existe d'autres communications globales qui sont soit d'ordre du contrôle de l'exécution du programme, comme la **synchronisation**, soit qui rajoutent une combinaison entre les données échangées (*global combine*).

Nous allons maintenant voir quelques outils issus de la théorie des graphes, permettant de concevoir des algorithmes de communications[1]. Nous donnons quelques exemples d'algorithmes de communication globale illustrant leur utilisation. Tous les algorithmes décrits ci-dessous supposent un modèle Δ-port, *full-duplex* (voir partie I chapitre 1).

1. Une description plus précise de certains de ces outils pourra être trouvé dans [96] et [78].

1.2 Communication point-à-point

Nous supposons qu'un nœud A veut envoyer un message de longueur L à un nœud B situé à une distance d. La distance utilisée est la longueur du chemin le plus court parmi tous les chemins existant entre A et B.

1.2.1 Le pipeline de message

Il s'agit de réduire le temps de transmission du message lorsque le mode de commutation utilisé est la commutation de messages. Pour cela, le message est découpé en q paquets, ceux-ci étant envoyés les uns à la suite des autres de façon pipeline. Il existe une taille optimale de paquet permettant de minimiser le temps de transfert du message. Ce temps minimum pour transférer un message de taille L est alors :

$$\left(\sqrt{(d-1)\beta} + \sqrt{L\tau}\right)^2$$

Dans le cas de messages très longs, le temps de communication permet de masquer l'influence de la distance, et nous ramène à une communication proche du *wormhole* puisque le chemin est bloqué pendant le transfert.

1.2.2 Les chemins disjoints

Il s'agit de trouver le plus grand nombre de chemins à arêtes disjointes de longueur minimale entre les nœuds A et B. Supposons qu'il existe θ chemins de longueur d', $\theta \leq \Delta$. Alors il est possible de diviser le message en θ paquets et de router ces paquets sur chacun des chemins. Cela conduit à un temps de :

$T = d'\left(\beta + \frac{L}{\theta}\right)$	en commutation de messages
$T = \alpha + d'\delta + \frac{L}{\theta}$	en *wormhole*
$T = \left(\sqrt{(d'-1)\beta} + \sqrt{\frac{L}{\theta}\tau}\right)^2$	en commutation de messages pipeline (ou commutation de paquets)

Nous présentons sur la figure 1.2 quatre chemins à arêtes disjointes de longueur $d' = d + 2$ sur le tore [108] et trois chemins à arêtes disjointes sur un 3-cube sur la figure 1.3.

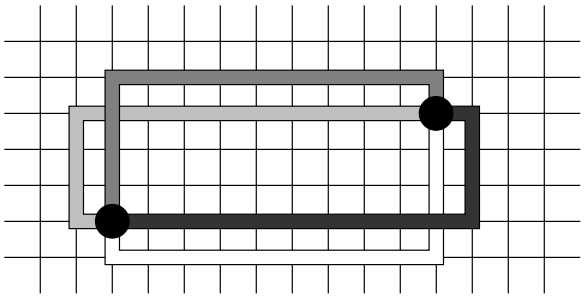


FIG. 1.2 - *Quatre chemins à arêtes disjointes sur le tore.*

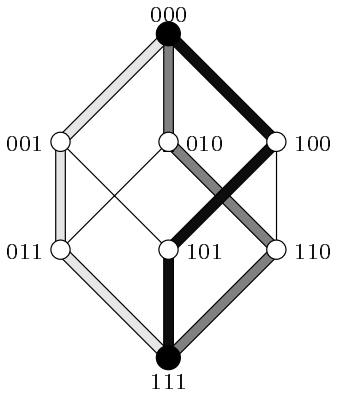


FIG. 1.3 - *Trois chemins à arêtes disjointes sur le 3-cube.*

1.3 Communications globales

1.3.1 Commutation de messages

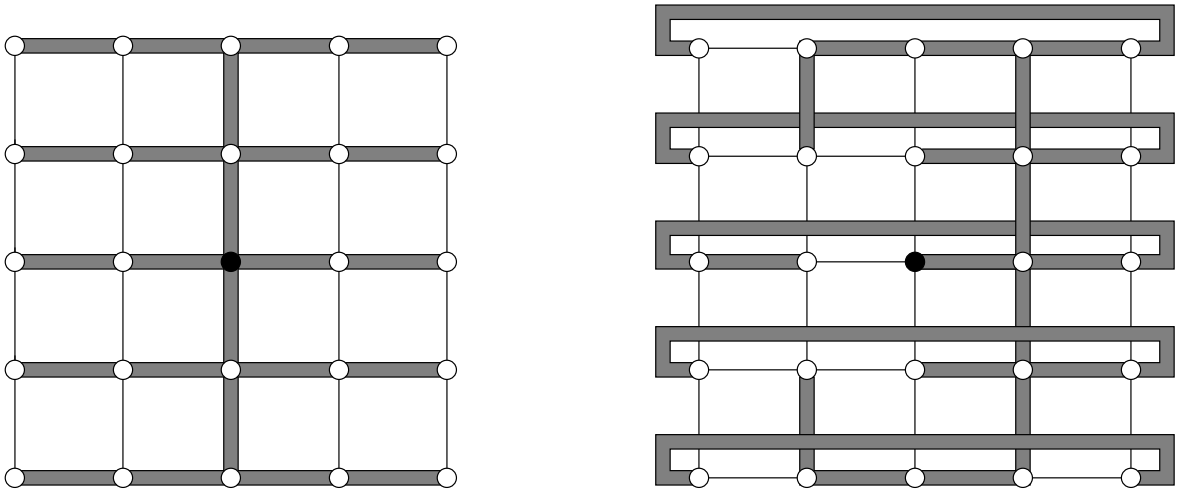
Les arbres couvrants

Les algorithmes de communication globale en commutation de messages ont été très largement étudiés et sont source d'une bibliographie importante. Le lecteur pourra se référer à quelques ouvrages ou articles de référence pour plus de détails [39, 59, 60, 78, 82, 90, 108, 113]. Nous ne prétendons pas citer toutes les techniques, ni décrire tous les algorithmes de communication globale, mais plutôt donner quelques outils généralement utilisés que nous illustrons par des exemples d'algorithmes.

Parmi toutes les techniques utilisées dans ces algorithmes, un premier outil est **l'arbre de recouvrement**. Nous allons rappeler deux définitions utiles pour la suite :
Définition 1 :

- *Étant donnés un graphe $G = (V, E)$ et un arbre T de racine x, $(x \in V)$, on dit que T est un **arbre de recouvrement** (ou **arbre couvrant**) s'il contient tous les nœuds du graphe.*
- *Deux arbres couvrants de G sont à arêtes disjointes s'ils n'ont pas d'arêtes en commun.*

Il s'agit de trouver un maximum d'arbres de profondeur minimale à arêtes disjointes. Nous présentons deux exemples d'arbres de recouvrement sur le tore $(5, 5)$ sur la figure 1.4. Le premier arbre couvrant (figure de gauche) est de profondeur minimale (c'est-à-dire égale au diamètre). Le second est plus profond $(D + 2)$, cependant il en existe 4 de ce type (par rotation de $\frac{\pi}{2}$) qui sont à arêtes disjointes [96].



Arbre de recouvrement du tore de profondeur D — Arbre de recouvrement du tore de profondeur $D+2$

FIG. 1.4 - *Arbres de recouvrement du tore.*

Il est possible dans la plupart des cas de construire des arbres de recouvrement du tore de profondeur minimale, c'est-à-dire égale à $D+1$ [96].

Sur la figure 1.5 nous présentons trois arbres binomiaux de recouvrement de l'hypercube de dimension 3 [82]. Il est possible d'en trouver autant que le degré de l'hypercube et de profondeur minimale, par rotation successive suivant les dimensions. Ces arbres sont notés SBT_i (pour *Spanning Binomial Tree*) où i indique la première dimension à emprunter.

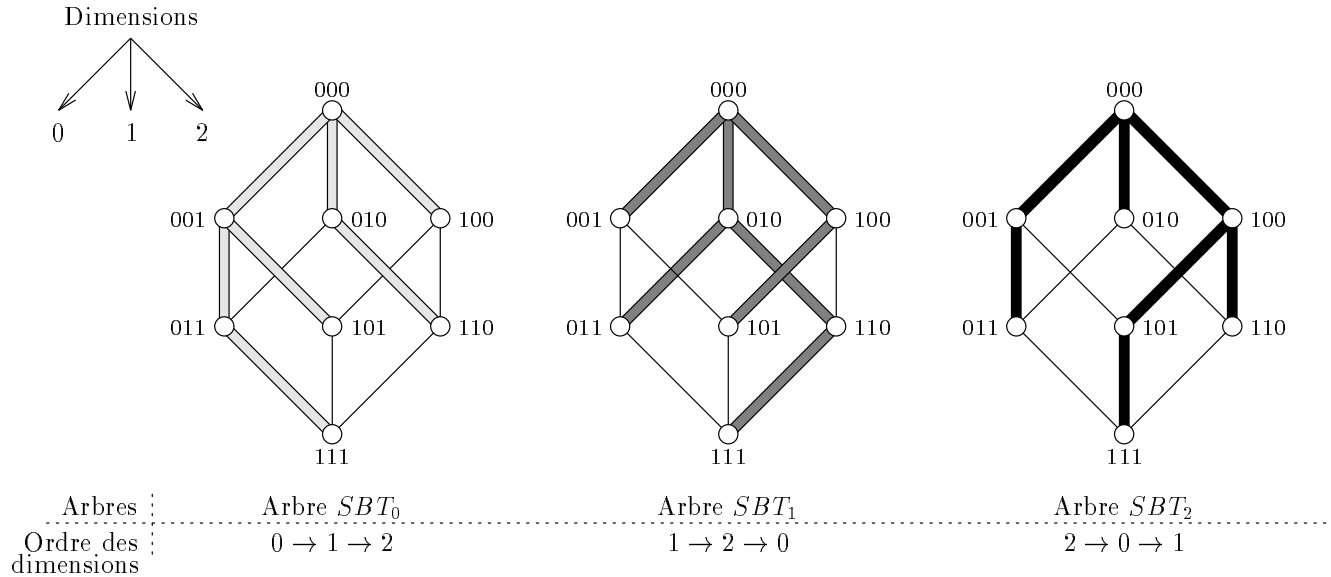


FIG. 1.5 - *Arbres binomiaux de recouvrement de l'hypercube*

Le nombre d'arbres couvrants dans un graphe quelconque est donné par le théorème d'Edmonds [52].

Les arbres couvrants sont fort utiles pour diffuser un message ou réaliser un échange total. Il est possible d'utiliser des techniques de pipeline ou de diviser le message à diffuser en autant de paquets qu'il existe d'arbres couvrants. Nous allons voir quelques exemples d'algorithmes basés sur ces arbres couvrants.

Application à la diffusion dans le tore

Nous présentons sur la figure 1.6 deux algorithmes de diffusion dans un tore 2D. Ces deux algorithmes reposent sur le premier arbre de recouvrement (schéma de gauche sur la figure 1.4). Sans technique de pipeline des messages [39, 96] cela conduit à un algorithme en nombre d'étapes égal au diamètre de la grille. Le premier de ces deux algorithmes est basé sur la diffusion pipeline du message suivant la verticale, puis ensuite chacun des nœuds atteint lors de cette première phase ré-effectue une diffusion pipeline sur l'horizontale (voir schéma de gauche sur la figure 1.6). Le temps de ce premier algorithme, pour des tores de dimension $(\sqrt{P}, \sqrt{P})$ est :

$$2\left(\sqrt{\left(\left\lfloor\frac{\sqrt{P}}{2}\right\rfloor - 1\right)\beta} + \sqrt{L\tau}\right)^2$$

Le deuxième algorithme effectue une diffusion pipeline sur les deux dimensions en même temps, les processeurs étant informés suivant des fronts successifs (voir schéma de droite sur la figure 1.6). Le temps de ce deuxième algorithme est alors :

$$\left(\sqrt{\left(2\left\lfloor\frac{\sqrt{P}}{2}\right\rfloor - 1\right)\beta} + \sqrt{L\tau}\right)^2$$

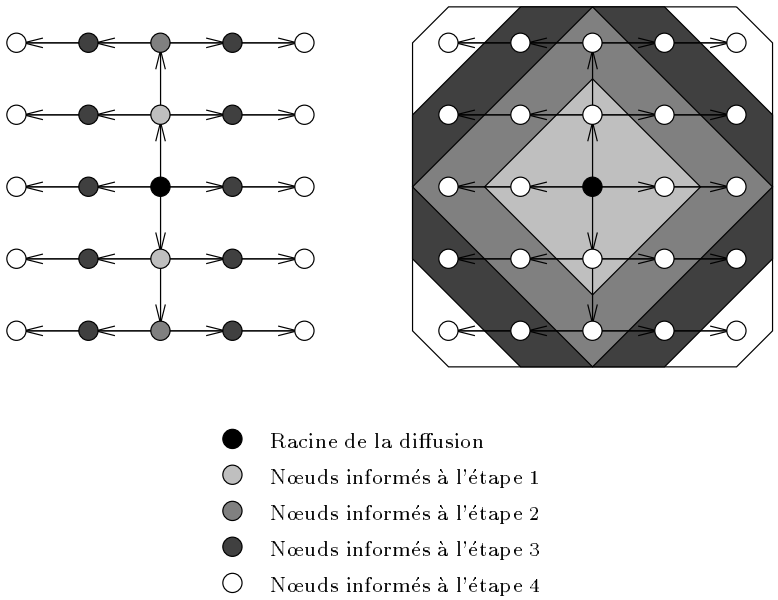


FIG. 1.6 - *Algorithmes de diffusion dans le tore.*

Un algorithme optimal a été donné par Michallon, Trystram et Villard [97, 98] sous les mêmes hypothèses, et étudié également par Bermond, Michallon et Trystram dans le cas de liens mono-directionnels [12]. Ces algorithmes sont basés sur les 4 arbres couvrants présentés *supra* (schéma de droite sur la figure 1.4). Le temps de cet algorithme est le suivant :

$$\left(\sqrt{\left(2\left\lfloor\frac{\sqrt{P}}{2}\right\rfloor-1\right)\beta}+\sqrt{\frac{L}{4}\tau}\right)^2$$

Application à la diffusion dans l'hypercube

Les algorithmes de diffusion dans l'hypercube reposent sur les arbres de recouvrement binomiaux (voir figure 1.5). Un premier algorithme consiste à utiliser un arbre couvrant et à effectuer une diffusion pipeline sur cet arbre. Le temps de cet algorithmes est le suivant :

$$\left(\sqrt{(log_2(P)-1)\beta}+\sqrt{L\tau}\right)^2$$

Un deuxième algorithme consiste à utiliser les $log_2(P)$ arbres binomiaux de façon simultanée (voir la figure 1.7), ce qui conduit à un algorithme dont le temps est :

$$log_2(P)\left(\beta+\frac{L}{log_2(P)}\tau\right)$$

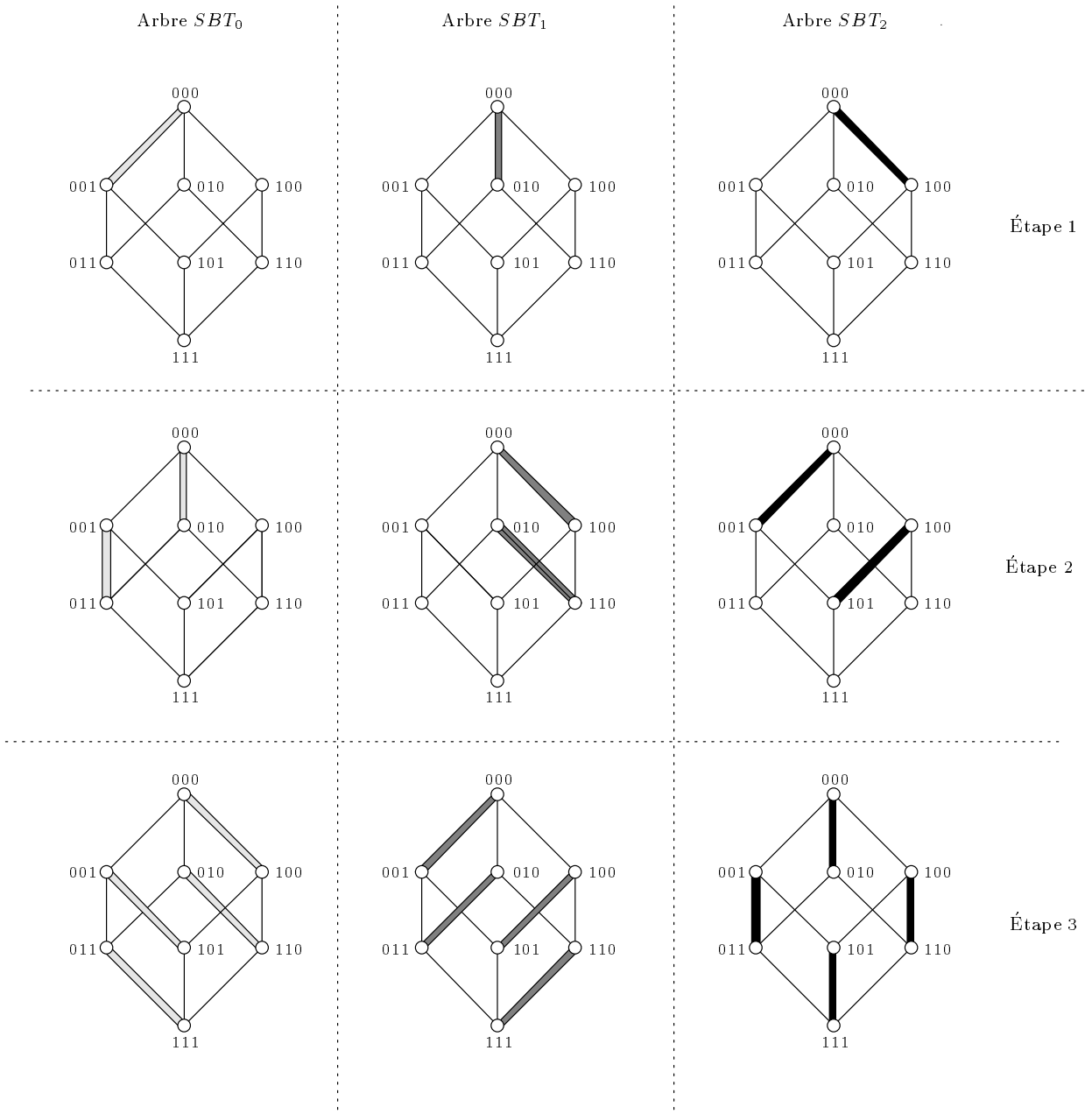


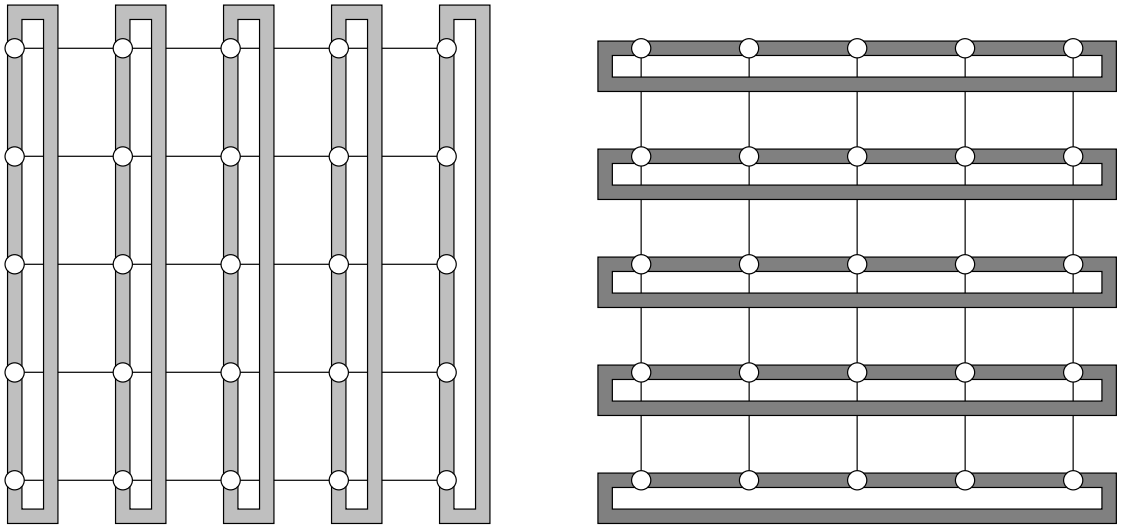
FIG. 1.7 - *Algorithme de diffusion dans l'hypercube basé sur les arbres SBT.*

Une autre technique appliquée au problème de l'échange total : la composition de graphe

Dans le cas de l'échange total sur le tore, les arbres présentés ne conviennent pas. Une première solution consiste alors à considérer le tore comme un produit cartésien d'anneaux. Ceci est un exemple d'un autre outil de la théorie des graphes utilisé dans la conception d'algorithmes de communication : **la composition de graphes** [78, 90]. Nous présentons le principe de cet algorithme d'échange total sur le tore $(5, 5)$ sur la figure 1.8. L'algorithme

consiste dans une première phase à effectuer un échange total des messages détenus par chaque sommet suivant les anneaux verticaux (*gossip* avec des messages de longueur L) puis à faire de même sur les anneaux horizontaux avec l'information accumulée dans la phase précédente (*gossip* avec des messages de longueur $\sqrt{P} \times L$). Le temps de cet algorithme est le suivant :

$$\left\lfloor \frac{\sqrt{P}}{2} \right\rfloor \left(2\beta + (\sqrt{P}+1)L\tau\right)$$



Échange total suivant les anneaux verticaux avec des messages de longueur L

Échange total suivant les anneaux horizontaux avec des messages de longueur $\sqrt{P} \times L$

FIG. 1.8 - *Algorithme d'échange total dans le tore basé sur le produit cartésien d'anneaux.*

1.3.2 Commutation de circuit et *wormhole*

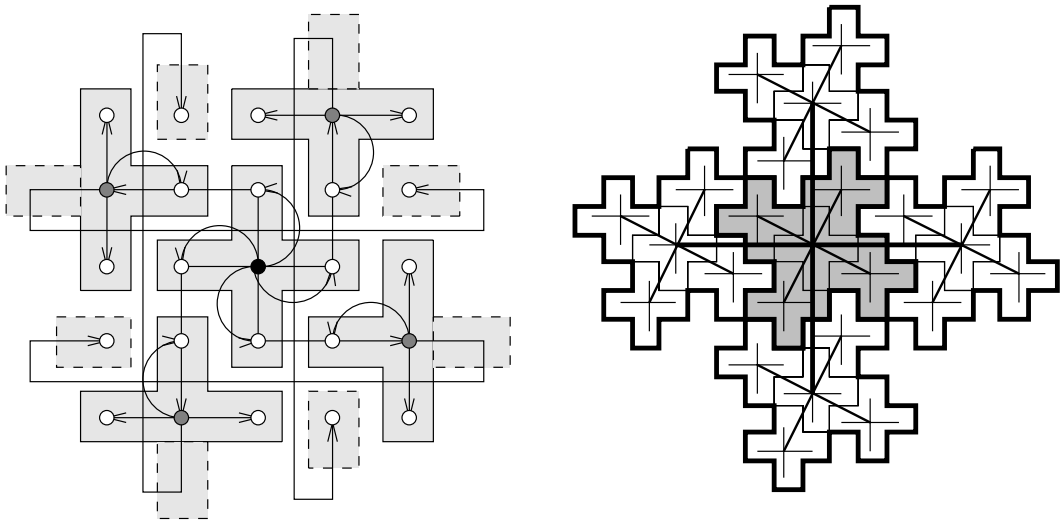
Le problème en commutation de circuits et *wormhole* n'est pas tout à fait le même. En effet, si en commutation de messages la distance est un facteur critique, il devient négligeable en *wormhole*. En revanche le problème de la contention sur les liens devient prépondérant dans ce dernier mode de commutation [120]. Les outils utilisés vont avoir pour but d'éviter les contentions. Par exemple, en commutation de messages, il est possible d'utiliser des arbres de recouvrement à arêtes disjointes dans le temps. C'est-à-dire à une étape de temps donnée il n'y a pas une arête utilisée par deux chemins. La contrainte est plus forte dans le cas du *wormhole* et de la commutation de circuit, puisqu'un chemin est bloqué pendant toute la durée de la communication. Par contre la profondeur des arbres utilisés sera moins discriminante.

Une première façon d'envisager les algorithmes de communication globale en *wormhole* est l'adaptation directe des algorithmes conçus en commutation de messages à ce nouveau mode de commutation, en effectuant uniquement des communications entre voisins directs (c'est-à-dire à distance 1). Cela conduit généralement à des algorithmes ayant un bon facteur de

taux de transmission (qui reste optimal si c'est déjà le cas en commutation de messages) mais souvent catastrophique en nombre d'étapes. L'essentiel du travail consiste alors à trouver de nouveaux algorithmes profitant de l'insensibilité de la distance, propre au *wormhole*, tout en gardant un taux de transmission « acceptable ».

Algorithme de diffusion dans les tores

Nous allons présenter un premier exemple d'algorithme optimal en nombre d'étapes réalisant une diffusion dans les tores 2D carrés dont la taille est une puissance de 5. Cet algorithme proposé par Peters et Syska [104] nous sera utile par la suite. L'idée de base repose sur la décomposition récursive du tore. Un exemple de cet algorithme est donné sur le tore $(5,5)$ sur la figure 1.9 ainsi que la décomposition récursive sur un tore $(25,25)$.



Algorithme de diffusion sur le tore $(5,5)$

Principe appliqué récursivement

FIG. 1.9 - *Algorithme de diffusion dans le tore en wormhole.*

Le principe de l'algorithme est donné par le schéma de gauche de la figure 1.9. La racine de la diffusion (nœud en noir) envoie son message au centre des différentes croix (nœuds en gris) par un « schéma en diagonale ». La deuxième étape consiste alors à diffuser à partir de chaque centre de croix vers les 4 voisins par un « schéma en croix ». Pour des tailles de tore plus grandes, on utilise un pavage du tore en croix (voir schéma de droite sur la figure 1.9), et on applique récursivement le principe de l'algorithme du tore $(5,5)$ par une succession de diffusions en utilisant des « schémas en croix » et des « schémas en diagonale »[2]. Cela conduit à un algorithme optimal en nombre d'étapes dont le temps est pour des tores carrés de taille $(5^k, 5^k)$:

$$\log_5(P)\alpha + (\sqrt{P}-1)\delta + 2\log_5(P)L\tau$$

2. Nous verrons plus en détail dans le chapitre 3 le pavage du tore et les propriétés qui lui sont associées.

Une idée très intéressante que l'on peut tirer de cet algorithme, est la mise en évidence du rôle particulier de certains nœuds (les centres des croix dans l'algorithme précédent). Tsai et McKinley se sont basés sur une notion de **nœud dominant** pour définir une méthodologie de conception d'algorithmes de communication globale appliquée aux grilles et aux tores multi-dimensionnels en commutation *wormhole* [121, 122].

Un outil pour la conception d'algorithmes de communication globale en commutation de circuit et *wormhole*

Rappelons la définition d'un ensemble de nœuds dominants :

Définition 2 :

> *Un ensemble $\mathcal{D}$ de nœuds dominants dans un réseau direct est un sous-ensemble de nœuds tel que chaque sommet du réseau, soit appartient à $\mathcal{D}$, soit est au moins voisin avec un des nœuds de $\mathcal{D}$.*

Nous présentons sur la figure 1.10 un ensemble de nœuds dominants sur la grille 4×4.

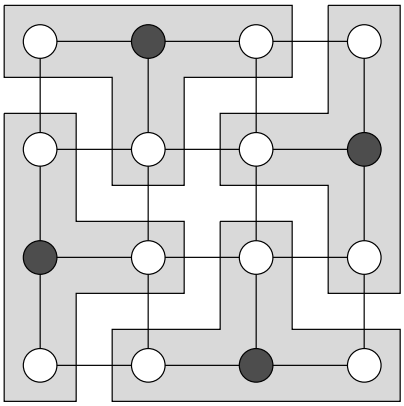


FIG. 1.10 - *Ensemble de nœuds dominants de la grille* 4×4.

Tsai et McKinley étendent cette définition [121] pour profiter de l'avantage du *wormhole* qui rend les algorithmes « insensibles » à la distance. De plus, ils tiennent compte d'un autre facteur qui est souvent négligé : l'algorithme de routage. Sans entrer dans les détails, l'algorithme de routage calcule la route à emprunter pour acheminer un message à travers le réseau [102]. Dans les réseaux directs de type grille ou tore, l'algorithme de routage adopté est généralement celui par **ordre des dimensions** (*dimension order routing*) [45]. En effet, ce type de routage déterministe (cela implique qu'un message bloqué ne peut changer de route) favorise les problèmes de contention. Tsai et McKinley introduisent donc la notion d'**ensemble dominant étendu** (*Extended Dominating Set*, noté EDS):

Définition 3 :

Considérons un réseau direct Δ-port à mode de commutation wormhole. Nous notons V l'ensemble des sommets du réseau et $\mathcal{R}$ l'algorithme de routage utilisé. Considérons deux ensembles de nœuds $\mathcal{D}_1$ et $\mathcal{D}_2$ tels que $\mathcal{D}_1 \subseteq V$ et $\mathcal{D}_2 \subset \mathcal{D}_1$. $\mathcal{D}_2$ est un ensemble dominant étendu, si et seulement si, il existe un ensemble de chemins $\mathcal{P}$ à arêtes disjointes suivant le routage $\mathcal{R}$ tel que, pour chaque sommet v de $\mathcal{D}_1 - \mathcal{D}_2$, il existe un sommet x, $x \in \mathcal{D}_2$, et un chemin p, $p \in \mathcal{P}$, menant de x à v.

Nous donnons un exemple d'EDS sur la grille $(4,4)$ sur la figure 1.11.

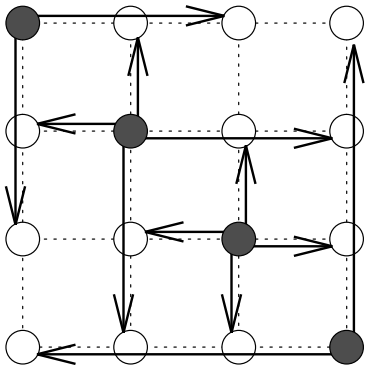


FIG. 1.11 - *Ensemble EDS de la grille* $(4,4)$.

Un élément d'un EDS est appelé **nœud dominant étendu** (*Extended Dominating Node* noté EDN). Si tous les EDN d'un réseau détiennent une copie d'un même message, ils peuvent alors la diffuser en un seul envoi à tous les autres sommets du réseau. L'idée est alors de trouver un sous-ensemble de ces EDN, appelons-le $\mathcal{D}_2$, tel que les éléments de $\mathcal{D}_\in$ soient capables d'envoyer un message aux autres EDN en une seule étape, sans générer de contention. Si les éléments de $\mathcal{D}_2$ contiennent une copie d'un même message, ils peuvent l'envoyer alors aux autres EDN qui ensuite eux-même le diffusent aux autres nœuds du graphe : nous obtenons un algorithme récursif de diffusion, d'où la définition de **niveaux** d'EDS :

Définition 4 :

Considérons un réseau direct Δ-port à mode de commutation wormhole. Nous notons V l'ensemble des sommets du réseau et $\mathcal{R}$ l'algorithme de routage utilisé. Considérons un ensemble $X \subset V$ et un ensemble $\mathcal{D}_i \subset V$. $\mathcal{D}_i$ est un ***ensemble dominant étendu de niveau i*** *(noté EDS_i) de X si et seulement si $\mathcal{D}_i$ est un EDS_1 d'un EDS_{i-1} de X.*

Nous donnons un exemple d'utilisation de ces outils pour la diffusion dans la grille $(8,8)$ en routage de type XY[3] sur la figure 1.12.

3. Le routage XY est un algorithme de routage par ordre de dimension qui consiste à acheminer d'abord le message suivant l'horizontale (axe des X), puis suivant la verticale (axe des Y).

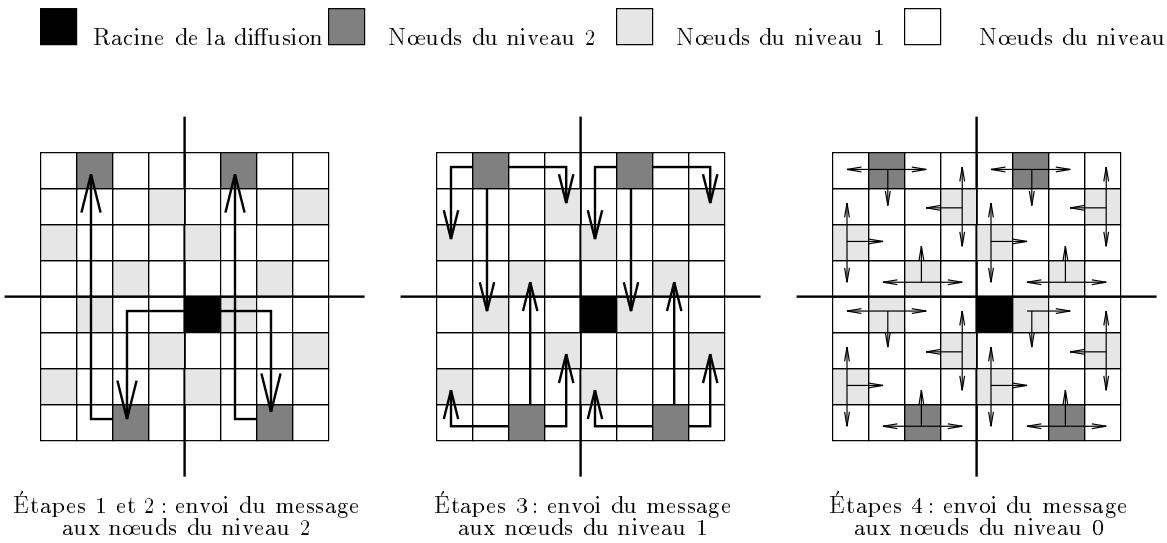


FIG. 1.12 - *Diffusion dans la grille* $(8,8)$.

Le nombre d'étapes dépend bien évidemment de la position relative de la racine par rapport aux nœuds du niveau 2. Mises à part ces une ou deux premières étapes d'envoi de la racine vers les nœuds du niveau le plus haut, le nombre d'étapes est en $\mathcal{O}(k)$ pour des grilles carrées de côté $q \times 2^k$.

Cette méthodologie est appliquée à d'autres types d'algorithmes de communication globale, comme le regroupement, la transposition de matrices [121], ou à d'autres réseaux comme les tores 2D et 3D [121, 122].

Cette formalisation est très intéressante car elle s'applique à un grand nombre d'algorithmes qui ont été conçus sans l'utiliser mais qui y correspondent totalement [29, 104], comme nous le verrons dans le chapitre 3.

1.4 Conclusion

Nous avons présenté, dans ce chapitre, quelques outils et techniques issus de la théorie des graphes permettant de concevoir des algorithmes de communication efficaces. Nous avons illustré leur utilisation sur quelques exemples, notamment pour la conception d'algorithmes sur les tores.

L'essentiel des recherches sur les algorithmes de communication globale se fait maintenant en commutation de circuit ou en *wormhole* car c'est le modèle utilisé par les machines de la dernière génération. Cependant, les algorithmes en commutation de messages restent intéressants car ils s'avèrent très efficaces lorsque la taille des messages est grande.

Comme nous avons pu le voir les méthodes utilisées ne sont plus les mêmes, car les contraintes ont changé. De plus, il est évident que d'autres outils de la théorie des graphes, ou d'autres domaines, devront être employés. En effet, il faut noter qu'aujourd'hui aucun algorithme de communication globale ne prend en compte la différence entre la commutation

de circuits et le *wormhole*. Un autre facteur souvent négligé dans les algorithmes de communication est le temps de gestion des messages. Le temps de transfert entre le réseau et la mémoire n'est pas négligeable et la recomposition des messages peut s'avérer très coûteuse. C'est-à-dire qu'un algorithme optimal en théorie, mais ne tenant pas compte de ce facteur risque d'être catastrophique en pratique.

Il y a certainement un grand enjeu à concevoir des communications globales efficaces, déjà du fait de leur incoporation dans des standards de bibliothèques comme MPI. De plus, la majorité des dernières machines parallèles disposant de nœud de communication, les temps d'initialisation sont souvent très grands par rapport aux débits possibles des liens. Cela implique qu'il est souvent très coûteux de réaliser une communication globale par une suite de communications point-à-point. Cependant, pour être efficaces ces schémas doivent être implémentés directement au niveau des nœuds de communication, ou au moins utiliser les protocoles de plus bas niveau de la machine.

Chapitre 2

Transposition de matrices allouées par blocs

Nous nous intéressons dans ce chapitre au problème de la transposition de matrices carrées allouées par blocs sur machines parallèles à mémoire distribuée. La méthodologie proposée est basée sur la décomposition des graphes modélisant les réseaux d'interconnexion en chemins ou en cycles. Nous utilisons alors des schémas élémentaires de communication pour réaliser des transpositions partielles sur ces chemins et cycles. Ce travail a été réalisé en collaboration avec D. Trystram[a] *[30].*

[a] D. Trystram est professeur d'informatique à l'École Supérieure de Génie Industriel de Grenoble.

2.1 Introduction

La parallélisation de la plupart des méthodes numériques entraîne des mouvements de données réguliers [36, 55, 63, 108, 113]. L'opération de transposition matricielle est courante dans de nombreux algorithmes numériques. Le schéma de communication engendré par cette opération dépend directement alors de la distribution des données [36, 51, 69, 82]. Nous allons étudier dans ce chapitre, la mise en œuvre de ce schéma pour des matrices allouées par blocs, sur des machines parallèles à mémoire distribuée. Nous proposons un schéma général basé sur la décomposition des réseaux de communications, en chemins ou cycles élémentaires, permettant la réalisation de la transposition. Cette méthode nous permet d'obtenir des algorithmes efficaces pour des réseaux qui admettent une décomposition de ce type. Nous appliquons ce schéma au tore et au de Bruijn (*cf* définitions des réseaux partie I, chapitre 1).

Nous considérons P processeurs connectés entre eux par un réseau de communication. Nous supposons que le mode de routage est soit en commutation de messages, soit *wormhole* et que les liens sont bi-directionnels (*cf* partie I, chapitre 1). Nous supposons également un modèle k-port, avec k=2 et k=4.

Nous rappelons dans une première partie les définitions de deux types d'allocations par blocs (consécutive et cyclique). Nous décrivons, ensuite, une série de travaux précédents en commutation de messages : nous commençons d'abord par rappeler le principe de l'algorithme

de transposition récursive et son implantation sur l'hypercube. La version pipelinée de cet algorithme, est à l'heure actuelle, le meilleur résultat connu sur cette topologie. Vient ensuite un algorithme sur la grille décrit par Johnsson dans [81]. Enfin, nous montrons comment adapter des résultats de routage de permutations sur l'hypercube [15] au tore 2D et au de Bruijn en utilisant des méthodes d'émulation [90].

Le paragraphe suivant est consacré à des résultats préliminaires sur deux échanges élémentaires de transposition sur un chemin et un cycle. Nous proposons alors un schéma général pour réaliser une transposition de matrice sur n'importe quel réseau partitionné en ces chemins ou cycles. Cette méthodologie est appliquée au tore bi-dimensionnel et au de Bruijn dans le paragraphe 4. Nous améliorons ces algorithmes en utilisant les liens inoccupés et nous en calculons les complexités. Nous présentons ensuite les bornes inférieures de ce problème en commutation de message et nous les comparons aux complexités de nos algorithmes.

Cette méthodologie conçue initialement pour un mode de commutation de type *store-and-forward* (qui correspondait aux machines à base de Transputers) s'adapte également à un mode de commutation de type *wormhole*. Avant de conclure, nous montrons donc comment étendre les différents algorithmes présentés à cet autre mode de commutation, et nous comparons les complexités de ces algorithmes aux bornes inférieures. Nous rappelons également les travaux existants sur le problème de la transposition de matrices en *wormhole* sur des grilles bi-dimensionnelles.

2.2 Description des allocations de données

L'opération de transposition d'une matrice consiste à associer l'élément en ligne i et colonne j, avec l'élément situé en ligne j et en colonne i. Le schéma de communication généré par cette opération va donc dépendre de l'allocation des données sur les processeurs.

Etant donnés une matrice carrée A de taille N et P processeurs, il existe de nombreuses façons de distribuer les éléments de la matrice sur les processeurs de façon équilibrée [33, 55, 84, 99, 101]. Par souci de simplicité, nous supposons dans la suite de ce chapitre, que $\frac{N}{P}$ est entier.

Allocation par lignes (colonnes) :

La matrice est subdivisée en P blocs de tailles équilibrées de $\frac{N}{P}$ lignes (respectivement colonnes) (voir figure 2.1). Pour ce type d'allocation, le problème de la transposition correspond à un échange total personnalisé (ou multi-distribution) [82] : chaque processeur doit envoyer une partie des lignes (respectivement colonnes) qu'il possède à chacun des autres processeurs, c'est-à-dire : $\frac{1}{P} \times \left(N \times \frac{N}{P}\right) = \left(\frac{N}{P}\right)^2$ données. Nous rappelons que le coût des meilleurs algorithmes de multi-distribution en commutation de message avec des messages de longueur L sont respectivement pour le tore et l'hypercube $\sqrt{P}\left(\beta + \frac{PL}{8}\tau\right)$ [108] et $\log_2(P)\left(\beta + \frac{L}{2\log_2(P)}\tau\right)$ [51, 82]. Les temps de ces algorithmes appliqués à la transposition de matrice, avec des messages de longueur $\left(\frac{N}{P}\right)^2$ sont donnés dans la table 2.1 [1].

1. À notre connaissance, il n'existe aucun algorithme de multi-distribution sur le de Bruijn.

Tore [108]	$\sqrt{P}\beta + \frac{N^2\sqrt{P}}{8}\tau$
Hypercube [82]	$\log_2(P)\beta + \frac{N^2}{2P}\tau$

TAB. 2.1 - *Coûts des algorithmes de multi-distribution appliqués à la transposition de matrice.*

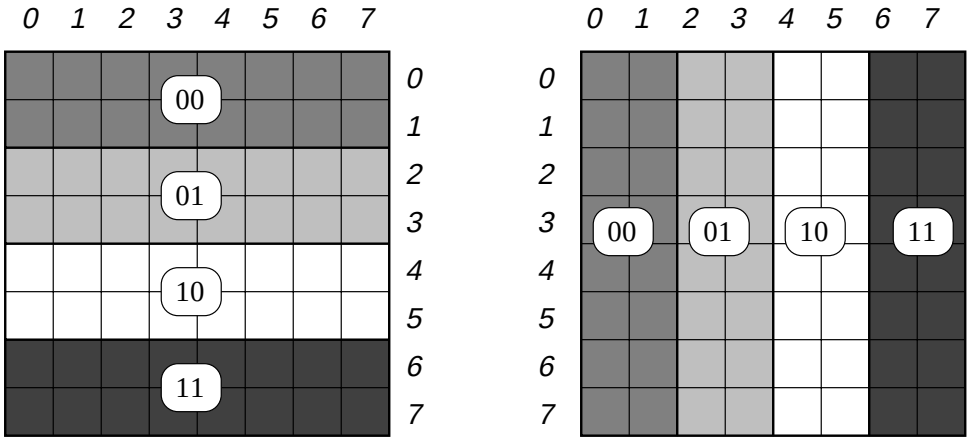


FIG. 2.1 - *Exemples d'allocation par lignes et par colonnes pour $N = 8$ et $P = 4$.*

Allocations par blocs :

Une autre façon d'allouer une matrice est de la subdiviser en P sous-blocs, et d'associer chacun des sous-blocs à un processeur. En général, du fait que les sous-blocs sont rectangulaires, le problème de la transposition de matrices correspond à une multi-distribution [33]. Cependant, si nous supposons que $\sqrt{P}$ est entier, alors les sous-blocs sont carrés, et la transposition de matrices se ramène à un ensemble d'échanges entre paires de processeurs. Nous supposerons donc, dans le reste de ce chapitre que P est tel que $\sqrt{P}$ soit entier[2].

Nous avons représenté dans la figure 2.2 une *allocation consécutive par blocs* et une *allocation cyclique par blocs*. Soit $(i)_2$ l'écriture en base 2 de l'entier i, et $|$ l'opérateur de concaténation de bits. Nous désignons par *Alloc(i,j)* le processeur qui contient l'élément (i, j), quels que soient i et j tels que $0 \leq i, j \leq \frac{N}{\sqrt{P}} - 1$. Les fonctions d'allocations par blocs utilisées sont définies de la façon suivante :

2. Ce qui implique que $log_2(P)$ est un pair.

$$Alloc(i,j) = \left(\left\lfloor i \times \frac{\sqrt{P}}{N} \right\rfloor\right)_2 | \left(\left\lfloor j \times \frac{\sqrt{P}}{N} \right\rfloor\right)_2 \quad \text{Allocation consécutive}$$

$$Alloc(i,j) = (i \bmod \sqrt{P})_2 | (j \bmod \sqrt{P})_2 \quad \text{Allocation cyclique}$$

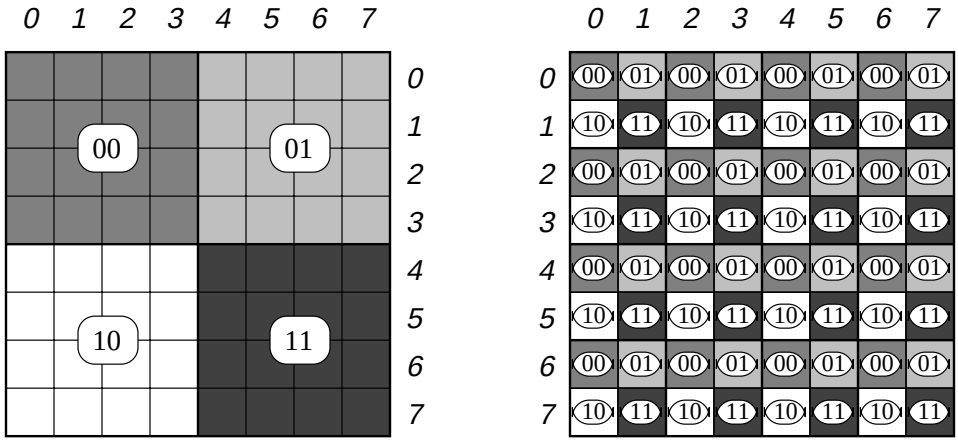


FIG. 2.2 - *Exemples d'allocation consécutive et cyclique pour $N = 8$ et $P = 4$.*

Les algorithmes que nous proposons pour transposer une matrice sont valides pour les deux types d'allocation par blocs présentés. Par souci de clarté, nous ne considérons que l'allocation consécutive par blocs, tout en sachant que les résultats pour la seconde allocation peuvent facilement s'obtenir par adaptation des résultats de la première. Il existe bien évidemment d'autres allocations par blocs, notamment celles qui permettent de réduire le problème de la transposition de matrices à de simples échanges entre voisins directs [71, 91]. Cependant, nous nous intéressons dans ce chapitre aux allocations définies ci-dessus, qui correspondent aux allocations « standard » comme elles ont été définies dans HPF ou dans ScaLAPACK [33, 56]. De plus, la plupart des allocations spécifiques qui sont « optimales » pour un schéma de communication, ne le seront plus pour d'autres schémas, et c'est donc repousser le problème. Cela ne veut pas dire que ce genre de méthodes est inintéressante, mais tout simplement qu'il faut regarder le problème dans sa globalité. L'avantage de prendre des allocations standard est qu'elles correspondent à des « normes » spécifiées dans des langages ou bibliothèques standard.

D'après la définition de l'opération de transposition, et en utilisant la fonction d'allocation par blocs précédente, un processeur doit alors échanger son bloc de données seulement avec un seul processeur. De façon plus précise, le processeur *Alloc(i,j)* échange son bloc avec le processeur *Alloc(j,i)*. Nous appelons cette opération, l'**opération de transposition directe**.

2.3 L'algorithme de transposition récursive et travaux précédents en commutation de message

2.3.1 L'algorithme de transposition récursive

Etant donnée la décomposition par bloc définie précédemment, la transposition de la matrice peut être réalisée récursivement en $\frac{\log_2(N)}{2}$ étapes de la façon suivante [53] : à la première étape, la matrice A à transposer, est subdivisée en 4 sous-blocs :

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$$

L'algorithme de transposition appliqué à la matrice A est le suivant :

```
Procédure Transpose(A)
début
  Échange des sous-blocs sur-diagonaux A01 et A10
  Transpose(A00)
  Transpose(A01)
  Transpose(A10)
  Transpose(A11)
fin
```

FIG. 2.3 - *Algorithme de transposition récursive par blocs.*

2.3.2 Implantation sur l'hypercube en commutation de message

La construction récursive de l'hypercube [107] permet une implantation directe de l'algorithme précédent sur l'hypercube [73]. L'étape S de l'algorithme consiste à échanger les bits S et $\left(S + \frac{\log_2(N)}{2}\right)$ de l'écriture binaire de l'élément (i, j) de la matrice A, pour $S = 0 \ldots \frac{\log_2(N)}{2} - 1$ (voir figure 2.4).

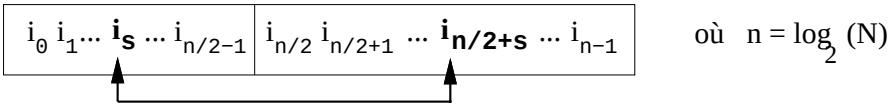


FIG. 2.4 - *Echange de bits au cours de l'étape S dans l'algorithme de transposition récursive.*

Les $\frac{\log_2(P)}{2}$ premières étapes génèrent des communications entre les processeurs. Il est clair que chacune des étapes de récursivité peut être réalisée par des étapes de communications entre voisins (voir figure 2.5) avec des messages de taille $\frac{N^2}{P}$, d'où le temps total de communication de l'algorithme est égal à :

$$T_1^{Hypercube} = \log_2(P)\left(\beta + \frac{N^2}{P}\tau\right)$$

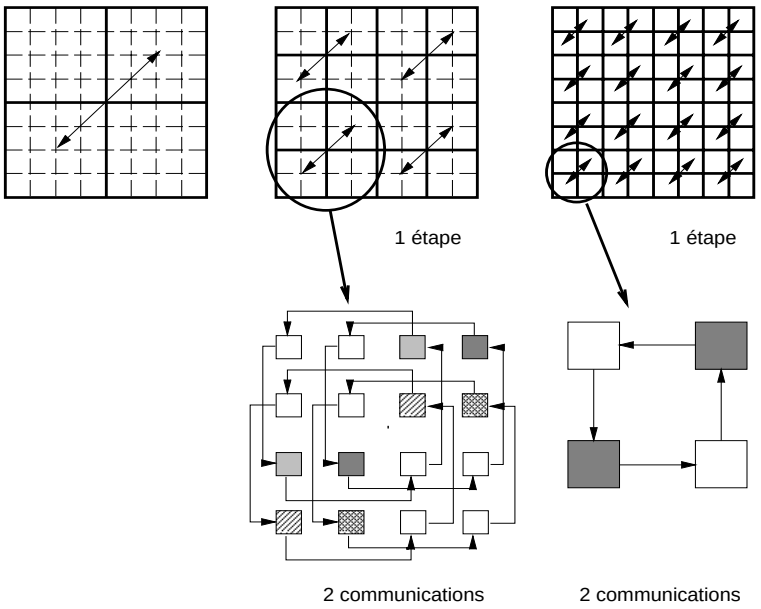


FIG. 2.5 - *Implantation de l'algorithme de transposition récursive sur l'hypercube.*

Si nous utilisons des liens bi-directionnels (*cf* partie I, chapitre 1), les messages peuvent être divisés en deux parties de longueur égale. Chaque message est alors envoyé dans les deux directions à chaque étape. Le coût de l'algorithme devient alors :

$$T_2^{Hypercube} = \log_2(P)\left(\beta + \frac{N^2}{2P}\tau\right)$$

Remarquons, au passage, que le facteur du taux de transmission (τ) est proportionnel à la moitié du diamètre. Nous verrons par la suite que nous obtenons de meilleurs résultats avec notre méthode dans le cas du tore et du de Bruijn.

Johnsson et Ho ont décrit dans [84] l'algorithme « *Multiple Paths Recursive Transpose* » qui utilise des chemins différents entre les nœuds sources et les nœuds destinations. En utilisant une technique de pipeline, cela conduit à un algorithme très efficace, puisque son coût est de (pour de grandes valeurs de N) :

$$T_3^{Hypercube} = \left(\sqrt{\beta} + \sqrt{\frac{N^2}{2P}\tau}\right)^2$$

Remarquons, qu'asymptotiquement, il n'y a qu'un seul temps d'initialisation, et que le taux de transmission est optimal.

2.3.3 Travaux précédents sur la grille en commutation de message

Dans [81] Johnsson décrit un algorithme de transposition sur des grilles de taille $(2^k, 2^k)$ en 2^k étapes de communications entre voisins directs, en décalant successivement les sur-diagonales[3] vers les indices décroissants des colonnes puis vers les indices croissants des

3. sur-diagonale$(s) = \{(i,j)/0 \leq i < (N-s), s \leq j < N\}$.

lignes. De façon symétrique, les sous-diagonales[4] sont d'abord décalées en direction des indices de lignes décroissants, puis vers les indices de colonnes croissants (*cf* figure 2.6 pour les étapes successives de l'algorithme). Il propose une idée d'extension de cet algorithme au tore, en utilisant des chemins disjoints et en pipelinant sur chacun d'eux.

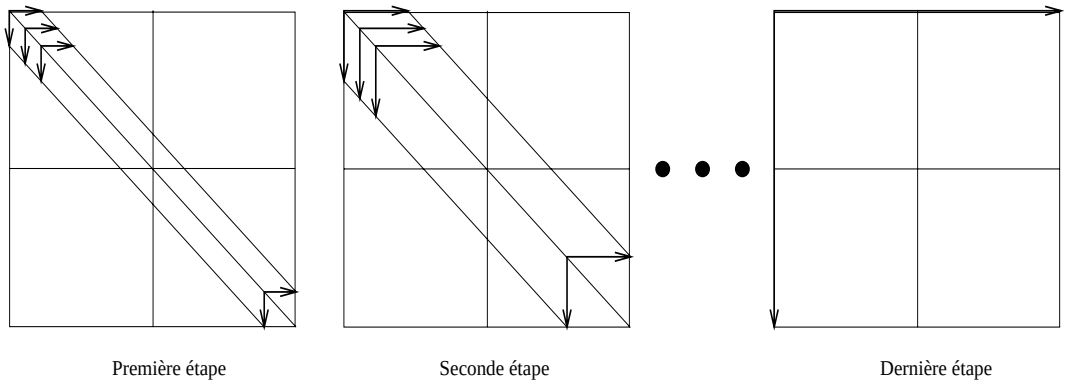


FIG. 2.6 - *Principe de l'algorithme de transposition de Johnsson.*

En utilisant le modèle de temps défini précédemment, le coût de cet algorithme est de :

$$T_1^{Grille} = 2(\sqrt{P} - 1)\left(\beta + \frac{N^2}{P}\tau\right)$$

2.3.4 Utilisation des résultats généraux sur le routage de permutation

Le problème de la transposition de matrices correspond à une permutation qui appartient à la classe des **permutations linéaires complémentées**. Cette classe est définie de la façon suivante :

Définition 5 :

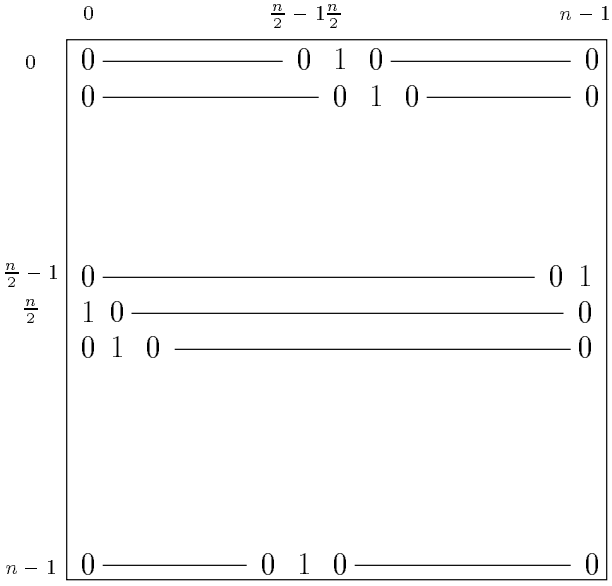
Une permutation sur $V = \{0, 1, \ldots, N-1\}$, *avec* $N = 2^n$, *est dite une permutation linéaire, s'il existe une matrice booléenne* $\mathcal{Q}_{n\times n}$ *non singulière, telle que, pour chauque* x *in*V, *son image,* y, *est donnée par l'équation :*

$$y^T = Qx^T$$

La transposition de matrice est donc une permutation linéaire, puisque qu'à chaque numéro de processeur $(i_0 i_1 \ldots i_{\frac{n}{2}-1} i_{\frac{n}{2}} \ldots i_{n-1})$ son image est le numéro $(i_{\frac{n}{2}} \ldots i_{n-1} i_0 i_1 \ldots i_{\frac{n}{2}-1})$. La

4. sous-diagonale$(s) = \{(i,j)/s \leq i < N\ , 0 \leq j < (N-s)\}$.

matrice booléenne Q est alors la suivante :



Boppana et Raghavendra proposent dans [15] un algorithme optimal de routage de n'importe quelle permutation linéaire complémentée sur l'hypercube (en au plus $\log_2(P)$ étapes). En appliquant cet algorithme de routage au problème de la transposition , cela donne un temps égal à :

$$T_4^{Hypercube} = \log_2(P)\left(\beta + \frac{N^2}{2P}\tau\right)$$

Nous pouvons en déduire un algorithme sur le de Bruijn. En effet, puisque le routage proposé par Boppana et Raghavendra est effectué dimension par dimension, nous pouvons utiliser cet algorithme pour émuler le réseau de de Bruijn à partir de l'hypercube [90]. Donc, nous obtenons un algorithme de transposition en $2\log_2(P)$ étapes sur le de Bruijn (chaque étape de communication suivant une dimension sur l'hypercube, peut être réalisée en 2 étapes sur le de Bruijn), ce qui est proportionnel à deux fois le diamètre. Le coût total de l'algorithme est le suivant :

$$T_1^{DeBruijn} = 2\log_2(P)\left(\beta + \frac{N^2}{2P}\tau\right)$$

La même méthode peut être appliquée au tore, chaque étape concernant la dimension i de l'hypercube correspond à un processeur situé à distance $\frac{\sqrt{P}}{2^i}$ sur le tore. D'où un temps total égal à :

$$T_1^{Tore} = \sum_{i=1}^{log_2(P)} \frac{\sqrt{P}}{2^i} = 2\sqrt{P}\frac{P-1}{P}$$

Les algorithmes spécifiques à la transposition de matrice que nous proposons sont meilleurs que ces derniers en ce qui concerne le tore er le réseau de de Bruijn.

Nous allons, dans un premier temps, présenter l'étude du problème en commutation de message. Nous verrons par la suite comment adapter les algorithmes présentés en mode *wormhole*.

2.4 Quelques résultats préliminaires

Dans ce paragraphe, nous allons définir deux algorithmes d'échanges élémentaires de données qui vont nous être utiles pour donner ensuite un algorithme général de transposition de matrice sur un réseau de communication quelconque. Ces échanges ne permettent pas eux-mêmes de réaliser une transposition, ils ne constitutent que la brique de base pour la construction du schéma global de communication.

2.4.1 Transposition sur un chemin

Définition 6 :

Considérons un chemin de longueur l, avec l pair (où chaque processeur est numéroté de 0 à l). La « Transposition sur un Chemin » (notée TCh) est le schéma de communication au cours duquel chaque processeur q envoie sa donnée au processeur $(l-q)$, $0 \leq q \leq l$.

Nous détaillons sur la figure 2.8 les étapes successives de l'algorithme pour $l = 4$. Le couple sur chaque arc indique la source et le destinataire de chaque message. L'algorithme est exprimé dans la figure 2.7.

```
Procédure TCh(A,chemin k) (Programme du proc. q sur le chemin k)
début
  B = bloc local de la matrice A
  pour S = 0 à l − 1 faire
    si (q = S) alors
      Envoyer B au processeur (q + 1) sur le chemin k.
    si ((q > S) et (q < l/2 + ⌊(S+1)/2⌋)) alors
      Envoyer B au processeur (q + 1) sur le chemin k.
      Recevoir B du processeur (q − 1) sur le chemin k.
    si (q = l/2 + ⌊(S+1)/2⌋) alors
      Recevoir B du processeur (q − 1) sur le chemin k.
    si (l − q = S) alors
      Envoyer B au processeur (q − 1) sur le chemin k.
    si ((l − q > S) et (l − q < l/2 + ⌊(S+1)/2⌋))) alors
      Envoyer B au processeur (q − 1) sur le chemin k.
      Recevoir B du processeur (q + 1) sur le chemin k.
fin
```

FIG. 2.7 - *Algorithme TCh.*

On peut facilement vérifier que l'algorithme TCh sur un chemin de longueur l, s'exécute en en l étapes.

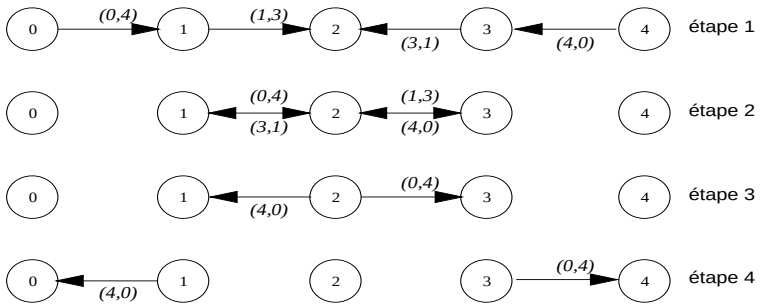


FIG. 2.8 - *Exemple de transposition sur un chemin de longueur 4.*

Amélioration : algorithme TCh pipeliné

Nous pouvons remarquer que lors de l'exécution du précédent algorithme, tous les liens ne sont pas utilisés. Afin de maximiser le nombre de liens utilisés à chaque étape, nous divisons chaque message en ν paquets de taille $\frac{L}{\nu}$ et nous pipelinons l'envoi des paquets. L'utilisation d'une technique de pipeline standard directement sur l'algorithme TCh est impossible car tous les processeurs intermédiaires ayant un message à envoyer, il y aurait collision entre les paquets des différents processeurs. Cependant, il est possible de pipeliner en retardant de façon périodique les envois. Nous présentons sur la figure 2.9 l'algorithme pipeliné (noté TChP) de façon détaillée (par souci de clarté, nous n'avons fait figurer que les envois de messages et pas les échanges).

À la première étape de l'algorithme, le processeur 0 envoie son premier paquet. Le processeur 1 envoie son premier seulement lorsqu'il a transmis celui du processeur 0, c'est-à-dire à l'étape 2. Le processeur 0 envoie son second paquet quand son premier paquet a atteint le processeur $\frac{l-1}{2}$, etc.

Il est clair qu'aucun conflit n'a lieu, par construction, ce qui garantit la correction de l'algorithme. Remarquons deux choses : la première est que le lien reliant les processeurs $\frac{l}{2}-1$ et $\frac{l}{2}$ est occupé en permanence, la deuxième est que tous les processeurs finissent leurs échanges au même moment.

Calculons maintenant le coût de cet algorithme. Le temps nécessaire pour que le premier paquet du processeur atteigne le processeur l est : $l(\beta + \frac{L}{\nu}\tau)$. Le i^{eme} paquet du processeur 0 est envoyé après $\left(\frac{l}{2}\right) \times (i-1)$ étapes. Donc le temps total de l'algorithme est :

$$T_{TChP}(\nu) = l\left(\beta + \frac{L}{\nu}\tau\right) + \left(\frac{l}{2}\right)(\nu - 1)\left(\beta + \frac{L}{\nu}\tau\right)$$

Nous pouvons alors facilement calculer le nombre optimal de paquets $\nu_{opt} = \sqrt{\frac{L\tau}{\beta}}$
D'où une durée totale de l'algorithme TChP pour un nombre optimal de paquets :

$$T_{TChP}(\nu_{opt}) = \left(\frac{l}{2}\right)\left(\sqrt{\beta} + \sqrt{L\tau}\right)^2$$

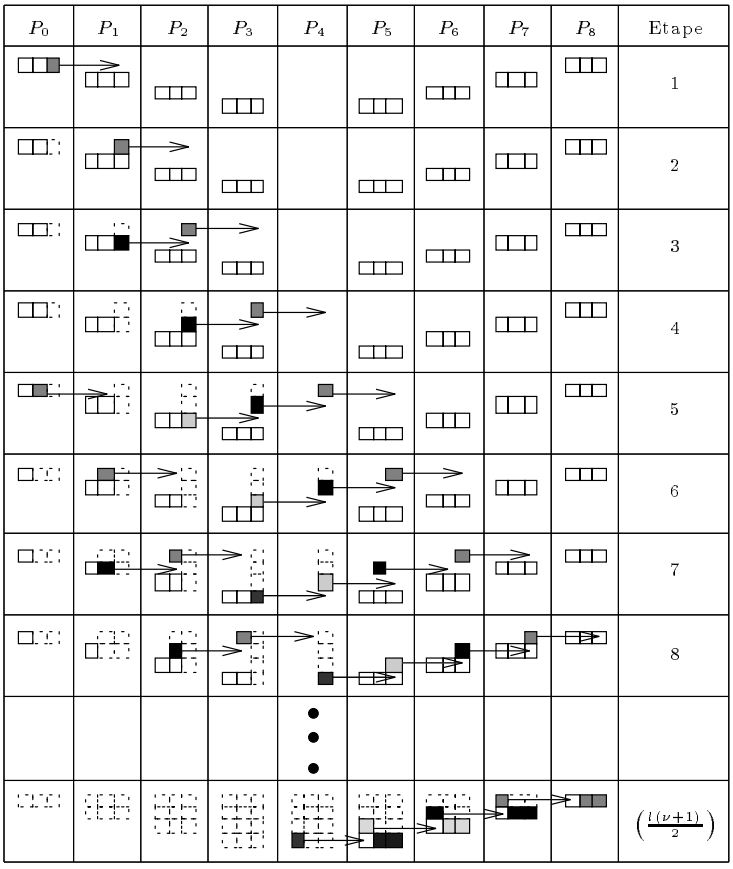


FIG. 2.9 - *TChP sur un chemin de longueur 8.*

2.4.2 Transposition cyclique

Définition 7 :

Considérons un cycle de longueur l, avec l impair. La « Transposition Cyclique » (notée TCy) est le schéma de communication où chaque processeur q envoie ses données au processeur $\left((q + \frac{l+1}{2}) \bmod [l]\right)$, $0 \leq q \leq l$.

L'algorithme réalisant le schéma TCy est donné dans la figure 2.10, et est détaillé pour $l = 5$ sur la figure 2.11.

```
Procédure TCy(A,cycle k) (Programme du proc. q sur le cycle k)
début
  B = bloc local de la matrice A
  pour S = 0 à (l+1)/2
    si (q = S) alors
      Envoyer B au processeur (q + 1) sur le cycle k.
      Recevoir B du processeur (q − 1) sur le cycle k.
fin
```

FIG. 2.10 - *Algorithme TCy.*

Ce schéma de communication élémentaire peut être effectué en $\frac{l+1}{2}$ étapes, sur un cyle de longueur l.

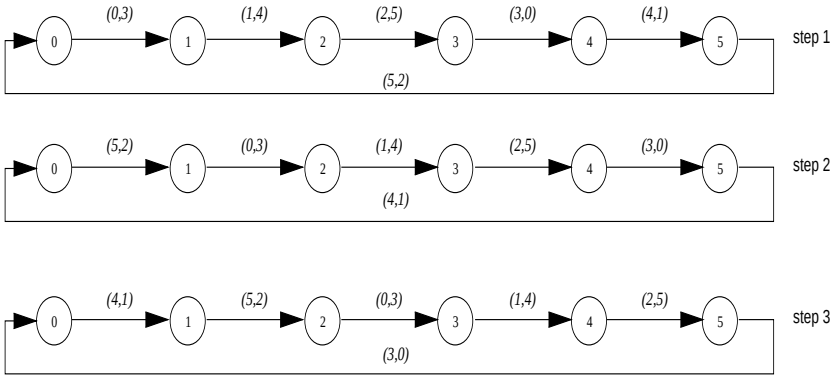


FIG. 2.11 - *TCy sur 6 processeurs.*

Remarquons au passage, que dans ce cas, nous ne pouvons améliorer cet algorithme par une quelconque technique de pipeline, du fait qu'à tout moment, tous les liens sont occupés.

2.4.3 Méthodologie générale

D'après les définitions précédentes des mouvements élémentaires de données, nous présentons maintenant une méthodologie générale permettant de réaliser une « Transposition Directe »[5] sur un réseau quelconque :

1. Trouver une partition (au sens des nœuds) du réseau en C cycles ou chemins, telle que chaque nœud et son transposé soient sur le même chemin ou cycle :
   - Si C est un chemin, alors la contrainte suivante doit être vérifiée :
     $C = X_0 X_1 \dots X_l$ avec l pair, tel que $(X_i^t) = X_{l-i}$ pour $i = 0 \dots l$
   - Si C est un cycle alors il doit vérifier :
     $C = X_0 X_1 \dots X_l$ avec l impair, tel que $(X_i^t) = X_{i+\frac{l+1}{2}}$ pour $i = 0 \dots \frac{l-1}{2}$

5. Voir la définition que nous avons donnée précédemment de la « Transposition Directe ».

2. Appliquer ensuite, les schémas élémentaires de communication précédents sur ces C cycles ou chemins.

Par conséquent, le coût d'une « Transposition Directe » est soit égal au coût de l'algorithme TChP sur le plus long des C chemins, soit égal au coût de l'algorithme TCy sur le plus long des C cycles du réseau.

2.5 Application aux réseaux usuels

2.5.1 Transposition Directe sur le tore

Préliminaire : la grille

Nous allons commencer par présenter une partition de la grille carrée de P processeurs. Nous définissons $\sqrt{P}-1$ chemins par la liste des processeurs composant chacun de ces chemins, de la façon suivante (se reporter à la figure 2.13 pour voir la disposition de ces chemins dans la grille) :

Chemin i, $(i = 1 \ldots \sqrt{P}-1)$
$Alloc(i, \sqrt{P}) \rightarrow Alloc(i, \sqrt{P}-1) \rightarrow \ldots \rightarrow Alloc(i,i) \rightarrow Alloc(i+1,i) \rightarrow \ldots \rightarrow Alloc(\sqrt{P}, i)$

FIG. 2.12 - *Description des chemins.*

Au cours de la transposition de matrice, pendant que le processeur $Alloc(i, \sqrt{P})$ doit envoyer ses données au processeur $Alloc(\sqrt{P}, i)$, le processeur $Alloc(i, \sqrt{P}-1)$ doit envoyer les siennes au processeur $Alloc(\sqrt{P}-1, i)$, etc. Nous utilisons donc l'algorithme TCh le long de ces chemins.

Le chemin i contient $2(\sqrt{P}-i)+1$ nœuds, donc le nombre total de nœuds contenus dans ces chemins est égal à :

$$\sum_{i=1}^{\sqrt{P}-1} \Big(2(\sqrt{P}-i)+1\Big) = P-1$$

Si nous considérons le nœud $Alloc(\sqrt{P}, \sqrt{P})$ comme un chemin dégénéré réduit à un seul processeur, l'ensemble de tous ces chemins constitue une partition de la grille.

Nous avons donc une « Transposition Directe » sur une grille carrée de taille $\sqrt{P}$ en $2(\sqrt{P}-1)$ étapes de communication (correspondant au chemin le plus long de la partition) avec des messages de taille $\frac{N^2}{P}$. Le coût d'une transposition sur une grille carrée en utilisant des chemins TChP est égal à[6] :

$$T_2^{Grille} = 2(\sqrt{P}-1)\left(\beta + \frac{N^2}{P}\tau\right)$$

6. Cette construction est une autre façon de voir la version pipelinée de l'algorihme de Johnsson [81].

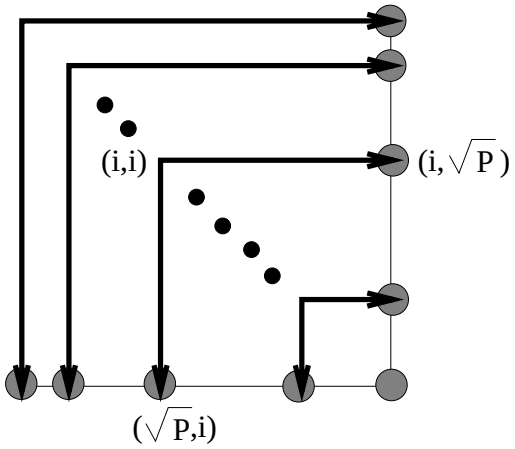


FIG. 2.13 - *Partition de la grille en utilisant les chemins TCh.*

Adaptation au tore

Les résultats sur le tore sont adaptés à partir de ceux obtenus sur la grille, mais nous utilisons les liens suplémentaires qui relient les nœuds du bord. Nous décrivons dans la figure 2.14 les chemins utilisés et nous les visualisons sur la figure 2.15.

Pour le chemin i ($i = 1$ to $\frac{\sqrt{P}}{2} - 1$ (domaine A_1 sur la figure 2.15))
$Alloc(i, (\frac{\sqrt{P}}{2} + i)) \rightarrow Alloc(i, (\frac{\sqrt{P}}{2} + i + 1)) \rightarrow \ldots \rightarrow Alloc(i, \sqrt{P}) \rightarrow Alloc(i, 1) \rightarrow Alloc(i, 2) \rightarrow \ldots \rightarrow$
$Alloc(i, i) \rightarrow Alloc(i - 1, i) \rightarrow \ldots \rightarrow Alloc(1, i) \rightarrow Alloc(\sqrt{P}, i)) \rightarrow Alloc(\sqrt{P} - 1, i) \rightarrow \ldots \rightarrow Alloc\ (\frac{\sqrt{P}}{2} + i, i)$
Pour le chemin i ($i = \frac{\sqrt{P}}{2} + 1$ to $\sqrt{P}$ (domaine A_2 sur la figure 2.15))
$Alloc(i, (i - \frac{\sqrt{P}}{2}) + 1) \rightarrow Alloc\ (i, (i - \frac{\sqrt{P}}{2}) + 2) \rightarrow \ldots \rightarrow$
$Alloc(i, i) \rightarrow Alloc(i - 1, i) \rightarrow \ldots \rightarrow Alloc((i - \frac{\sqrt{P}}{2}) + 1, i)$

FIG. 2.14 - *Description des chemins.*

Calculons maintenant le nombre de nœuds contenus dans ces chemins :

- Chemins de A_1: $\sum_{i=1}^{\frac{\sqrt{P}}{2}} \left(\sqrt{P} + 1\right) = \frac{1}{2}(P + \sqrt{P})$

- Chemins de A_2: $\sum_{i=\frac{\sqrt{P}}{2}+1}^{\sqrt{P}} \left(\sqrt{P} + 1\right) = \frac{1}{2}(P - \sqrt{P})$

Donc le nombre total de nœuds contenus dans ces chemins est égal à P. Tous ces chemins sont disjoints, et nous obtenons alors une partition du tore. La longueur de ceux-ci est égale à $\sqrt{P} - 1$ dans A_1 et $\sqrt{P} - 3$ dans A_2. Nous pouvons donc réaliser une « Transposition Directe » sur un tore carré en $\sqrt{P}$ étapes élémentaires avec des messages de taille $\frac{N^2}{P}$, d'où

un coût de :

$$T_2^{Tore} = \sqrt{P}\left(\beta + \frac{N^2}{P}\tau\right)$$

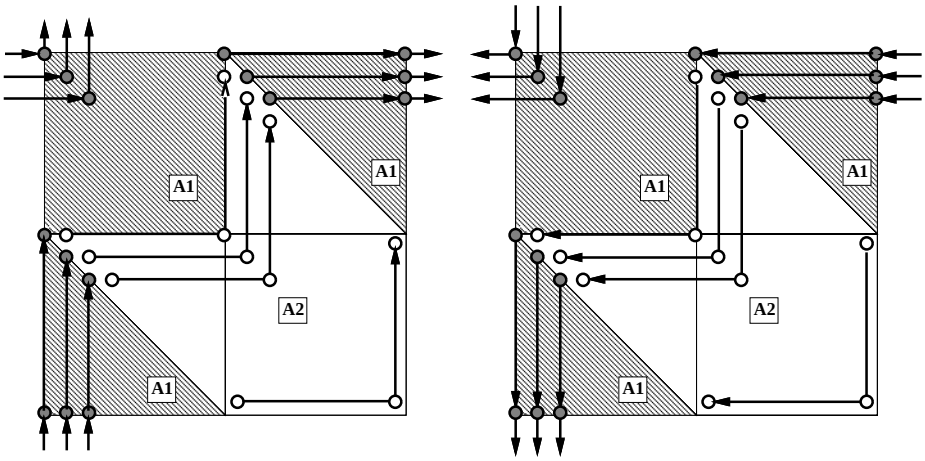


FIG. 2.15 - *Partition du tore en utilisant des chemins TCh.*

Nous pouvons améliorer ce résultat en remarquant que la moitié des liens n'est pas utilisée (voir figure 2.16). Les blocs sont divisés en deux parties de longueur égale qui sont envoyées simultanément sur les liens verticaux et horizontaux. Nous utilisons donc deux schémas de « Transposition Directe » en parallèle avec des messages de longueur $\frac{N^2}{2P}$. Par conséquent le coût de cet algorithme est :

$$T_3^{Tore} = \sqrt{P}\left(\beta + \frac{N^2}{2P}\tau\right)$$

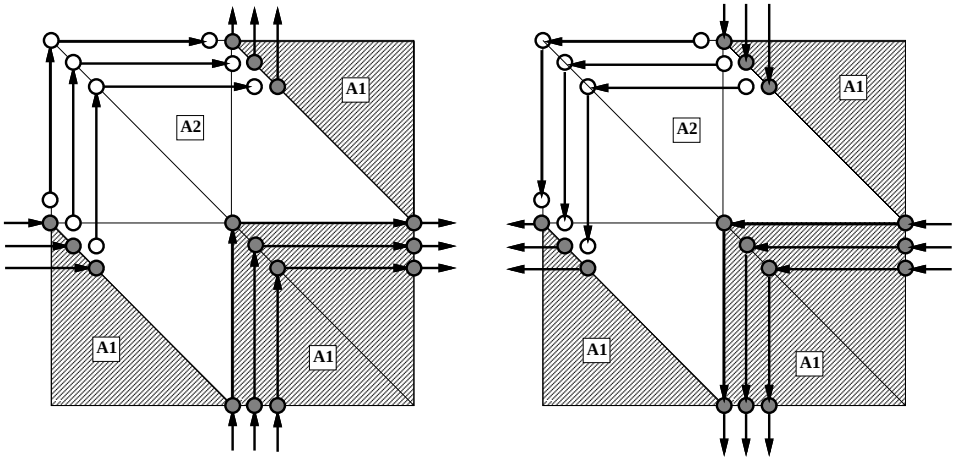


FIG. 2.16 - *Partition complémentaire du tore avec des chemins TCh sur les liens orthogonaux.*

En utilisant l'algorithme pipeliné TChP sur ces chemins, nous obtenons finalement un

temps égal à :

$$T_4^{Tore} = \frac{\sqrt{P}}{2}\left(\sqrt{\beta} + \sqrt{\frac{N^2}{2P}\tau}\right)^2$$

Nous voyons que le temps est asymptotiquement proportionnel au quart du diamètre du tore.

2.5.2 Transposition Directe sur le de Bruijn

Nous commençons tout d'abord par donner quelques définitions [62].

Définition 8 :

Cycle pur :
Un cycle pur, est un cycle où le nombre de bits égaux à 1 dans la numérotation des nœuds est conservé.

Générateur de cyle :
Un générateur de cyle est un nœud qui peut être utilisé pour décrire un cycle, en décalant successivement à gauche les n bits de son numéro.

Par exemple, si nous prenons le nœud (1100) comme générateur, le cycle pur correspondant est le suivant :

$$[(1100) \quad (1001) \quad (0011) \quad (0110)]$$

Il est possible de définir un opérateur qui permet d'énumérer tous les générateurs. Cet opérateur est défini dans [62]. Nous donnons dans le tableau 2.2 tous les génerateurs ainsi que les cycles purs correspondants pour $n = 6$.

Maintenant, il nous faut prouver que les cycles ainsi générés contiennent tous les nœuds du de Bruijn.

Proposition 1 : [65, 93]

Le nombre total de générateurs est égal à $NG(n)$ avec :

$$NG(n) = \frac{1}{n}\sum_{d \in D} \phi(d) 2^{\frac{n}{d}}$$

où D est l'ensemble des diviseurs de n et ϕ est la fonction d'Euler. Nous rappelons que la fonction d'Euler est définie ainsi :
$\forall n \geq 1$, $\phi(n)$ = nombre d'entiers x, $1 \leq x \leq n$, tels que x et n sont premiers entre eux.

Proposition 2 :

Le nombre total de nœuds contenus dans les cyles obtenus par les générateurs de cycles est égal à 2^n.

Par exemple, pour $n = 6$

- $D = \{6, 3, 2, 1\}$
- $\phi(6) = 2$, $\phi(3) = 2$, $\phi(2) = \phi(1) = 1$
- $NG(6) = \frac{1}{6}\left(\phi(6)2^{\frac{6}{6}} + \phi(3)2^{\frac{6}{3}} + \phi(2)2^{\frac{6}{2}} + \phi(1)2^{\frac{6}{1}}\right) = 14.$

Preuve :
$NG(n)$ est le nombre de façons différentes d'associer les nombres 0 et 1, afin de composer des mots cycliques de longueur n. Ainsi, lorsque nous considérons ces mots cycliques, nous énumérons tous les mots de longueur n sur l'alphabet $\{0, 1\}$. Ceci revient à décrire d'une autre manière les nœuds du de Bruijn. Donc les $NG(n)$ cycles contiennent tous les nœuds du graphe et sont disjoints par construction. Nous avons donc défini une partition du de Bruijn en $NG(n)$ cycles de longueur maximale égale à n. $\diamond$

Par conséquent nous pouvons réaliser une « Transposition Directe » sur un de Bruijn de dimension n en $\frac{n}{2}$ étapes de communication (voir figure 2.17), et le coût total de l'algorithme est alors de :

$$T_2^{DeBruijn} = \frac{\log_2(P)}{2}\left(\beta + \frac{N^2}{P}\tau\right)$$

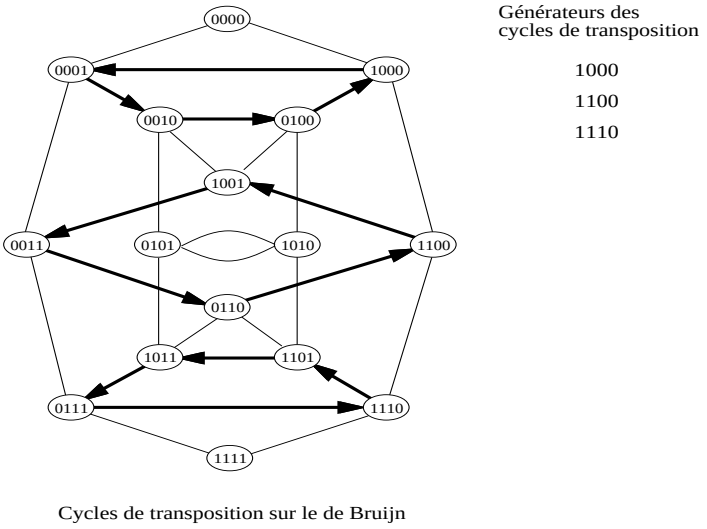


FIG. 2.17 - *Transposition Directe sur un de Bruijn (2,4).*

Ce résultat peut être amélioré en remarquant que nous utilisons seulement la moitié des liens[7], comme dans le cas du tore, que l. Aussi, nous pouvons diviser chaque message en deux parties de longueur égale, et utiliser les liens dans le sens inverse pour transmettre une moitié des messages. Le temps devient alors :

$$T_3^{DeBruijn} = \frac{\log_2(P)}{2}\left(\beta + \frac{N^2}{2P}\tau\right)$$

7. Nous avons supposé au départ que nous disposions de liens bi-directionnels.

Finalement, nous obtenons pour de grands messages, un temps qui est asymptotiquement proportionnel au quart du diamètre du de Bruijn, comme pour le tore.

Niveau	Longueur du cycle	Générateur
C_6	1	111111
C_5	6	111110
C_4	6	111100
C_4	6	111010
C_3	6	111000
C_4	3	110110
C_3	6	110100
C_3	6	110010
C_2	6	110000
C_3	2	101010
C_2	6	101000
C_2	3	100100
C_1	6	100000
C_0	1	000000

111111					
111110	111101	111011	110111	101111	011111
111100	111001	110011	100111	001111	011110
111010	110101	101011	010111	101110	011101
111000	110001	100011	000111	001110	011100
110110	101101	011011			
110100	101001	010011	100110	001101	011010
110010	100101	001011	010110	101100	011001
110000	100001	000011	000110	001100	011000
101010	010101				
101000	010001	100010	000101	001010	010100
100100	001001	010010			
100000	000001	000010	000100	001000	010000
000000					

TAB. 2.2 - *Table des générateurs de cycles purs et cycles purs correspondants pour $n = 6$.*

2.6 Résultats d'optimalité

2.6.1 Bornes inférieures en commutation de message

Nous donnons, ci-dessous, les bornes inférieures des temps de transposition sur les différentes topologies.

Proposition 3 :

Les bornes inférieures des coûts d'une transposition de matrice allouée par blocs de façon consécutive ou cyclique, en commutation de message, sont données dans la table 2.3 pour les facteurs du temps d'initialisation et du taux de transmission.

	Start-up	Taux de transmission
Grille	$2(\sqrt{P}-1)$	$\left\lceil \frac{\sqrt{P}}{2} \right\rceil \times \frac{N^2}{2P}$
Tore	$2\left\lfloor \frac{\sqrt{P}}{2} \right\rfloor$	$\left\lceil \frac{\sqrt{P}}{2} \right\rceil \times \frac{N^2}{4P}$
Hypercube	$\log_2(P)$	$\left\lceil \frac{N^2}{2P} \right\rceil$
De Bruijn $(2,n)$	$\frac{\log_2(P)}{2}$	$\Theta\left(\left\lceil \frac{N^2 \log_2(P)}{2P}(1-\frac{\sqrt{P}}{P}) \right\rceil\right)$

TAB. 2.3 - *Table des bornes inférieures en terme de temps d'initialisation et de taux de transmission.*

Preuve :

1. Facteur du taux de transmission

   - Pour la grille et le tore, il faut au moins échanger les sous-matrices anti-diagonales, c'est à dire les sous-matrices $A_{(0,\sqrt{P}-1)}$ et $A_{(\sqrt{P}-1,0)}$ (voir figure 2.19). Le nombre total de données contenues dans chacun des blocs est égal à $\frac{N^2}{P} \times \frac{P}{4}$ et le nombre maximum de liens utilisables est $2 \times \frac{\sqrt{P}}{2}$ pour la grille [72] et $4 \times \frac{\sqrt{P}}{2}$ pour le tore.
   - Pour l'hypercube, la borne inférieure est obtenue en remarquant que le nombre de liens quittant un bloc anti-diagonal d'une sous-matrice est égal à $2 \times \frac{P}{4} = \frac{P}{2}$ pour échanger $\frac{N^2}{P} \times \frac{P}{4}$ données (voir figure 2.19).
   - Pour le de Bruijn, le résultat est un corollaire direct du lemme suivant. Avant de l'enoncer, nous rappelons la définition de la bisection d'un graphe.

     Définition 9 :

     La bisection d'un graphe est le nombre d'arcs (ou d'arêtes) qu'il faut ôter pour déconnecter le graphe en deux ensembles ayant le même nombre de sommets (à un près).

     Lemme :
     La bisection du de Bruijn binaire de dimension n, noté de Bruijn $(2, n)$, *est égale à* $\Theta\left(\frac{P}{\log_2(P)}\right)$ *[90]*

     Nous construisons alors une bi-partition du de Bruijn de la façon suivante : nous appelons V l'ensemble des nœuds du de Bruijn duquel on a ôté les nœuds symétriques[8]. La construction des deux ensembles Q et Q^t de la bi-partition consiste à supprimer un nœud q de V pour le mettre dans Q et supprimer alors le nœud transposé de q, q^t, pour le mettre dans Q^t. Nous continuons ce processus jusqu'à ce que $V = \emptyset$. Nous ajoutons alors la moitié des nœuds symétriques dans Q et l'autre moitié dans Q^t. Il est clair, par construction, que $Q \cap Q^t = \emptyset$, que $Q \cup Q^t$ est égal à l'ensemble des nœuds du de Bruijn et enfin que $|Q| = |Q^t| = \frac{P}{2}$. Donc, Q et Q^t constituent une bi-partition du de Bruijn.

     Nous devons maintant montrer que la bi-partition ainsi construite réalise bien la bisection. Pour cela nous rappelons que la preuve de la valeur de la bisection du de Bruijn repose sur sa représentation dans le plan complexe [90] (voir figure 2.18). Il faut donc montrer que tous le nœuds x de Q appartiennent à une moitiée du plan complexe, et que tous les nœuds x^t de Q^t appartiennent à l'autre moitiée.

8. Un nœud q est dit symétrique si et seulement si $q^t = q$.

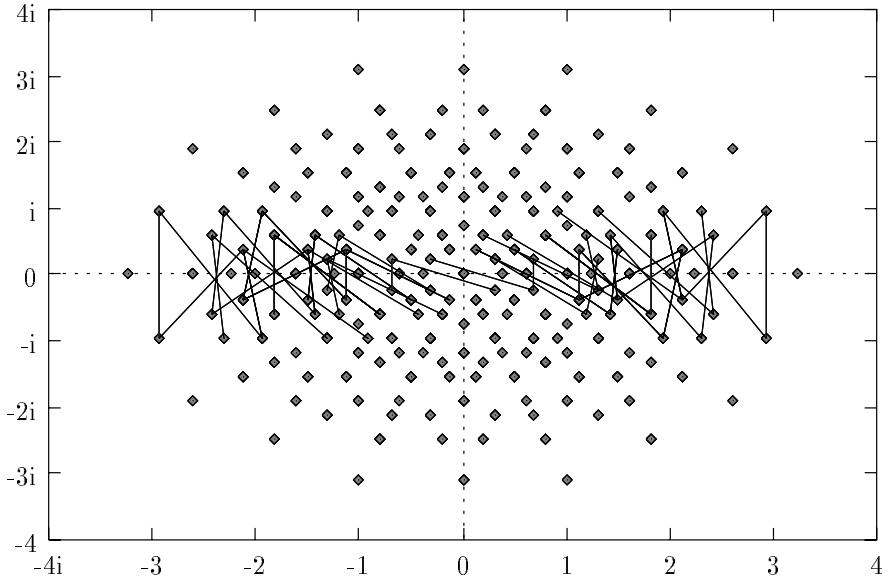


FIG. 2.18 - *Représentation dans le plan complexe du de Bruijn binaire de dimension 10. Les arêtes représentées appartiennent à la bisection.*

Considérons $x = x_{n-1} \dots x_0$ un nœud du de Bruijn, alors sa représentation dans le plan complexe est $\overline{x} = \sum_{j=0}^{n-1} x_j \times \omega^j$ où $\omega^j = e^{-\frac{2i\pi j}{n}}$.
Alors prouver que $\overline{x}$ et $\overline{x^t}$ appartiennent à des moitiées différentes du plan complexe revient à montrer que $\overline{x^t} = \omega^{\frac{n}{2}} \times \overline{x}$.

Proposition 4 :

Soit x un nœud quelconque de Q et soit x^t son nœud transposé, $x^t \in Q^t$. Alors $\overline{x^t} = \omega^{\frac{n}{2}} \times \overline{x}$.

Preuve : Si $x = x_{n-1} \dots x_{\frac{n}{2}} x_{\frac{n}{2}-1} \dots x_0$, alors $x^t = x_{\frac{n}{2}-1} \dots x_0 x_{n-1} \dots x_{\frac{n}{2}}$ et

$$\begin{aligned}
\overline{x^t} &= x_{\frac{n}{2}} + x_{\frac{n}{2}+1}\omega^1 + \dots + x_{n-1}\omega^{\frac{n}{2}-1} + x_0\omega^{\frac{n}{2}} + \dots + x_{\frac{n}{2}-1}\omega^{n-1} \\
&= x_0\omega^{\frac{n}{2}} + \dots + x_{\frac{n}{2}-1}\omega^{n-1} + x_{\frac{n}{2}}\omega^n + x_{\frac{n}{2}+1}\omega^{n+1} + \dots + x_{n-1}\omega^{n+\frac{n}{2}-1} \\
&= \omega^{\frac{n}{2}} \left(x_0 + \dots + x_{\frac{n}{2}-1}\omega^{\frac{n}{2}-1} + x_{\frac{n}{2}}\omega^{\frac{n}{2}} + x_{\frac{n}{2}+1}\omega^{\frac{n}{2}+1} + \dots + x_{n-1}\omega^{n-1} \right) \\
&= \omega^{\frac{n}{2}} \times \overline{x}
\end{aligned}$$

□

Donc le nombre maximum de liens qui connectent Q à Q^t est égal à $\Theta\left(\frac{P}{\log_2(P)}\right)$. Le nombre de nœuds dans Q qui doivent échanger leur $\frac{N^2}{P}$ données avec un nœud de Q est exactement égal à $\frac{P}{2} - \frac{\sqrt{P}}{2}$ et ils peuvent utiliser pour cela au plus $\Theta\left(\frac{P}{\log_2(P)}\right)$ liens. Donc la borne inférieure pour le taux de transmission est égale à :
$\Theta\left(\frac{N^2 \log_2(P)}{2P}(1 - \frac{\sqrt{P}}{P})\right)$

2. Le nombre minimum d'étapes en commutation de message correspond à la distance maximale à traverser.

   - Pour la grille, le tore et l'hypercube, le nombre minimum d'étapes est évident puisqu'il correspond au diamètre.
   - Pour le de Bruijn, il suffit de remarquer que cette distance maximale est égale à $\frac{\log_2(P)}{2}$.

◇

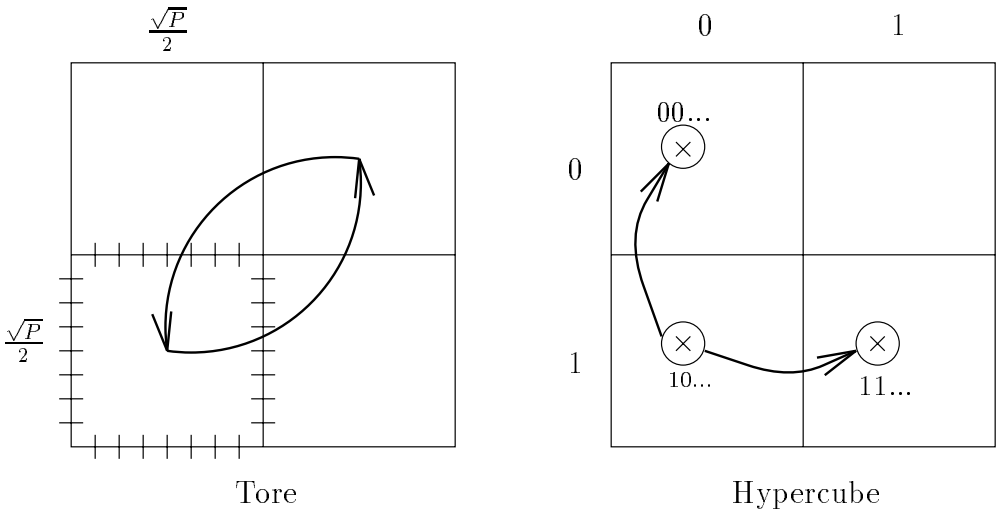


FIG. 2.19 - *Borne inférieure du facteur de bande passante pour le tore et l'hypercube.*

2.6.2 Résultats d'optimalité

D'après les résultats précédents, nous vérifions dans le tableau 2.4 que l'algorithme présenté pour le de Bruijn est optimal en terme de temps d'initialisation et asymptotiquement optimal (quand la taille de la matrice augmente pour un nombre de processeurs fixé) de taux de transmission. Pour le tore, l'algorithme atteint asymptotiquement la borne inférieure en temps d'initialisation (lorsque la taille de la matrice augmente) et est à un facteur 2 de l'optimal pour le taux de transmission. L'algorithme *multiple recursive paths transpose* proposé dans [84] pour les hypercubes est asymptotiquement optimal en taux de transmission.

	Temps
Tore	$\frac{\sqrt{P}}{2}\left(\sqrt{\beta}+\sqrt{\frac{N^2}{2P}\tau}\right)^2$
De Bruijn $(2,n)$	$\frac{\log_2(P)}{2}\left(\beta+\frac{N^2}{2P}\tau\right)$

TAB. 2.4 - *Temps d'exécution des différents algorithmes.*

2.7 Adaptation au *wormhole*

Le problème de la transposition de matrices sur des grilles 2D en mode *wormhole* pour des allocations par blocs a été déjà étudié par Ho [72] et par Tsai et McKinley [121]. Ces deux approches diffèrent par le fait que la première minimise le facteur du taux de transmission et que la deuxième optimise le nombre d'étapes.

L'idée de base de la première approche est d'utiliser l'algorithme de transposition récursive en pipelinant les phases entre elles. Le facteut du taux de transmission de l'algorithme pour des grilles carrées de P nœuds, avec $P = 2^k$, est $\mathcal{O}\left(\frac{N^2}{3\sqrt{P}}\right)$.

Tsai et McKinley utilisent quant à eux, « l'approche étendue de nœuds dominants » afin de minimiser le nombre d'étapes. C'est une nouvelle illustration de l'outil décrit dans le chapitre 1 de cette même partie. Ils définissent des nœuds particuliers (les nœuds diagonaux) sur lesquels ils concentrent l'information à échanger, puis la redistribue. Cet algorithme récursif est utilisable pour des grilles carrées de P nœuds, avec $P = 2^k$. Le nombre total d'étapes est égal à $\log_2(\sqrt{P})$ et le facteur du taux de transmission est égal à $\left\lfloor \frac{5^{\frac{\log_2 P}{4}}}{2} \right\rfloor \times \frac{N^2}{P}$.

La méthodologie introduite dans ce chapitre peut être étendue au mode *wormhole*. En effet, il suffit pour cela de changer les deux algorithmes de base TCh et TCy afin de minimiser le nombre d'étapes. Le principe de ces deux algorihtmes est présenté dans les figures 2.20 et 2.21.

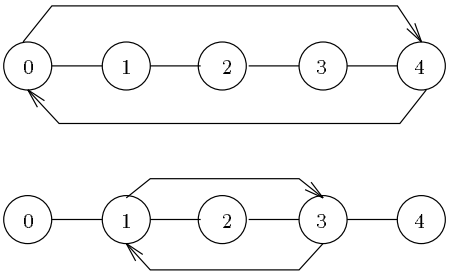


FIG. 2.20 - *Principe de l'algorihtme TCh en wormhole sur un chemin de 5 nœuds.*

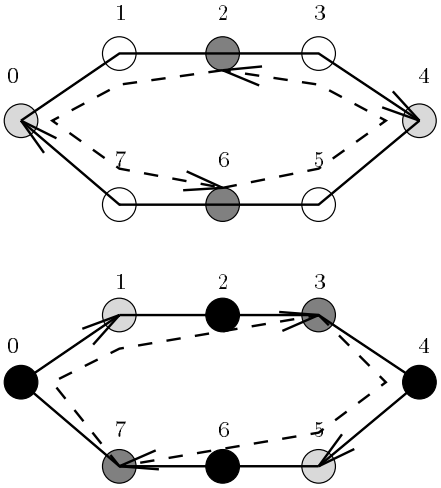


FIG. 2.21 - *Principe de l'algorihtme TCy en wormhole sur un cycle de 8 nœuds.*

Les temps d'exécution de ces deux algorithmes sont les suivants : $T_{TCh-Worm} = \frac{l-1}{2}(\beta + L\tau)$ $T_{TCy-Worm} = \left\lceil \frac{l}{4} \right\rceil (\beta + L\tau)$
Maintenant il suffit d'utiliser ces algorithmes sur les différentes topologies, pour obtenir les temps d'exécution suivants :

$$T^{Grille-Worm} = (\sqrt{P} - 1)\left(\beta + \frac{N^2}{P}\tau\right)$$
$$T^{Tore-Worm} = \left(\frac{\sqrt{P}-1}{2}\right)\left(\beta + \frac{N^2}{2P}\tau\right)$$
$$T^{DeBruijn-Worm} = \frac{\log_2(P)}{4}\left(\beta + \frac{N^2}{2P}\tau\right)$$

Comme dans le cas du mode de commutation *store-and-forward*, nous pouvons établir les bornes inférieures en terme des facteurs du nombre d'étapes et du taux de transmission (voir la table 2.5). Comme on peut le remarquer, les bornes inférieures du facteur du taux de transmission sont égales en commutation de messages et en *wormhole*.

	Nombre d'étapes	Taux de transmission
Grille	$\log_5\left(\frac{\sqrt{P}-1}{2}\right)$	$\left\lceil \frac{\sqrt{P}}{2} \right\rceil \times \frac{N^2}{2P}$
Tore	$\log_5\left(\frac{\sqrt{P}-1}{4}\right)$	$\left\lceil \frac{\sqrt{P}}{2} \right\rceil \times \frac{N^2}{4P}$
Hypercube	$\log_{\log_2(P)+1}\left(2\frac{P-\sqrt{P}}{P}\right)$	$\left\lceil \frac{N^2}{2P} \right\rceil$
De Bruijn $(2,n)$	$\Theta\left(\log_5\left(\log_2(P) \times \left(\frac{P-\sqrt{P}}{2P}\right)\right)\right)$	$\Theta\left(\left\lceil \frac{N^2 \log_2(P)}{2P}(1 - \frac{\sqrt{P}}{P}) \right\rceil\right)$

TAB. 2.5 - *Bornes inférieures en wormhole.*

Pour établir les bornes inférieures sur le nombre d'étapes, nous utilisons le lemme suivant :

Lemme :
Étant donné un graphe G de bissection $B(G)$ et de degré maximum $\Delta(G)$ de P sommets, le nombre minimum d'étapes pour réaliser une « transposition directe » est : $\left\lceil \log_{(\Delta(G)+1)} \left(\frac{P-\sqrt{P}}{2B(G)} \right) \right\rceil$

Preuve :
Nous considérons la bi-partition du graphe G formée par les ensembles Q et Q^t définis précédemment pour le de Bruijn. Pour réaliser une « transposition directe », il faut faire passer $\frac{P-\sqrt{P}}{2}$ messages entre Q et Q^t. Considérons un arc e qui appartient à l'ensemble des arcs entre Q et Q^t. À la première étape, le nombre maximum de messages pouvant passer par e est égal à $(\Delta(G)+1)^0$, à l'étape t, $(\Delta(G)+1)^{t-1}$ messages sont passés par e. Puisqu'il n'existe au plus que $B(G)$ arcs entre Q et Q^t, il faut au moins t étapes, telles que $(\Delta(G)+1)^{t-1} \times B(G) \geq \frac{P-\sqrt{P}}{2}$. Donc le nombre minimum d'étapes est plus grand ou égal à : $\left\lceil \log_{(\Delta(G)+1)} \left(\frac{P-\sqrt{P}}{2B(G)} \right) \right\rceil$ $\diamond$

Nous pouvons alors en déduire les résultats de la table 2.5 du lemme précédent et des valeurs des différentes bissections des graphes considérés rappelées dans la table ci-contre.

Graphe	Bissection
Grille	$\sqrt{P}$
Tore	$2\sqrt{P}$
Hypercube	$\frac{P}{2}$
De Bruijn $(2,n)$	$\Theta\left(\frac{P}{\log_2(P)}\right)$

2.8 Conclusion

Nous avons présenté dans ce chapitre une méthodologie de conception d'algorithmes de transposition de matrices allouées par blocs de façon cyclique ou consécutive sur réseaux directs. Cette méthodologie est basée sur le partitionnement des réseaux en chemins ou en cycles sur lesquels nous appliquons des schémas élémentaires de communication.

Nous avons tout d'abord appliqué cette méthodologie pour un mode de commutation de type commutation de messages sur le tore et le de Bruijn. Dans ces deux cas, nous avons amélioré les résultats précédents en utilisant les liens libres pour envoyer la moitié des messages. De plus, une stratégie pipeline nous a permis de décroître sensiblement le temps de transposition sur le tore. Ces deux algorithmes sont les meilleurs à ce jour.

Nous avons également montré comment adapter cette méthodologie au mode *wormhole* en modifiant les schémas élémentaires sur les chemins et les cycles pour tenir compte des spécificités de ce mode de commutation. Ces algorithmes ne sont pas optimaux en nombre d'étapes mais restent très efficaces en bande passante.

Chapitre 3

Échange total dans une grille torique.

Nous présentons dans ce chapitre des algorithmes efficaces de communication d'échange total dans des tores bi-dimensionnels. Nous nous intéressons à la résolution du problème sous les hypothèses d'un mode de commutation wormhole et d'un modèle Δ-port. Nous présentons un algorithme optimal en nombre d'étapes pour des tores carrés dont la taille est une puissance de 5. Nous présentons également un algorithme améliorant le facteur du taux de transmission tout en restant efficace en nombre d'étapes. Ce chapitre se termine par l'extension de ces algorithmes à d'autres tailles de tore et par des résultats de simulation permettant de comparer les différents algorithmes. Ce travail a été effectué en collaboration avec S. Perennes[a] et D. Trystram [29].

[a] S. Perennes est doctorant au laboratoire I3S de Nice.

3.1 Introduction

Nous nous intéressons dans ce chapitre au problème de **l'échange total** (*cf* chapitre 1de cette même partie) dans les grilles toriques. Ce schéma de communication est fréquemment utilisé en algorithme numérique parallèle comme dans le calcul d'un produit matrice-vecteur [37] ou dans l'algorithme parallèle du gradient conjugué [96]. De plus, comme nous allons le voir, il y a un très grand intérêt à concevoir des algorithmes performants de communication globale car ils sont désormais spécifiés ou implantés dans les biblitohèques de communication [9, 95]. Les algorithmes présentés peuvent être considérés comme des bases d'implantations efficaces de l'échange total sur grille torique. Cette topologie présente un avantage indéniable si l'on désire inteconnecter un très grand nombre de processeurs : son degré est constant (voir le chapitre 1 de la partie I). De plus le tore offre un intérêt par rapport à la grille : c'est un graphe symétrique régulier. C'est à dire que tous le nœuds du tore jouent un rôle équivalent, ce qui n'est pas le cas de la grille.

Nous considérons le problème sous les hypothèses suivantes : un mode de commutation *wormhole* ou commutation de circuits, et un modèle de communication Δ-port avec des liens *full-duplex* (se reporter au premier chapitre de cette même partie pour les définitions). Nous étudions dans un premier temps uniquement des tores carrés dont la dimension est

une puissance de 5. Nous montrons par la suite comment étendre les algorithmes présentés à d'autres tailles de tore.

Le reste du chapitre est organisé de la façon suivante : nous débutons par l'établissement de quelques propriétés qui nous sont utiles pour la conception de nos algorithmes. Dans la section suivante nous présentons quelques résultats déjà établis ainsi que l'adaptation d'algorithmes de diffusion en mode *wormhole* ou d'algorithmes en mode par commutation de messages. Nous établissons ensuite les bornes inférieures du problème en fonction du nombre d'étapes, de la distance et du taux de transmission. La section suivante est consacrée à la description d'un nouvel algorithme, optimal en nombre d'étapes. Ce dernier est légèrement modifié pour améliorer de façon notable le facteur du taux de transmission. Nous finissons ce chapitre par la description de l'extension des algorithmes précédents à d'autres tailles de tore ainsi que par la comparaison des différents algorithmes proposés entre eux et avec une implémentation « naïve » sur un Cray T3D.

3.2 Quelques résultats préliminaires

Pour la présentation des algorithmes nous nous restreignons dans un premier temps aux tores carrés dont la dimension est une puissance de 5. Le graphe correspondant est noté $WM(k) = (V, E)$, où le nombre n de nœuds est égal à 5^{2k}. Afin de concevoir nos algorithmes nous utilisons les définitions et propositions suivantes :

Définition 10 :

Nous définissons les ensembles S_i, $i = 0, ..., 4$ *de la façon suivante :*
$S_i = \{(x, y) \in V / x + 2y \equiv i[5]\}$, *où* $[.]$ *correspond à l'opérateur modulo et* $\equiv$ *est la relation de congruence (voir figure 3.1).*
Étant donné $i = 0, ..., 4$, *nous définissons également les sous-ensembles* $S_{i,j}$ *de* S_i, $j = 0...4$, *ainsi :* $S_{i,j} = \{(x, y) \in S_i / x \equiv j[5]\}$ *(la figure 3.2 représente les sous-ensembles* $S_{1,j}$ *de* $WM(2)$*)*

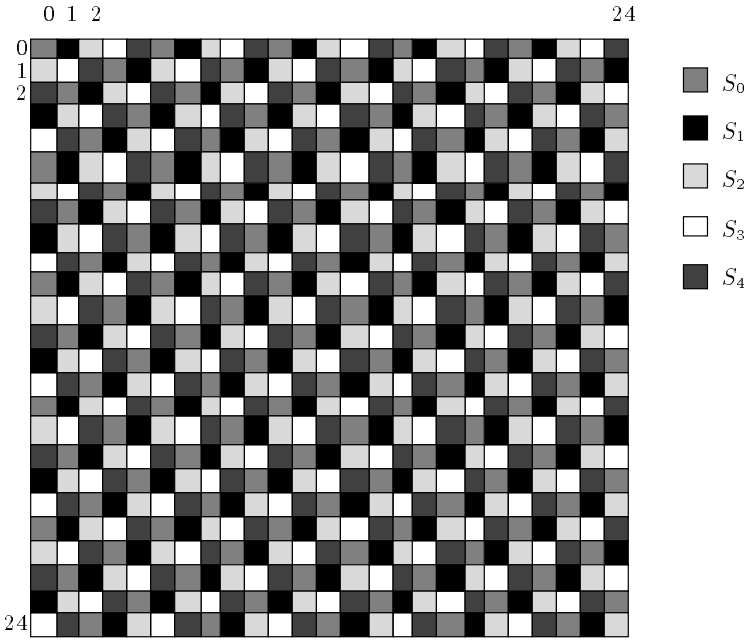


FIG. 3.1 - *Partition de $WM(2)$ en S_i.*

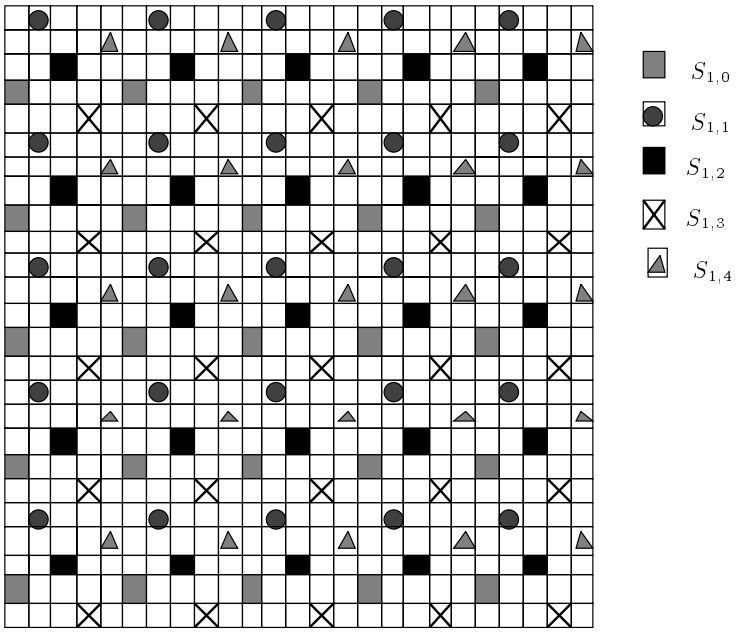


FIG. 3.2 - *Les sous-ensembles $S_{1,j}$ de $WM(2)$.*

Proposition 5 :

La décomposition de V en S_i est une sommet-partition du tore. De plus, étant donné un nœud $(x,y) \in S_i$, alors tous les voisins directs de (x,y) appartiennent à des sous-ensembles S_j (où $i \neq j$).

Preuve

A partir de la définition des ensembles S_i, il est facile de vérifier qu'ils constituent une sommet-partition du tore. La relation suivante montre que les quatre voisins directs d'un nœud quelconque (x,y) de S_i appartiennent à des ensembles S_j différents, avec $j \neq i$:

- $((x-1)[5], y) \in S_{(i+1)[5]}$
- $((x+1)[5], y) \in S_{(i-1)[5]}$
- $(x, (y-1)[5]) \in S_{(i+2)[5]}$
- $(x, (y+1)[5]) \in S_{(i-2)[5]}$

□

Proposition 6 :

Deux nœuds appartenant au même ensemble S_i, mais éléments de deux sous-ensembles $S_{i,j}$ ne sont, ni sur la même ligne, ni sur la même colonne du tore.

Preuve

Etant donnés $(x,y) \in S_{i,j}$ et $(x',y') \in S_{i,j'}$ avec $j \neq j'$, alors :
$x + 2y \equiv i[5]$ avec $x \equiv j[5]$, et $x' + 2y' \equiv i[5]$ avec $x' \equiv j'[5]$.
Il est donc clair que $x \not\equiv x'$. Supposons maintenant que $y = y'$, alors :
$x + 2y' \equiv i[5] \Rightarrow x + i - x' \equiv i[5] \Rightarrow x \equiv x'[5]$, ce qui est une contradiction. □

Proposition 7 :

Soit le graphe $WM_{i,j}(k) = (S_{i,j}, E_{i,j})$, où $E_{i,j}$ est l'ensemble des arêtes défini de la façon suivante : si (x,y) et (w,z) appartiennent à $S_{i,j}$, alors il existe une arête entre ces deux nœuds dans $WM_{i,j}(k)$, si et seulement si, il existe un chemin dans $WM(k)$ tel que tous les nœuds intermédiaires, (u,v), vérifient la propriété suivante : $u = x = w$ ou $v = y = z$. Les $WM_{i,j}(k)$ sont des tores.

La preuve est une conséquence directe de la définition. Remarquons que la distance entre deux voisins directs de $WM_{i,j}(k)$ est égale à 5 dans $WM(k)$.

3.3 Travaux précédents

Comme nous avons pu le voir dans le premier chapitre de cette partie, les travaux concernant la conception d'algorithmes de communications globales en mode de commutation *wormhole* sont assez récents. Le problème de l'échange total personnalisé a été étudié pour les grilles et les tores dans [34, 72] ainsi que dans les hypercubes [111]. Une première

solution au problème de la diffusion a été proposée dans [10] et une autre solution pour les tores dans [104]. Nous rappelons, pour mémoire, les travaux récents de Tsai et McKinley sur la méthodologie pour la conception d'algorithmes efficaces [121, 122].

Il n'existe pas, à l'heure actuelle d'algorithmes d'échange total en mode de commutation *wormhole* ou commutation de circuits. Une étude expérimentale a été faite par Bokhari sur un iPSC/860 [14]. Cependant de nombreux travaux ont été consacrés au même problème mais pour un mode de commutation par messages [108]. Dans la suite, nous décrivons rapidement un algorithme en commutation de messages qui est adapté de façon simple au mode de commutation *wormhole* (cet algorithme a été décrit dans le chapitre 1). Cet algorithme est évidemment très coûteux en nombre d'étapes, en revanche, il atteint la borne inférieure en taux de transmission. Un autre algorithme d'échange total peut être une adaptation de l'algorithme de diffusion de Peters et Syska [104] ce qui conduit à une solution logarithmique en nombre d'étapes mais très mauvaise en bande passante.

Algorithme ADSF (Adaptation Directe du *Store-and-Forward*)

L'idée de cet algorithme présentée dans [108] est d'utiliser la décomposition du tore en produit cartésien d'anneaux (*c.f.* le premier chapitre de cette partie). L'échange total est obtenu après deux phases : un premier échange de messages de taille $\frac{L}{2}$ effectué simultanément sur les anneaux verticaux et horizontaux suivi d'un échange total de messages de taille $\frac{nL}{2}$, de nouveau, sur les anneaux verticaux et horizontaux. Le temps total est la somme des temps des deux phases. Tous les processeurs font exactement la même chose à chaque étape et les échanges ne concernent que des processeurs voisins. Le temps de cet algorithme est alors le suivant [1] :

$$2\left\lfloor\frac{n}{2}\right\rfloor\alpha + 2\left\lfloor\frac{n}{2}\right\rfloor\delta + \left\lceil\frac{(n^2-1)L}{4}\right\rceil\tau$$

Algorithme RD (Regroupement-Diffusion)

L'echange total peut être vu comme l'enchaînement d'une phase de regroupement de toute l'information sur un nœud particulier puis d'une diffusion à partir de ce dernier. Nous utilisons pour cela l'algorithme récursif de diffusion de Peters et Syska [104] (voir le chapitre 1 de cette même partie), le regroupement étant considéré comme un schéma de diffusion inversé. La complexité de cet algorithme est la suivante :

$$2\log_5(n^2)\alpha + 4\left\lfloor\frac{n}{2}\right\rfloor\delta + \left(\frac{n^2}{4} + n^2\log_5(n^2)\right)L\tau$$

Ces deux algorithmes sont opposés dans le sens où le premier s'effectue en un grand nombre d'étapes mais avec un taux de transmission optimal, alors que le second a un nombre logarithmique d'étapes mais un facteur de transmission très important. Si nous nous intéressons à l'échange total de messages de petite taille (comme c'est le cas dans une synchronisation où les messages sont unitaires ou dans certains algorithmes numériques parallèles comme dans le gardient conjugué [96]) il est important de trouver un algorithme ayant un minimum d'étapes tout en conservant un taux de transmission faible.

1. Le modèle de temps utilisé est celui adopté classiquement en *wormhole* et commutation de circuit (*cf* partie I chapitre 1).

3.4 Bornes inférieures

Nous analysons les bornes inférieures théoriques pour le problème de l'échange total dans un tore (n, n) dans un modèle Δ-port *full-duplex* dans um mode de commutation *wormhole*. Les résultats sont résumés ci-dessous :

Théorème 1 :

1. *La longueur maximale des chemins utilisés à chacune des étapes est au moins égale au diamètre, c'est à dire :* $2\left\lfloor\frac{n}{2}\right\rfloor$.
2. *Le temps minimum de transmission pour effectuer un échange total de messages de taille L est :* $\left\lceil\frac{(n^2-1)L}{4}\right\rceil\tau$.
3. *Le nombre minimum d'étapes pour réaliser l'échange total est :* $\log_5(n^3) - \log_5 8 + 1$

Preuve
Les démonstrations des deux premiers points sont directes par l'utilisation des remarques suivantes : chaque nœud doit atteindre au moins les quatre sommets du graphe les plus éloignés situés à une distance égale au diamètre du tore ; et chacun des nœuds doit recevoir $(n^2 - 1)$ messages de taille L en utilisant au plus 4 liens.
La preuve du troisième point est basée sur l'arc-bissection du tore (la définition de l'arc-bissection a été rappelée dans cette même partie dans le chapitre 2). Pour un tore carré de dimension n la valeur de cette bissection, notée b, est $2 \times n$. Notons S et D deux ensembles réalisant la bi-partition de la bissection. Le nombre total à échanger afin de réaliser l'échange total est $\left(\frac{n^2}{2}\right)^2$. En effet chaque nœud de S doit envoyer un message vers un nœud de D. Considérons deux nœuds $x \in S$ et $y \in D$, et notons a l'arc entre x et y. a est donc un arc de la bissection. À la première étape, le nombre de messages pouvant passer par a est égal à $5^0 = 1$. À l'étape suivante, puisque x a 4 voisins, il peut donc passer par x, et par voie de conséquence par a, $4 + 1 = 5^1$ messages. En continuant le même raisonnement, le nombre de messages passant par a à l'étape t est égal à : 5^{t-1}. Puisqu'il y a b arcs a, le nombre total de messages qui ont été envoyés pendant les t premières étapes est : $b \times 5^{t-1}$. Donc le nombre minimum d'étapes pour effectuer un échange total (noté $t^\star$) vérifie :

$$\begin{array}{lll} 2 \times n \times 5^{t^\star-1} & = & \left(\frac{n^2}{2}\right)^2 \\ - \quad 5^{t^\star-1} & = & \frac{n^3}{8} \\ - \quad t^\star & = & \log_5(n^3) - \log_5(8) + 1 \end{array}$$

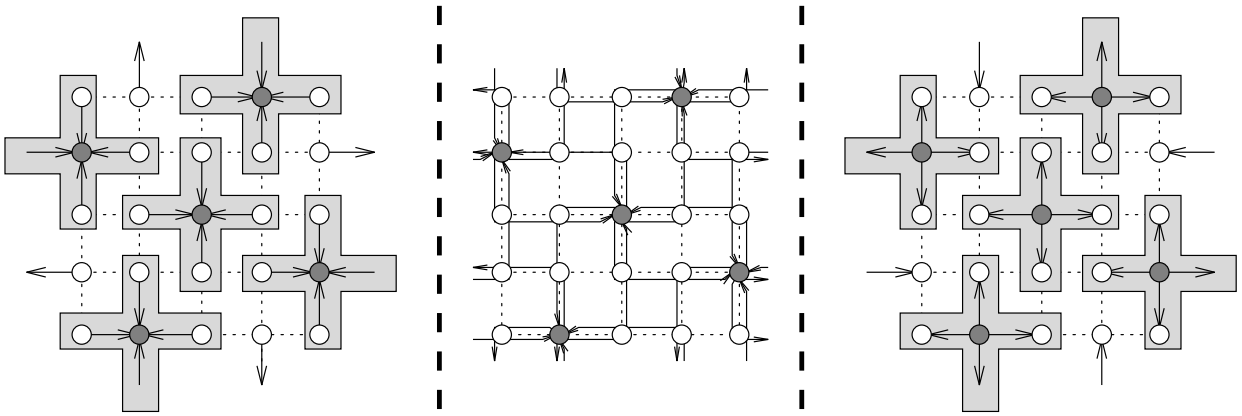
□

3.5 Deux nouveaux algorithmes

Nous présentons dans cette section un nouvel algorithme, optimal en nombre d'étapes (noté OE pour Optimal en nombre d'Étapes).

3.5.1 Description de l'algorithme

Nous détaillons tout d'abord le principe de base sur $WM(1)$ (voir figure 3.3). Il comporte trois étapes. Nous allons choisir des nœuds « privilégiés » qui serviront de nœuds de concentration. Nous allons prendre les éléments de S_1[2]. Nous aurions pu choisir un autre ensemble S_i, le raisonnement est identique. Les nœuds de S_1 son représentés en gris sur la figure 3.3. La première étape consiste à concentrer l'information (message de taille L) sur les nœuds de S_1. Les nœuds de S_1 échangent alors entre eux les informations qu'ils possèdent (message de taille $5L$), pour ensuite la rediffuser à chacun de leurs 4 voisins respectifs (message de taille $25L$).



Étape 1 : Concentration sur les nœuds de S_1 — Étape 1 : Échange entre les nœuds de S_1 — Étape 1 : Diffusion depuis les nœuds de S_1

FIG. 3.3 - *Les trois étapes de l'algorithme d'échange total sur* $WM(1)$.

Pour des tores $WM(k)$ de dimension plus grande, c'est à dire $k \geq 2$, le principe de l'algorithme est d'appliquer récursivement l'algorithme d'échange total décrit ci-dessus. Afin de décrire totalement cet algorithme, nous allons étendre les définitions des ensembles S_i et $S_{i,j}$ (définition 8) de la façon suivante :

Définition 11 :

Pour $\lambda \leq k$, *nous définissons les ensembles* $S_i(\lambda) = \{(x,y)/x+2y \equiv i[5^{k-\lambda}]\}$ $(i = 0,...,4)$ *et les sous-ensembles* $S_{i,j}(\lambda) = \{(x,y) \in S_i(\lambda)/x \equiv j[5^{k-\lambda}]\}$ $(i,j = 0,...,4)$ *sur lesquels l'algorithme d'échange total est appliqué à l'étape* λ *de récursion.*

L'algorithme est décrit dans la figure 3.5 et toutes les étapes de l'algorithme sur $WM(2)$ sont données en annexe. Comme dans le cas de $WM(1)$, les nœuds de concentration sont les éléments de $S_1(\lambda)$, mais il aurait pu s'agir de n'importe quel autre ensemble $S_i(\lambda)$ de façon équivalente.

2. Remarquons au passage que les S_i sont des ensembles de nœuds dominants au sens défini par Tsai et McKinley (*c.f.* chapitre 1.)

```
Procédure Gossip(λ)
début
  si (λ = 1) alors
    Concentration(λ,5^{k-1})   Concentration sur S_1(1)
    Échange(λ,5^{k-1})         Échanges entre S_{1,j}(1), pour tous j
    Diffusion(λ,5^{k-1})       Diffusion depuis S_1(1) vers S_{i≠1}(1)
  sinon
    Concentration(λ,5^{k-λ})   Concentration sur S_1(λ)
    Gossip(λ − 1)
    Échange(λ,5^{k-λ})         Échanges entre S_{1,j}(λ), pour tous j
    Diffusion(λ,5^{k-λ})       Diffusion depuis S_1(λ) vers S_{i≠1}(λ)
  fin si
```

FIG. 3.4 - *Algorithme d'échange total dans $WM(k)$.*

L'appel initial de cet algorithme pour réaliser un échange total dans $WM(k)$ est `Gossip(k)`. Nous détaillons dans les figures 3.5, 3.6 et 3.7 les schémas de communication élémentaires.

```
Procédure Concentration(λ,d)
début
  si (x, y) ∈ S_1(λ) alors
    faire en parallèle
      recevoir de (x + d[5^k], y)
      recevoir de (x − d[5^k], y)
      recevoir de (x, y + d[5^k])
      recevoir de (x, y − d[5^k])
    fin faire
  fin si
fin
```

FIG. 3.5 - *Algorithme `Concentration`.*

```
Procédure Échange(λ,d)
début
  si (x,y) ∈ S_1(λ) alors
    faire en parallèle
      échanger avec (x + d[5^k], y − 2d[5^k])
      échanger avec (x + 2d[5^k], y + d[5^k])
      échanger avec (x − 2d[5^k], y − d[5^k])
      échanger avec with (x − d[5^k], y + 2d[5^k])
    fin faire
  fin si
fin
```

FIG. 3.6 - *Algorithme* `Échange`.

```
Procédure Diffusion(λ,d)
  si (x,y) ∈ S_1(λ) alors
    faire en parallèle
      envoyer à (x + d[5^k], y)
      envoyer à (x − d[5^k], y)
      envoyer à (x, y + d[5^k])
      envoyer à (x, y − d[5^k])
    fin faire
  fin si
fin
```

FIG. 3.7 - *Algorithme* `Diffusion`.

3.5.2 Preuve de la correction de l'algorithme OE

Nous présentons dans la table 3.1 la trace d'exécution de l'algorithme pour $k = 3$. Dans cette table, nous notons $I(s)$ le volume d'informations contenu par chaque nœud pendant l'étape s, et par $N(s)$ le nombre de nœuds qui contiennent $I(s)$ informations. λ correspond à l'étape de récursion et d à la dilation du tore sur lequel l'algorithme s'exécute par rapport au tore initial. Les deux dernières colonnes de cette table contiennent respectivement la taille des messages envoyés et la distance à laquelle les messages sont envoyés.

Étape s	λ	d	$I(s)$	$N(s)$	Taille	Distance	Routine
1	3	1	5	3125	1	1	Concentration(3,1)
2	2	5	25	625	5	5	Concentration(2,5)
3	1	25	125	25	25	25	Concentration(1,25)
4	1	25	625	25	25	75	Échange(1,25)
5	1	25	625	625	625	25	Diffusion(1,25)
6	2	5	3125	625	625	15	Échange(2,5)
7	2	5	3125	3125	3125	5	Diffusion(2,5)
8	3	1	15625	3125	3125	3	Échange(3,1)
9	3	1	15625	15625	15625	1	Diffusion(3,1)

TAB. 3.1 - *Trace d'exécution de l'algorithme pour $k = 3$.*

Afin de prouver la correction de l'algorithme, nous devons montrer qu'à la fin tous les nœuds ont reçu 5^{2k} informations non-redondantes.

Pour s = 2,...,k nous calculons $I(k)$ et $N(k)$ après les appels à `Concentration`. $I(s) = 5 \times I(s-1)$ et $I(1) = 5$. Donc $I(k) = 5^k$. $N(s) = \frac{N(s-1)}{5}$ et $N(1) = 5^{2k}$. Donc $N(k) = 5^k$. De plus les informations rassemblées sont différentes puisque les nœuds qui envoient leurs données sont différents d'une étape à l'autre.

Pour s = k+1,...,3k $I(s) = 5 \times I(s-1)$ et $N(s) = N(s-1)$ pour $s = k+1, k+3, ..., 3K$. $I(s) = I(s-1)$ et $N(s) = 5 \times N(s-1)$ pour $s = k+2, k+4, ..., 3k-1$. Donc $I(3k) = 5^{2k}$ et $N(3k) = 5^{2k}$.
Les informations sont non-redondantes à la fin de l'algorithme. En effet à une étape de récursion λ donnée, la routine `Échange` implique des nœuds appartenant à des sous-ensembles $S_{1,j}(\lambda)$ différents et la routine `Diffusion` diffuse l'information précédemment collectée aux nœuds de $S_{i\neq 1}(\lambda)$ qui sont tous différents pour chaque λ.

3.5.3 Étude de la complexité

Nous notons $T(k)$ le temps pour réaliser un échange total dans $WM(k)$. Ce temps est composé de trois termes : le nombre d'étapes, la distance et le taux de transmission. Donc $T(k) = F_\alpha(k)\alpha + F_\delta(k)\delta + F_\tau(k)\tau$. Les équations de récurrence en fonction de k sont les suivantes :

$$\begin{array}{lcll} F_\alpha(k) & = & 1 + F_\alpha(k-1) + 1 + 1 & F_\alpha(0) = 0 \\ F_\delta(k) & = & 1 + 5 \times F_\delta(k-1) + 3 + 1 & F_\delta(0) = 0 \\ F_\tau(k) & = & 1 + 5 \times F_\tau(k-1) + 5^{2k-1} + 5^{2k} & F_\tau(0) = 0 \end{array}$$

Ces équations sont faciles à déduire de l'algorithme et des propositions 3, 4 et 5. La résolution de chacune d'entre elles conduit à la complexité suivante :

$$T(k) = 3k\alpha + \left(5\frac{5^k - 1}{4}\right)\delta + \left(\frac{2 \times 5^{2k} - 5^{k+1} - 1}{4} + 5^{2k}\right) L\tau$$

Remarquons que le taux de transmission n'est pas optimal. Il peut être amélioré en décomposant chaque dernière étape de récursion en deux phases impliquant des messages de taille plus petite. Nous décrivons cette amélioration dans la section suivante.

3.5.4 Amélioration de l'algorithme

Dans l'algorithme OE, la taille des messages est maximale ($5^{2\lambda}$) à chaque étape de diffusion. De plus, lors de cette phase, tous les liens ne sont pas utilisés. L'idée est alors d'améliorer l'algorithme OE en décomposant l'étape `Diffusion` en deux phases successives avec des messages de taille deux fois plus petite (c'est à dire de taille $5^{2\lambda-1}$). Nous appellerons ATT ce nouvel algorithme (pour Amélioration du Taux de Transmission).
Après `Échange`(λ, d), chaque nœud de $S_1(\lambda)$ contient toute l'information de $WM(\lambda)$. Puisque chaque voisin d'un nœud $(x, y) \in S_1(\lambda)$ appartient à des sous-ensembles $S_i(\lambda)$ (*c.f.* proposition 3), lw nœud (x, y) envoie toute l'information (*i.e.* un message de taille $5^{2\lambda-1}$) de $S_i(\lambda)$ à son voisin qui appartient à $S_i(\lambda)$ (voir figure 3.8).

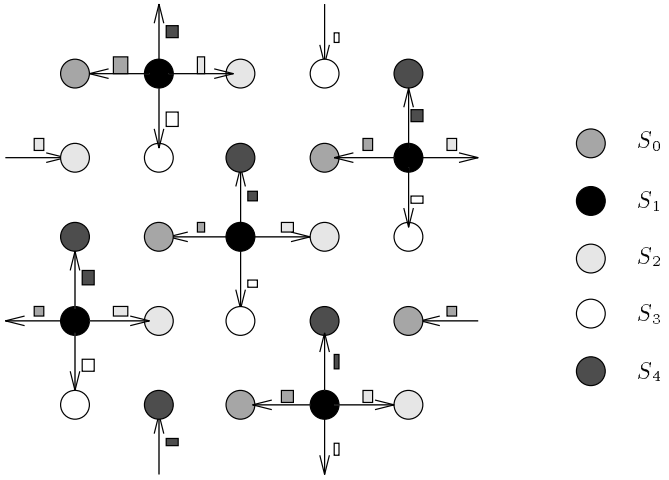


FIG. 3.8 - *Première étape de `Diffusion` de l'algorithme ATT.*

Après ces étapes, chaque nœud de $WM(\lambda)$ contient au moins toute l'information de son sous-ensemble d'appartenance $S_i(\lambda)$. Alors, chaque nœud de $WM(\lambda)$ envoie tout l'information qu'il détient à chacun de ses voisins, ce qui correspond à un message de longueur $5^{2\lambda-1}$. Remarquons qu'il est inutile d'envoyer l'information aux nœuds de $S_1(\lambda)$, et ces nœuds n'envoient à leur voisin uniquement que l'information de $S_1(\lambda)$. Nous présentons cette deuxième étape sur la figure 3.9, uniquement vu des nœuds appartenant à $S_0(\lambda)$ pour plus de clarté, mais les nœuds des autres ensembles $S_i(\lambda)$, $i \neq 1$, font exactement la même chose.

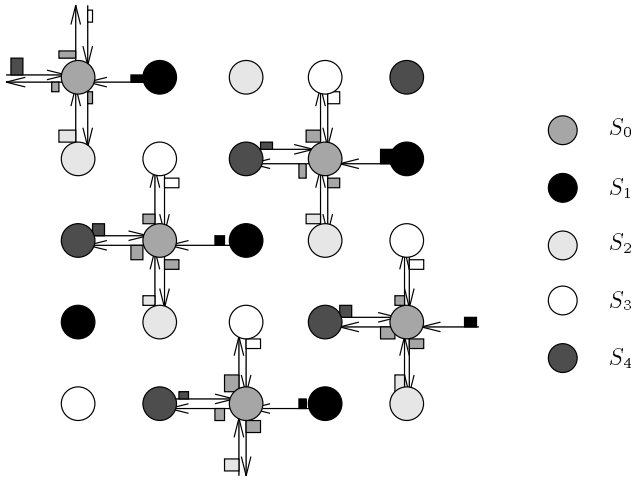


FIG. 3.9 - *Deuxième étape de* `Diffusion` *de l'algorithme ATT.*

Le calcul de la complexité de l'algorithme ATT est basé sur le même principe que pour l'algorithme OE :

$$\begin{array}{lcll} F_\alpha(k) & = & 1 + F_\alpha(k-1) + 1 + 1 + 1 & F_\alpha(0) = 0 \\ F_\delta(k) & = & 1 + 5 \times F_\delta(k-1) + 3 + 1 + 1 & F_\delta(0) = 0 \\ F_\tau(k) & = & 1 + 5 \times F_\tau(k-1) + 5^{2k-1} + 5^{2k-1} & F_\tau(0) = 0 \end{array}$$

La complexité de l'algorithme ATT est alors la suivante :

$$T(k) = 4k\alpha + \left(3\frac{5^k - 1}{2}\right)\delta + \left(3\frac{5^{2k} - 5^k}{4} + \frac{5^k - 1}{4}\right)L\tau$$

3.6 Extension à d'autres tailles de tore

Les algorithmes précédents peuvent s'adapter directement à toutes les tailles de tore de type (p^k, p^k) [3]. Le problème est de trouver les ensembles S_i et $S_{i,j}$ vérifiant les propositions 3, 4 et 5. Nous donnons sur la figure 3.10 de telles décompositions pour quelques tores de taille (p^k, p^k).

3. Dans la suite nous ne donnerons que les complexités de l'algorithme OE, le principe étant le même pour l'algorithme ATT.

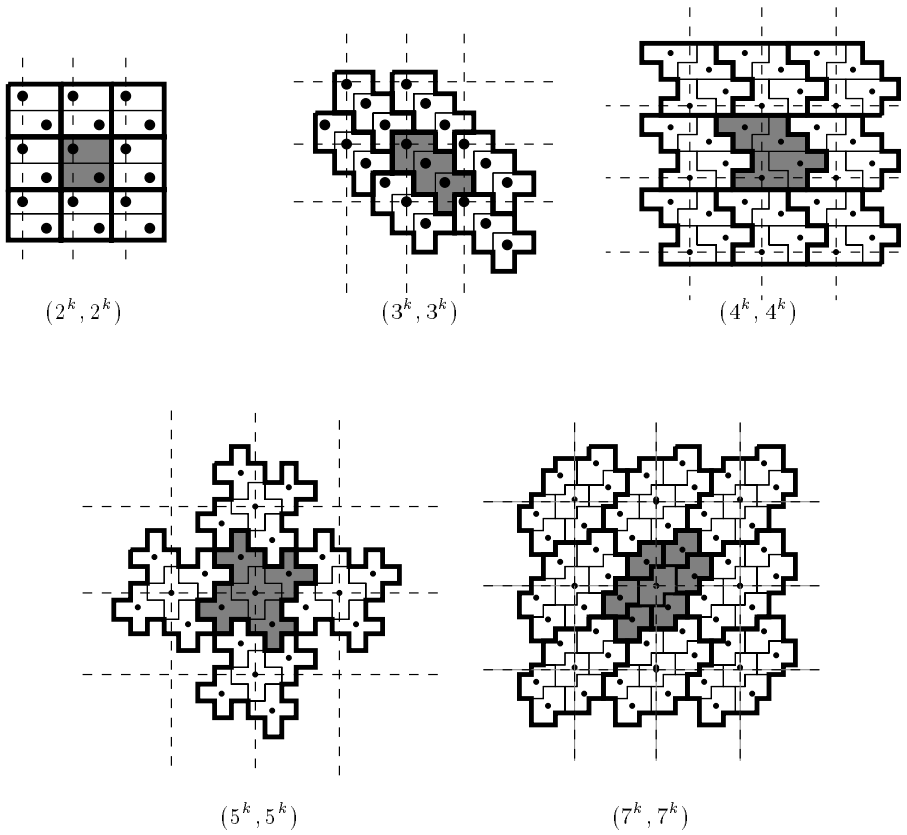


FIG. 3.10 - *Découpage récursif de tores* (p^k, p^k)

La définition des ensembles S_i (et donc $S_{i,j}$) se dérive très naturellement à partir de la définition 8 de la façon suivante :

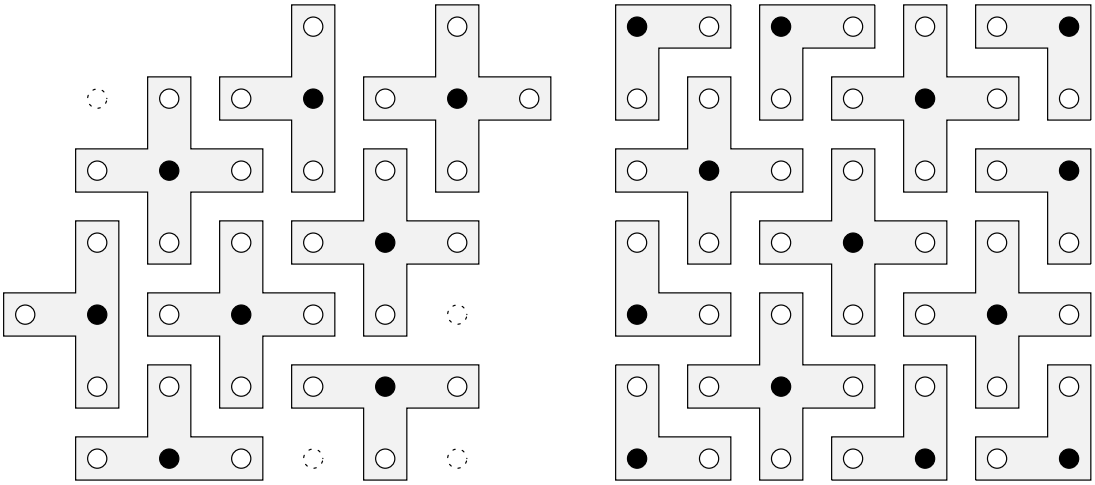
Définition 12 :

Nous définissons les ensembles S_i, $i = 0, ..., (p-1)$, *avec* p *impair, de la façon suivante :*
$S_i = \left\{(x, y) \in V / x + \frac{p-1}{2} y \equiv i[p]\right\}$.
Étant donné $i = 0, ..., (p-1)$, *nous définissons également les sous-ensembles* $S_{i,j}$ *de* S_i, $j = 0...(p-1)$, *ainsi :* $S_{i,j} = \{(x, y) \in S_i / x \equiv j[p]\}$

Il est facilement démontrable que ces ensembles vérifient bien les propositions 3, 4 et 5. Cela conduit donc à un algorithme d'échange total sur les tores de dimension (p^k, p^k) de complexité égale à :

$$T(p,k) = \left\lceil \frac{p-1}{4} \right\rceil \left(3k\alpha + \left(\left(3 + \left\lfloor \frac{p}{2} \right\rfloor \right) \left(\frac{p^k - 1}{p-1} \right) \right) \delta + \left(\frac{2 \times p^{2k} - p^{k+1} - 1}{p-1} + p^{2k} \right) L\tau \right)$$

Remarquons que le nombre d'étapes n'est pas optimal, hormis pour $p = 5$. Cependant il est possible de trouver « à la main » des décompositions spécifiques (hélàs non récursives) réalisant un nombre minimal d'étapes. Nous donnons des exemples de décomposition pour quelques tailles de tore dans la figure 3.11.



Tore $(6,6)$ Tore $(7,7)$

FIG. 3.11 - *Exemples de pavage spécifique des tores* $(6,6)$ *et* $(7,7)$ *donnant des algorithmes optimaux en nombre d'étapes.*

À partir de ces décompositions il nous est alors possible d'étendre nos algorithmes à toutes les tailles de tore de type $(p_1^{k_1} \times p_2^{k_2})$ par utilisation des décompositions du tore $(p_1^{k_1} \times p_1^{k_1})$ puis $(p_2^{k_2} \times p_2^{k_2})$. Donc il est possible d'étendre les algorithmes d'échange total à tous les tailles de tore carré pour peu que l'on sache trouver des décompositions récursives pour tous les tores carrés de dimension p^k où p est premier.

Nous présentons sur la figure 3.12 la décomposition du tore $(15,15)$ en tores $(5,5)$ puis $(3,3)$, et dans la figure 3.13 la décomposition symétrique.

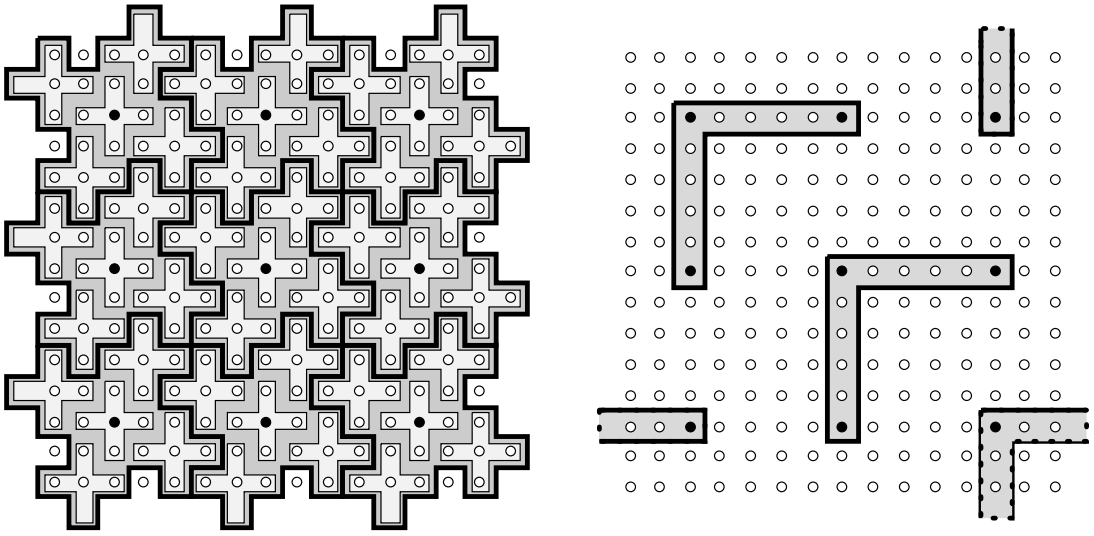


FIG. 3.12 - *Décomposition du tore* $(15,15)$ *en tores* $(5,5)$ *puis* $(3,3)$

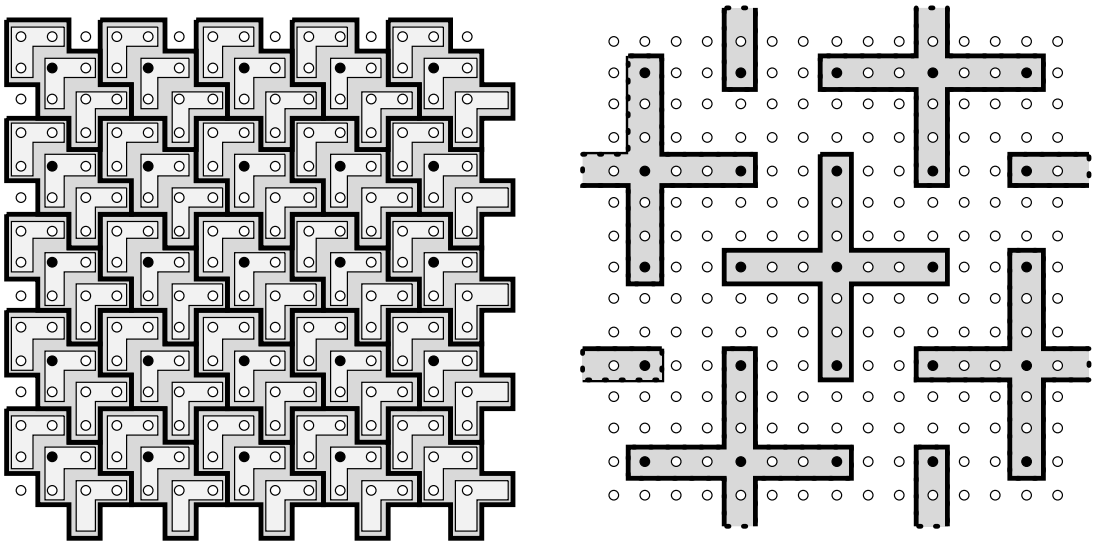


FIG. 3.13 - *Décomposition du tore* $(15,15)$ *en tores* $(3,3)$ *puis* $(5,5)$

Le coût de l'algorithme d'échange total pour des tores carrés de dimension $(p_1^{k_1} \times p_2^{k_2})$ se dérive directement et est égal à :

$$T(p_1,k_1,p_2,k_2) = F_\alpha(p_1,k_1,p_2,k_2)\alpha + F_\delta(p_1,k_1,p_2,k_2)\delta + F_\tau(p_1,k_1,p_2,k_2)\tau$$

$$F_\alpha(p_1,k_1,p_2,k_2) = \left(3\left\lceil\frac{p_1-1}{4}\right\rceil k_1 + 3\left\lceil\frac{p_2-1}{4}\right\rceil k_2\right)$$

$$F_\delta(p_1,k_1,p_2,k_2) = \left\lceil\frac{p_1-1}{4}\right\rceil\left(3+\left\lceil\frac{p_1}{2}\right\rceil\right)\left(\frac{p_1^{k_1}-1}{p_1-1}\right) + p_1\left[\left\lceil\frac{p_2-1}{4}\right\rceil\left(3+\left\lceil\frac{p_2}{2}\right\rceil\right)\left(\frac{p_2^{k_2}-1}{p_2-1}\right)\right]$$

$$F_\tau(p_1,k_1,p_2,k_2) = \left\lceil\frac{p_1-1}{4}\right\rceil\left(\frac{p_1^{k_1}-1}{p_1-1}\right)\left(1+p_1^{k_1}p_2^{2k_2}(p_1+1)\right) + p_1^{k_1}\left\lceil\frac{p_2-1}{4}\right\rceil\left[\frac{2p_2^{2k_2}-p_2^{k_2+1}-1}{p_2-1} + p_2^{2k_2}\right]$$

Toutes les décompositions ne sont pas équivalentes. En effet suivant l'ordre de décomposition choisie, le facteur de bande passante et du nombre d'étapes n'est pas le même[4].
Prenons un exemple. Considérons un tore carré $(45,45)$. Ce tore peu être décomposé en tore $(3^2,3^2)$ puis $(5,5)$ ou l'inverse. Calculons les différents facteurs de bande passante: $F_\tau(5,1,3,2) = 3171$, $F_\tau(3,2,5,1) = 3883$.

Nous avons donc tout intérêt à utiliser pour un tore (p^k,p^k), la décomposition suivante: $p = p_1 \times p_2$ tel que $p_1 > p_2$.

3.7 Quelques résultats de simulation

Afin de mieux se rendre compte des différences de temps entre les différentes versions de l'algorithme que nous proposons, nous avons effectué quelques simulations en utilisant les paramètres d'un Cray T3D. Ces paramètres ont été déterminés expérimentalement.[5] Nous

4. Par contre le nombre d'étapes reste identique.

5. Nous présentons dans le chapitre 2 de la partie III les résultats d'évaluation et de modélisation des performances de communication du Cray T3D.

présentons tout d'abord sur la figure 3.14 la comparaison entre les différents algorithmes proposés en prenant 625 processeurs et en utilisant les temps élémentaires de communication obtenus avec PVM.

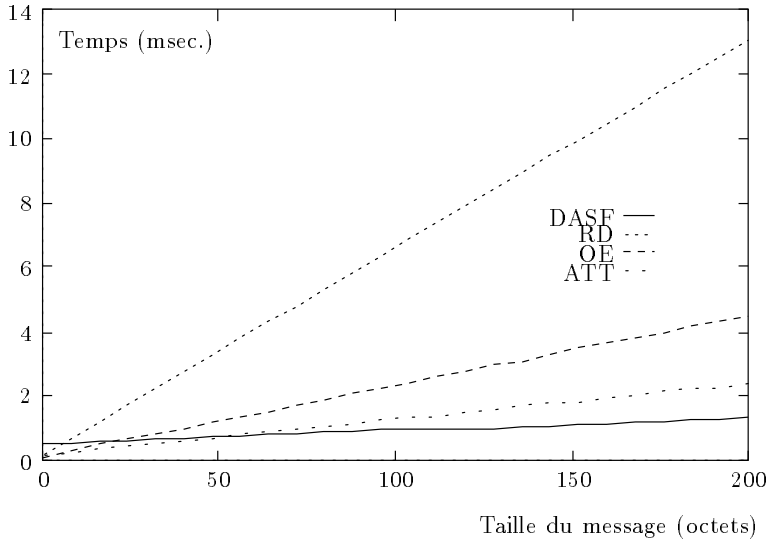


FIG. 3.14 - *Comparaison des différents algorithmes sur 625 processeurs avec les paramètres de PVM sur T3D*

La figure 3.15 effectue la même comparaison, mais en utilisant les paramètres de la bibliothèque de mémoire virtuelle paratagée, ShMem (*cf* le chapitre 2 de la partie III).

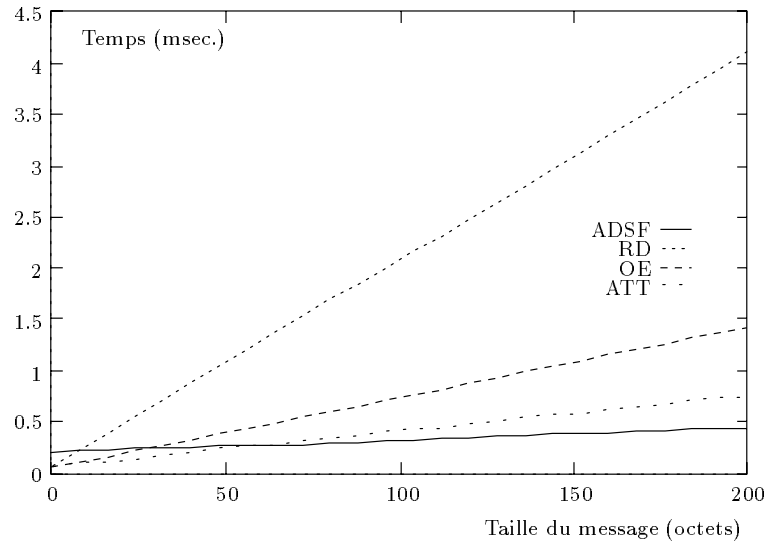


FIG. 3.15 - *Comparaison des différents algorithmes sur 625 processeurs avec les paramètres de ShMem sur T3D*

Comme on peut le constater, le facteur de la bande passante est très important dès que la taille des messages grandit. En effet l'algorithme OE est meilleur que l'algorithme ADSF

jusqu'à des messages de taille 25. Cette taille passe à 64 pour l'algorithme ATT. Ceci est vrai quelle que soit la bibliothèque utilisée.

Il faut cependant modérer ces résultats : premièrement il s'agit de simulations, et nous ne prenons pas en compte le temps interne de recomposition des messages. Ce temps est proportionnel au nombre d'étapes, ce qui est à l'avantage des algorithmes OE et ATT. Deuxièmement, si l'on prend comme exemple l'algorithme du produit matrice-vecteur qui utilise un échange total, un message de taille 64 sur 625 processeurs correspond à des tailles de matrice et de vecteur environ égale à 5000, ce qui donne une idée de la taille du problème correspondant. Troisièmement, l'échange total n'est pas seulement utilisé dans les algorithmes numériques avec des messages longs, mais aussi avec des messages très courts, comme par exemple dans l'algorithme du gradient conjugué, où les messages sont de taille 1. De plus le schéma est exécuté un certain nombre de fois puiqu'il s'agit d'un algorithme itératif. Dans ce dernier exemple, on tout intérêt à avoir un algorithme avec un nombre minimal d'étapes.

Dans la figure 3.16 nous présentons une comparaison entre un échange total effectivement implémenté en PVM sur T3D (voir les résultats du chapitre 2 de la partie III) avec des simulation de l'algorithme OE avec les paramètres de PVM et de ShMem. Ces comparaisons ont été faites sur un tore carré de taille 16 (donc 64 processeurs). Comme on peut le voir, il y a un très grand intérêt à implémenter des algorithmes efficaces de communication globale, puisque dans cet exemple nous obtenons des facteurs 100 et 1000 entre les différents algorithmes. Comme dans les figures précédentes, il faut modérer ces résultats par le fait que nos simulations ne prennent pas en compte le temps de gestion interne des messages, cependant les différences de temps entre les algorithmes resteront toujours très importantes.

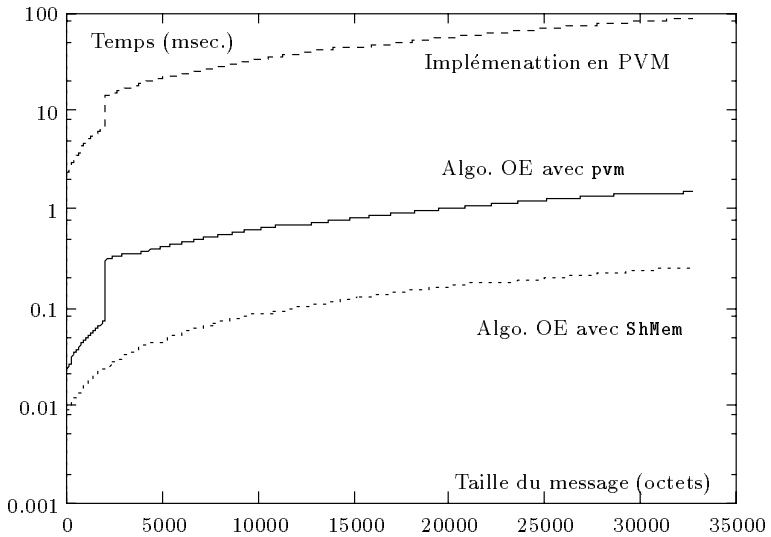


FIG. 3.16 - *Comparaison entre les temps d'un échange total réalisé sur 64 processeurs d'un Cray T3D et la simulation des temps de l'algorithme OE avec les paramètres du T3D suivant la bibliothèque utilisée (PVM ou Shared Memory). Les temps sont exprimés en millisecondes et l'axe des ordonnées suit une échelle logarithmique.*

3.8 Conclusion

Nous avons présenté dans ce chapitre un nouvel algorithme d'échange total sur machine parallèle à mémoire distribuée configuré en tore 2D. L'analyse théorique a conduit à plusieurs versions suivant le nombre d'étapes, la distance maximale à traverser ou la facteur du taux de transmission. Nous avons proposé des algorithmes qui atteignent les bornes inférieures dans un ou deux des facteurs précédents. Finalement nous avons décrit un algorithme intermédiaire qui représente un bon compromis, puisqu'il est en un nombre logarithmique d'étapes, et des facteurs de distance et de taux de transmission proches de l'optimal.

Les algorithmes ont été décrits pour des tores carrés de taille 5^k, et nous avons montré comment il était possible de les étendre assez facilement à n'importe quelle taille de tore carré. Les résultats de simulation faites sur un T3D montrent tout d'abord quel gain peut apporter la conception d'algorithmes efficaces d'échange total par rapport à ce que l'on peut faire actuellement, mais également que l'algorithme adapté de la commutation de messages est très efficace dès que la taille des messages grandit. Il ne faut pas cependant minimiser l'importance d'algorithmes spécifiques au *wormhole* ayant un nombre optimal d'étapes, car les schémas de communication globales sont souvent utilisés sur des messages très petits, voire unitaires.

3.9 Annexe

Nous donnons les étapes détaillées de l'algorithme OE pour un tore $(25, 25)$ $(WM(2))$ dans les figures suivantes.

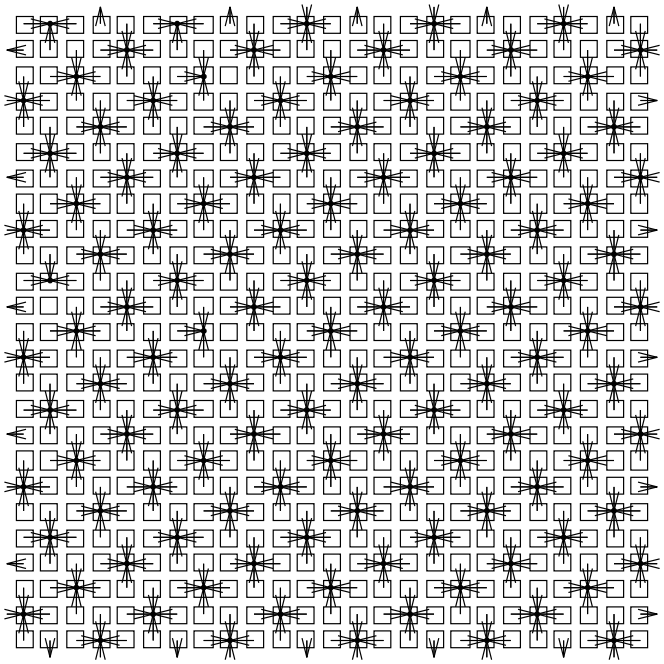


FIG. 3.17 - *Première étape :* `Concentration(2,1)`

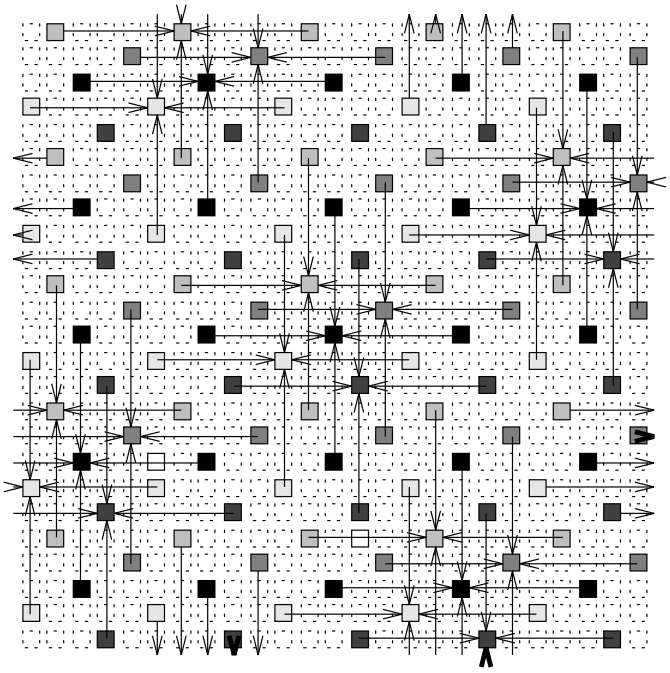


FIG. 3.18 - *Deuxième étape:* `Concentration(1,5)`

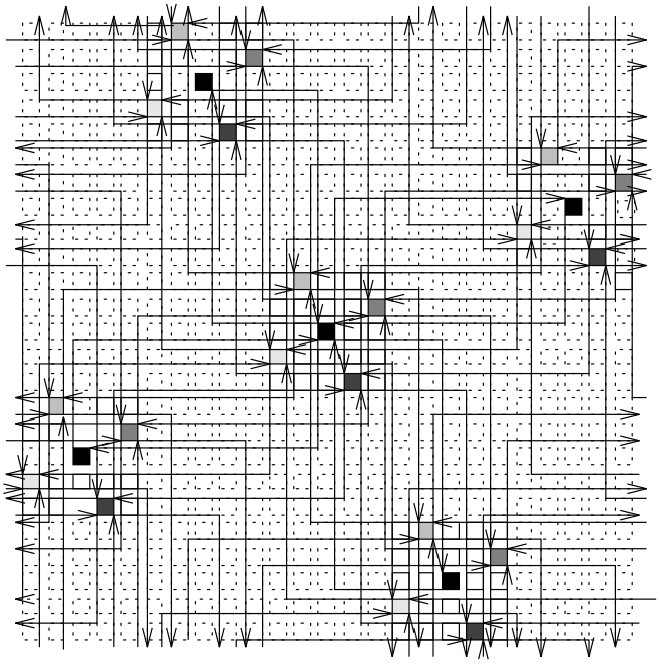


FIG. 3.19 - *Troisième étape:* `Échange(1,5)`

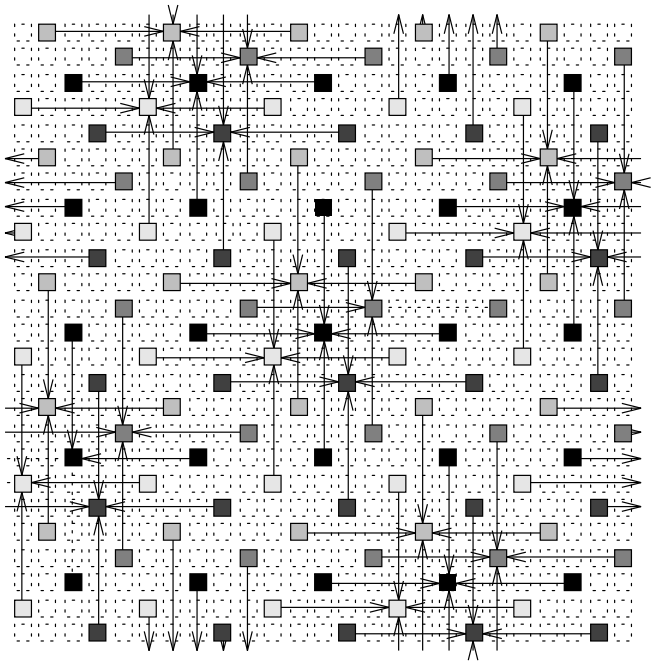


FIG. 3.20 - *Quatrième étape :* `Diffusion(1,5)`

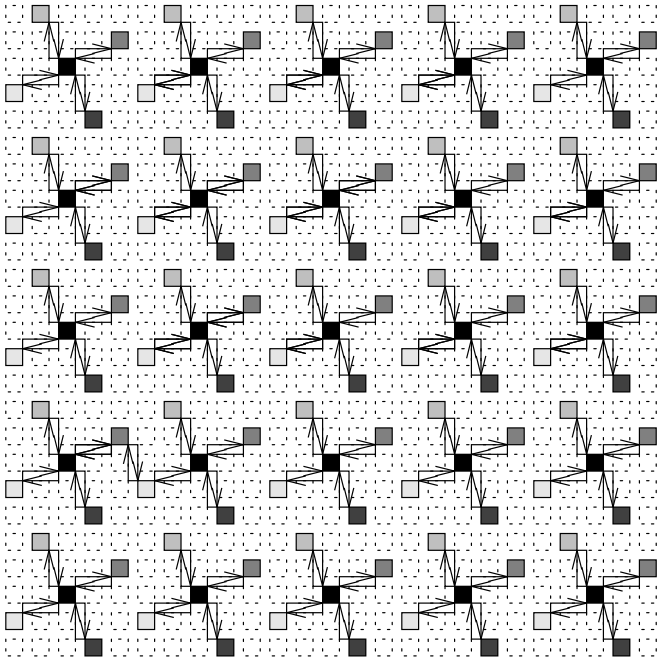


FIG. 3.21 - *Cinquième étape :* `Échange(2,1)`

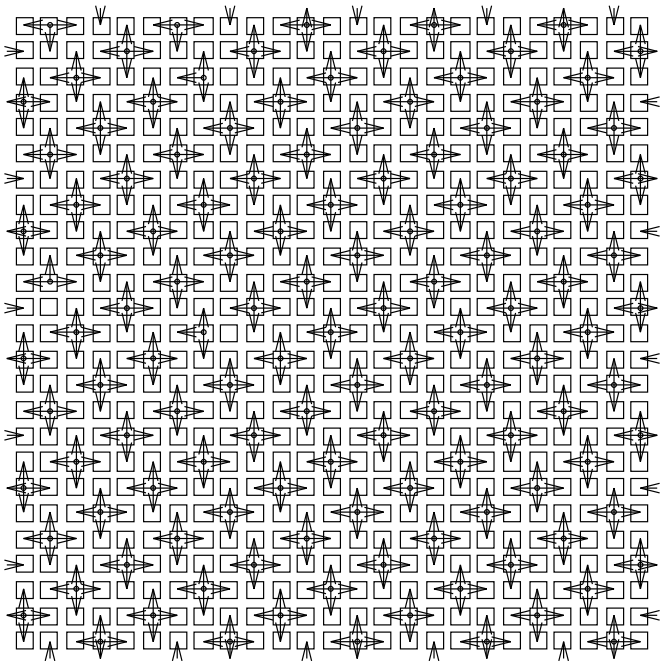


FIG. 3.22 - *Sixième étape :* `Diffusion(2,1)`

Partie III

Masquage des communications dans les algorithmes parallèles

Nous allons présenter dans cette partie d'autres voies permettant de diminuer la part de la communication dans le temps parallèle total. Ces méthodes ne se fondent plus sur l'optimisation des algorithmes de communication, comme nous avons pu le voir dans la partie précédente, mais cherchent à diminuer l'importance des communications dans l'algorithme parallèle lui-même. Une de ces méthodes est le recouvrement des communications par du calcul.

Nous allons décrire, dans un premier chapitre, les principales méthodes utilisées pour minimiser le temps de communication, en nous intéressant plus particulièrement aux techniques permettant de masquer les communications. Ces techniques nécessitent une bonne modélisation des performances des machines. Nous proposons, dans un second chapitre, une évaluation des performances des communications du Cray T3D. Enfin, dans un troisième temps, nous illustrons les techniques présentées par différents algorithmes numériques de calcul de transformée de Fourier.

Chapitre 1

Principes généraux et méthodologie

Nous présentons, dans ce chapitre, quelques méthodes de réduction du sur-coût des communications dans un algorithme parallèle. Nous nous attardons notamment sur les techniques permettant de recouvrir le temps de communication en le masquant par du calcul. Le travail sur l'utilisation de routines mixant communication et calcul à été réalisé avec L. Colombet[a], F. Desprez[b], P. Michallon[c] et D. Trystram [26]. Dans un second temps, nous nous sommes intéressés avec L. Colombet et P. Michallon, à des techniques algorithmiques masquant au mieux les communications, indépendamment du schéma de communication [27].

[a] L. Colombet est ingénieur de recherche au CEA de Grenoble
[b] F. Desprez est maître de conférences à l'Université de Bordeaux
[c] P. Michallon est ingénieur de recherche à l'ETCA d'Arcueil

1.1 Introduction

Nous avons vu, dans la partie précédente, une première méthode pour minimiser l'influence des communications, par optimisation de l'algorithme de communication lui-même. Nous avons pu constater quelques limites d'une telle méthode : les algorithmes sont souvent complexes à mettre en œuvre et, de plus, le niveau de programmation à utiliser pour pouvoir implémenter de manière efficace ces algorithmes n'est pas directement accessible au niveau utilisateur. Autre inconvénient : cette méthodologie est complètement à l'opposé de toute notion de portabilité, car l'algorithme dépend directement de la topologie de la machine cible.

Il nous faut donc proposer une autre approche permettant à l'utilisateur d'obtenir des parallélisations portables et efficaces de ses applications. L'idée est la suivante : la perte de portabilité provient de la prise en compte de la topologie du réseau d'inteconnexion de la machine, il ne nous faut donc plus travailler au niveau de la communication mais plutôt sur l'algorithme lui-même, afin de minimiser l'influence du coût de la communication sur le temps d'exécution parallèle total.

De plus, la parallélisation d'une application donne souvent des codes qui sont des alternances de séquences de calculs et de communications. Aucune bibliothèque numérique ac-

tuelle, qu'elle soit séquentielle (comme les BLAS [89] ou LAPACK [5]) ou parallèle (comme ScaLAPACK [33]) ne permet de mélanger efficacement communications et calculs.

Pour cela, nous proposons de modifier les algorithmes parallèles afin d'occuper les processeurs de calcul pendant qu'une communication s'effectue. Il s'agit donc, d'un point de vue temporel, de **masquer la communication par du calcul** en profitant du parallélisme potentiel entre le processeur de calcul et le réseau d"interconnexion.

Le but de ce chapitre est de proposer des schémas algorithmiques sur lesquels nous appliquons des techniques basées sur le découpage en tâches indépendantes, le regroupement et le ré-ordonnancement local de tâches. Ces techniques sont paramétrables en fonction des machines cibles sur lesquelles on désire implanter les algorithmes, permettant ainsi d'optimiser au mieux le recouvrement.

Nous montrons, tout d'abord, l'intérêt des communications non-bloquantes et pourquoi la simple utilisation de celles-ci ne permet pas forcément de masquer le temps de communication. Puis, dans un second temps, nous décrivons quelques méthodes utilisées pour diminuer le sur-coût des communications lorsqu'on ne désire pas optimiser l'algorithme de communication. Enfin, nous présentons la méthodologie adoptée et les techniques employées pour masquer au mieux les communications.

1.2 Le recouvrement n'est pas que la simple utilisation de communications asynchrones.

Le temps d'exécution d'un programme parallèle est une fonction des temps de calcul et de communication. Si dans un programme parallèle les communications ne sont pas recouvertes, le temps total est la somme de ces deux temps. Puisque le temps de calcul d'un algorithme donné ne peut être réduit, l'idée de base dans le recouvrement est de minimiser le temps d'inactivité du processeur par l'utilisation de communications non-bloquantes, de techniques de pipelines [87] ou plus généralement en changeant la granularité de calcul de l'algorithme [123].

L'étude du recouvrement des communications a connu un grand intérêt depuis l'apparition de la dernière génération de machines parallèles qui possèdent des mécanismes matériels autorisant les communications et les calculs de façon concurrente.

Nous illustrons l'intérêt de l'utilisation de communications non-bloquantes sur deux exemples simples. Nous mettons également en évidence, dans ces exemples, le fait que l'utilisation de communications asynchrones n'est pas suffisante. Considérons deux processeurs P_1 et P_2. Chaque processeur doit exécuter deux tâches T_i^j (où i est le numéro du processeur et j est le numéro de la tâche à exécuter) et une communication. Nous supposons également que T_i^2 a besoin du résultat de T_i^1 pour pouvoir commencer. Nous considérons deux cas (schémas de gauche sur la figure 1.1) : dans le premier cas P_2 a fini son premier calcul avant P_1, mais il doit attendre la fin de T_1^1 pour pouvoir commencer T_2^2. Dans le deuxième cas, le processeur P_1 ne peut envoyer ses résultats tant que T_2^1 n'est pas finie (à cause de la synchronisation entre émetteur et récepteur : nous sommes dans un modèle de communication bloquante et

donc le message ne peut être envoyé que lorsque le récepteur est prêt à le recevoir *c.f.* chapitre 2 de la partie I). Dans ces deux cas, un des deux processeurs doit attendre l'autre pour pouvoir envoyer ou recevoir un message. En utilisant des communications non-bloquantes, nous pouvons réduire ce temps d'inactivité (voir les schémas de droite sur la figure 1.1).

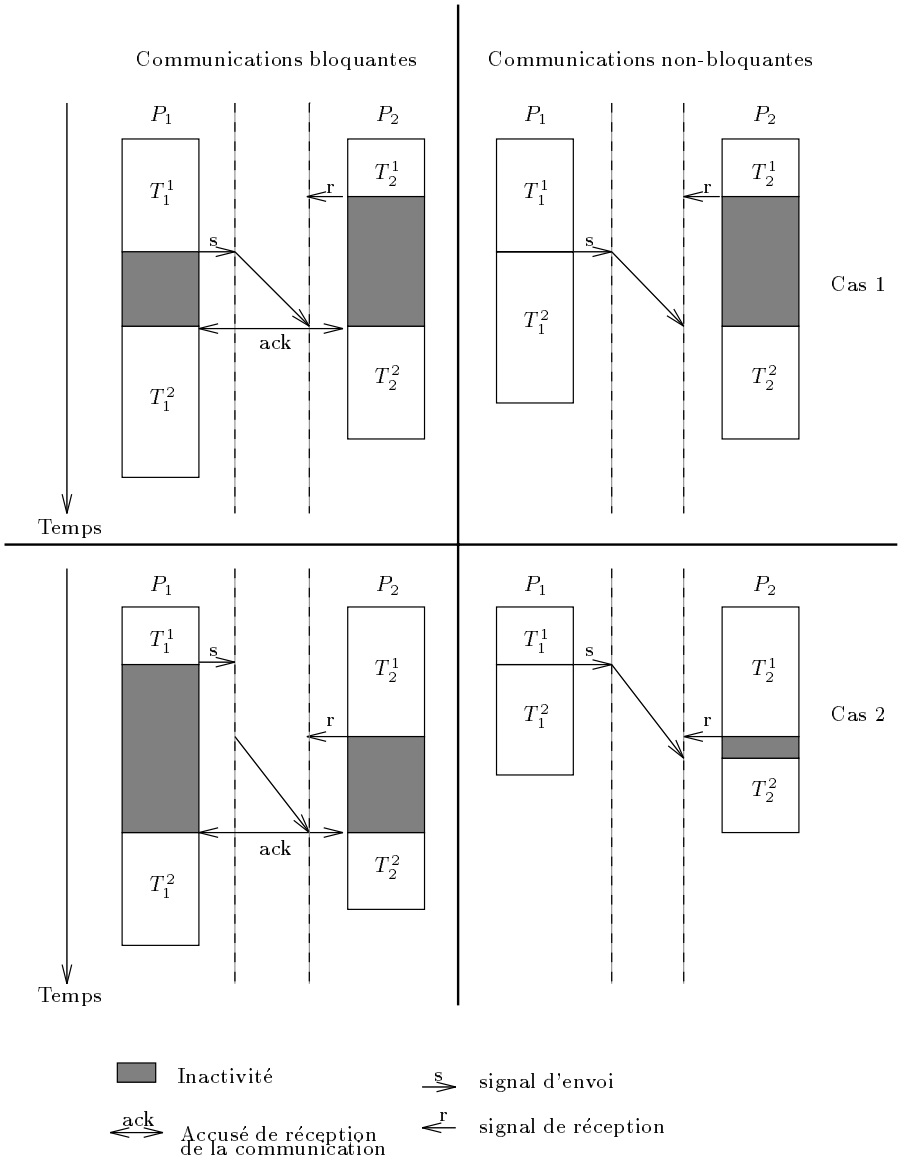


FIG. 1.1 - *Comparaison entre communications bloquantes et non-bloquantes.*

Cependant, il n'est pas possible d'obtenir un recouvrement total des communications dans le premier cas, même en utilisant des communications non-bloquantes. En recourant, par exemple, à des techniques de pipeline, il est possible de réduire au maximum le temps d'inactivité du processeur P_2 dans le cas 1 (voir figure 1.2). Cette technique est simple : il s'agit de découper la tâche T_1^1 en sous-tâches indépendantes et d'envoyer dès que possible un résultat partiel. Il y a enchaînement des envois et des calculs (d'où le terme de pipeline),

le calcul d'un résultat masquant le temps d'envoi du précédent.

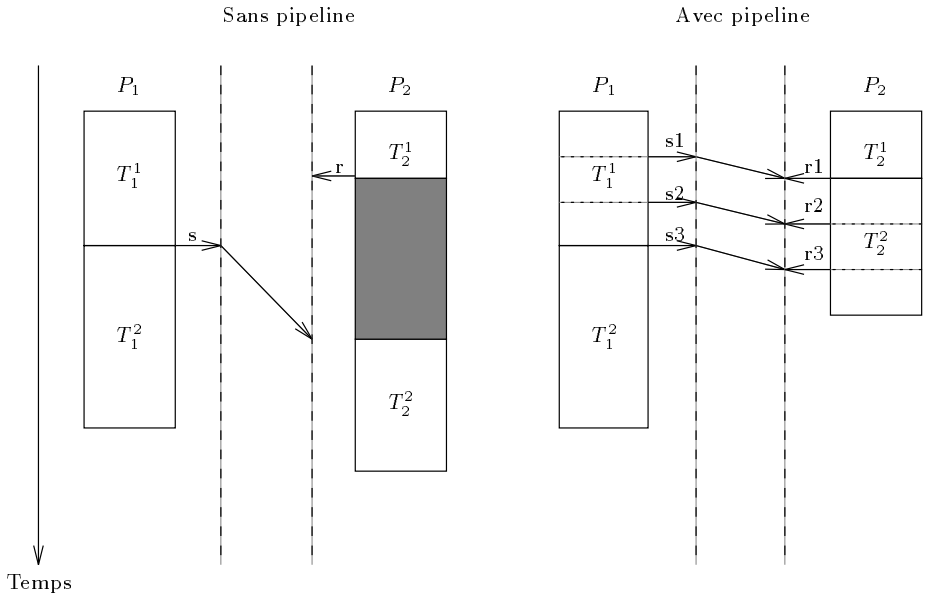


FIG. 1.2 - *Technique de pipeline avec des communications non-bloquantes.*

Cette méthode n'est pas la seule possible. Nous allons en présenter maintenant quelques autres utilisées pour minimiser le temps des communications, en nous intéressant plus particulièrement aux techniques permettant le masquage des communications.

1.3 Les différentes méthodes pour minimiser le temps de communication

On peut classer ces différentes méthodes en trois grandes catégories. La première consiste à utiliser des mécanismes du système d'exploitation de la machine parallèle. La deuxième catégorie regroupe toutes les optimisation qui peuvent être effectuées de manière automatique, sans que l'utilisateur ait à modifier son code. Enfin, la troisième consiste en l'utilisation de bibliothèques permettant d'exprimer plus simplement certains algorithmes avec recouvrement des communications.

1.3.1 Le masquage des communications au niveau système

Les messages actifs[126] : Le principe est d'associer aux données envoyées une adresse d'une séquence d'instructions, définie par l'utilisateur, qui sera exécutée sur les données contenues dans le message. Ce système permet donc de déclencher un traitement à distance sur les données envoyées et ainsi, de mixer calculs et communications. Ce système est employé dans le langage *Split-C* [43] afin de minimiser le sur-coût de communication. Une illustration de cette utilisation est présentée sur l'implémentation

de l'algorithme parallèle du produit de matrices [126]. Ce mécanisme est également utilisé par une bibliothèque de communication sur la machine parallèle CM5.

Les messages actifs sont un outil permettant de masquer les communications et de minimiser les temps d'inactivité des processeurs, mais le travail reste à la charge de l'utilisateur.

Le *multithreading* : Cette technique consiste à faire du multi-processus (*multiprocessing* en anglais) sur les processeurs. Nous avons donc une liste de processus en attente de calculer qui sont activés lorsque le processus qui était actif est bloqué en attente d'une donnée.

Ce système, basé sur un modèle de programmation par parallélisme de tâches (voir partie I chapitre 1), est totalement indépendant de l'algorithme, et c'est à l'utilisateur de concevoir le mieux possible son programme afin qu'il y ait toujours un processus qui puisse s'exécuter. Une implémentation de ce modèle est proposé dans TPVM [54] et dans le langage ATHAPASCAN [35, 105].

1.3.2 Optimisation des communications dans la génération de code Fortran parallèle

Le modèle de programmation *data-parallel* consiste en une suite d'opérations appliquées de façon identique sur un grand nombre de données différentes. Le parallélisme peut être implicite ou explicite avec des annotations indiquant la manière de répartir les données sur les processeurs.

Ce modèle de programmation étant particulièrement adapté aux programmes de type scientifique, il a été implémenté par des langages Fortran. Il existe de nombreux langages et compilateurs Fortran parallèles : on peut citer bien sûr *High Performance Fortran* (HPF) [56], PARADIGM [103] ou encore *Fortran D* [58]. Nous nous intéressons ici plus particulièrement aux optimisations des communications dans Fortran D [123].

L'approche de base du compilateur Fortran D est de convertir un programme Fortran en programmes de type SPMD exécutables sur chacun des processeurs. Ces programmes SPMD contiennent des ordres explicites d'échanges de messages qui sont générés à chaque fois que le programme veut accéder à une donnée non locale. Les principales méthodes utilisées, lors de l'optimisation des communications générées par le compilateur de Fortran parallèle, sont **l'analyse de dépendance** et **l'analyse de flots de données**.

Les optimisations des communications sont de deux types :

1. Réduction du sur-coût de communication :

   La vectorisation de messages : cette technique utilise une analyse de dépendance de données afin de regrouper tous les éléments d'un message destiné à un même processeur dans un vecteur unique.

   La fusion de messages : une fois la vectorisation effectuée, la fusion permet elle d'éliminer les messages redondants

L'agrégation : ici, le compilateur essaye de détecter si plusieurs messages différents sont destinés au même processeur afin de les regrouper dans un seul envoi.

Les communications collectives : il s'agit de générer une communication globale lorsqu'un même message doit être envoyé à plusieurs processeurs.

Autres optimisations : la duplication des calculs afin d'éviter une communication, la possibilité qu'un processeur effectue des calculs sur une donnée qui ne lui appartient pas, ou encore des optimisations du même type que précédemment mais qui ne se font qu'à l'exécution, dans le cas où le flot de données ne peut être connu à la compilation.

2. Le masquage du sur-coût de communication : Toutes les techniques pour recouvrir les communications sont basées sur des méthodes de pipeline, soit à grain fin (*fine grain pipelining*) soit à gros grain (*coarse grain pipelining*) lorsqu'on tient compte du temps d'initialisation du pipeline [88, 103].

   Le pipeline de messages : Cela consiste à insérer dans le code un ordre d'envoi dès qu'une référence non locale est détectée et à mettre l'ordre de réception correspondant juste avant que la donnée ne soit utilisée.

   Le pipeline de messages vectorisés : Il s'agit d'appliquer le même genre de technique mais sur un message vectorisé. C'est à dire qu'un message vectorisé destiné à un processeur est découpé en petits bouts afin d'être envoyé par une succession d'ordres d'envois (*send*), ceci permet alors de masquer le temps de transit du message. C'est le principe présenté précédemment sur la figure 1.2.

   Le ré-ordonnancement d'itérations : Le pipeline de messages vectorisés ne marche pas dans tous les cas, notamment lorsque la structure des calculs ne permet pas de masquer le temps de transit de messages. Il est possible alors de réordonner dans une certaine mesure les boucles, afin que le bon positionnement des ordres d'envoi permettent effectivement de masquer les communications. Le compilateur PARADIGM tient compte en plus d'une estimation des coûts de calcul et de communication afin d'ajuster la granularité du pipeline [103].

Comme on a pu le voir, le nombre d'optimisations que l'on peut apporter aux communications générées par un compilateur est essentiellement de l'ordre de la correction des mauvais choix faits par ce dernier : élimination de communications redondantes et vectorisation de messages. Seule l'optimisation consistant à recouvrir dans une certaine mesure les communications apporte une amélioration au code « naïf » qu'aurait pu écrire un utilisateur. Ces optimisations ne sont pas réalisables à la compilation lorsque le flot de données et les communications sont irréguliers, mais à l'exécution, ce qui en limite forcément l'efficacité. De plus, le recouvrement des communications reste limité à des échanges de boucles et à des techniques de pipeline. On peut remarquer, cependant, que les compilateurs commencent à prendre en compte les coûts de calcul et de communication grâce à des modèles de machines afin d'améliorer la granularité du pipeline [103].

1.3.3 Les bibliothèques mixant calculs et communications

En vue de faciliter le travail des programmeurs, une idée séduisante a été de concevoir des bibliothèques de routine permettant le masquage des communications. Walker propose en 1988 une bibliothèque de routines de communications non-bloquantes VMLSCS (*Virtual Machine Loosely Synchronous Communication System*) [128] permettant l'utilisation efficace des hiérarchies mémoires et le recouvrement des communications par du calcul. L'auteur illustre l'utilisation de cette bibliothèque sur deux exemples : le calcul d'une Transformée de Fourier mono et bi-dimensionnelles. Cependant ces routines ne constituent que des outils et le travail algorithmique reste à la charge du programmeur.

Une amélioration, au niveau facilité d'expression des algorihtmes avec pipeline, est apportée par les LOCCS (*Low Overhead Communication and Computations Subroutines*) [26, 47, 49]. Cette bibliothèque, spécifiée par Desprez et Tourancheau, fournit un ensemble de routines (8 au total) qui mixent communication et calcul. Le principe utilisé pour recouvrir les communications est basé (comme son nom l'indique) sur du pipeline à gros grain. Ces routines correspondent aux principaux schémas de communication que l'on trouve dans les algorithmes numériques parallèles et dispensent l'utilisateur d'exprimer lui-même le pipeline de son algorithme. Cependant le calcul du grain reste à sa charge, et la mise en place des paramètres à passer aux routines s'avère quelquefois délicat lorsque le schéma de communication est complexe, comme c'est le cas dans les algorithmes parallèles de transformée de Fourier. L'utilisateur est donc obligé, à un moment ou un autre, d'écrire l'algorithme pipeline afin de déterminer le « grain optimal » du pipeline.

Un inconvénient majeur à ce type de bibliothèque provient du fait qu'elles sont basées sur des routines de communications, et donc limitées dans leur possibilité d'expression. De plus les LOCCS ne fournissent pas de calcul de grain et ne permettent le recouvrement que grâce à des techniques de macro-pipeline. Elles ont, en revanche l'avantage, de pouvoir exprimer très simplement certains types d'algorithmes [26] rendant ainsi l'écriture plus simple et le code plus lisible.

1.4 L'approche proposée

L'enjeu, nous semble-t-il, est clair : il faut proposer aux utilisateurs des moyens simples et efficaces de profiter de ces mécanismes et ainsi améliorer de façon significative l'efficacité de leurs applications parallèles, sans que cela nécessite un investissement personnel trop important. La solution idéale est sans aucun doute une « approche compilateur ». Seulement, comme nous avons pu le voir, cette approche, bien que prometteuse, est encore beaucoup trop limitée actuellement. La solution des bibliothèques mixant calculs et communications est une approche séduisante mais réservée à un programmeur non néophyte en informatique parallèle. Le problème du recouvrement des communications est complexe car il fait intervenir des techniques algorithmiques assez lourdes à mettre en œuvre et il nécessite une modélisation précise de la machine pour que l'implémentation puisse être efficace.

Notre but est de proposer un ensemble de techniques générales permettant de recouvrir les communications. Celles-ci sont applicables à de nombreux algorithmes numériques, afin d'améliorer l'efficacité de leur parallélisation. Elles peuvent servir ensuite de base à la conception d'heuristique à intégrer dans un compilateur. Ces techniques sont une généralisation des différents cas que nous avons pu remarquer dans les algorithmes numériques que nous avons étudiés; elles sont basées, bien sûr sur **des techniques de pipeline**, mais également **de regroupement ainsi que de ré-ordonnancement des calculs**. Il nous semble possible d'**exprimer la granularité des calculs à effectuer permettant un recouvrement maximal, en fonction de coûts élémentaires des tâches de calcul et des coûts de communication**. Ceci nous permet alors d'implémenter des algorithmes numériques parallèles efficaces avec un masquage maximal du temps de communication pour peu que l'on soit capable de modéliser correctement le comportement de la machine cible.

Il nous semble donc, à l'heure actuelle, que l'intérêt pour les utilisateurs est de pouvoir disposer d'un ensemble de routines numériques efficaces dont ils peuvent se servir lors de la parallélisation de leurs applications. Ces routines doivent être paramétrées en fonction des coûts de calcul et de communication de base de la machine utilisée. Ces paramètres sont soit fournis par le constructeur, soit déterminés par une série de jeux d'essais destinés à modéliser le comportement de la machine.

1.4.1 Schémas algorithmiques généraux

Nous nous plaçons dans un modèle de programmation où le parallélisme est guidé par la distribution des données, de type SPMD (voir partie I chapitre 1). Plusieurs raisons à ce choix : en premier lieu, ce modèle est le plus utilisé actuellement dans le monde du calcul parallèle scientifique. De plus, si nous nous plaçons dans l'optique de proposer des routines de calculs efficaces et prêtes à l'emploi, la distribution initiale des données nous est imposée. Sous ces hypothèses les noyaux numériques peuvent s'exprimer sous forme de schémas de dépendance entre les tâches de calcul et les communications, le placement des tâches sur les processeurs étant *a priori* décidé par la distribution initiale des données.

Cette première approche est, bien évidemment, destinée à être étendue à d'autres formes de parallélisme, en particulier au parallélisme de contrôle. Cependant, il nous a semblé, dans un premier temps, utile de nous fixer quelques contraintes qui permettent de faciliter le problème. Nous allons présenter dans la suite trois de ces schémas que l'on retrouve dans la plupart des algorithmes numériques. Le but n'étant pas de résumer en trois schémas tous les cas possibles, mais plutôt les techniques utilisables pour effectuer du recouvrement.

Schéma avec distribution intermédiaire de données

Ce premier schéma correspond au cas où les processeurs doivent s'échanger des résultats intermédiaires avant de commencer une nouvelle phase de calcul. Ce schéma correspond aux algorithmes avec redistribution intermédiaire des données, comme dans la FFT bi-dimensionnelle [28, 36] ou encore dans la parallélisation de l'algorithme de transformée spectrale [57]. Ces noyaux numériques sont importants en traîtement du signal et d'images, mais également pour le calcul des solutions des équations de Poisson etc. Ce schéma se

retrouve également dans tous les algorithmes avec diffusion partielle ou totale d'un résultat intermédiaire comme dans certains algorithmes parallèles de résolution de systèmes linéaires [48].

Si l'une au moins des deux phases de calcul peut être découpée en tâches indépendantes de granularité plus faible, alors chacun des résultats partiels de ces tâches peut être envoyé dès qu'il a été calculé. Donc l'échange d'un résultat partiel est masqué par le calcul du résultat suivant. Le masquage de la communication est réalisé par une technique de pipeline classique (voir figure 1.3).

Le cas où seulement la première phase de calcul peut être découpée correspond à une synchronisation avant le début d'une nouvelle phase de calcul. Alors, le masquage du temps de communication n'est pas total (schéma de droite sur la figure 1.3).

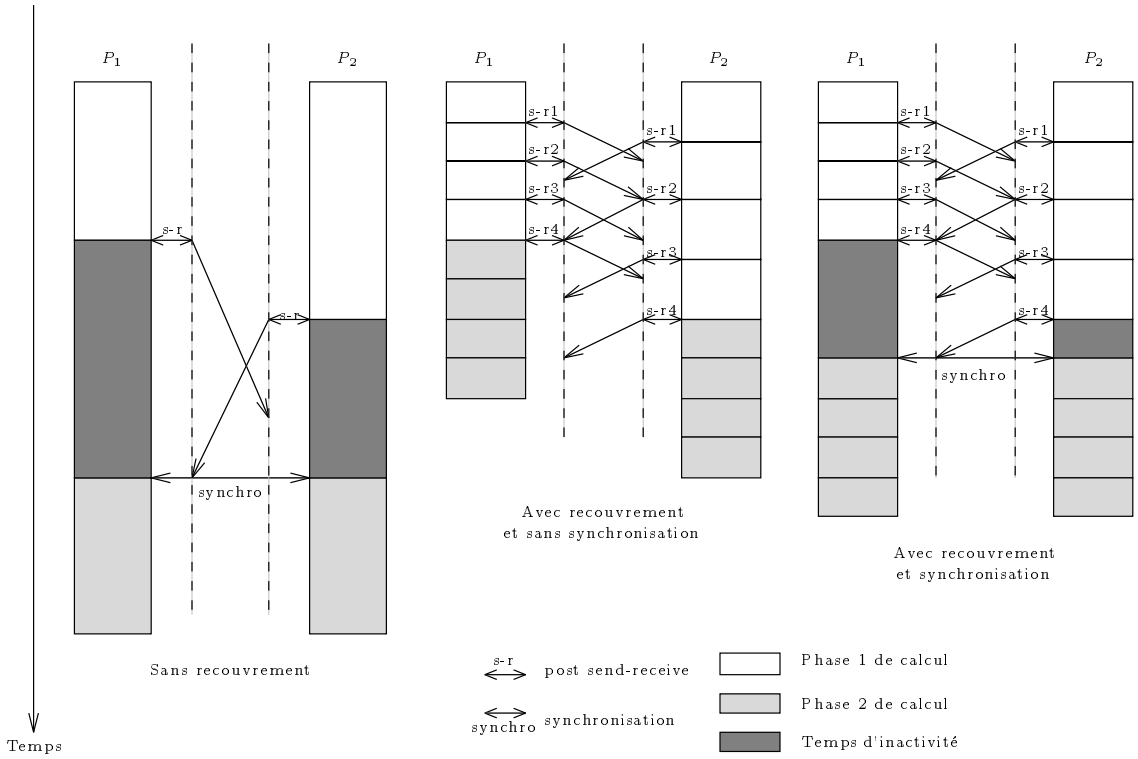


FIG. 1.3 - *Schéma avec distribution intermédiaire de données.*

Le calcul du grain de découpage est simple, il doit être tel que le temps de calcul d'une tâche doit être au moins égal au temps de communication du résultat partiel précédent. Il faut remarquer qu'à l'inverse d'un pipeline classique, tel qu'on le trouve par exemple dans les opérateurs pipelines des processeurs, la granularité du découpage n'est pas limitée par le découpage possible de la phase de calcul, mais il est nécessaire de prendre en compte les temps d'initialisations (*start-up*) induits par chaque communication. Ceci implique une granularité moins fine de découpage: c'est ce que l'on appelle du **pipeline à gros grain** [103]. Une illustration de la différence et de l'influence du grain est donnée dans la figure 1.4.

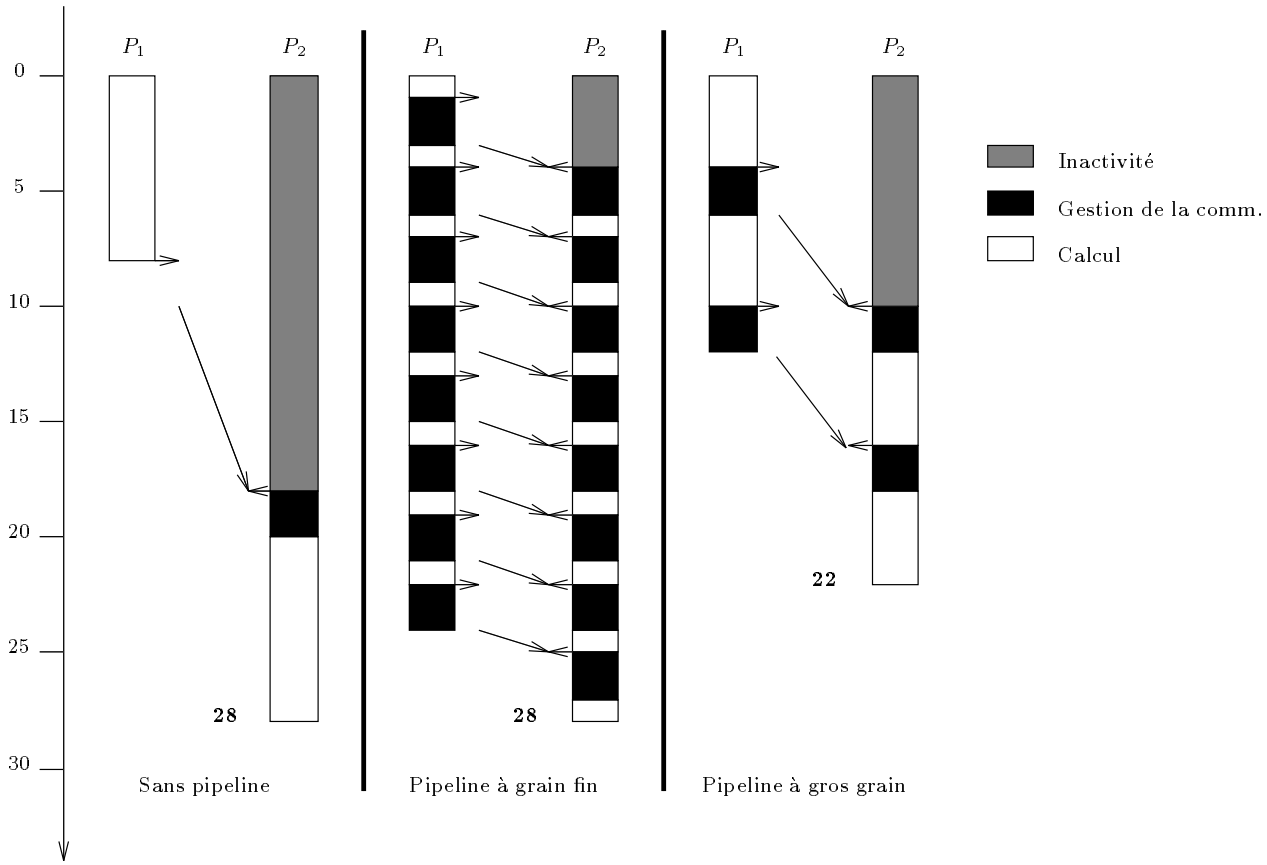


FIG. 1.4 - *Pipeline à grain fin versus pipeline à gros grain.*

Dans cet exemple, nous supposons que le temps de gestion de la communication (en envoi et en réception) est grand par rapport au volume de communication à effectuer. Si le grain du pipeline est trop fin, le sur-coût rajouté par les temps de *start-up* conduit à une inefficacité du pipeline, bien que le temps d'inactivité du processeur P_2 ait largement diminué. Il faut donc, lors du calcul du grain, prendre en compte ce facteur et raisonner en temps total de « non-calcul » du processeur, plutôt qu'en temps d'inactivité. C'est ce temps de « non-calcul » que nous appelons **sur-coût de communication**.

Il faut remarquer que ces techniques ne sont pas nécessairement applicables sans modifications. En effet, il peut être intéressant de s'occuper de l'ordonnancement local des tâches sur chacun des processeurs afin de pouvoir masquer au mieux le sur-coût des communications. La figure 1.5 en est une illustration.

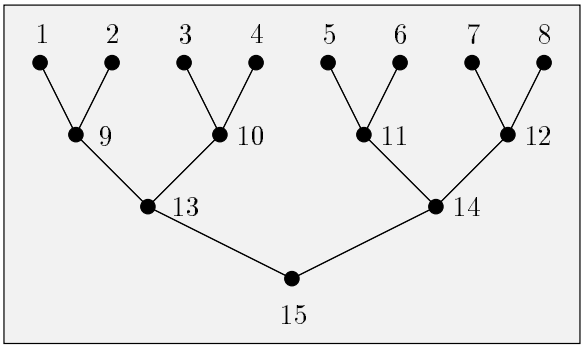


Schéma 1 : Graphe de précédence (orientation vers le bas)

P_0 : {1,5,9,13,15}
P_1 : {2,6,11}
P_2 : { 3,7,10}
P_3 : {4,8,12,14}

Schéma 2 : Placement des tâches

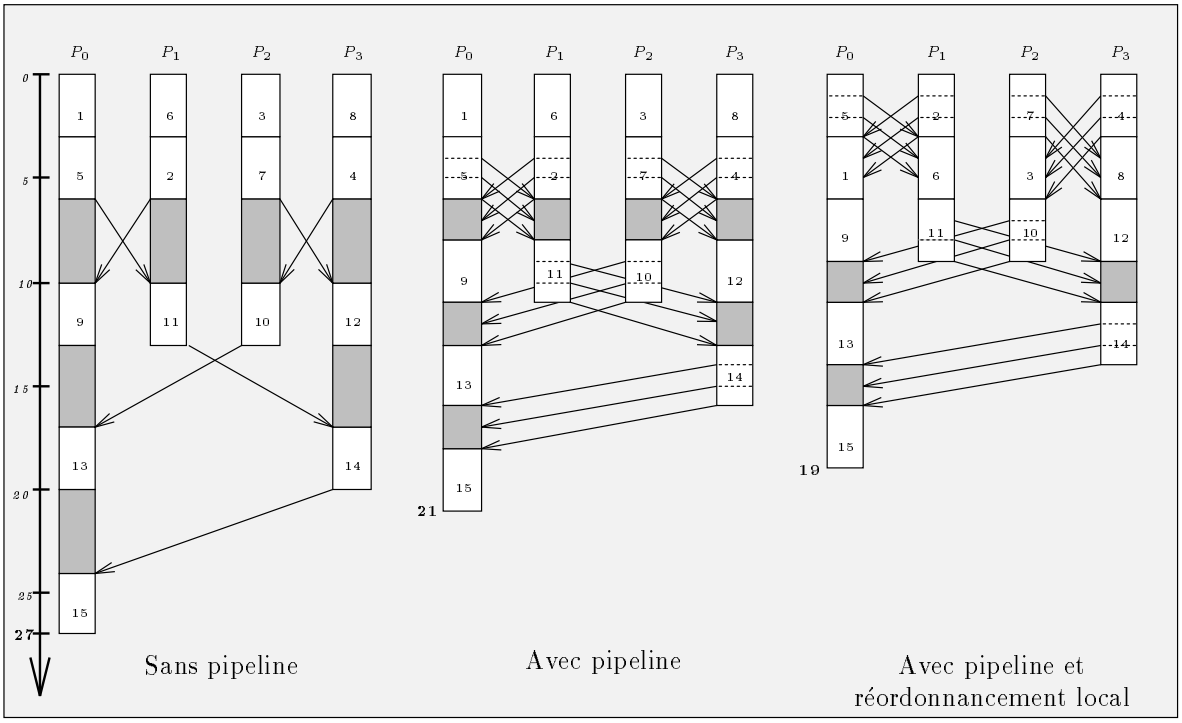


Schéma 3 : Ordonnancement et schéma d'exécution sur 4 processeurs

FIG. 1.5 - *Illustration de l'intérêt du ré-ordonnancement local des tâches.*

Considérons un graphe de dépendance logique sous forme d'arbre binaire complet composé de 15 tâches toutes identiques[1] (nous sommes dans un modèle SPMD) (voir schéma 1, figure 1.5). La distribution des données étant fixée, le placement des tâches sur les processeurs nous est imposé. Nous considérons donc le placement illustré par le schéma 2 de la figure 1.5. Nous nous fixons un ordonnancement local des tâches sur chaque processeur (premier schéma d'exécution du schéma 3). En appliquant une technique de pipeline classique nous pouvons masquer une partie des communications (deuxième schéma d'exécution).

1. Ce type de graphe de dépendance correspond aux algorithmes d'évaluation d'expression arithmétique, ou encore aux algorithmes de tri.

Cependant, il est utile de s'intéresser à l'ordonnancement local des tâches sur les processeurs. En effet le simple échange des premières et secondes tâches calculées sur chacun des processeurs permet d'optimiser le masquage des communications (troisième schéma d'exécution). On peut remarquer, dans cet exemple, que le ré-ordonnancement des calculs suffit à masquer les premières communications, le pipeline n'apportant un gain que pour les communications suivantes.

Notons que ces techniques de pipelines sont en partie intégrées dans un compilateur, cependant le ré-ordonnancement local de tâches semble être un travail plus difficile.

Schéma à phases répétées

Ce schéma correspond à la répétition de phases identiques correspondant à une succession de tâches de calculs avec échanges de résultats partiels. Ce schéma se retrouve, par exemple, dans les algorithmes de traitement d'images [44, 92] ou dans le calcul d'un ensemble de FFT mono-dimensionnelles [22, 28].

L'idée de base pour masquer les communications est **l'enchevêtrement** des tâches des phases suivantes avec les tâches des phases précédentes. Nous présentons sur la figure 1.6 un exemple de schéma à phases répétées.

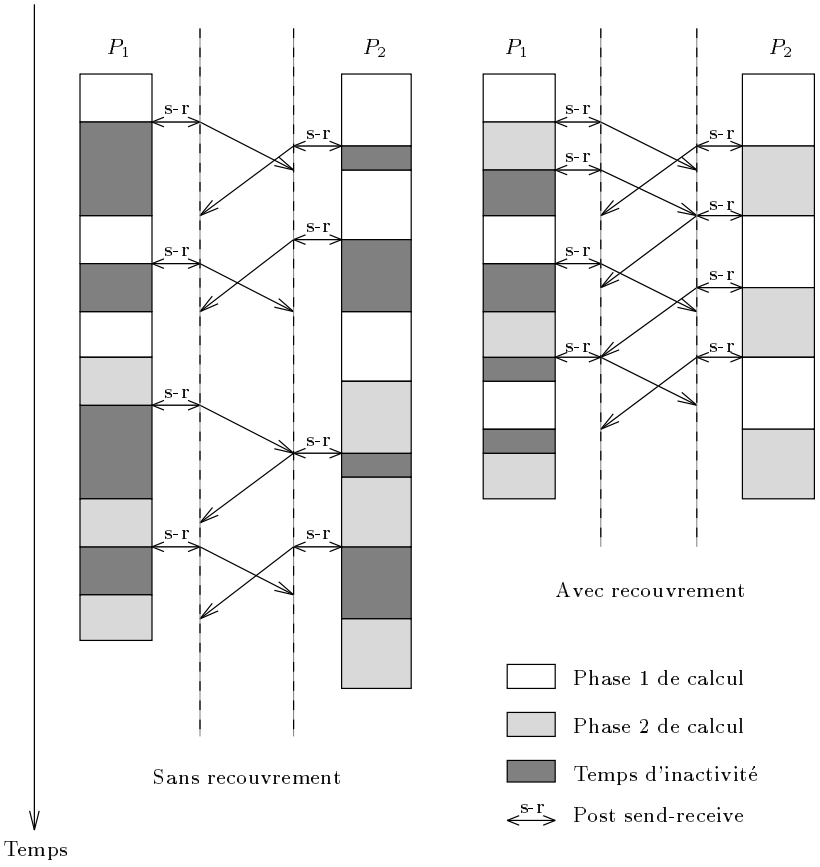


FIG. 1.6 - *Schéma à phases répétées.*

Dans ce cas, il nous est possible de calculer le nombre de tâches de calcul à entremêler afin de recouvrir au maximum le temps de communication, mais on peut envisager une exécution plus dynamique des choses. En effet, si l'on utilise des primitives de réception non-bloquantes, alors le processeur qui a fini une tâche de calcul ne restera pas en attente du résultat, mais pourra commencer un autre calcul (relatif à une autre phase) qui ne nécessite pas de résultat intermédiaire.

Remarquons que le recouvrement peut, dans ce cas, se faire de manière automatique à l'aide de mécanisme de *multi-threading*. En effet, si on reprend le schéma d'exécution de la figure 1.6, il est possible d'initier un certain nombre de *threads*, chacun étant chargé de calculer une phase. Lorsqu'une tâche est en attente d'une communication, alors le *thread* est désactivé et un autre *thread* commence une nouvelle tâche, etc. Un exemple est donné dans la figure 1.7 sur deux processeurs avec 3 *threads*.

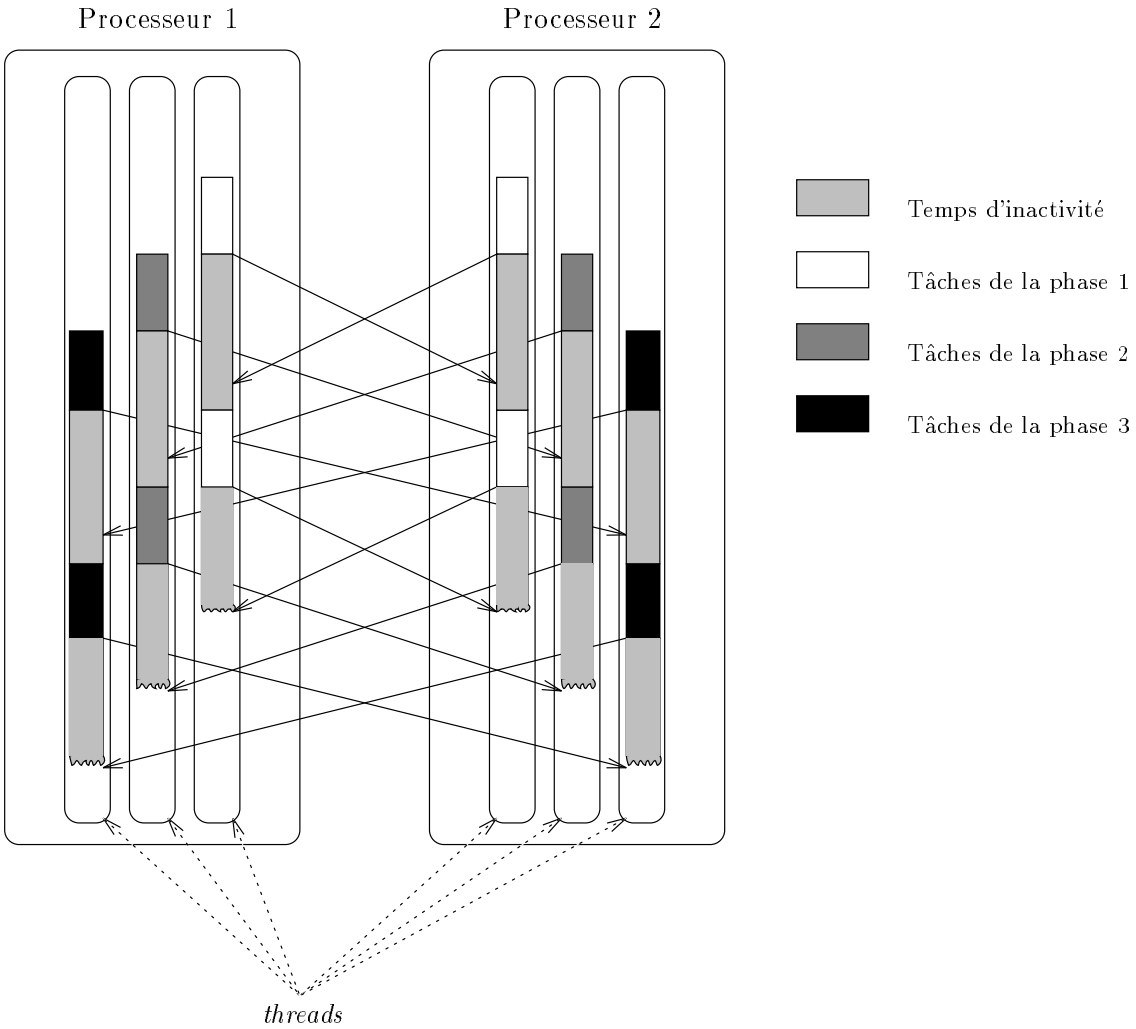


FIG. 1.7 - *Schéma à phases répétées avec multi-threading.*

Schéma à dépendances fortes

Ce dernier schéma est celui qui pose le plus de problèmes pour masquer les communications. Nous considérons ici une succession de tâches « unitaires », c'est à dire indivisibles ou à granularité très fine, qui ont besoin de résultats partiels précédents avant de pouvoir s'exécuter. Dans ce cas le seul moyen d'effectuer du recouvrement est d'anticiper la communication lorsque cela est possible. Par exemple, dans le cas où les processeurs ne s'échangent pas des résultats intermédiaires issus du calcul qui est en train de se faire. Nous donnons un exemple d'un tel schéma sur la figure 1.8.

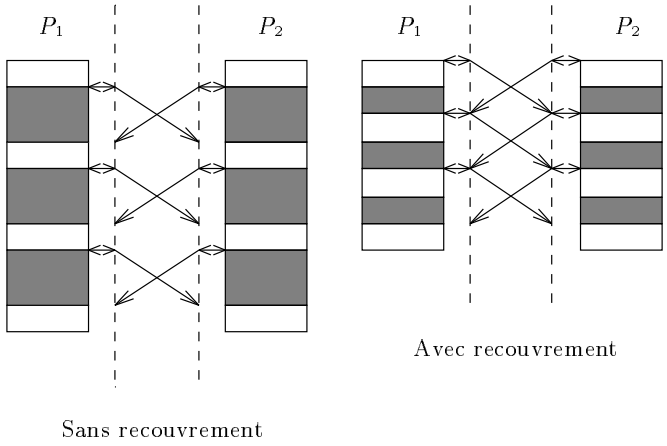


FIG. 1.8 - *Schéma à dépendances fortes.*

Remarquons que ce schéma peut être vu comme une dégénérescence du premier. En effet, il correspond à un schéma pipeline avec une granularité minimale, alors il peut être intéressant de ré-augmenter la granularité du pipeline (voir figure 1.4), soit à un schéma avec redistribution intermédiaire, mais avec des tâches unitaires, et sur lesquelles il est impossible, ou inefficace, d'appliquer des techniques de pipeline.

1.5 Conclusion

Nous avons vu dans ce chapitre quelques techniques permettant de minimiser le surcoût des communications. Nous nous sommes attardés plus particulièrement sur celles qui permettent de masquer les communications. Bien que les compilateurs de langage parallèle aient progressé, les possibilités qu'ils offrent sont encore limitées. Il est encore nécessaire d'implémenter « à la main » des algorithmes optimisés afin d'obtenir une parallélisation efficace.

Cependant, il est possible d'utiliser quelques techniques générales, comme celles que nous avons présentées, et qui offrent l'avantage d'être paramétrables en fonction des coûts de calcul et de communication, permettant ainsi d'obtenir des algorithmes parallèles efficaces qui masquent au mieux les communications. Nous avons également donné quelles étaient les possibilités d'une automatisation éventuelle de ces techniques. Il est clair que cette auto-

matisation ne sera performante que si elle intègre un modèle le plus précis possible des performances de la machine cible.

Nous présentons dans le chapitre suivant, une évaluation et une modélisation des performances des communications de la machine Cray T3D. C'est le genre de modèle qu'il faut étudier et mettre en place, en vue d'une automatisation des techniques de recouvrement.

Chapitre 2

Détermination des paramètres élémentaires de communication : modélisation des temps de communication du Cray T3D

Le but de ce chapitre est de présenter des résultats de jeux de tests de communication afin d'évaluer et de modéliser les performances des communications de la machine Cray T3D MPP. Ces modèles nous permettent ensuite d'optimiser les paramètres de recouvrement au moment de l'implantation des algorithmes avec recouvrement des communications. Nous avons testé les principaux schémas de communication globale en utilisant toutes les bibliothèques d'échange de messages disponibles sur cette machine. Ces tests nous ont permis de donner un modèle d'évaluation des performances en fonction de la taille des messages envoyés et du nombre de processeurs y participant. Ce travail a été effectué avec L. Colombet [25].

2.1 Introduction

Comme nous avons pu le voir dans le chapitre précédent, il est très important d'avoir une bonne modélisation des performances des machines. La plupart des jeux d'essais (*benchmarks* en anglais) qui existent sont constitués de noyaux et d'applications numériques (comme les *benchmarks* NAS, LINPACK etc [3, 50, 109, 125]). Ils consistent à implémenter et à optimiser les différents composants du jeux d'essais sur différentes machines parallèles [50, 96]. Cette méthodologie ne permet pas de différencier la part des communications de celle des calculs et n'offre qu'une vision globale et déformée des performances qu'un utilisateur pourra obtenir avec sa propre application. En effet, ce n'est pas parce la machine X offre une performance de Y Mflops sur un test LINPACK de taille 1000, que l'utilisateur pourra obtenir des performances semblables avec son application parallèle.

Il existe quelques jeux d'essais incluant des tests de communication [1, 2], ou la comparaison des performances de communication de différentes machines parallèles [75], mais ils

sont réduits à des mesures de type échange de messages entre deux processeurs. Il n'existe, à l'heure actuelle, que très peu d'études plus poussées sur la modélisation des communications. Nous pouvons cependant citer les travaux de C. Tron sur la modélisation des performances des communications en réseau chargé [118, 119].

Nous présentons dans ce chapitre des résultats de jeux d'essais de communication point-à-point et globale sur Cray T3D. Ces schémas sont ceux qui sont les plus utilisés dans les algorithmes numériques parallèles [79]. De ces expérimentations, nous en déduisons un modèle de prédiction les temps de communication en fonction de la taille des messages et du nombre de processeurs. Nous vérifions également que les modèles proposés sont proches de la réalité.

La suite de ce chapitre est organisée ainsi : dans un premier temps nous déataillons les caratéristiques de communication de l'architecture du Cray T3D et sur les différentes bibliothèques de communication par échange de messages disponibles. Nous nous intéressons plus particulièrement aux principales différences dans les implémentations des primitives de communications point-à-point. La section suivante est dédiée aux résultats des expérimentations et à la présentation des modèles.

2.2 La machine Cray T3D

Le Cray T3D est la première machine massivement parallèle de Cray Research Inc. Cette machine peut contenir jusqu'à 2048 processeurs, chacun possédant sa propre mémoire locale.

Le Cray T3D possède quatre types de composants : les nœuds de calcul, le réseau d'interconnexion, les portes d'entrées/sorties et une horloge globale. Nous ne détaillons que les deux premiers composants [40], car ce sont ceux qui nous intéressent plus particulièrement pour évaluer les performances des communications.

2.2.1 Les nœuds de calcul et le réseau d'interconnexion

Les éléments de calcul

Chaque élément de calcul (*Processing Element* en anglais, noté PE), contient un microprocesseur et une mémoire locale. Un nœud de calcul est constitué de deux PE, d'un support de communication composé d'une interface réseau *Network Interface*) et d'une unité de transfert (*Block Transfer Engine*: BLT) (voir la figure 2.1). Le microprocesseur est de type RISC 64-bit développé par Digital Equipment Corporation, sa fréquence d'horloge est de 150 Mhz. La mémoire locale est de type DRAM (*Dynamic Random Access Memory*) et d'une taille de 64 Moctets. Cette mémoire locale fait partie de la mémoire totale physiquement distribuée, mais elle est considérée comme logiquement partagée, du fait que le microprocesseur d'un PE peut accéder à la mémoire d'un autre PE sans interrompre le microprocesseur du PE accédé[1]. Les deux PE d'un nœud de calcul sont identiques, mais fonctionnent de façon indé-

1. Cette remarque conceptuelle, issue d'un choix architectural, revêt toute son importance lorsqu'on désire évaluer les performances des communications, et mieux encore, lorsqu'on s'intéresse aux recouvrements des communications.

pendante. Les accès à l'interface du réseau et à l'unité de transfert se font, eux, de manière concurrente. C'est le BLT qui s'occupe de calculer les adresses, ainsi que de lire et écrire les données dans les mémoires locales, et ceci sans interrompre le processeur de calcul.

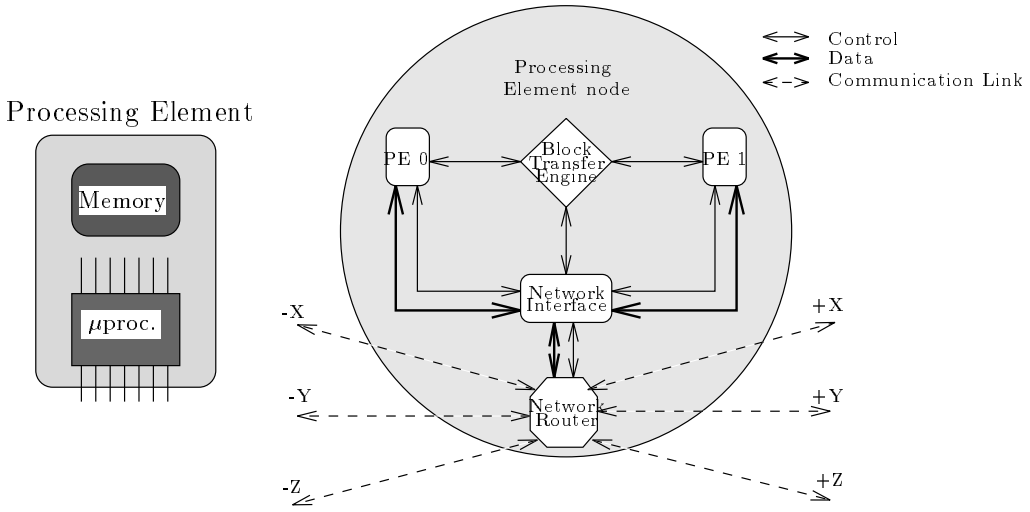


FIG. 2.1 - *Nœud de calcul du Cray T3D*

Le réseau d'interconnexion

Le réseau d'interconnexion est un réseau direct avec une topologie de tore tri-dimensionnel (voir la figure 2.2).

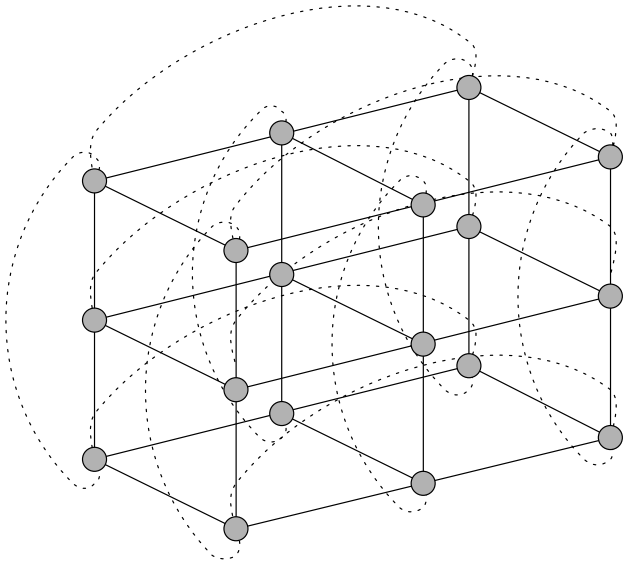


FIG. 2.2 - *Tore 3D de 18 nœuds*

Les sommets du graphe représentant le réseau d'interconnexion sont composés par les routeurs de chaque PE, et les arêtes sont les liens de communication qui relient les sommets

dans les trois dimensions : X, Y et Z (voir la figure 2.1). Les liens sont bi-directionnels et *full-duplex* avec un débit maximal de 150 Moctets/s dans une direction. Le mode de commutation du réseau est de type *wormhole* avec une fonction de routage classique par « ordre des dimensions » (*dimension order routing*) [102]. Cet algorithme de routage est le suivant : lorsqu'une information quitte un nœud, il voyage à travers le réseau en prenant la dimension X, puis Y et enfin la dimension Z. Ce type de routage est déterministe du fait qu'un message est arrivé à destination lorsqu'il a fini son transfert suivant la dimension Z (*c.f.* figure 2.3).

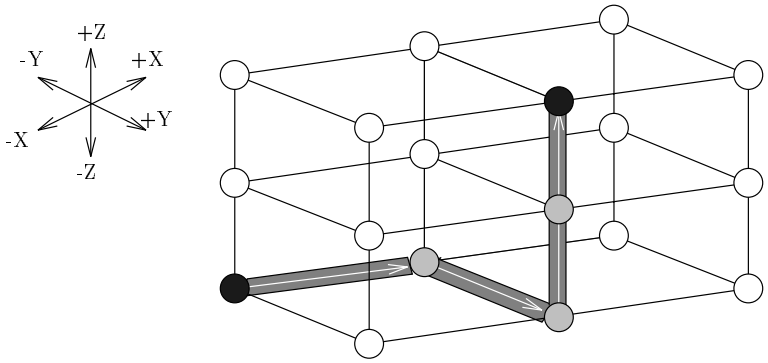


FIG. 2.3 - *Routage par ordre de dimension.*

2.2.2 Les bibliothèques de communication

Nous pouvons distinguer deux niveaux de bibliothèques de communication : un niveau bas constitué par la bibliothèque **ShMem** (*Shared Memory*) [42] et un niveau haut qui est une implémentation spécifique de PVM version 3.3.4. [11].

Les opérateurs de base de la bibliothèque ShMem sont les primitives `put` et `get` qui permettent d'écrire et de lire des données situées dans une mémoire distante. PVM a été implémentée sur cette couche de bas niveau [41]. Le modèle utilisé par PVM est bien sûr le modèle de communication par tampon, ce qui implique un contrôle de flux transparent (voir la partie I chapitre 2). Ce n'est pas le cas lorsqu'on utilise directement les primitives `put` et `get`. De plus la cohérence des données est à la charge de l'utilisateur. Nous pouvons distinguer deux façons différentes d'envoyer un message par PVM :

- `pvm_send` : c'est la primitive classique d'envoi de messages en communication point-à-point avec PVM. Deux variations sont possibles dans la façon d'emballer les données avant l'envoi : soit tout le message est emballé (option `PvmDataRaw`), soit seulement l'adresse et la taille du messages (option `PvmDataInPlace`)[2].

- `pvm_psend` : dans ce cas le message n'est pas emballé avant d'être envoyé[2].

Dans les deux cas le principe de l'implémentation est le suivant : si un nœud A envoie un message de taille L à un nœud B, alors une première requête est envoyée vers B *via* un petit

2. Il est évident que dans ce cas les données à envoyer doivent être contiguës en mémoire.

message de taille PVM_DATA_MAX. Dans ce premier message se trouvent l'adresse et la taille des données qui doivent être envoyées. Si $L \leq$PVM_DATA_MAX alors les données sont envoyées avec la première requête. Sinon, lorsque B reçoit la requête, il la déballe et va récupérer les données distantes à l'aide d'une lecture à distance (*remote load*) (voir figure 2.4).

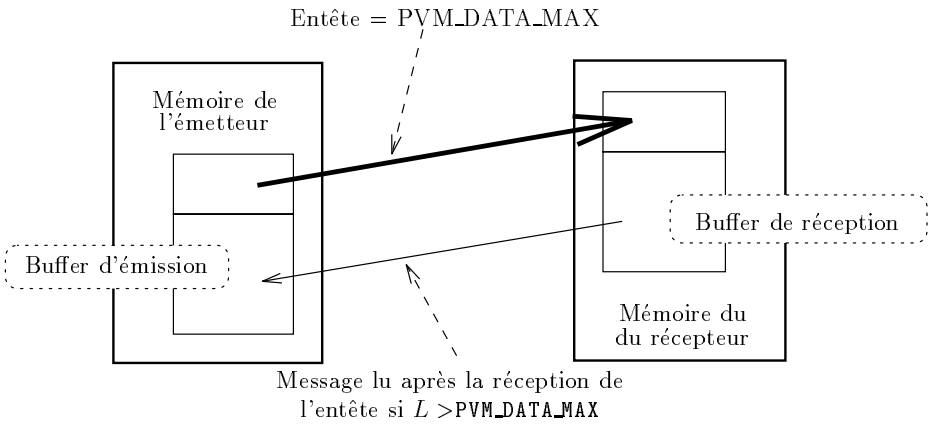


FIG. 2.4 - *Principe de l'envoi d'un message de taille L avec PVM sur T3D.*

Cray Research a également implémenté une extension à PVM : les pvm_channels. Ils permettent des envois directs émetteur/récepteur (*remote stores*) en établissant un « canal » de communication entre les deux mémoires. Il y a deux avantages à l'utilisation de ces canaux : le premier est l'utilisation d'écriture distante (*remote store*) plutôt que de lecture distante (*remote load*), ce qui est plus rapide. Le second est la suppression de copie mémoire ou contrôle qui est entraînée par l'utilisation des primitives classiques. Ce canal est orienté, et il est établi une fois pour toute entre deux adresses précises de la mémoire de l'émetteur vers la mémoire du récepteur (voir la figure 2.5). L'envoi est non-bloquant et la réception est bloquante. L'utilisation de ces canaux est très intéressante lorsqu'on met en œuvre un schéma de communication identique de façon intensive, car l'établissement du canal est assez coûteux.

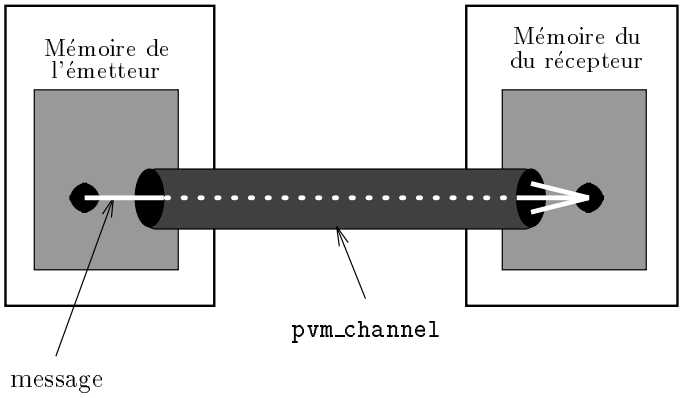


FIG. 2.5 - *Les canaux de communication PVM.*

2.3 Évaluation et modélisation des temps de communication

2.3.1 Présentation du protocole expérimental

Les différents schémas de communication évalués sont les suivants :

1. L'échange entre deux processeurs (*One-To-One*, OTO)
2. La diffusion (*One-To-All*, OTA)
3. L'échange total (*All-To-All*, ATA)
4. La multi-distribution (*Personalized-All-To-All*, PATA)

Tous ces schémas ont été présentés en détail dans le chapitre 1 de la partie II. Le modèle de temps utilisé est celui classiquement adopté pour modéliser le temps d'une communication en mode de commutation *wormhole* [96, 102]. C'est-à-dire que le temps d'envoi d'un message de taille L entre deux processeurs situés à distance d est : $T = \beta + d\delta + L\tau$ où β est le temps d'initialisation de la communication, δ le temps de commutation des commutateurs (*switchs*) intermédiaires et τ le taux de transmission du lien. Comme c'est le cas dans la plupart des machines actuelles utilisant un mode de commutation *wormhole*, et comme nous avons pu le vérifier, $\delta << \beta$. Nous pouvons donc supposer que le temps de communication est indépendant de la distance. Nous utilisons par conséquent un modèle de temps plus simple qui est le suivant : $T = \beta + L\tau$.

Tous les tests ont été exécutés 500 fois et moyennés. Nous avons remarqué, de façon expérimentale, que la variabilité des mesures est très faible ($< 2\%$), il nous semble donc que la moyenne sur 500 mesures est largement suffisante. Afin de valider les modèles par rapport aux expérimentations effectuées, nous avons choisie pour exprimer l'erreur relative, le rapport entre l'écart type empirique et la moyenne empirique. Cette erreur, donnée en pourcentage (notée e) définie comme suit : nous notons $T_m(i)$ le ième temps mesuré[3] et $T_s(i)$ le ième temps simulé à partir du modèle, $i = 0...n-1$. Alors :

$$e = \frac{\sqrt{\sum_{i=0}^{n-1} \left(T_m(i) - T_s(i)\right)^2}}{\sum_{i=0}^{n-1} T_m(i)} \times 100$$

Dans la suite nous notons par $T^{schéma}_{primitive}$ le temps pour effectuer le schéma de communication *schéma* en utilisant la primitive *primitive*. Par exemple, nous noterons T^{OTA}_{bcast} le temps pour diffuser un message en utilisant la primitive `pvm_bcast` [11].

3. $T_m(i)$ est en fait la valeur moyennée du temps de communication pour un message de taille i sur 500 expériences. Il faut noter que la variabilité des mesures est très faible ($< 2\%$), ce qui justifie le fait que cette erreur n'est pas reportée sur les graphiques.

2.3.2 *One-To-One*

Le but de ce test est de déterminer les deux paramètres du modèle de temps donné ci-dessus, à savoir le temps d'initialisation (*start-up*) β et le taux de transmission τ. Nous avons utilisé les différentes primitives d'envoi et de réception disponibles :

1. `pvm_send` avec l'option d'emballage `PvmDataRaw`.
2. `pvm_send` avec l'option d'emballage `PvmDataInPlace`.
3. `pvm_psend`.
4. `shmem_put`, qui correspond à une écriture distante (*remote store*).
5. `shmem_get`, qui correspond à une lecture distante (*remote load*).
6. `pvm_channel`.

Nous reportons dans les figures 2.6, 2.7 et 2.8 les différents résultats d'expérimentation.

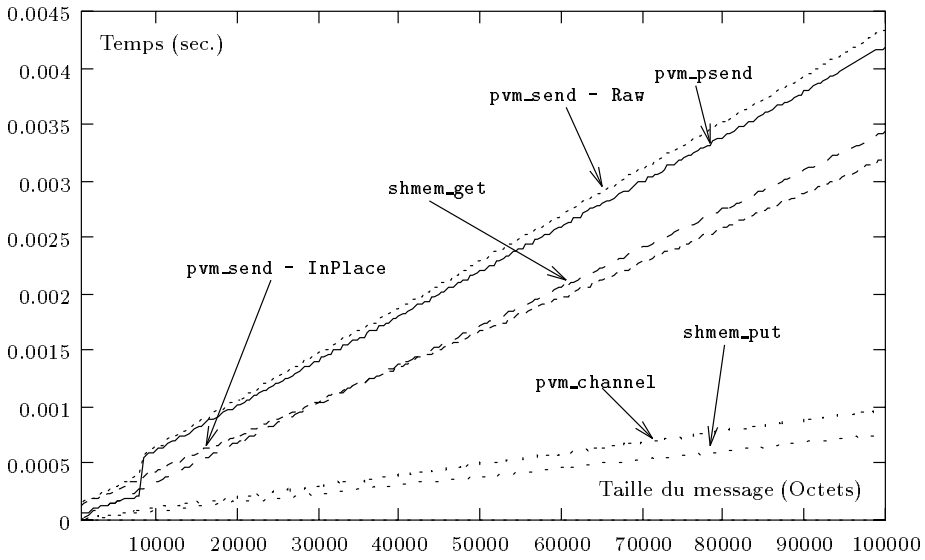


FIG. 2.6 - *Comparaison des temps du test One-To-One.*

Nous pouvons observer, pour des tailles importantes de message, trois groupes principaux : le premier constitué des primitives `shmem_put` et `pvm_channel`, réalise les meilleures performances. Le deuxième est composé des fonctions `shmem_get` et `pvm_send - InPlace` et le dernier, avec les primitives `pvm_psend` and `pvm_send - DataRaw`, réalisent les moins bonnes performances. Remarquons également la très grande régularité du comportement, ce qui renforce l'idée d'une modélisation par une fonction affine du temps.

En revanche, nous observons un comportement différent pour de petites tailles de messages (inférieures à `PVM_DATA_MAX` qui est dans ce test égal à 8192 octets). Nous nous intéressons de plus près à cette zone dans la figure 2.7.

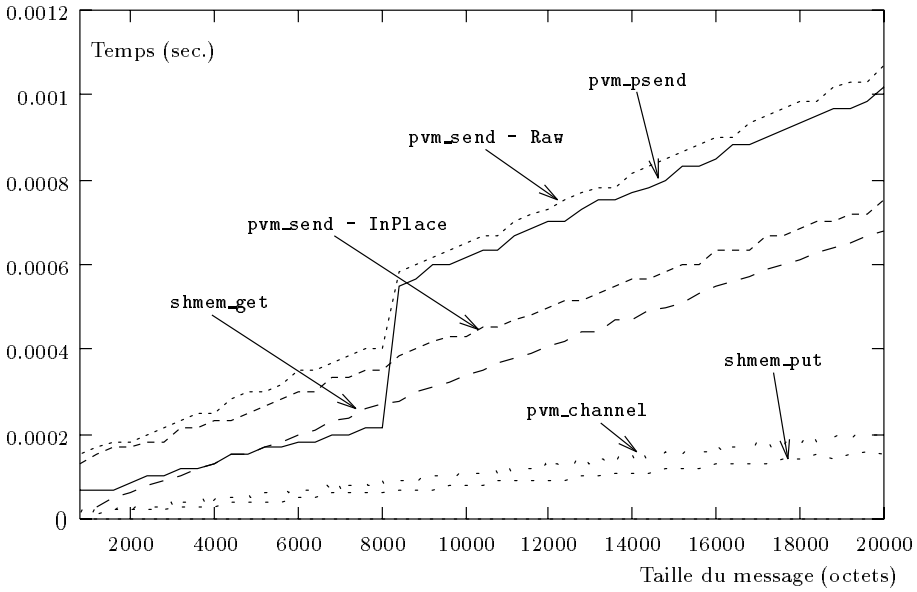


FIG. 2.7 - *Zoom pour des petits messages.*

Comme nous l'avions expliqué précédemment, les primitives `pvm_psend` et `pvm_send - DataRaw` utilisent des lectures à distance pour des messages dont la taille dépasse `PVM_DATA_MAX`. Ceci explique les sauts de la figure 2.7. Dans le cas où les messages sont de taille inférieure, nous pouvons remarquer l'amélioration notable des performances de la fonction `pvm_psend`. Les temps d'échange de messages avec les `pvm_channel` sont toujours excellents que les messages soient petits ou grands, ce qui confirme l'intérêt de l'utilisation d'écritures distantes à la place de lectures distantes.

Nous présentons dans la figure 2.8 les résultats des bandes passantes.

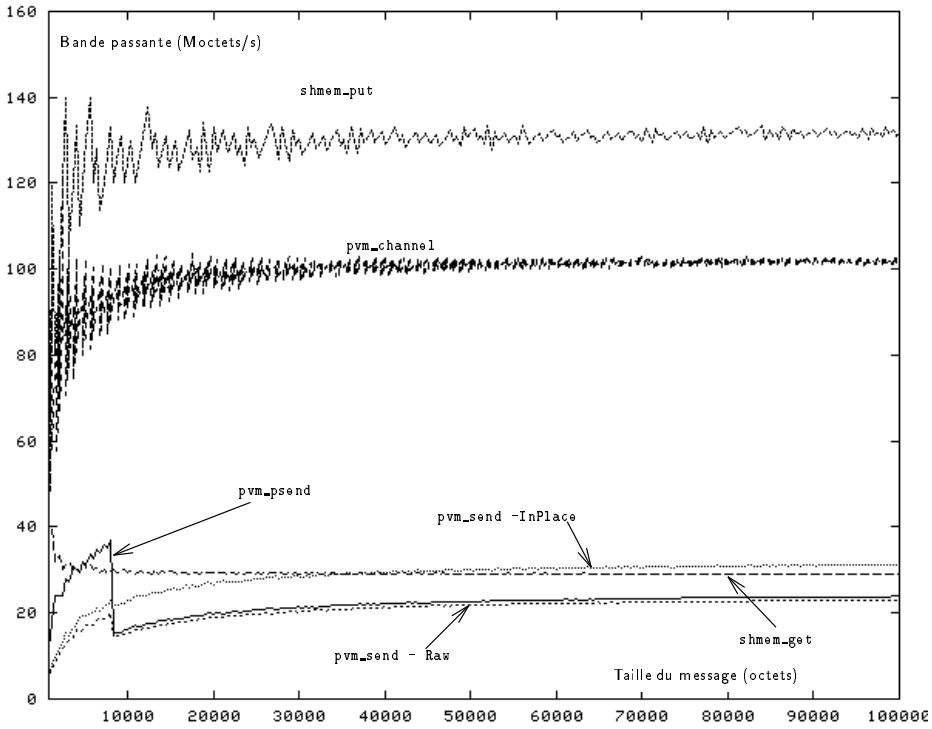


FIG. 2.8 - *Comparaison des bandes passantes obtenues.*

Nous résumons dans la table 2.1 les valeurs des différents paramètres en fonction des routines et de la taille des messages. La dernière colonne du tableau indique l'erreur du modèle par rapport à la réalité.

	$L \leq$ PVM_DATA_MAX		$L >$ PVM_DATA_MAX		Bande passante max.	e
	β (μs)	τ ($\mu s/octets$)	β (μs)	τ ($\mu s/octets$)	$Moctets/s$	
`pvm_send - DataRaw`	119	0.0351	239	0.0409	20 - 23	0.34%
`pvm_send - DataInPlace`	104	0.0309	104	0.0309	31	0.94%
`pvm_psend`	23.3	0.0242	217	0.04	37 - 24	0.9%
`shmem_put`	8.57	0.0076	8.57	0.0076	140 - 132	1.5%
`shmem_get`	1.72	0.0345	1.72	0.0345	29	0.55%
`pvm_channel`	10.23	0.0097	10.23	0.0097	102	0.85%

TAB. 2.1 - *Valeurs des paramètres en fonction des routines utilisées.*

Nous avons également vérifié l'insensibilité du temps d'échange d'un message en fonction de la distance, sauf dans le cas où l'échange s'effectue entre deux processeurs se trouvant sur le même nœud de calcul. Dans ce cas, le modèle change et doit tenir compte des bandes passantes mémoire ainsi que du rangement interne des données.

2.3.3 *One-To-All*

Nous avons utilisé trois types d'implémentations de la diffusion. La première, la plus naïve, consiste à effectuer $P-1$ envois *via* la fonction `pvm_psend`. Les deux autres versions utilisent les primitives `pvm_mcast` et `pvm_bcast`. Nous comparons les résultats obtenus sur la figure 2.9 sur 32 processeurs. Comme on peut le remarquer, l'utilisation de la fonction `pvm_psend` est la plus efficace pour des petits messages, simplement parce que le coût d'initialisation est bien inférieur. En revanche pour des messages dont la taille est supérieure à `PVM_DATA_MAX`, il n'y a plus d'intérêt à utiliser cette primitive, pour les mêmes raisons que précédemment.

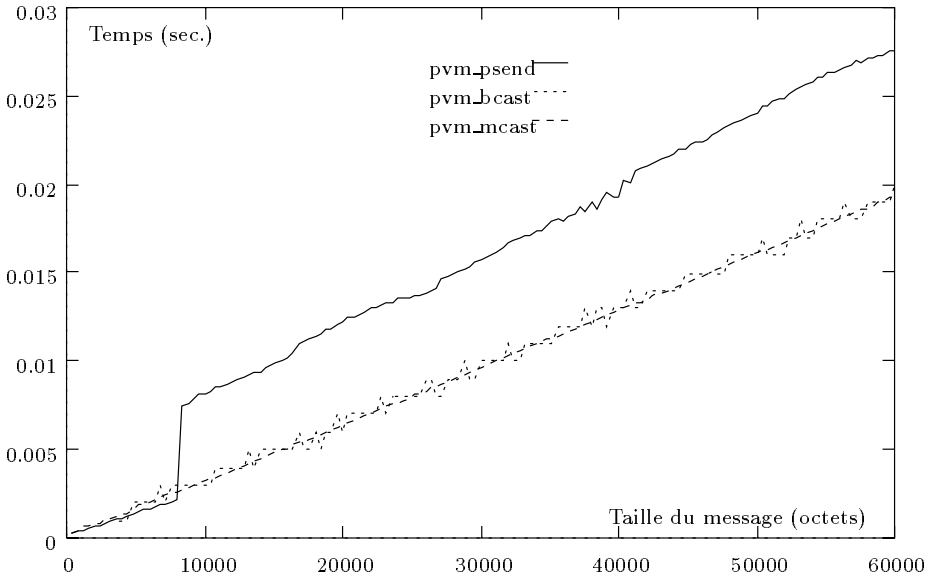


FIG. 2.9 - *Comparaison des temps du One-To-All sur 32 PEs*

À partir de ces expérimentations nous avons déduit les modèles de temps suivants en fonction de la taille des messages, du nombre de processeurs et de la primitive utilisée. Nous indiquons également l'erreur du modèle.

$$T_{OTA}^{psend} = (P-1) \times \left(\beta_{OTA}^{psend} + L\tau_{OTA}^{psend}\right)$$
$e = 1.73\%$

avec
$$\begin{cases} \beta_{OTA}^{psend} = 2.96\mu s \;\; if \;\; L \leq PVM\_DATA\_MAX \\ \beta_{OTA}^{psend} = 123\mu s \;\; if \;\; L > PVM\_DATA\_MAX \\ \tau_{OTA}^{psend} = 0.009\mu s/octet \;\; if \;\; L \leq PVM\_DATA\_MAX \\ \tau_{OTA}^{psend} = 0.013\mu s/octet \;\; if \;\; L > PVM\_DATA\_MAX \end{cases}$$

$$T_{OTA}^{mcast} = \left(\beta_{OTA}^{mcast} + (P-1) \times L\tau_{OTA}^{mcast}\right)$$
$e = 2.1\%$

avec
$$\begin{cases} \beta_{OTA}^{mcast} = 137\mu s \\ \tau_{OTA}^{mcast} = 0.01\mu s/octet \end{cases}$$

$$T_{OTA}^{bcast} = \left(\beta_{OTA}^{bcast} + (P-1) \times L\tau_{OTA}^{bcast}\right)$$
$e = 4.2\%$

avec
$$\begin{cases} \beta_{OTA}^{bcast} = 137\mu s \\ \tau_{OTA}^{bcast} = 0.01\mu s/octet \end{cases}$$

Nous présentons sur la figure 2.10 les résultats d'expérimentation sur 64 et 128 processeurs avec la fonction `pvm_psend` que nous comparons avec les expérimentations analytiques en utilisant les modèles.

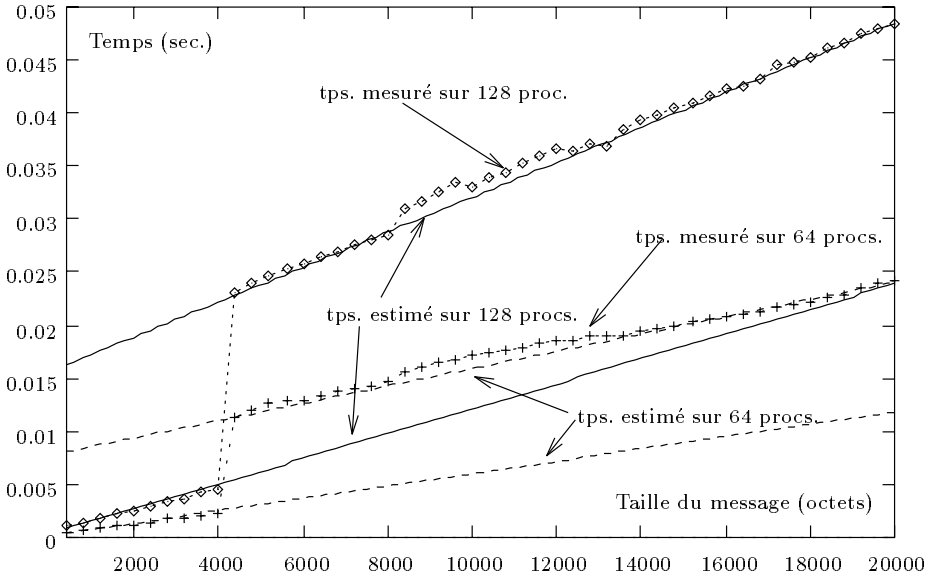


FIG. 2.10 - *Comparaison des temps simulés et mesurés sur 64 et 128 processeurs.*

2.3.4 *All-To-All*

Comme pour la diffusion, nous avons utilisé trois implémentations différentes pour le All-To-All. Nous présentons sur la figure 2.11 les résultats en fonction du nombre de processeurs et de la taille des messages avec l'utilisation de la fonction `pvm_mcast` (les résultats sont similaires avec les autres fonctions).

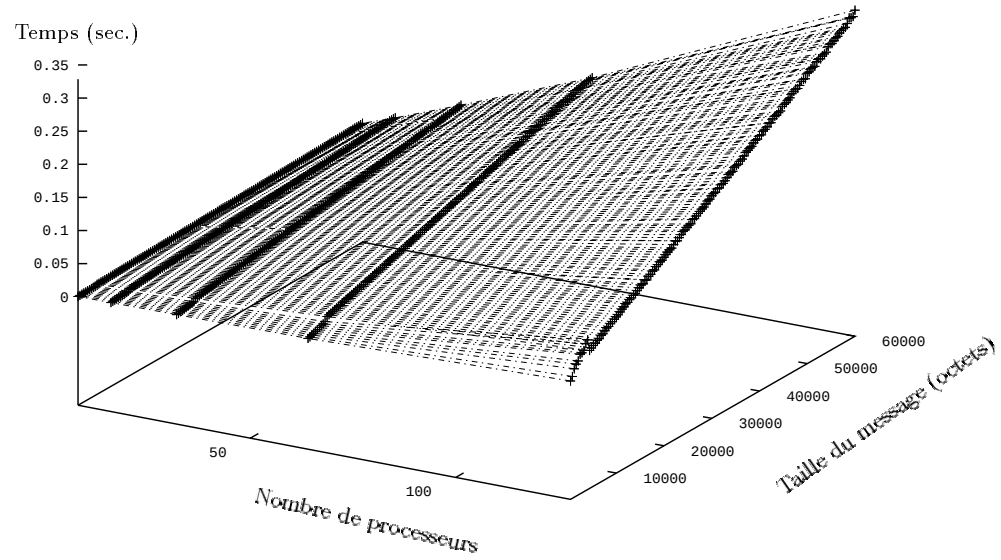


FIG. 2.11 - *Résultats du All-To-All en utilisant la fonction* `pvm_mcast` *en fonction du nombre de processeurs et de la taille des messages.*

D'après l'étude des résultats d'expérience, étant donné un nombre de processeurs, le temps est linéairement dépendant de la taille des messages (pour une taille supérieure à `PVM_DATA_MAX`). Ce fait est confirmé ensuite par la valeur des erreurs des modélisations. Nous nous intéressons sur la figure 2.12 aux messages plus petits.

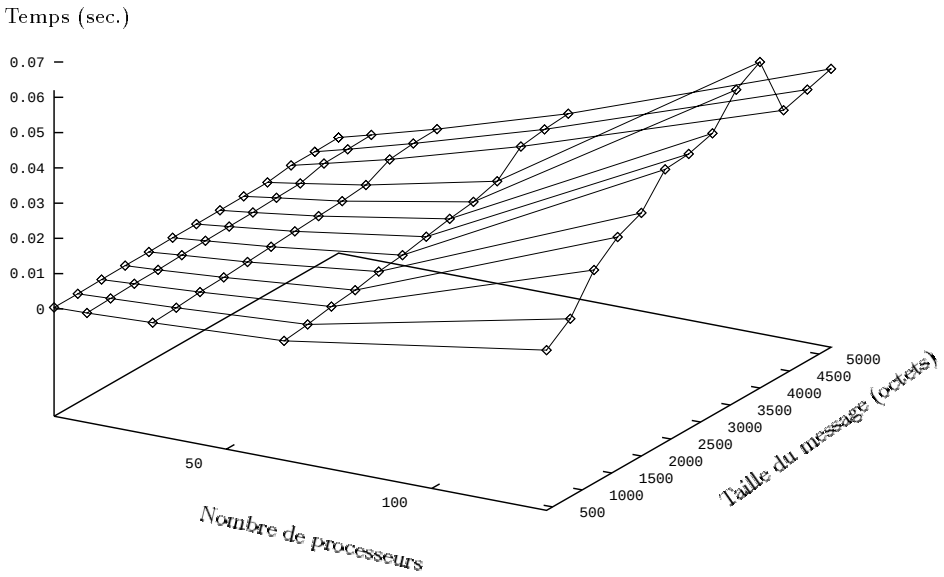


FIG. 2.12 - *Zoom pour les petits messages.*

Nous observons un comportement différent suivant le nombre de processeurs. En effet, pour des tailles de message inférieures à `PVM_DATA_MAX`, la fonction `pvm_mcast` utilise des écritures distantes. Cela génére des conflits d'accès mémoire au niveau de la réception (*hot spots*). Ce phénomène s'accroît d'autant plus lorsque le nombre de processeurs est grand. Lorsque la taille des messages est plus importante, le système utilise des lectures à distance et ainsi la réception devient séquentielle, ce qui évite les conflits d'accès. De ce fait il est moins coûteux d'envoyer des messages un peu plus grands que `PVM_DATA_MAX` plutôt que d'envoyer des messages un peu plus petits (voir la figure 2.12, `PVM_DATA_MAX` est égal à 4096 dans ce cas).

Pour ces raisons, il est assez difficile de trouver un modèle général fonction du nombre de processeurs. Nous avons donc détaillé dans les tables suivantes les paramètres des différents modèles dépendants de la taille des messages pour chaque nombre de processeurs :

`pvm_psend` : $T_{ATA}^{psend} = \beta_{ATA}^{psend} + L\tau_{ATA}^{psend}$

P	$L < PVM\_DATA\_MAX$	$L \geq PVM\_DATA\_MAX$	e
16	$\beta = 492$ $\tau = 0.462$	$\beta = 3740$ $\tau = 0.665$	2.84%
32	$\beta = 814$ $\tau = 1.3$	$\beta = 7307$ $\tau = 1.445$	1.16%
64	$\beta = 2970$ $\tau = 2.58$	$\beta = 15550$ $\tau = 3.07$	3.6%
128	$\beta = 7480$ $\tau = 17.1$	$\beta = 37300$ $\tau = 6.067$	1.4%

β en $\mu seconds$ et τ en $\mu s/octet$.

pvm_mcast : $T_{ATA}^{mcast} = \beta_{ATA}^{mcast} + L\tau_{ATA}^{mcast}$

P	$L < PVM\_DATA\_MAX$	$L \geq PVM\_DATA\_MAX$	e
16	$\beta = 367.5$ $\tau = 0.584$	$\beta = 2100$ $\tau = 0.54$	1.7%
32	$\beta = 760$ $\tau = 1.2$	$\beta = 4340$ $\tau = 1.12$	0.7%
64	$\beta = 2205$ $\tau = 2.45$	$\beta = 9450$ $\tau = 2.36$	0.9%
128	$\beta = 10795$ $\tau = 13.35$	$\beta = 21590$ $\tau = 4.98$	1.08%

β en $\mu seconds$ et τ en $\mu s/octet$.

pvm_bcast : $T_{ATA}^{bcast} = \beta_{ATA}^{bcast} + L\tau_{ATA}^{bcast}$

P	$L < PVM\_DATA\_MAX$	$L \geq PVM\_DATA\_MAX$	e
16	$\beta = 479$ $\tau = 0.55$	$\beta = 131$ $\tau = 0.656$	5%
32	$\beta = 102$ $\tau = 1.19$	$\beta = 250$ $\tau = 1.5$	4.2%
64	$\beta = 239.4$ $\tau = 2.5$	$\beta = 492.5$ $\tau = 3.6$	4.82%
128	$\beta = 895$ $\tau = 5$	$\beta = 1260$ $\tau = 7.6$	4%

β en $\mu seconds$ et τ en $\mu s/octet$.

Nous comparons sur la figure 2.13 les différentes implémentations sur 64 processeurs.

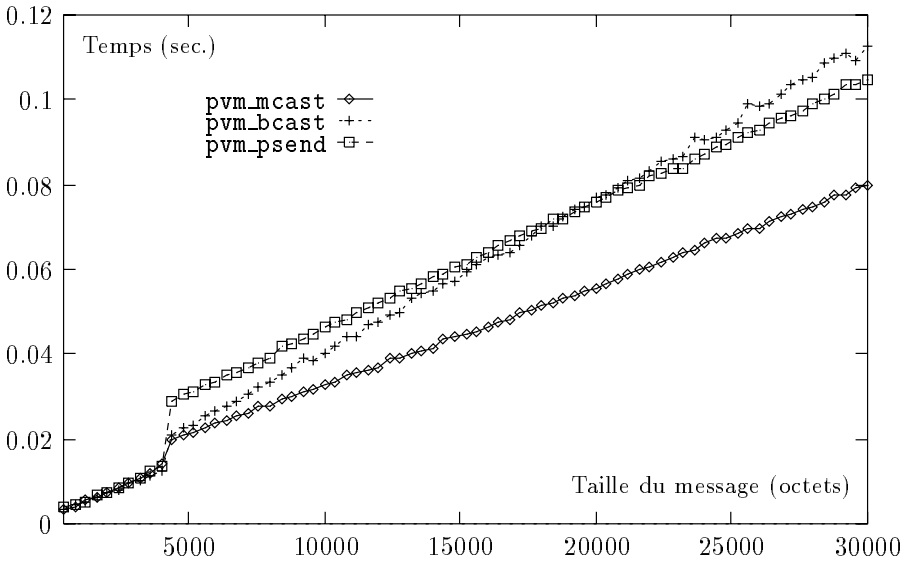


FIG. 2.13 - *Comparaison des différentes implémentations sur 64 processeurs.*

Nous remarquons sur cette figure une dégradation des performances en utilisant la fonction pvm_bcast. En effet, celle-ci est basée sur un arbre binaire de recouvrement pour réaliser la diffusion [82]. Bien qu'en théorie cette technique permette d'améliorer les performances

d'un schéma de diffusion, cela a pour conséquence de générer des contentions sur le réseau lorsqu'il est utilisé par tous les processeurs en même temps, comme c'est le cas dans un schéma d'échange total (voir le chapitre 1). En effet les sommets des arbres de recouvrement se retrouvent placés sur les mêmes nœuds physiques du réseau. Ceci pourrait être évité en utilisant des arbres rotatifs [78, 82].

2.3.5 *Personalized-All-To-All*

Ce schéma a été implémenté à l'aide des primitives `pvm_psend` et `pvm_send - DataRaw`. Comme pour les tests précédents du ATA, nous observons des contentions similaires pour des tailles de messages juste inférieures à `PVM_DATA_MAX`. Nous résumons les différents paramètres des modèles dans les tables suivantes :

`pvm_psend` : $T_{pATA}^{psend} = \beta_{pATA}^{psend} + L\tau_{pATA}^{psend}$

P	$L < PVM\_DATA\_MAX$	$L \geq PVM\_DATA\_MAX$	e
16	$\beta = 1666$ $\tau = 0.833$	$\beta = 2833$ $\tau = 0.719$	3.7%
32	$\beta = 3222$ $\tau = 1.9441$	$\beta = 6430$ $\tau = 1.492$	3.2%
64	$\beta = 6221$ $\tau = 4.44$	$\beta = 14219$ $\tau = 3.579$	3.54%
128	$\beta = 11109$ $\tau = 12.2$	$\beta = 35515$ $\tau = 6.4747$	1.6%

β en $\mu seconds$ et τ en $\mu s/octet$.

`pvm_send` : $T_{pATA}^{send} = \beta_{pATA}^{send} + L\tau_{pATA}^{send}$

P	$L < PVM\_DATA\_MAX$	$L \geq PVM\_DATA\_MAX$	e
16	$\beta = 1888$ $\tau = 0.277$	$\beta = 3833$ $\tau = 0.719$	2.9%
32	$\beta = 2555$ $\tau = 1.1108$	$\beta = 7272$ $\tau = 1.528$	3.38%
64	$\beta = 4667$ $\tau = 3.33$	$\beta = 15300$ $\tau = 3.111$	0.93%
128	$\beta = 11554$ $\tau = 11.1$	$\beta = 34751$ $\tau = 6.4209$	0.9%

β en $\mu seconds$ et τ en $\mu s/octet$.

2.4 Conclusion

Nous avons présenté dans ce chapitre une série d'expériences de communication destinée à prédire les temps de communication du Cray T3D en fonction des bibliothèques de communication utilisées. Nous avons évalué non seulement les fonctions point-à-point, mais aussi des schémas de communication globale. Pour chacun des schémas expérimentés nous avons proposé des modèles fins, puisque les erreurs mesurées sont de 2% en moyenne.

Bien que les conditions et les hypothèses des expériences effectués soient les plus proches possibles des conditions réelles d'une application parallèle, les résultats peuvent varier du

fait de l'asynchronisme du fonctionnement de la machine et du rangement des données en mémoire.

De ces expériences nous pouvons tirer quelques avantages de l'utilisation des communications basées sur des systèmes de lecture à distance (les *remote loads*) qui permettent d'éviter les problèmes de contention mémoire[4]. Par contre ces mécanismes limitent dans une certaine mesure les possibilités de recouvrement des communications.

Dans des travaux futurs il faudrait envisager l'utilisation d'outils d'évaluation de performance plus complexes afin de modéliser les problèmes de contention au niveau des nœuds [118]. Un autre problème, encore plus complexe, consiste à trouver des modélisations des performances de calcul (comme la gestion de la mémoire, l'influence du cache, les performnaces des opérateurs pipelines etc.).

Cependant ces premières études nous ont permis de montrer qu'il était possible d'obtenir des modélisations de performance des communications d'une machine. Ainsi, nous les utiliserons par la suite dans la conception d'algorithmes numériques parallèles efficaces masquant les communications.

4. Phénomène qui ne devrait pas se produire dans une véritable machine Δ-ports, mais qui arrive lorsque les liens de communication sont parallèles mais pas les accès mémoires.

Chapitre 3

Application à la transformée de Fourier

Nous décrivons dans ce chapitre une application des techniques de recouvrement présentées lors des chapitres précédents. Il s'agit d'algorithmes de transformée de Fourier avec minimisation du sur-coût de communication. Les techniques ont été employées sur des transformations mono-dimensionnelles [21] et bi-dimensionnelles [22, 28].

3.1 Introduction

La transformée de Fourier (TF) est utilisée dans de nombreux domaines scientifiques, tels que les mathématiques appliquées, le traitement du signal, le traitement d'images etc. Elle constitue le noyau de base dans le calcul d'une convolution, dans l'analyse spectrale ... Les TF multi-dimensionnelles sont utilisées en dynamique des fluides, dans la solution d'équations de Poisson [16] ainsi que dans le traitement d'images *via* des techniques de filtrage. Un des algorithmes les plus connus pour calculer la TF est l'algorithme de Transformée de Fourier Rapide (connu sous le terme de FFT pour *Fast Fourier Transform*) [38].

Les calculs de FFT mono et multi-dimensionnelles ont été largement étudiés sur plusieurs types de machines hautes performances, comme les machines à mémoire partagée [7], les ordinateurs vectoriels [6], ou encore les machines parallèles à mémoire distribuée de types SIMD [83] et MIMD [31, 69]. Ces articles décrivent différentes implémentations d'algorithmes de FFT selon l'architecture cible de la machine, ou implémentent de nouveaux algorithmes dérivés de méthodes séquentielles différentes. Plusieurs algorithmes ont été proposés pour recouvrir les communications, soit dans le cas de transformée de Fourier mono-dimensionnelle [8, 69, 128, 129], soit dans le cas de transformée de Fourier bi-dimensionnelle [28]. Cependant, le recouvrement des communications n'est pas effectué de manière efficace dans le cas mono-dimensionnel, du fait du schéma de dépendance entre les données et les calculs [8, 69, 128].

Chu [36] présente plusieurs algorithmes FFT bi-dimensionnelles sur un hypercube. Il met en évidence le fait que des améliorations peuvent être obtenues en recouvrant des phases de communication par des phases de calcul. Walker [127, 128] présentent des versions par blocs

d'algorithmes de FFT bi-dimensionnelles en utilisant du recouvrement, mais il n'exploite pas l'éventuel parallélisme au niveau des liens de communication.

Nous montrons dans ce chapitre comment les différents algorithmes de FFT, que cela soit dans le cas mono ou bi-dimensionnel, sont des illustrations des techniques de masquage des communications présentées dans le chapitre 1 de cette même partie. Cela nous conduit à utiliser un nouvel algorithme avec redistribution intermédiaire des données dans le cas mono-dimensionnel, nous permettant ainsi de recouvrir le sur-coût des communications. Dans le cas bi-dimensionnel nous montrons comment recouvrir les communications et nous présentons de plus des méthodes utilisant de façon maximale la bande passante du réseau dans le cas d'une architecture k-port 1.

Nous allons présenter, dans un premier, temps les algorithmes séquentiels de calcul de la transformée de Fourier mono et bi-dimensionnelles. Dans le deuxième paragraphe nous décrirons quelques résultats préliminaires sur les correspondances entre l'hypercube, la grille et le tore en commençant par des résultats d'émulations des communications sur l'hypercube sur ces autres topologies. Les paragraphes 3 et 4 concernent la parallélisation des algorithmes de transformée de Fourier sur l'hypercube, la grille et le tore. Les deux paragraphes suivants présentent les versions des précédents algorithmes avec recouvrement des communications par du calcul et utilisation maximale de la bande passante des réseaux. Nous étudions la complexité de ces algorithmes en fonction de la topologie, et nous calculons les tailles optimales de paquets afin de maximiser le recouvrement. Enfin, nous finissons par des résultats d'expérimentations sur diverses machines : Intel iPSC/860 et Paragon, CRAY T3D et IBM SP-1.

3.2 Algorithmes séquentiels de calcul de la Transformée de Fourier

3.2.1 FFT mono-dimensionnelle

La Transformée de Fourier Discrète (notée TFD) d'un vecteur x de $\mathbb{R}$ ou $\mathbb{C}$ de taille n, est le vecteur X défini de la façon suivante :

$$X_j = \sum_{k=0}^{n-1} x_k \omega^{jk} \quad avec \quad \omega^{jk} = e^{-\frac{2i\pi jk}{n}} \quad et \quad i^2 = -1$$

Le premier algorithme de FFT a été proposé par Cooley et Tuckey [38]. Depuis, de nombreuses variations ont été déduites de cet algorithme [114]. La principale différence provient de la façon dont sont stockés les éléments intermédiaires. L'idée principale des ces algorithmes est le découpage des données en entrée, x, à chaque étape de l'algorithme en deux sous-ensembles, et de les combiner en utilisant un **schéma papillon**. Nous avons représenté dans la figure 3.1 un exemple de l'algorithme pour $n = 8$.

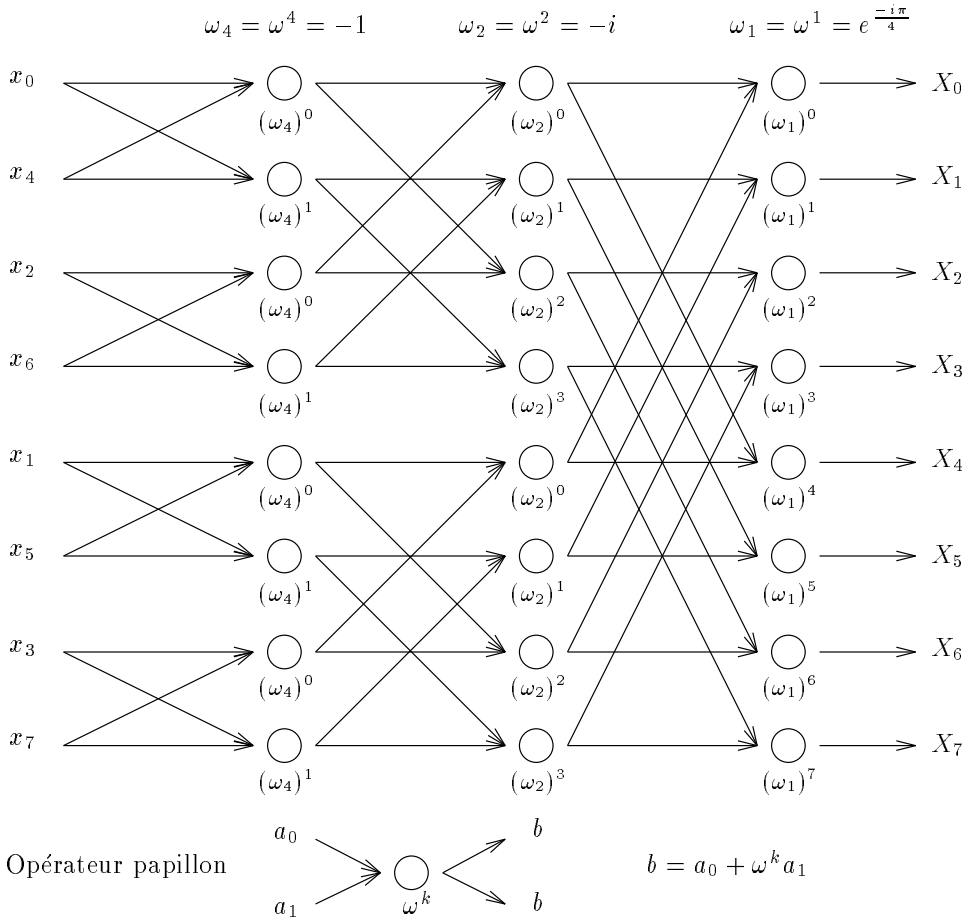


FIG. 3.1 - *Graphe d'exécution de l'algorithme de Cooley-Tuckey pour* $n = 8$.

Nous utilisons dans la suite une version de l'algorithme qui implique que les données en entrée sont en « ordre bit-inversé » et les données en sortie sont en ordre normal. L'algorithme est décrit dans la figure 3.2 [69]. Nous supposons que $n = 2^r$, et nous notons par $(i_0 \dots i_{r-1})$ la représentation binaire de i.

```
Procédure fft1d_seq(x,n)
début
  pour i = 0 à n − 1 faire
    X[i] = x[i]

  pour l ← 0 to r − 1 faire
    pour i ← 0 à n − 1 faire
      S[i] ← X[i]
    pour i ← 0 à n − 1 do
      X[(i_0 ... i_{r−1})] ← S[(i_0 ... i_{l−1} 0 i_{l+1} ... i_{r−1}]+
        ω^(i_l i_{l−1} ... i_0 0 ... 0) × S[(i_0 ... i_{l−1} 1 i_{l+1} ... i_{r−1}]
fin
```

FIG. 3.2 - *Algorithme séquentiel de FFT mono-dimensionnel.*

Nous notons τ_a le temps de calcul d'un schéma papillon. Le coût de l'algorithme précédent est le suivant :

$$\boxed{T_{seq}^{fft1d} = n \log_2(n) t_a}$$

3.2.2 FFT bi-dimensionnelle

Considérons une matrice carrée A de dimension n. Nous notons $\tilde{A}$ la transformée de Fourier de la matrice A. Le calcul de la TFD bi-dimensionnelle est donnée par l'équation [112] :

$$\tilde{A}(j_1, j_2) = \sum_{k_1=0}^{n-1} \left(\sum_{k_2=0}^{n-1} A(k_1, k_2) \times \exp\left(\frac{(-2i\pi(k_1.j_1 + k_2.j_2))}{n} \right) \right)$$

Cette équation peut être transformée afin de se ramener au calcul de deux TFD mono-dimensionnelles classiques :

$$\tilde{A}(j_1, j_2) = \sum_{k_1=0}^{n-1} \exp\left(\frac{(-2i\pi k_1.j_1)}{n} \right) \times Y_{j_2}(k_1)$$

Où $Y_{j_2}(k_1)$ est la TFD de A par rapport à la variable k_1.
L'algorithme de FFT bi-dimensionnelle est alors naturel, il consiste à calculer des FFT mono-dimensionnelles suivant les deux dimensions de la matrice. Donc le coût d'une FFT bi-dimensionnelle est :

$$\boxed{T_{seq}^{fft2d} = 2n^2 \log_2(n) t_a}$$

3.3 Préliminaires

Avant de présenter les algorithmes sur chaque topologie, nous commençons par décrire les différentes fonctions d'allocation utilisées ainsi que quelques résultats d'émulation des communications sur hypercube, la grille et le tore.

3.3.1 Quelques définitions

Dans ce chapitre nous utilisons les notations et définitions suivantes :

- Nous définissons une fonction d'allocations, *Alloc*, comme une bijection de l'ensemble des éléments de la matrice vers l'ensemble des processeurs. Nous notons $Alloc_{topo}$ la fonction d'allocation correspondant à la topologie *topo*.
- La fonction *Bit Reverse*, notée BR, est définie de la façon suivante : $BR(i_0 i_1 \dots i_{k-1}) = (i_{k-1} i_{k-2} \dots i_0)$ où $(i_0 i_1 \dots i_{k-1})$ est la représentation binaire de l'entier i.
- Nous notons $\oplus$ l'opérateur « ou-exclusif » sur les bits, $|$ l'opérateur de concaténation de bits et GR l'opérateur défini de la façon suivante [107] : $GR(\alpha) = \alpha \oplus \left\lfloor \frac{\alpha}{2} \right\rfloor$ où α est un entier.
- La représentation binaire de l'entier i est notée $(i)_2$.
- La notation $(i,j) \longrightarrow (\overline{i},\overline{j})$ indique une communication du nœud (i,j) de la grille, ou du tore, vers le nœud $(\overline{i},\overline{j})$.

3.3.2 Correspondance entre les fonctions d'allocation sur les différentes topologies

Afin de pouvoir directement émuler les communications de l'hypercube sur la grille et le tore, il nous faut donner des correspondances entre la fonction d'allocation des données sur l'hypercube et celles sur la grille et le tore. Ces correspondances doivent être telles qu'elles maximisent le nombre de communication entre voisins directs sur les différentes topologies pendant le déroulement de l'algorithme. Étant donnée une fonction d'allocation sur l'hypercube, $Alloc_{Hyp}(i) = q_h$, nous avons les correspondances suivantes avec les fonctions d'allocation sur les autres topologies :

$$\text{si } Alloc_{Hyp}(i) = q_h \text{ alors } \begin{cases} Alloc_{Grille}(i) = \left(\left\lfloor \frac{q_h}{P_2} \right\rfloor , q_h \bmod P_2 \right) \\ Alloc_{Tore}(i) = \left(GR\left(\left\lfloor \frac{q_h}{P_2} \right\rfloor \right) , GR(q_h \bmod P_2) \right) \end{cases}$$

Nous présentons un exemple de la fonction d'allocation sur le tore dans la figure 3.3 pour un tore 4×2 et un vecteur de 8 éléments.

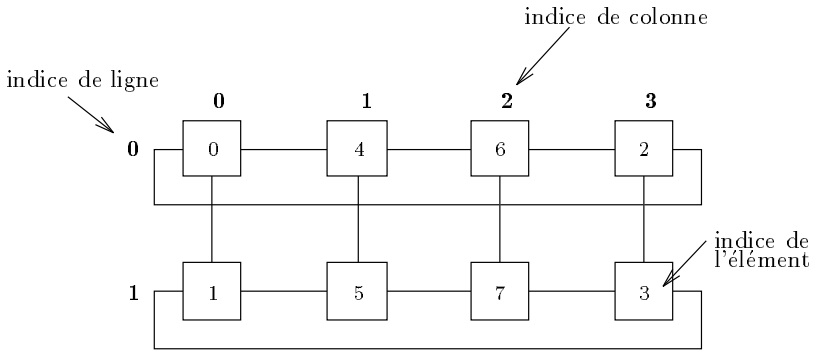


FIG. 3.3 - *Fonction d'allocation pour un tore* 4×2 *et un vecteur de 8 éléments.*

3.3.3 Résultats d'émulation

Maintenant que nous avons défini les fonctions d'allocations sur les différentes topologies en fonction de celle de l'hypercube, nous avons besoin de définir la correspondance d'une communication suivant une dimension d sur l'hypercube avec la communication sur la grille ou le tore. Nous supposons que toutes les topologies ont le même nombre de processeurs.

Émulation des communications de l'hypercube sur la grille 2D

Une communication suivant la dimension d de l'hypercube correspond à la communication sur la grille (voir figure 3.4) : $(i, j) \longrightarrow (\overline{i}, \overline{j})$ où:

$$\begin{aligned} \overline{i} &= \left\lfloor \frac{[(i \times P_1) + j] \oplus 2^d}{P_2} \right\rfloor \\ \overline{j} &= \left([(i \times P_1) + j] \oplus 2^d \right) \bmod P_2 \end{aligned}$$

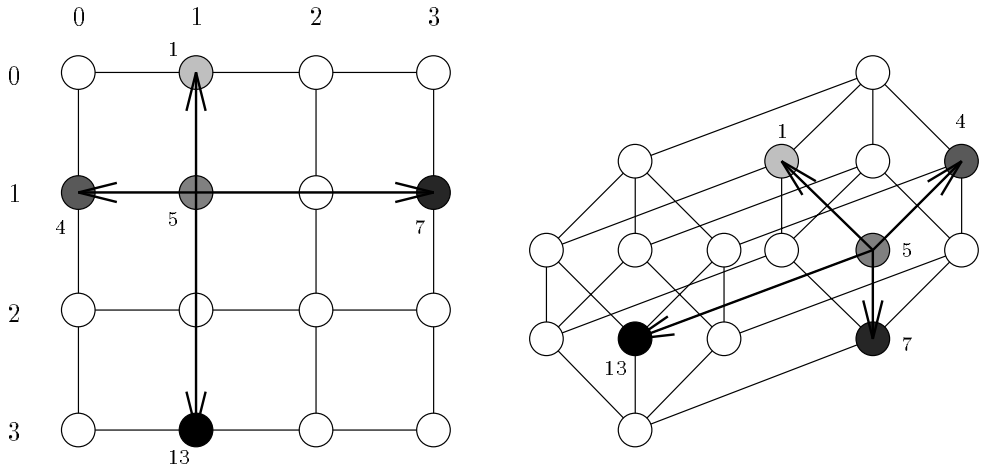


FIG. 3.4 - *Émulation des communications 4-cube sur la grille* 4×4.

Prenons un exemple : considérons un hypercube de dimension 4, alors une communication suivant chacune des dimensions de l'hypercube correspond à deux communications à distance 1 et à deux communications à distance 2 sur la grille 4×4 (voir figure 3.4).

Émulation des communications de l'hypercube sur le tore 2D

Une communication suivant la dimension d de l'hypercube correspond à la communication sur le tore (voir figure 3.5) : $(i, j) \longrightarrow (\overline{i}, \overline{j})$ où :

$$\begin{aligned} \overline{i} &= GR\left(\left\lfloor \frac{[(i \times P_1) + j] \oplus 2^d}{P_2} \right\rfloor\right) \\ \overline{j} &= GR\left(\left([(i \times P_1) + j] \oplus 2^d\right) \bmod P_2\right) \end{aligned}$$

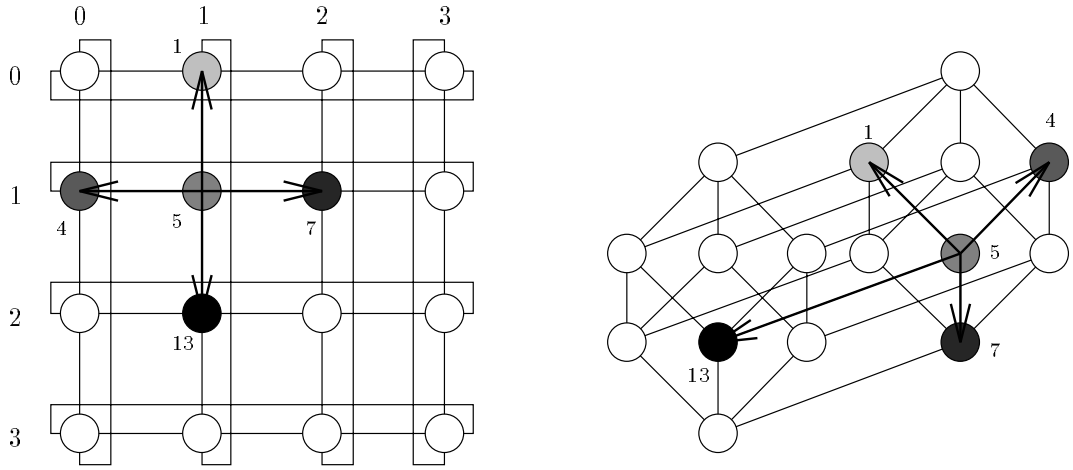


FIG. 3.5 - *Émulation des communications du 4-cube sur le tore* 4×4.

Par exemple, si nous considérons un hypercube de dimension 4, alors une communication suivant chacune des dimensions de l'hypercube équivaut à quatre communications à distance 1 sur le tore 4×4 (voir figure 3.5).

Nous faisons remarquer au lecteur que ces résultats d'émulation ne sont corrects que si les correspondances entre les fonctions d'allocations sur les différentes topologies sont utilisées. Dans la suite de cette partie, chaque algorithme décrit sur l'hypercube peut être ainsi directement implémenté sur la grille et le tore 2D en utilisant les fonctions d'allocations et les résultats d'émulation présentés précédemment.

3.4 Algorithmes parallèles de FFT mono-dimensionnelle

3.4.1 Algorithme général

Le vecteur x est divisé en P blocs de taille $\frac{n}{P}$ [1] alors la fonction d'allocation s'exprime de cette façon citeCha88:

$$Alloc_{Hyp}(i) = BR\left((\tfrac{i \times P}{n})_2\right) \qquad i = 0 \ldots n-1$$

où i est l'indice d'un élément de x.
Le programme d'un processeur q est écrit dans la figure 3.6.

```
Procédure fft1d_par(x, n, P)
début

    /* Calcul fft1d sur mes données de taille n/P */
  fft1d_loc(x, n/P)

  pour l = 0 à log2 (P) − 1 faire
      /* Échange des données avec le processeur q ⊕ l */
    Échange(x, x_recv, n/P, q ⊕ l)
      /* Calcul d'une opération papillon */
      /* entre mon bloc et le bloc reçu */
    Mise-à-jour(x, x_recv, n/P)

fin
```

FIG. 3.6 - *Algorithme de FFT parallèle.*

Dans cet algorithme, la routine `fft1d_loc` correspond au calcul d'une FFT partielle mono-dimensionnelle sur les $\frac{n}{P}$ données locales.

3.4.2 Implémentations et analyses de la compléxité sur différentes topologies

L'hypercube

L'allocation *bit-reverse* implique que toutes les communications au cours de l'algorithme se font entre voisins directs [31]. Nous avons décrit les étapes successives de l'algorithme pour $n = 16$ et $P = 8$ dans la figure 3.7.

1. Nous supposons que n et P sont des puissances de 2, aussi $\frac{n}{P}$ est entier.

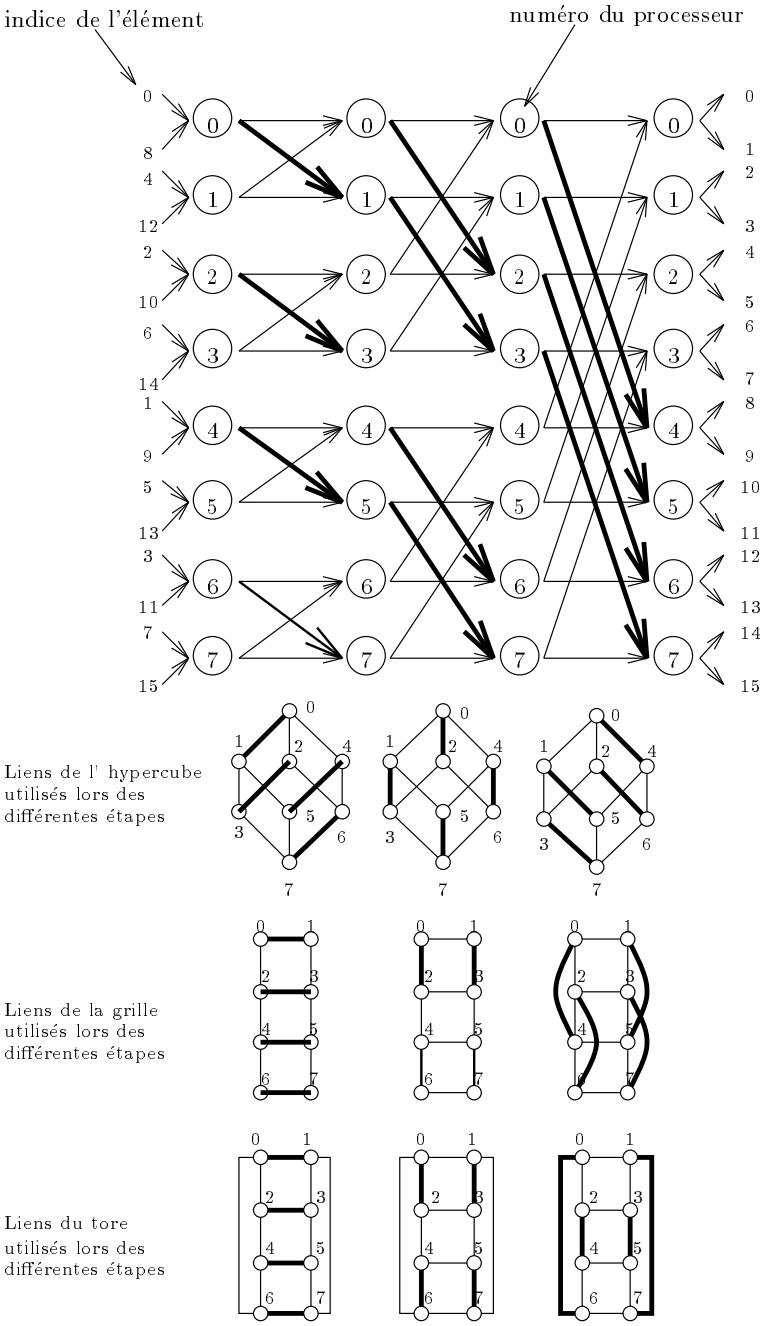


FIG. 3.7 - *Exécution de l'algorithme parallèle de FFT mono-dimensionnelle pour* $n = 16$ *sur un 3-cube, une grille et un tore* 4×2.

Le coût de l'algorithme est :

$$T_{Hyp}^{fft1d} = \frac{n}{P}\log_2\left(\frac{n}{P}\right)t_a + \log_2(P)\left[\left(\beta + \frac{n}{P}\tau\right) + \frac{n}{P}t_a\right]$$

$$\boxed{T_{Hyp}^{fft1d} = \frac{n}{P}\log_2(n)t_a + \log_2(P)\left(\beta + \frac{n}{P}\tau\right)}$$

La grille

Considérons une grille de $P = P_1 \times P_2$ processeurs, telle que $P_1 = 2^{p_1}$ et $P_2 = 2^{p_2}$. Nous pouvons directement adapter l'algorithme de l'hypercube en utilisant l'émulation des communications de l'hypercube sur la grille [69] (voir figure 3.7). Le coût de l'algorithme est :

$$T_{Grille}^{fft1d} = \left(\frac{n}{P}\right)\log_2\left(\frac{n}{P}\right)t_a + \sum_{d=0}^{p_1-1}\left[\frac{n}{P}t_a + 2^d\left(\beta + \frac{n}{P}\tau\right)\right] + \sum_{d=0}^{p_2-1}\left[\frac{n}{P}t_a + 2^d\left(\beta + \frac{n}{P}\tau\right)\right]$$

$$\boxed{T_{Grille}^{fft1d} = \left(\tfrac{n}{P}\right)\log_2(n)t_a + (P_1 + P_2 - 2)\left(\beta + \tfrac{n}{P}\tau\right)}$$

Le tore

Considérons un tore de $P = P_1 \times P_2$ processeurs, tel que $P_1 = 2^{p_1}$ et $P_2 = 2^{p_2}$. Comme pour la grille, nous pouvons directement adapter l'algorithme de l'hypercube. La complexité de cet algorithme est égale à :

$$T_{Tore}^{fft1d} = \left(\frac{n}{P}\right)\log_2\left(\frac{n}{2P}\right)t_a + \sum_{d=0}^{p_1-1}\left[\frac{n}{2P}t_a + 2^{\lfloor\frac{d}{2}\rfloor}\left(\beta + \frac{n}{P}\tau\right)\right] + \sum_{d=0}^{p_2-1}\left[\frac{n}{2P}t_a + 2^{\lfloor\frac{d}{2}\rfloor}\left(\beta + \frac{n}{P}\tau\right)\right]$$

$$\boxed{T_{Tore}^{fft1d} = \left(\tfrac{n}{2P}\right)\log_2(n)t_a + \left(\tfrac{P_1+P_2}{2} - 2\right)\left(\beta + \tfrac{n}{P}\tau\right)}$$

3.5 Algorithmes parallèles de calcul de la FFT bi-dimensionnelle

Après avoir décrit les principales fonctions d'allocation [28, 80], nous présentons trois méthodes pour calculer une FFT bi-dimensionnelle [28, 36]. Afin d'améliorer la clarté de l'implémentation de ces méthodes sur la grille et le tore, nous supposons dans la suite que $P_1 = P_2 = \sqrt{P}$.

3.6 Allocation de données

Pour chaque allocation nous définissons la fonction d'allocation *Alloc(élément)* qui à *élément* (*élément* est soit une ligne (colonne) i, soit un élément (i, j) de la matrice) associe un numéro de processeur (voir figure 3.8).

- Allocation par blocs de lignes consécutives (LC)

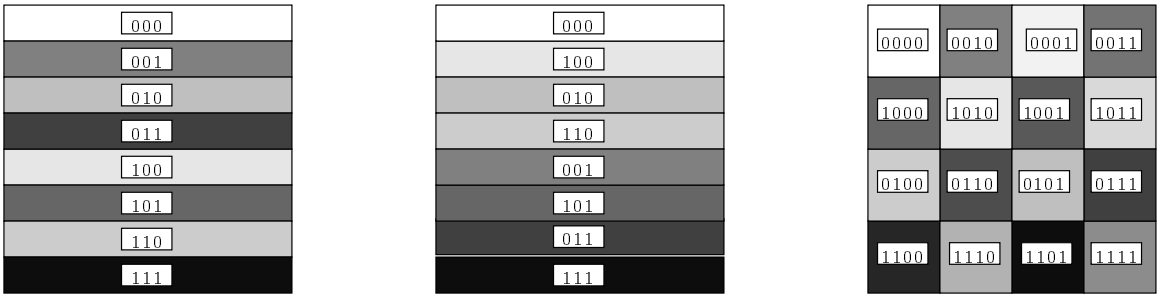
$$Alloc_{hyp}(i) = \left(\left\lfloor\frac{i \times P}{n}\right\rfloor\right)_2 \qquad i = 0 \ldots n-1$$

- Allocation *bit-reverse* par blocs de lignes consécutives (BRLC)

$$Alloc_{hyp}(i) = BR\left(\left(\left\lfloor\frac{i \times P}{n}\right\rfloor\right)_2\right) \qquad i = 0 \ldots n-1$$

– Allocation *bit-reverse* par blocs (BRB)

$$Alloc_{hyp}(i,j) = BR\left[\left(\left\lfloor\frac{i\sqrt{P}}{n}\right\rfloor\right)_2\right] | BR\left[\left(\left\lfloor\frac{j\sqrt{P}}{n}\right\rfloor\right)_2\right] \qquad i,j = 0\ldots n-1$$



Allocation LC d'une matrice de 16 lignes sur 8 processeurs

Allocation BRLC d'une matrice de 16 lignes sur 8 processeurs

Allocation BRB matrice de 16×16 sur 8 processeurs

FIG. 3.8 - *Les différentes fonctions d'allocation.*

3.7 Méthode *Transpose Split* (TS)

La matrice A est distribuée sur les processeurs suivant l'allocation LC. L'algorithme est une adaptation directe de la version séquentielle : après avoir calculé une FFT mono-dimensionnelle sur chacune des lignes, la matrice est transposée. Ensuite, une autre FFT mono-dimensionnelle est calculée sur chacune des lignes de cette matrice transposée. Afin de retrouver les données dans leur ordre initial, une dernière transposition est effectuée. Dans la suite, nous ne considèrerons qu'une seule transposition (voir figure 3.10). L'algorithme de cette méthode est exprimé dans la figure 3.9.

```
Procédure Transpose_Split
début

    /* FFT-1d sur les lignes  */
  pour i = 0 à n/P − 1 faire
    fft1d_seq(X[i][],n)

    /* Transposition de la matrice */
  Transpose(X,X^t)

    /* FFT-1d sur les lignes de la matrice transposée  */
  pour i = 0 à n/P − 1 faire
    fft1d_seq(X^t[i][],n)

fin
```

FIG. 3.9 - *Algorithme Transpose Split.*

Le coût de cet algorithme est :

$$T_{topo}^{TS} = 2\frac{n^2}{P}\log_2(n)t_a + T_{topo}^{comm}\left(\frac{n^2}{P}\right)$$

Nous détaillons dans la table 3.1 la complexité sur chaque topologie.

Topologie	T_{topo}^{comm}	T_{topo}^{TS}
Hypercube	$\log_2(P)\beta + \frac{n^2}{2P}\tau$ [82]	$2\frac{n^2}{P}\log_2(n)t_a + \log_2(P)\beta + \frac{n^2}{2P}\tau$
Grille	$2\sqrt{P}\beta + \frac{n^2\sqrt{P}}{4}\tau$ [108]	$2\frac{n^2}{P}\log_2(n)t_a + 2\sqrt{P}\beta + \frac{n^2\sqrt{P}}{4}\tau$
Tore	$\sqrt{P}\beta + \frac{n^2\sqrt{P}}{8}\tau$ [108]	$2\frac{n^2}{P}\log_2(n)t_a + \sqrt{P}\beta + \frac{n^2\sqrt{P}}{8}\tau$

TAB. 3.1 - *Complexités de l'algorithme TS en fonction de chaque topologie.*

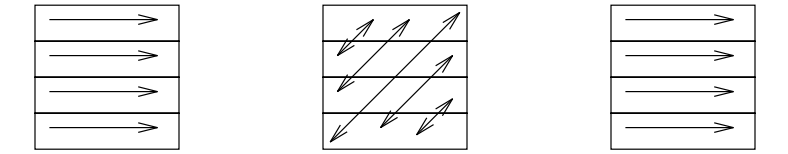


FIG. 3.10 - *Méthode Transpose Split.*

3.8 Méthode *Local Distributed* (LD)

La distribution de la matrice A est est de type BRLC. L'algorithme LD consiste à calculer une FFT mono-dimensionnelle sur chaque ligne de la matrice, mais contrairement à la méthode précédente, au lieu d'effectuer une transposition, une FFT mono-dimensionnelle distribuée est calculée suivant l'autre dimension de la matrice (voir figure 3.12). L'algorithme est exprimé dans la figure 3.11.

```
Procédure Local_Distributed
début

    /* FFT-1d sur les lignes */
  pour i = 0 à n/P − 1 faire
    fft1d_seq(X[i][],n)

    /* FFT-1d distribuée sur les colonnes */
  pour j = 0 à n − 1 faire
    fft1d_par(X[][j],n,P)

fin
```

FIG. 3.11 - *Algorithme LD.*

Nous détaillons dans la table 3.2 la complexité pour chaque topologie. Ces compléxités se déduisent directement de celles de la FFT mono-dimensionnelle.

Topologie	T_{topo}^{LD}
Hypercube	$2\frac{n^2}{P}\log_2(n)t_a + \log_2(P)\left(\beta + \frac{n^2}{P}\tau\right)$
Grille	$2\frac{n^2}{P}\log_2(n)t_a + 2(\sqrt{P}-1)\left(\beta + \frac{n^2}{P}\tau\right)$
Tore	$2\frac{n^2}{P}\log_2(n)t_a + \sqrt{P}\left(\beta + \frac{n^2}{P}\tau\right)$

TAB. 3.2 - *Complexité de l'algorithme LD en fonction des topologies.*

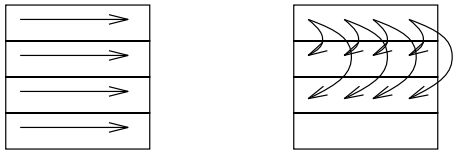


FIG. 3.12 - *Méthode Local Distributed.*

3.9 Méthode *Block* (Bl)

La troisième et dernière méthode consiste à calculer des FFT mono-dimensionnelles distribuées suivant les deux dimensions (voir figure 3.14). L'allocation de la matrice est de type BRB.

```
Procédure Block
début

    /* FFT-1d distribuée sur les lignes */
  pour i = 0 à n/√P − 1 faire
    fft1d_par(X[i][],n,√P)

    /* FFT-1d distribuée sur les colonnes */
  pour j = 0 à n/√P − 1 faire
    fft1d_par(X[][j],n,√P)

fin
```

FIG. 3.13 - *Algorithme Block*

Nous détaillons dans la table 3.3 la complexité sur chaque topologie. Comme pour l'algorithme précédent les compléxités se déduisent directement de celles de la FFT mono-dimensionnelle.

Topologie	T^{Bl}_{topo}
Hypercube	$2\frac{n^2}{P}\log_2(n)t_a + \log_2(P)\left(\beta + \frac{n^2}{P}\tau\right)$
Grille	$2\frac{n^2}{P}\log_2(n)t_a + 2(\sqrt{P}-1)\left(\beta + \frac{n^2}{P}\tau\right)$
Tore	$2\frac{n^2}{P}\log_2(n)t_a + \sqrt{P}\left(\beta + \frac{n^2}{P}\tau\right)$

TAB. 3.3 - *Complexités de l'algorithme Bl en fonction des topologies.*

Il faut remarquer que la méthode Bl revient à exécuter deux phases de FFT mono-dimensionnelle distribuée respectivement sur un $\log_2(\sqrt{P})$-cube pour l'hypercube, un réseau linéaire de taille $\sqrt{P}$ pour la grille et un anneau de taille $\sqrt{P}$ pour le tore.

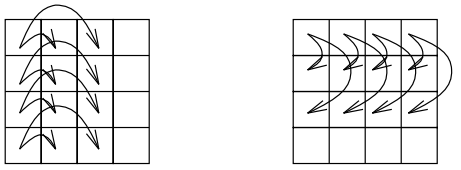


FIG. 3.14 - *Méthode Bl.*

3.10 Masquage des communications dans le cas mono-dimensionnel

3.10.1 Calcul d'une FFT mono-dimensionnelle

L'inconvénient majeur de l'algorithme parallèle classique présenté précédemment est la présence de $\log_2(P)$ étapes d'échanges de données entre les processeurs. Ces multiples échanges peuvent être pénalisants car ils nécéssitent une initialisation (*start-up*, voir partie I, chapitre 1) à chaque étape. De plus, comme nous allons le voir, l'algorithme classique ne permet pas de masquer le temps de communication. L'idée est alors d'utiliser une autre méthode, basée sur une redistribution intermédiaire des données [70]. Nous proposons alors un nouvel algorithme reposant sur cette méthode permettant de recouvrir au mieux les communications grâce à des techniques de pipeline et de réordonnncement des calculs locaux (voir chapitre 1). Mais revenons dans une premier temps sur le problème de la distribution des données.

Distribution des données

Le problème de la distribution de données sur une machine parallèle à mémoire distribuée consiste à faire correspondre certains bits de l'adresse d'un élément avec les numéros de processeur. Une de ces distributions est la *distribution cyclique*: le tableau de données est partagé en blocs d'éléments consécutifs de taille égale. Ceux-ci sont alors distribués de façon cyclique. Supposons que nous ayons à distribuer un vecteur x de N éléments sur P processeurs, alors l'allocation cyclique est définie ainsi :

$$\begin{aligned} Cyclic(indice, taille\,du\,bloc) &= \{(processeur), (adresse\,locale)\} \\ Cyclic(i,b) &= \left\{\left(\left\lfloor\frac{i}{b}\right\rfloor \bmod P\right), \left((i \bmod b) + \left\lfloor\frac{i}{P\times b}\right\rfloor \times b\right)\right\} \end{aligned}$$

Nous présentons dans le tableau ci-contre des exemples de distribution cyclique pour $b = 2, 1, 4$, $N = 8$ et $P = 4$.

b	Proc. 0	Proc. 1	Proc. 2	Proc. 3
2	0,1,8,9	2,3,10,11	4,5,12,13	6,7,14,15
1	0,4,8,12	1,5,9,13	2,6,10,14	3,7,11,15
4	0,1,2,3	4,5,6,7	8,9,10,11	12,13,14,15

Supposons qu'un tableau de 2^n éléments est distribué de façon cyclique sur 2^p processeurs. Alors les p bits nécessaires pour l'adresse du processeur sont choisis parmi les n bits utilisés pour coder les indices des éléments. Si $i = (i_{n-1} \dots i_0)$ est l'indice d'un élément, alors $Cyclic(i, 2^b)$ revient à choisir les bits $(i_{b+p-1} \dots i_b)$ comme adresse du processeur.

$$(\underbrace{i_{n-1} \dots i_{b+p}}_{adresse\ locale}\ \overbrace{i_{b+p-1} \dots i_b}^{processeur}\ \underbrace{i_{b-1} \dots i_0}_{adresse\ locale})$$

Une redistribution de données de $Cyclic(i, 2^b)$ à $Cyclic(i, 2^{b'})$, implique l'utilisation des bits $(i_{b'+p-1} \dots i'_b)$ pour l'adresse globale au lieu des bits $(i_{b+p-1} \dots i_b)$. Dans la plupart des cas, cette redistribution de données correspond à un échange total personnalisé (ou multi-distribution) comme schéma de communication.

Algorithme parallèle avec redistribution intermédiaire

Une façon classique d'implanter l'algorithme de FFT de Cooley et Tuckey sur une machine parallèle à mémoire distribuée est d'utiliser une allocation cyclique avec une taille de blocs égale à $\frac{N}{P}$. Nous pouvons déterminer les communications nécessaires à partir des traces en terme d'adresse globale : entre les étapes 0 à $n-p-1$, les calculs sont locaux, par contre les p dernières étapes nécessitent des échanges entre processeurs. Ce schéma d'exécution correspond à *un schéma à forte dépendance* [27]. C'est à dire que nous sommes en présence d'une séquence de tâches directement dépendantes les unes des autres. L'anticipation des communications n'étant alors pas possible, il est très difficile de recouvrir les communications [8, 69, 128]. L'idée est alors de changer d'algorithme pour obtenir un nouveau schéma de dépendance données/calculs.

Si $n \geq p - \ N \geq P^2$, alors en choisissant les p bits de poids fort pour les $n-p$ premières étapes, et les p bits de poids faible pour les p dernières étapes comme adresse de processeur, alors les calculs restent locaux pendant les n étapes. Ceci implique que le tableau d'entrée x est initialement distribué suivant une allocation $Cyclic(i, 2^n)$, et après $n-p$ étapes, sa distribution est changée en $Cyclic(i, 1)$. La trace de l'algorithme est la suivante :

Etape	:	*Adresse globale*
0	:	$(\underbrace{i_{n-1} \dots i_{n-p}}_{processor\ @}\ \underbrace{i_{n-p-1} \dots i_1\ \overset{\downarrow}{i_0}}_{local\ @})$
1	:	$(\underbrace{i_{n-1} \dots i_{n-p}}_{processor\ @}\ \underbrace{i_{n-p-1} \dots \overset{\downarrow}{i_1}\ i_0}_{local\ @})$
		$\vdots$
$(n-p-1)$	:	$(\underbrace{i_{n-1} \dots i_{n-p}}_{processor\ @}\ \underbrace{\overset{\downarrow}{i}_{n-p-1} \dots i_1 i_0}_{local\ @})$

Etape	:	*Adresse globale*
redistribution	:	$(\underbrace{i_{n-1} \dots i_p}_{local\ @}\ \underbrace{i_{p-1} \dots i_0}_{processor\ @})$
$(n-p)$	:	$(\underbrace{i_{n-1} \dots \overset{\downarrow}{i}_{n-p}}_{local\ @}\ \underbrace{i_{n-p-1} \dots i_1 i_0}_{processor\ @})$
		$\vdots$
$(n-1)$	:	$(\underbrace{\overset{\downarrow}{i}_{n-1} \dots i_{n-p}}_{local\ @}\ \underbrace{i_{n-p-1} \dots i_1 i_0}_{processor\ @})$

Nous présentons dans la figure 3.15 les schémas d'exécution des deux algorithmes pour $P = 4$ et $N = 8$.

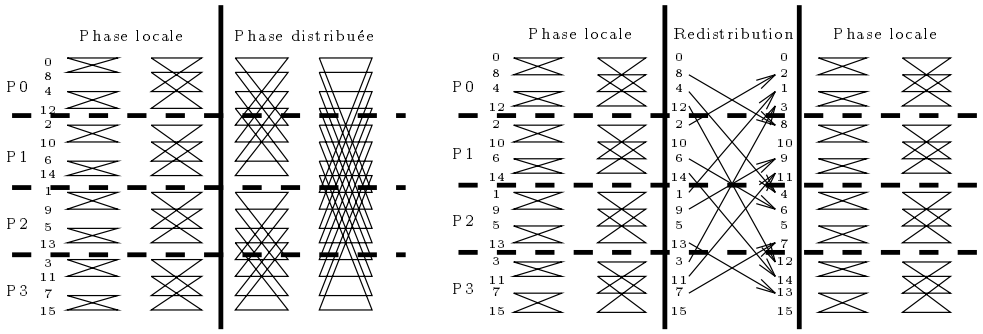


FIG. 3.15 - *Schémas d'exécution des algorithmes parallèles classiques et avec redistribution pour* $P = 4$ *et* $N = 8$.

Le schéma de dépendance de cet algorithme est *un schéma avec redistribution intermédiaire des données* (*cf* le premier chapitre de cette partie). Dans ce cas il est facile de recouvrir la phase de communication en enchaînant communication de résultats partiels avec calculs locaux [27]. En effet, il suffit de commencer par calculer les données qui vont être échangées, et de les communiquer dès leur mise à jour. Pendant cet échange, on peut alors calculer un papillon sur d'autres données à transférer. Nous présentons dans la figure 3.17 le schéma d'exécution de cet algorithme. Le papillon en pointillé étant celui à calculer en premier. L'algorithme avec recouvrement est décrit dans la figure 3.16.

```
pour s = 0 à p − 1 faire
  pour pap= 0 à N/P − 1 faire
    calculer(pap)
pour pap = 1 à N/P − 1 faire
  calculer(pap)
  envoyer(pap)
calculer(0)
pour pap= 1 à N/P − 1 faire
  recevoir(pap)
pour s = p à n − p − 1 faire
  pour pap= 0 à N/P − 1 faire
    calculer(pap)
```

FIG. 3.16 - *Algorithme avec recouvrement.*

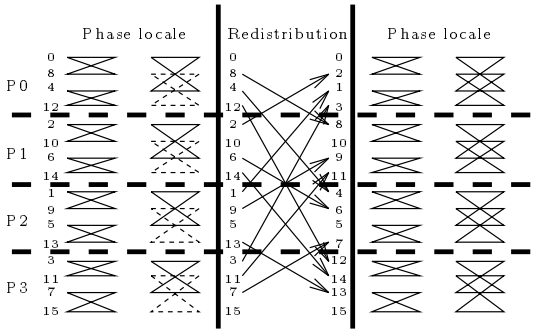


FIG. 3.17 - *Schéma d'exécution de l'algorithme avec recouvrement.*

3.10.2 Calcul de plusieurs FFT mono-dimensionnelles

Supposons que nous ayons à calculer m FFT mono-dimensionnelles de taille n sur P processeurs[2]. Chaque processeur possède un bloc de $\frac{m \times n}{P}$ données. Le but de l'algorithme asychrone (ASYNC) que nous proposons, est de maximiser le recouvrement des communications, de calculer plusieurs lignes au même moment afin de réduire le nombre de *start-up* et d'utiliser la bande passante maximale du réseau.

2. Nous supposons dans la suite que $\frac{m}{P}$ et que $\frac{n}{P}$ sont entiers.

Walker donne dans [127] quelques remarques pour recouvrir les communications en calculant plusieurs blocs de lignes à la fois. Cette méthode, par blocs de lignes, permet ainsi de réduire le nombre de *start-up* et de recouvrir les communications d'un bloc de lignes, par le calcul d'un autre bloc. Dans ce paragraphe, nous généralisons la méthode décrite par Walker en augmentant le nombre de communications recouvertes en utilisant plusieurs liens de communication à la fois.

Pour cela, nous devons trouver une distribution des données qui permettent d'effectuer des phases de communication sur des liens de communication différents de blocs de lignes différents lors d'une même étape de l'algorithme de FFT.

Supposons que nous ayons une machine k-port (*cf* partie I, chapitre 1). Alors la fonction d'allocation du bloc (i,j), de s lignes, est la suivante :

$$Alloc_{hyp}(i,j) = SL^{i \bmod k}(BR(j)), \quad 0 \leq i < \frac{m}{s}, 0 \leq j < \frac{n}{s}$$

où $SL^i(x)$ correspond à appliquer i fois un décalage à gauche sur les bits de x :

$$SL^i(x_0 x_1 \ldots x_i \ldots x_{k-1}) = (x_i \ldots x_{k-1} x_0 x_1 \ldots x_{i-1}).$$

Un exemple de cette distribution est donné sur la figure 3.18 sur 8 processeurs et $k = 3$.

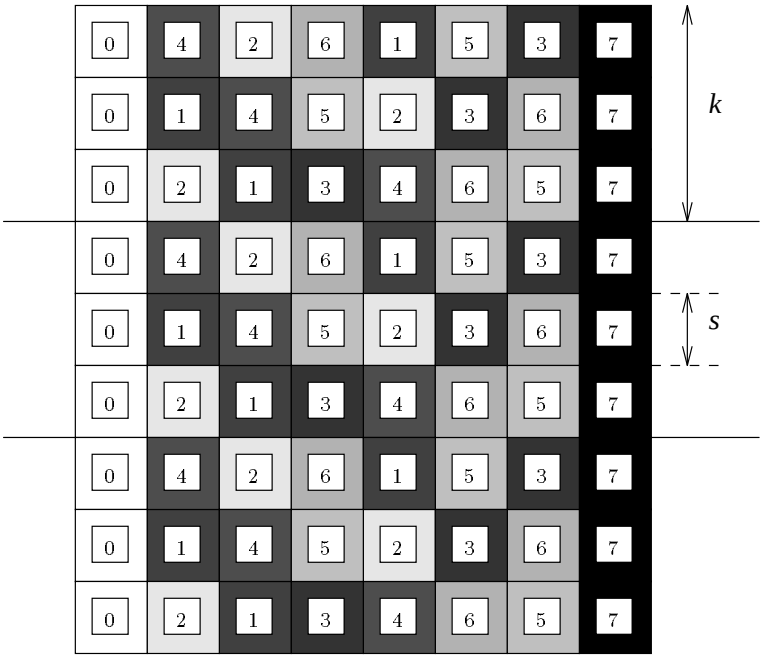


FIG. 3.18 - *Distribution d'une matrice $m \times n$ sur 8 processeurs avec $k = 3$ et des blocs de s lignes.*

L'algorithme ASYNC est donné dans la figure 3.19. Dans cet algorithme nous notons :

- `fft1d_seq(frow,L,s)` la routine séquentielle de calcul d'une FFT mono-dimensionnelle sur un bloc de `s` lignes de taille `L` commençant à la ligne `frow`.
- `Update(frow,L,s,step)` l'opération de mise à jour sur le paquet de lignes reçu à l'étape `step`, c'est-à-dire le calcul d'un papillon.

- `NBexchange(nbr,frow,s,L,step)` l'échange non-bloquant entre le processeur appelant le processeur `nbr` d'un paquet de `s` lignes de taille `L` commençant à la ligne `frow` au cours de l'étape `step`.

- `wait(`j`)` est la fonction d'attente de réception d'un message (*cf* partie I, chapitre 2).

Il faut remarquer que nous sommes capables de savoir quel est le lien à utiliser dans l'opération si nous connaissons le numéro de ligne `frow` et l'étape. Cet algorithme est bien sûr valable dans le cas de liens 1-port (*cf* partie I, chapitre 2) en travaillant sur deux blocs de s lignes à la fois.

```
Procédure fft1d_async(frow,k,s,n,P)
début
  pour i = 1 à k faire
    fft1d_seq(frow + (i − 1)s, n/P, s)
    j=NBexchange(nbr, frow + (i − 1)s, s, n/P, 0)

  pour step = 1 à log2(P) − 1 faire
    pour i = 1 à s faire
      wait(j)
      Update(frow + (i − 1)s, n/P, s, step)
      j=NBexchange(nbr, frow + (i − 1)s, s, n/P, step)

  pour i = 1 à k faire
    wait(j)
    Update(frow + (i − 1)s, n/P, s, log2(P))
fin
```

FIG. 3.19 - *Algorithme ASYNC.*

Nous présentons dans la figure 3.20 le schéma d'exécution d'un processeur avec un modèle 2-ports.

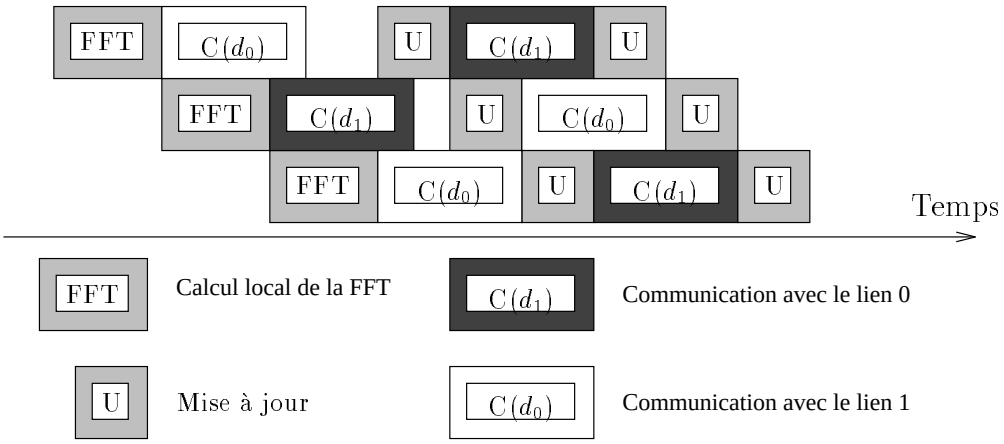


FIG. 3.20 - *Schéma d'exécution d'un processeur avec $k = 2$.*

Nous détaillons maintenant la complexité de cet algorithme. Nous supposons que nous calculons la FFT1D par blocs de s lignes et k blocs en même temps. Le temps total de l'algorithme est le suivant :

$$T_{topo}^{ASYNC} = s(k+1)\frac{n}{P}\log_2\left(\frac{n}{P}\right)t_a + \sum_{i=0}^{\log_2(P)-1}\left[\max\left(d_i\left(\beta + s\frac{n}{P}\tau\right), ks\frac{n}{P}t_a\right) + s\frac{n}{P}t_a\right]$$

où d_i est la distance entre un processeur et celui avec lequel il doit échanger ses données à l'étape i (par exemple, $d_i = 1$, $i = 0 \ldots \log_2(P) - 1$, pour l'hypercube) (*cf* algorithme de la figure 3.19).

Nous calculons maintenant le ratio calcul/communication. Afin d'obtenir le maximum de recouvrement, la contrainte ci-après doit être vérifiée :

$$sk\frac{n}{P}t_a \geq d_{max}\left(\beta + s\frac{n}{P}\tau\right) \qquad d_{max} = \max_{i=0}^{\log_2(P)-1}(d_i) \tag{3.1}$$

A partir de cette équation nous déduisons le nombre s de lignes à calculer à la même étape :

$$s \geq \left\lceil \frac{d_{max}P\beta}{n(kt_a - d_{max}\tau)} \right\rceil \tag{3.2}$$

k doit être égal au nombre de ports de communication utilisable en parallèle.

Nous détaillons dans la table 3.4 la complexité de l'algorithme ASYNC et la taille opti-

male du bloc de lignes, donnée par l'équation 3.2, pour chacune des topologies considérées.

Topologie	d_{max}	s
hypercube	1	$\left\lceil \frac{P\beta}{n(kt_a-\tau)} \right\rceil$
Grille	$\sqrt{P}$	$\left\lceil \frac{2P\sqrt{P}\beta}{n(kt_a-\sqrt{P}\tau)} \right\rceil$
Tore	$\frac{\sqrt{P}}{2}$	$\left\lceil \frac{P\sqrt{P}\beta}{n(kt_a-\frac{\sqrt{P}}{2}\tau)} \right\rceil$

TAB. 3.4 - *Détails des coûts de communication de la taille optimale de paquets pour chacune des topologies.*

Nous pouvons vérifier que le temps total pour calculer m FFT mono-dimensionnelles avec recouvrement total des communications pour n'importe quel réseau k-ports est de :

$$T_{topo}^{ASYNC}(m) = m \times \frac{n}{P}\log_2(n)t_a \tag{3.3}$$

Ce temps correspond exactement au temps de calcul. Remarquons que $1 \leq k \leq 2$ pour la grille et que $1 \leq k \leq 4$ pour le tore.

3.11 Application aux algorithmes de FFT bi-dimensionnelles

Nous allons décrire maintenant de nouvelles versions d'algorithmes de FFT bi-dimensionnelles permettant de recouvrir les communications.

3.11.1 Méthode TS avec recouvrement

Dans la méthode *Transpose Split*, la transposition de matrices correspond à un échange total personalisé, appelé aussi multi-distribution (*cf* partie I, chapitre 2 et partie II, chapitreCHAP:TRANSPO) [82, 108]. Si cette opération n'est pas recouverte par du calcul cela entraîne des pertes d'efficacité de l'algorithme parallèle. Celles-ci sont plus ou moins importantes suivant la rapidité du réseau de communication, comme on pourra le voir dans le paragraphe dédié aux expérimentations.

Algorithme général

Nous pouvons améliorer l'algorithme TS en recouvrant la phase de transposition de la matrice par le calcul local des FFT mono-dimensionnelles. En effet, nous pouvons commencer à communiquer un bloc de s lignes dès qu'il a été calculé et, au même moment, calculer une

FFT mono-dimensionnelle sur un autre bloc de lignes. Cet algorithme est exprimé dans la figure 3.21.

```
Procédure Transpose_Split_Overlapped
début

  fft1d_seq(X[0][],n)

  pour i = 1 à n/s − 1 faire
    Transpose(X[(i − 1) × s][],X^t[(i − 1) × s][])
    fft1d_seq(X[i × s][],n)

  Transpose(X[(n/s − 1) × s][],X^t[(n/s − 1) × s][])
  pour i = 0 à n/P − 1 faire
    fft1d_seq(X^t[i][],n)

fin
```

FIG. 3.21 - *Algorithme Transpose Split avec recouvrement.*

Complexité et implémentations sur diverses topologies

Nous notons $T^{com}_{topo}(n)$ le coût de communication d'un vecteur de taille n correspondant à une multi-distribution sur la topologie *topo*. Dans la figure 3.22 nous présentons les schémas d'exécution possibles d'un processeur avec deux blocs de lignes. Il y a trois étapes dans ce schéma :

- P_1: La première phase est le calcul d'une FFT mono-dimensionnelle sur s vecteurs de dimension n. Aussi le coût de cette phase est égal à :
  $T_1 = sT^{FFT-1d}_{seq}(n)$

- P_2: La seconde phase est le recouvrement de la communication de $\frac{n/P}{s}$ blocs avec le calcul de FFT mono-dimensionnelles sur ces blocs. Le coût de cette phase est donc égal à :
  $T_2 = \left(\frac{n/P}{s} - 1\right) \max\left(sT^{FFT-1d}_{seq}(n), T^{comm}_{topo}(n \times s)\right)$

- P_3: La dernière phase correspond à la communication du dernier bloc

Donc le coût de l'algorithme précédent avec des blocs de taille s est de :

$$\begin{aligned} T^{TSover}_{topo} &= sT^{FFT-1d}_{seq}(n) + \\ &\quad \left(\frac{n/P}{s} - 1\right) \max\left(sT^{FFT-1d}_{seq}(n), T^{comm}_{topo}(n \times s)\right) + \\ &\quad T^{comm}_{topo}(n \times s) + \frac{n}{P} T^{FFT-1d}_{seq}(n) \end{aligned}$$

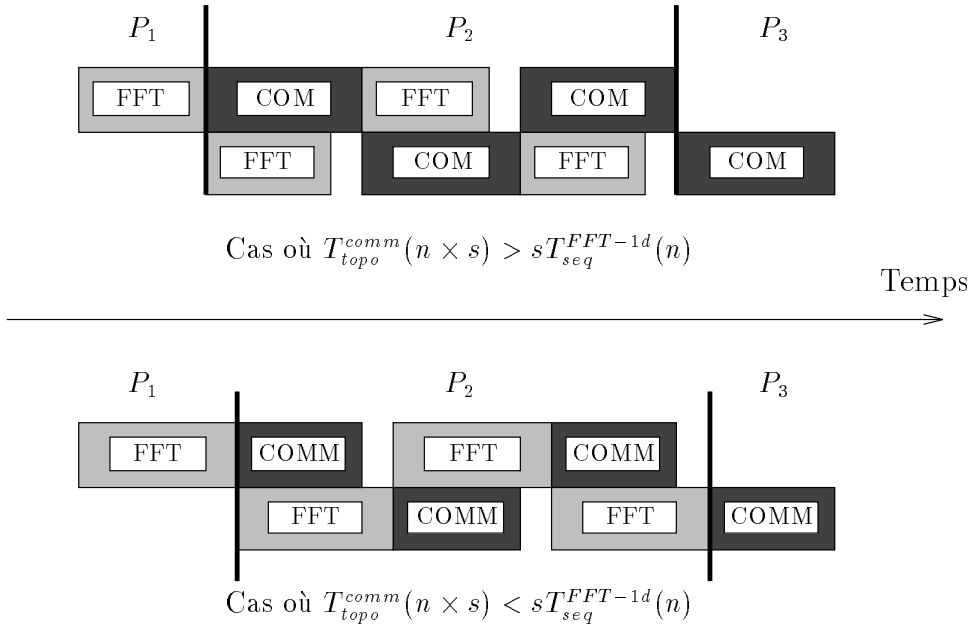


FIG. 3.22 - *Schémas d'exécution d'un processeur avec $s = 2$ pour la méthode TS avec recouvrement.*

Afin d'obtenir un recouvrement maximal, il faut que :

$$sT_{seq}^{FFT-1d}(n) \geq T_{topo}^{comm}(n \times s), \text{ où } T_{seq}^{FFT-1d}(n) = n \log_2(n).$$

Les complexités des différents algorithmes et les tailles optimales de paquets pour chaque topologie sont données dans la table 3.5.

Topologie	$T_{topo}^{comm}(n \times s)$	s
Grille	$2\sqrt{P}\beta + \frac{P\sqrt{P}ns\tau}{4}$ [108]	$\frac{2\sqrt{P}\beta}{n\left(log_2(n)t_a - \frac{P\sqrt{P}\tau}{4}\right)}$
Tore	$\sqrt{P}\beta + \frac{P\sqrt{P}ns\tau}{8}$[108]	$\frac{\sqrt{P}\beta}{n\left(\log_2(n)t_a - \frac{P\sqrt{P}\tau}{8}\right)}$
Hypercube	$\log_2(P)\beta + \frac{ns\tau}{2}$[82]	$\frac{\log_2(P)\beta}{n\left(log_2(n)t_a - \frac{\tau}{2}\right)}$

TAB. 3.5 - *Détails des coûts de communication et des tailles de paquet pour chaque topologie.*

Par conséquent, avec un recouvrement total, la complexité de cet algorithme est la suivante :

$$\boxed{T_{topo}^{TSover} = 2\frac{n^2}{P}\log_2(n)t_a + T_{topo}^{comm}(n \times s)}$$

Remarque : puisque nous ne pouvons recouvrir une phase de communication de taille $n \times s$, s doit être pris le plus petit possible, tout en respectant les conditions données dans la table 3.5.

3.11.2 Méthode LD avec recouvrement

Afin de recouvrir les communications par du calcul, nous utilisons l'algorithme ASYNC pour calculer les FFT mono-dimensionnelles distribuées de la seconde phase (*cf* voir algorithme figure 3.11). L'algorithme avec recouvrement est exprimé dans la figure 3.23.

```
Procédure Local_Distributed_Overlapped
début

  fft1d_seq(X[0][],n)

  fft1d_async(X[][0],k,s,n,P)

efin
```

FIG. 3.23 - *Algorithme de la méthode LD avec recouvrement.*

3.11.3 Méthode Bl avec recouvrement

Pour cette méthode le recouvrement est réalisé avec l'algorithme ASYNC pour les deux phases (*cf* figure 3.24).

```
Procédure Block_Overlapped
début

  fft1d_async(X[0][],k,s,n,P_2)

  fft1d_async(X[][0],k,s,n,P_1)

fin
```

FIG. 3.24 - *Algorithme de la méthode Bl avec recouvrement.*

3.12 Expérimentations

Dans toutes les tables résumant les temps d'expérimentation, nous comparons les temps des méthodes avec et sans recouvrement. Pour cela nous détaillons le temps de calcul (T_a) et le sur-coût de communication (T_c) pour la méthode sans recouvrement, ceci afin de pouvoir « mesurer » le gain possible en masquant les communications. Nous indiquons également le pourcentage de recouvrement, c'est à dire quelle proportion du sur-coût dû aux communications a pu être gagnée par rapport à la version sans recouvrement. Tous les temps sont donnés en secondes.

3.12.1 Cas mono-dimensionnel

Nous avons implanté les différentes versions des algorithmes de FFT mono-dimensionnelle sur un Cray T3D. Les résultats de ces expérimentations sont résumés dans la table 3.6 et la figure 3.25.

	Sans rec.			Avec rec.	% rec.
n	T_a	T_c	T	T	
16384	1.24	0.44	**1.68**	**1.51**	39%
32768	2.88	0.75	**3.63**	**3.21**	56%
65536	7.69	1.32	**9.01**	**8.13**	67%
131072	16.00	1.80	**17.80**	**16.50**	72%
262144	38.00	2.20	**40.20**	**39.00**	55%
524288	79.70	6.70	**86.40**	**81.60**	72%
1049376	166.90	10.8	**177.70**	**167.50**	95%

TAB. 3.6 - *Comparaison des algorithmes avec redistribution sur T3D (P=32).*

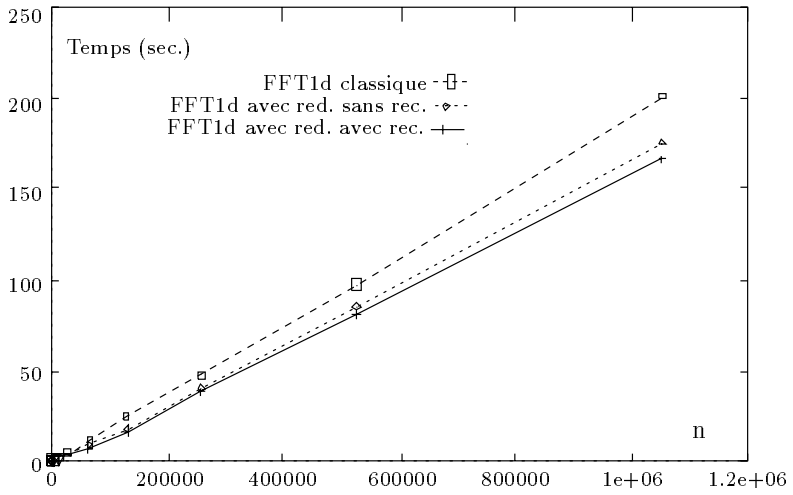


FIG. 3.25 - *Comparaison des trois versions d'algorithmes sur T3D (P=32).*

Comme on peut le constater, nous obtenons un très bon recouvrement des communications. La figure 3.25 compare les différentes versions des algorithmes. La version avec redistribution est plus performante que la version classique, et les résultats sont encore améliorés avec la version avec recouvrement.

3.12.2 Cas bi-dimensionnel

Nous avons implanté les méthodes présentées sur un iPSC/860 à 8 processeurs (3-cube), sur une Paragon à 16 processeurs (grille 4 par 4), sur un IBM SP1 à 16 nœuds (réseau

multi-étages) et un Cray T3D à 128 processeurs (tore 4 par 8 par 4).

Nous présentons, tout d'abord, les temps d'exécution des méthodes TS et LD sur l'iPSC (*c.f.* tables 3.7 et 3.8). Nous remarquons en premier lieu des différences de temps de communication dans l'une et l'autre méthode pour des temps de calcul équivalents. Pour plus de détails dans la comparaison des différents algorithmes, le lecteur intéressé pourra se reporter à l'article de Chu [36].

	Sans rec.			Avec rec.	% rec.
n	T_c	T_a	T	T	
128	0.057	0.121	**0.178**	**0.184**	0%
256	0.208	0.581	**0.789**	**0.675**	55%
512	1.063	2.612	**3.685**	**3.297**	36%
1024	3.623	11.601	**15.224**	**13.402**	50%
2048	16.130	51.028	**67.158**	**56.568**	65%

TAB. 3.7 - *Algorithme TS avec et sans recouvrement sur iPSC/860 avec 8 processeurs.*

	Sans rec.			Avec rec.	% rec.
n	T_c	T_a	T	T	
128	0.632	0.132	**0.764**	**0.583**	29%
256	0.915	0.637	**1.552**	**1.144**	45%
512	1.577	2.783	**3.360**	**3.292**	68%
1024	3.659	11.998	**15.657**	**12.944**	74%
2048	14.924	53.998	**68.922**	**57.544**	76%

TAB. 3.8 - *Algorithme LD avec et sans recouvrement sur iPSC/860 avec 8 processeurs.*

Comme nous pouvons le noter en examinant ces deux tables, le pourcentage de communication recouvert est important. Qu'en est-il du gain en temps d'exécution? Il est clair qu'il est d'autant plus intéressant que le rapport $\frac{T_c}{T_a}$ est important (voir la figure 3.26).

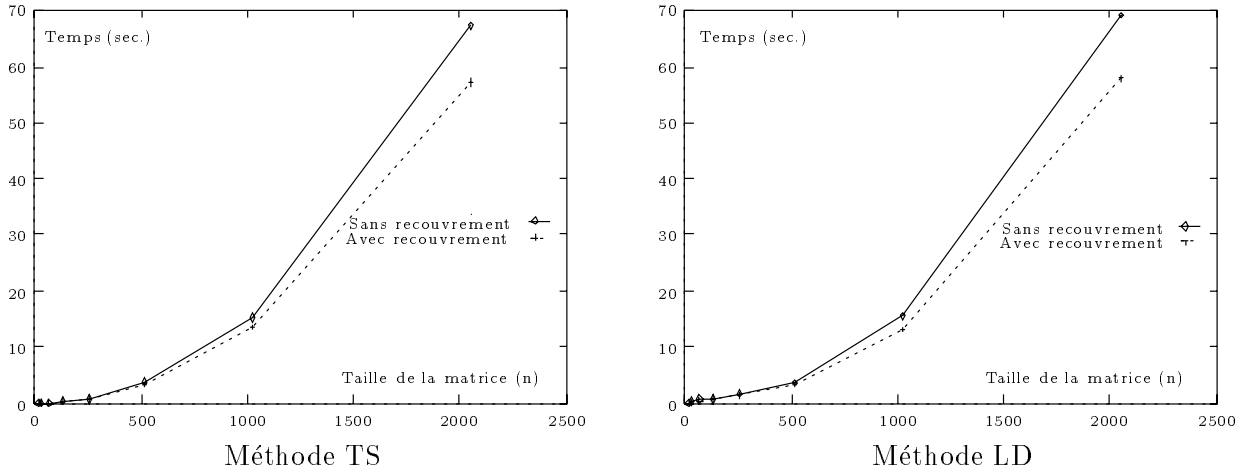


FIG. 3.26 - *Comparaison des temps d'exécutions des méthodes avec et sans recouvrement sur iPSC.*

Nous avons également expérimenté ces différentes méthodes sur une Paragon configurée en grille 4×2 et 4×4. Les résultats de ces expérimentations sont contenus dans la figure 3.27 sur laquelle nous avons visualisé la courbe du temps de calcul T_a. Comme on peut le constater, les courbes des temps d'exécution des méthodes avec recouvrement sont proches des courbes des temps de calcul, ce qui montre que l'on obtient un bon masquage du sur-coût des communications grâce aux méthodes avec recouvrement.

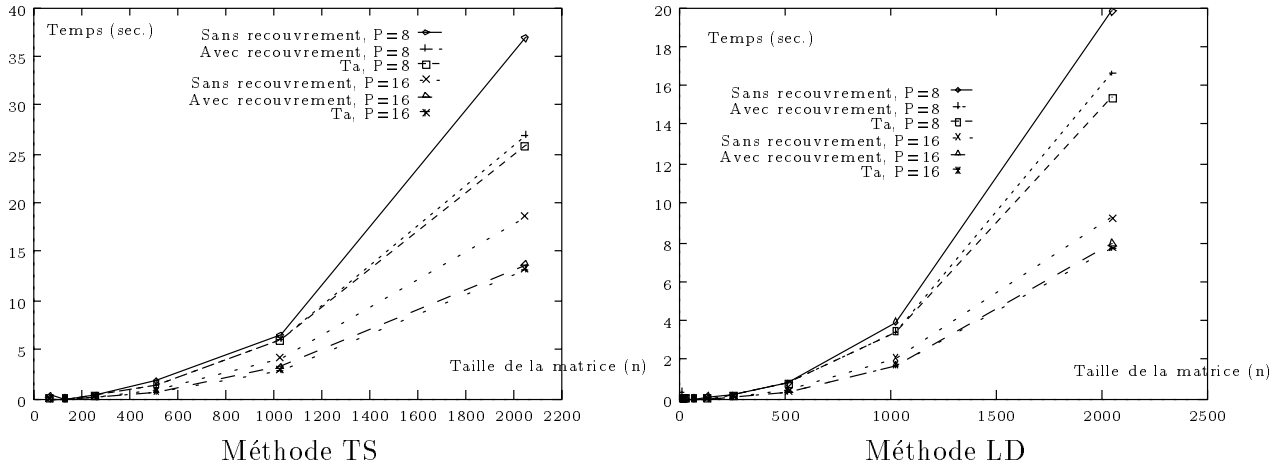


FIG. 3.27 - *Comparaison des méthodes avec et sans recouvrement pour les algorithmes TS et LD sur une Paragon avec 8 (grille 4×2) et 16 (grille 4×4) nœuds.*

Nous avons résumé dans la table 3.9 les résultats obtenus sur un SP1 à 16 processeurs. Comme on peut le constater, le recouvrement est très faible. Ceci s'explique par le fait qu'il

n'existe pas de mécanisme matériel déchargeant le processeur de calcul de la communication. Le seul gain obtenu provient du fait que l'on comble en partie le temps d'inactivité des processeurs.

	Sans rec.			Avec rec.	% rec.
n	T_a	T_c	T	T	
256	0.082	0.126	**0.212**	**0.241**	0%
512	0.396	1.066	**1.462**	**1.242**	20%
1024	2.818	4.53	**6.579**	**7.348**	0%
2048	12.996	9.668	**22.664**	**21.011**	18%

TAB. 3.9 - *Méthode TS sur SP1 avec 16 processeurs.*

En revanche nous obtenons de très bons recouvrements sur le T3D (tables 3.10 et 3.11) avec les deux algorithmes TS et LD. Il est intéressant de noter que, bien que le temps de communication soit plus important dans l'algorithme TS que dans l'algorithme LD, nous avons de meilleurs masquages dans le deuxième cas que dans le premier. Ceci s'explique par deux raisons : la première est que dans une transposition (algorithme TS) le temps de préparation et de mise en place des messages est très important (nous prenons en compte en effet le temps de préparation du message à envoyer ainsi que la transposition locale des données) et ce temps n'est pas masquable car c'est au processeur de calcul de le faire. La deuxième raison est que, dans l'algorithme LD, nous travaillons sur une granularité de calcul plus fine, ce qui influe sur le temps de calcul lui-même, ainsi le temps de calcul dans la méthode avec recouvrement est inférieur à celui de l'algorithme sans recouvrement [3].

	Sans rec.			Avec rec.	% rec.
n	T_a	T_c	T	T	
256	0.040	0.090	**0.130**	**0.100**	33%
512	0.240	0.190	**0.530**	**0.380**	26%
1024	1.040	3.400	**4.440**	**1.680**	81%
2048	4.610	14.020	**18.630**	**7.380**	80%

TAB. 3.10 - *Algorithme TS sur T3D avec 32 processeurs.*

	Sans rec.			Avec rec.	% rec.
n	T_a	T_c	T	T	
256	0.060	0.080	**0.140**	**0.100**	50%
512	0.280	0.130	**0.410**	**0.330**	60%
1024	1.100	0.650	**1.750**	**1.290**	70%
2048	5.450	2.080	**7.530**	**5.600**	93%

TAB. 3.11 - *Algorithme LD sur T3D avec 32 processeurs.*

3. Ceci est essentiellement dû au mode de fonctionnement du cache du processeur Alpha qui constitue le nœud de calcul du T3D

3.12.3 Le calcul de la taille de paquets

Les temps d'exécutions donnés pour les algorithmes avec recouvrement ont été obtenus en faisant varier la taille de paquet, qui est le paramètre de recouvrement, et en prenant le meilleur temps d'exécution. Ceci a été fait pour toutes les machines sauf pour le T3D. Nous avons pu remarquer, cependant, que cette taille de paquet dépendait très fortement de la taille du problème par rapport au nombre de processeurs, ce qui est encourageant pour calculer pratiquement une taille de paquet. Ceci n'a pas éte fait car nous n'avons pas une modélisation assez précise des performances de communication et de calcul pour ces machines.

En ce qui concerne le T3D, nous nous sommes servis bien sûr de l'étude qui a été présentée dans le chapitre 2 de cette même partie. Le rapport calcul/communication est relativement petit. En effet les communications sont plus rapides que les calculs (dans ce cas précis du calcul de FFT); ce qui implique que le recouvrement est réalisé pour d'assez petites tailles de paquets (voir les formules de calcul de tailles de paquets dans les paragraphes précédents, sections 3.10 et 3.11). Cependant, comme les temps d'initialisation des communications ne peuvent être masqués, le temps d'exécution minimum sera atteint pour un temps d'initialisation minimum. Ceci est vérifié dans le cas du T3D pour une taille de message égale à la constante `PVM_DATA_MAX` (voir le chapitre 2 de cette même partie).

Ce raisonnement s'est avéré exact en ce qui concerne l'algorithme TS, car dans ce cas le temps élémentaire de calcul ne dépend que de la taille des données. Dans le cas de l'algorithme LD, nous n'obtenons qu'un minimum local du temps d'exécution. En effet, pour cet algorithme, le temps de calcul va varier non seulement suivant la taille des données mais également en fonction de la granularité du calcul (ceci étant dû aux problèmes de cache sur cette machine).

3.13 Conclusion

Nous avons montré dans ce chapitre comment il était possible de recouvrir efficacement les communications dans un algorithme numérique qui est le calcul d'une transformée de Fourier.

Cet algorithme est une très bonne illustration des différentes méthodes générales pour recouvrir les communications présentées lors du premier chapitre. Nous avons conçu de nouveaux algorithmes de calcul de transformée de Fourier mono et bi-dimensionnelles qui recouvrent de façon maximale les communications. Ces algorithmes on été implantés sur de nombreuses machines parallèles à mémoire distribuée. Les expérimentations mettent en évidence un recouvrement des communications généralement très important (pour les machines offrant un mécanisme physique permettant au processeur de calcul de se décharger de la communication).

De plus, nous avons pu faire le lien entre le calcul théorique de la taille de paquets et le calcul pratique dans le cas du T3D, en utilisant les modélisations des temps de communication développées au chapitre précédent. Cependant il s'avère, comme nous pouvions le prévoir, qu'une modélisation très précise du temps de calcul est nécessaire pour obtenir des performances « optimales ».

Conclusion

Les machines parallèles à mémoire distribuée sont destinées à être utilisées de plus en plus en calcul scientifique. Dans cette optique, il est indispensable de fournir aux utilisateurs des outils et des techniques simples à mettre en œuvre afin d'obtenir de bonnes performances sur ce type de machine.

La parallélisation efficace d'applications et d'algorithmes numériques passe par une répartition équilibrée de la charge de calcul et par la minimisation du temps de « non-calcul ». Ce **sur-coût de communication**, a été défini comme le temps passé par le processeur de calcul à exécuter (respectivement ne pas exécuter) des instructions ne contribuant pas (respectivement contribuant) directement au calcul dont il a la charge. Ce temps « perdu » est dû à des attentes de données nécessaires à la poursuite du calcul et à des temps de transit et de gestion de messages.

Nous avons étudié, dans ce mémoire, deux voies, agissant à des niveaux différents, qui contribuent à la minimisation de ce sur-coût de communication. La première consiste à **optimiser l'algorithme de communication**, quant à la seconde, elle agit au niveau de l'algorithme parallèle lui-même pour **recouvrir les communications par du calcul**.

L'optimisation des schémas de communication globale

Comme illustration de la première méthode, nous avons proposé de nouveaux algorithmes efficaces de communication pour deux classes de problèmes très utiles dans la parallélisation de noyaux numériques : **la transposition de matrices allouées par blocs**, et **l'échange total sur les grilles toriques à deux-dimensions**.

Le premier de ces problèmes a été initialement étudié pour un mode de type commutation de messages, puis étendu au *wormhole*. Les solutions proposées reposent sur **une méthodologie générale basée sur le partitionnement du graphe** modélisant le réseau d'interconnexion, en cycles ou chemins. Nous avons appliqué sur ces derniers des schémas de communication élémentaires. Cette méthodologie a été appliquée aux grilles toriques et aux réseaux de de Bruijn et a conduit à des algorithmes optimaux en nombre d'étapes et asymptotiquement optimaux en bande passante pour le protocole *store-and-forward*. L'adaptation directe au *wormhole* mène à des algorithmes efficaces, mais qui ne profitent pas complètement des possibilités offertes par cet autre mode de commutation.

Le deuxième problème étudié a été le schéma d'échange total sur des grilles toriques en mode *wormhole*. La méthodologie adoptée repose sur **une décomposition récursive**

à arêtes disjointes du graphe. Elle permet l'écriture d'un nouvel algorithme optimal en nombre d'étapes pour ce mode de commutation. Nous avons proposé une variante de cet algorithme, qui tout en ayant un nombre faible d'étapes, améliore de façon non négligeable le taux de transmission. Des simulations de ces algorithmes, utilisant des paramètres réels de machine parallèle, ainsi que la comparaison avec des temps mesurés, confirment l'intérêt de concevoir des algorithmes efficaces de communication globale.

L'étude de méthodes de recouvrement des communications dans les algorithmes numériques parallèles

La deuxième méthode de minimisation consiste à masquer les communications en travaillant sur l'algorithme parallèle. Nous avons étudié des schémas algorithmiques sur lesquels nous proposons des méthodes de recouvrement des communications basées sur **un ré-ordonnancement local des tâches**, **des entrelacements de phases de calcul indépendantes** et **des techniques de pipeline**. Ces méthodes sont paramétrables en fonction de coûts élémentaires de calcul et de communication, qui eux, dépendent de la machine cible sur laquelle l'utilisateur désire implanter les algorithmes parallèles avec recouvrement. Elles peuvent également servir de base à des heuristiques utilisables dans des compilateurs de langages *data-parallel* afin d'optimiser le code parallèle généré.

Pour obtenir un recouvrement maximal au niveau de l'implantation, il est indispensable d'avoir la connaissance la plus précise possible des coûts élémentaires de calcul et de communication. Dans ce but, nous avons effectué **une analyse expérimentale des performances des communications** sur T3D de Cray, et nous proposons **des modèles de temps pour la plupart des schémas de communication globale**, en fonction de la taille des messages, et/ou du nombre de processeurs.

Afin d'illustrer les diverses méthodes de recouvrement proposées, nous avons étudié **différents algorithmes de calcul de transformée de Fourier mono et bi-dimensionnelle avec masquage des communications**. Pour chacun de ces nouveaux algorithmes, nous avons calculé les paramètres théoriques permettant d'obtenir un recouvrement maximal des communications. Ces algorithmes ont été implantés sur plusieurs machines parallèles, à architecture et caractéristiques différentes. Les expérimentations ont montré le gain apporté par les méthodes avec recouvrement. Le lien entre le calcul des paramètres théoriques de recouvrement et la modélisation des performances de communication, a été effectuée dans le cas du T3D.

La contribution principale de ce mémoire a été d'étudier de façon théorique et expérimentale deux méthodes de minimisation du sur-coût de communication, agissant à des niveaux différents dans la parallélisation. Nous avons notamment montré que l'utilisation de nouveaux outils dans la conception des algorithmes de communication globale qui tiennent compte des possibilités des modes de commutation, conduit à de très bonnes performances. Nous avons également dégagé des méthodologies algorithmiques générales permettant de

recouvrir les communications dans les algorithmes numériques parallèles. Nous avons expérimenté celles-ci sur un certain nombre d'exemples pratiques. Nous avons notamment mis en évidence le fait que les techniques de pipeline classiques ne sont pas suffisantes. Il nous semble que les méthodes de parallélisation classiques qui consistent à optimiser d'un côté le code de calcul, et améliorer les performances des communications de l'autre, ne permettent pas d'obtenir le maximum d'efficacité possible. Le fait de considérer l'algorithme parallèle dans son ensemble, et ainsi de mélanger les phases de calcul et de communication, apportent de bien meilleures performances.

Ces deux méthodes de minimisation sont véritablement complémentaires et apportent des gains considérables dans la parallélisation d'applications numériques. L'optimisation des communications globales s'avère de plus plus en plus intéressante depuis que ces dernières sont spécifiées dans les standards de bibliothèques de communication. De plus, le recouvrement de ces communications est d'autant plus aisé que le temps nécessaire pour les réaliser est faible.

Perspectives

Il n'existe, à l'heure actuelle, que peu de travaux sur la conception d'algorithmes de communication globale en *wormhole* et commutation de circuits. Il serait intéressant d'étudier d'autres schémas sur d'autres réseaux, notamment sur des réseaux multi-étages qui constituent le système d'interconnexion de nombreuses machines parallèles (comme le SP2 d'IBM). Le développement des nouvelles technologies dans la conception des réseaux, comme l'optique ou les réseaux à bus, ouvre de nouveaux horizons pour la communauté des chercheurs spécialistes des communications[4]. Dans ce domaine beaucoup de choses restent à faire, comme la spécification de nouveaux protocoles, mais aussi la réalisation de schémas efficaces de communication globale.

Il nous semble également important de prendre en compte les algorithmes de routage dans la conception des schémas de communication, ceci en vue de leur implantation matérielle dans les machines.

Les différentes expérimentations des algorithmes avec recouvrement ont mis en évidence la nécessité d'avoir de très bonnes modélisations des performances des machines parallèles. Un début de travail a été effectué sur le T3D, avec la modélisation des temps de communication, mais il serait nécessaire d'étendre ces études à d'autres machines et de s'attaquer au problème de la modélisation des temps de calcul. La définition de « protocoles » de tests permettant d'établir de façon la plus précise possible les paramètres fondamentaux des machines parallèles, et leur valeur, nous semble être d'une très grande utilité.

Après avoir étudié le recouvrement des communications sur un nombre important d'algorithmes numériques, nous avons dégagé les principales méthodologies permettant de masquer les échanges de données. Il serait très intéressant d'étudier leur intégration dans un compila-

4. Ce sont les principaux thèmes d'études de l'opération RUMEUR du PRC - Parallélisme Réseaux et Systéme.

teur. Elles peuvent, en effet, servir de base à des heuristiques permettant d'optimiser le code généré. Des premières études et implémentations ont été effectuées dans différents compilateurs de langages *data-parallel* comme Fortran D, Paradigm ou encore HPF, mais nous n'en sommes encore qu'au début.

Nous nous sommes, dans un premier temps, restreints à une parallélisation guidée par la distribution des données. Il serait certainement utile, de prendre un modèle plus général, tel que le parallélisme de contrôle[5], afin d'exploiter de nouvelles possibilités pour recouvrir les communications. Nous pensons notamment à essayer de faire le lien avec des modèles théoriques d'ordonnancement [68]. Ceux-ci tiennent compte des communications et la possibilité de les masquer. Cependant, ces modèles sont encore trop loin de la réalité pour pouvoir les utiliser directement. Partant des méthodes de recouvrement étudiées, le but serait de voir leur extension par des techniques utilisées dans le domaine de l'ordonnancement.

Il est évident qu'un autre but est d'utiliser ces techniques de recouvrement pour la parallélisation efficace d'applications numériques. Des premières réalisations ont été effectuées sur un code de reconstruction d'images tri-dimensionnelles [23]. Ce travail s'intègre directement dans les objectifs de groupes de travail pluri-disciplinaires regroupant des scientifiques ayant de fortes demandes en puissance de calcul et des spécialistes de parallélisme en vue de la parallélisation efficace de leurs applications[6].

5. Ce modèle de programmation est celui qui a été adopté au sein du projet APACHE, dont les principaux objectifs sont la conception et la mise au point d'un environnement de programmation pour machines parallèles [105].

6. Ce type d'initiative existe en région Rhônes-Alpes avec le groupe PARAPPLI ou encore à Bordeaux avec le groupe PARAMAP.

Bibliographie

[1] C. Addison, V. Getov, A. Hey, R. Hockney et I. Wolton, *The GENESIS Distributed Benchmarks*, in Computer Benchmarks, J. Dongarra et W. Gentzch, eds., North-Holland, 1993.

[2] ——, *Benchmarking for Distributed Memory Parallel Systems: Gaining Insight from Numbers*, Parallel Computing, (1994).

[3] A.J.G.Hey, *The GENESIS Distributed-Memory Benchmarks*, Parallel Computing, (1991), pp. 1275–1283.

[4] R. Allan et P. Lockey, *Survey of Parallel Software Packages of potential interest in Scientific Applications*, rapp. tech., Daresbury Laboratory, Warington UK, Nov. 1994. email: r.j.allan,p.lockey@daresbury.ac.uk.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney et D. Sorensen, *Lapack: a portable linear algebra library for high performance computers*, in Proceedings of Supercomputing'90, IEEE Press, 1990, pp. 1–10.

[6] M. Asworth et A. Lyne, *A Segmented FFT Algorithm for Vector Computers*, Parallel Computing, 6 (1988), pp. 217–224.

[7] A. Averbuch, E. Gabber, B. Gordissky et Y. Medan, *A Parallel FFT on an MIMD Machine*, Parallel Computing, 15 (1990), pp. 61–74.

[8] C. Aykanat et A. Dervis, *An Overlapped FFT Algorithm for Hypercube Multicomputers*, in International Conference on Parallel Processing, vol. 3, 1990, pp. 316–317.

[9] M. Barnett, S. Gupta, D. Payne, L. Schuler, R. van de Geijn et J. Watts, *Interprocessor Collective Communication Library (InterCom)*, in Proceedings of Scalable High Perfomance Computing Conference, IEEE Press, Mai 1994, pp. 357–364.

[10] M. Barnett, D. Payne et R. van de Geijn, *Optimal Broadcasting in Mesh-Connected Architectures*, rapp. tech., University of Texas, Austin, Texas 78712-1188, Déc. 1991.

[11] A. Beguelin, J. Dongarra, A. Geist, R. Manchek et V. Sunderam, *A User's Guide to PVM Parallel Virtual Machine (version 3)*, rapp. tech., Oak Ridge National Laboratory, Mai 1994.

[12] J. Bermond, P. Michallon et D. Trystram, *Broadcasting in wraparound meshes with parallel monodirectional links*, Parallel Computing Journal, 18 (1992).

[13] J. C. Bermond et C. Peyrat, *De Bruijn and Kautz networks: a competitor for the hypercube?*, in Proceedings of the 1st European Workshop on Hypercubes and Distributed Computers, Rennes, F. Andre and J.P. Verjus ed, North Holland, 1989, pp. 279–293.

[14] S. Bokhari, *Complete Exchange on the ipsc-860*, Technical Report 91-4, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center Hampton, Virginia, Jan. 1991.

[15] R. Boppana et C. Raghavendra, *Optimal Self-Routing of Linear-Complement Permutations in Hypercubes*, IEEE Transactions on Computers, (1990), pp. 800–808.

[16] A. Brass et G. Pawley, *Two and three dimensional FFTs on highly parallel computers*, Parallel Computing, 3 (1986), pp. 167–184.

[17] J. Bunch, J. Dongarra, C. Moler et G. Stewart, *LINPACK user's guide*, SIAM, 1979.

[18] R. Butler et E. Lusk, *User's Guide to the P4 Programming System*, Rapp. Tech. TM-ANL-92/17, Argonne National Laboratory, 1992.

[19] R. Butler et E. lusk, *Monitors, Messages , and Clusters: the P4 Parallel Programming System*, Parallel Computing, 20 (1994), pp. 547–564.

[20] R. Calkin, R. Hempel, H.-C. Hoppe et P. Wypior, *Portable Programming with the Parmacs Message-Passing Library*, Parallel Computing, special issue on message-passing interfaces, 20 (1994), pp. 615–632.

[21] C. Calvin, *Algorithmes parallèles de transformée de Fourier mono-dimensionnelle avec recouvrement des communications*, in Proceedings of RENPAR'7, 1995.

[22] ——, *Implementation of Parallel FFT Algorithms on Distributed Memory Machines with a Minimum Overhead of Communication*, rapp. tech., LMC - IMAG / APACHE, 1995. Submitted to "Journal of Parallel Computing".

[23] C. Calvin, J. Chassery, C. Laurent et F. Peyrin, *Efficient Implementation of Parallel Image Reconstruction Algorithms for 3D X-Ray Tomography*, in Proceedings of PARCO'95, 1995.

[24] C. Calvin et L. Colombet, *Introduction au modèle de programmation par processus communiquants deux exemples : PVM et MPI*, Rapp. Tech. Apache n^o12, LMC - IMAG / APACHE, Juil. 1994.

[25] ——, *Performance Evaluation and Modeling of Collective Communications on Cray T3D*, rapp. tech., LMC - IMAG / APACHE, 1995. Submitted to "Journal of Parallel Computing".

[26] C. Calvin, L. Colombet, F. Desprez, B. Jargot, P. Michallon, B. Tourancheau et D. Trystram, *Towards Mixed Computation/Communication in Parallel Scientific Libraries*, in Proceedings of CONPAR 94 - Linz - Austria, 1994.

[27] C. Calvin, L. Colombet et P. Michallon, *Overlapping Techniques of Communications*, in Proceedings of HPCN'95, Mai 1995.

[28] C. Calvin et F. Desprez, *Minimizing Communication Overhead Using Pipelining for Multi-Dimensional FFT on Distributed Memory Machines*, in Proceedings of PARCO'93, 1993.

[29] C. Calvin, S. Perennes et D. Trystram, *Gossiping in Torus with Wormhole-Like Routing*, in Proceedings of the "7^{th} IEEE Symposium on Parallel and Distributed Processing", 1995.

[30] C. Calvin et D. Trystram, *Matrix transpose for block allocations on torus and de bruijn networks*, Accepted in the "Journal of Parallel and Distributed Computing", (1995). Apache n^o2.

[31] R. Chamberlain, *Gray codes, Fast Fourier Transforms and hypercubes*, Parallel Computing, 6 (1988), pp. 225–233.

[32] J. Choi, J. Dongarra, R. Pozo et D. Walker, *ScaLAPACK: A Scalable Linera Algebra Library for Distributed Memory Concurrent Computers*, Rapp. Tech. LAPACK WN 55, University of Tennessee, 1992.

[33] J. Choi, J. Dongarra et D. Walker, *The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers*, in Environments and Tools for Parallel Scientific Computing, J. Dongarra et B. Tourancheau, eds., Elsevier Science Publishers, 1993, pp. 3–15.

[34] J. Choi, J. Dongarra et D. Walker, *Parallel Matrix Transpose Algorithms on Distrubuted Memory Concurrent Computers*, Rapp. Tech. ORNL/TM-12309, Oak Ridge National Laboratory, Oct. 1993.

[35] M. Christaller, *Athapascan-0a sur PVM3: définition et mode d'emploi*, Rapp. Tech. Apache 11, LMC - LGI - IMAG, Juin 1994.

[36] C. Y. Chu, *Comparison of Two-Dimensional FFT Methods on the Hypercube*, in The Third Conference on Hypercube Concurrent Computers and Applications, G. Fox, ed., vol. 2, 1988.

[37] L. Colombet, P. Michallon et D. Trystram, *Parallel Matrix-Vector Product on Rings with a minimum of Communications*, Research Report Apache 10, IMAG, Juin 1994.

[38] C. Cooley et J. Tuckey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput., 19 (1965), pp. 297–301.

[39] M. Cosnard et D. Trystram, *Algorithmes et Architectures Parallèles*, InterEditions, ed. iia, 1993.

[40] Cray Research Inc., *Cray T3D System Architecture Overview*, rapp. tech., 1993.

[41] ——, *PVM and HeNCE Programmer's Manual*, Rapp. Tech. SR-2501, 1993.

[42] ——, *ShMem user's Manual*, Rapp. Tech. SN-2517, 1993.

[43] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken et K. Yelick, *Parallel Programming in Split-C*, in Proceedings of the International Conference on SuperComputing, 1993.

[44] R. Cypher, A. Ho, S. Konstantinidou et P. Messina, *Architectural Requirements of Parallel Scientific Applications with Explicit Communication*, in Proceedings of the International Symposium on Computer Architecture, Mai 1993.

[45] W. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Davison et G. Fyler, *The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanism*, IEEE Micro, (1992), pp. 23–29.

[46] W. Dally et C. Seitz, *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*, IEEE Transactions on Computers, C-36 (1987), pp. 547–553.

[47] F. Desprez, *Procédures de Base pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée*, PhD thesis, Institut National Polytechnique de Grenoble, Jan. 1994.

[48] F. Desprez, J. Dongarra et B. Tourancheau, *Performance Complexity of LU Factorization with Efficient Pipelining and Overlap on a Multiprocessor*, Rapp. Tech. LAPACK Working Note 67, University of Tennessee, Fév. 1994.

[49] F. Desprez et B. Tourancheau, *LOCCS: Low Overhead Communication and Computation Subroutines*, Future Generation Computer Systems, 10 (1994), pp. 279–284.

[50] J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software*, Computer Science Technical Report CS-89-85, University of Tennessee, Mai 1989.

[51] A. Edelman, *Optimal Matrix Transposition and Bit Reversal on Hypercubes: All-to-All Personalized Communication*, rapp. tech., Thinking Machines Corporation, 1989.

[52] J. Edmonds, *Edges-disjoint branching, combinatorial algorithms*, Algorithms Press, 72.

[53] J. O. Eklundh, *A Fast Computer Method for Matrix Transposing*, IEEE Transactions on Computers, 21 (1972), pp. 801–803.

[54] A. Ferrari et V. Sunderam, *TPVM: Distributed Concurrent Computing with Lightweight Processes*, Rapp. Tech. CSTR-950201, Dept. of Math and Computer Science, Emory University, Atlanta, GA 30322, USA, Fév. 1995. ftp.mathcs.emory.edu:/pub/cstr.

[55] P. M. Flanders, *A Unified Approach to a Class of Data Movements on an Array Processor*, IEEE Transactions on Computers, 31 (1982), pp. 809–819.

[56] H. P. F. Forum, *High Performance Fortran language specification*, (version 1.0), 1993.

[57] I. Foster et P. H. Worley, *Parallelizing the Spectral Transform Method: A Comparison of Alternative Parallel Algorithms*, in Proceedings of the sixth SIAM conference on Parallel Processing For Scientific Computing., vol. 1, SIAM, 1992, pp. 100–107.

[58] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng et M. Wu, *Fortran D Language specification*, Rapp. Tech. TR90-141, Center for Research on Parallel Computation, Rice University, Déc. 1990.

[59] P. Fraignaud, *Communications Intensives dans les Architectures à Mémoire Distribuée et Algorithmes Parallèles pour la Recherche de Polynômes*, PhD thesis, LIP - ENSL, Déc. 1990.

[60] P. Fraigniaud et E. Lazard, *Methods and problems of communication in usual networks*, "Discrete Applied Mathematics", special issue on broadcasting and gossiping, 53 (1994).

[61] H. Franke, P. Hochschild, P. Pattnaik et M. Snir, *An Efficient Implementation of MPI*, in Proceedings of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, 1994.

[62] H. Fredricksen, *A new look at the de Bruijn graph*, Discrete Applied Mathematics, 37/38 (1992), pp. 193–203.

[63] D. B. Gannon et J. V. Rosendale, *On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms*, IEEE Transactions on Computers, 33 (1984), pp. 1180–1194.

[64] G. Geist, M. Hetah, B. Peyton et P. Worley, *PICL: A Portable Instrumented Communication Library.*, Rapp. Tech. ORNL/TM-11130, Oak Ridge National Laboratory, Juil. 1990.

[65] R. L. Graham, D. E. Knuth et O. Patasnik, *Concrete Mathematics - A Fundation for Computer Science*, Addison Wesley, Oct. 1990.

[66] W. Gropp, L. C. McInnes et B. Smith, *The PETSc package*, rapp. tech., Argonne National Laboratory, 1994. W3 site: http://www.mcs.anl.gov/home/gropp/petsc.html.

[67] W. Gropp et B. Smith, *Users Manual for the Chameleon Parallel PRogramming Tools*, Rapp. Tech. ANL93/30, Argonne National Laboratory, Août 1993.

[68] F. Guinand, *Ordonnancement avec communications pour architectures mulitprocesseurs dans divers modèles d'exécution*, PhD thesis, Institut National Polytechnique de Grenoble, 1995.

[69] A. Gupta et V. Kumar, *The Scalability of FFT on Parallel Computers*, IEEE Transactions on Parallel and Distributed Systems, 4 (1993), pp. 922–932.

[70] S. Gupta, C. Huang et P. Sadayappan, *Implementing Fast Fourier Transforms on Distributed-Memory Multiprocessors Using Data Redistributions*, Parallel Processing Letters, (1994).

[71] B. Hendrickson, R. Leland et S. Plimpton, *An Efficient Parallel Algorithm for Matrix-Vector Multiplication*, rapp. tech., Sandia National Laboratories, Albuquerque, NM 87185, 1994. Published in Intl. J. High Speed Comput.

[72] C. Ho, *Matrix Transpose on Meshes with Wormhole and XY Routing*, rapp. tech., IBM Research Division Almaden Research Center, Juin 1993.

[73] C. Ho et M. Raghunath, *Efficient communication primitives on hypercubes*, rapp. tech., IBM Almaden Research Center, Jan. 1991.

[74] C. Hoare, *Communicating Sequential Processes*, Communications of the ACM, 21 (1978), pp. 666–677.

[75] R. Hockney, *The communication challenge for MPP: Intel Paragon and Meiko CS-2*, Parallel Computing, (1994), pp. 389–398.

[76] K. Hwang, *Advanced Computer Architecture - Parallelism, Scalability, Programmability*, Mc Graw-Hill and MIT Press, 1993.

[77] IBM Corporation, *IBM AIX PVMe User's Guide and Subroutine Reference*, Mars 1994.

[78] Jean De Rumeur, *Communications dans les Réseaux de Processeurs*, Masson, 1994.

[79] S. Johnsson, *Massively Parallel Computing: Data distribution and communication*, in Proceedings of the First Heinz Nixdorf Symposium on "Parallel Architectures and Their Efficient Use", A. R. F. Meyer, B. Monien, ed., Springer-Verlag, Nov. 1992.

[80] S. Johnsson et C. Ho, *Algorithms for Matrix Transposition on Boolean n-cube Configured Ensemble Architectures*, in 1987 International Conference on Parallel Processing, S. K. Sahni, ed., Août 1987, pp. 621–629.

[81] S. L. Johnsson, *Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures*, Journal of Parallel and Distributed Computing, 4 (1987), pp. 133–172.

[82] S. L. Johnsson et C. Ho, *Optimum Broadcasting and Personalized Communications in Hypercubes*, IEEE Transactions on Computers, 38 (1989), pp. 1249–1267.

[83] S. L. Johnsson et R. L. Krawitz, *Cooley-Tuckey FFT on the Connection Machine*, Parallel Computing, 18 (1992), pp. 1201–1221.

[84] S. L. Johnssson et C. T. Ho, *Shuffle Permutations on Boolean Cubes*, rapp. tech., Yale University Department of Computer Science, Oct. 1988.

[85] K. Kennedy, *Parallel Computing: What we did wrong and what we did right*, HPCwire, (1995).

[86] P. Kermani et L. Kleinrock, *Virtual Cut-Trough: A New Computer Communication Switching Technique*, Computer Networks, 3 (1979), pp. 267–286.

[87] C. T. King, W. H. Chu et L. M. Ni, *Pipelined data parallel algorithms - Concept and modeling*, in Proceedings of the International Conference on Supercomputing, 1988, pp. 385–395.

[88] C.-T. King et L. Ni, *Large-Grain Pipelining on Hypercube Multiprocessors*, in Proceedings of The 3rd Conference on Hypercube Concurrent Computers and Applications, G. Fox, ed., vol. II - Software, ACM, Jan. 1988, pp. 1583–1591.

[89] C. Lawson, R. Hanson, D. Kincaid et F. Krogh, *Basic linear algebra subprograms for fortran usage*, ACM Trans. Math. Software, 3 (1979), pp. 308–371.

[90] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*, Morgan Kaufman Publishers, Inc., 1992.

[91] J. Lewis et R. van de Geijn, *Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms*, in Proceedings of the Supercomputing Conference, 1993.

[92] P. Li et D. Curkendall, *Parallel 3-D Perspective Rendering*, in Proceedings of the First Intel Delta Applications Workshop, 1992.

[93] P. A. MacMahon, *Application of a theory of permutations in circular procession to the theory of numbers*, in Proceedings of the London Mathematical Society, 1892, pp. 305–313.

[94] R. J. Manchek, *Design and Implementation of PVM version 3*, PhD thesis, University of Tennessee, Knoxville, Mai 1994.

[95] Message Passing Interface Forum, *Document for a Standard Message-Passing Interface*, Avr. 1994.

[96] P. Michallon, *Schémas de Communications Globales dans les Réseaux de Processeurs: Application à la Grille torique*, PhD thesis, Institut National Polytechnique de Grenoble, Fév. 1994.

[97] P. Michallon et D. Trystram, *Practical Experiments of Broadcasting Algorithms on a Configurable Parallel Computer*, Discrete Applied Mathematics, 53 (1994), pp. 291–298.

[98] P. Michallon, D. Trystram et G. Villard, *Optimal broadcasting algorithms on torus*, Rapp. Tech. RR8271, LMC-IMAG, Grenoble, 1991.

[99] R. Miller et S. Tanimoto, *Detecting Repeated Patterns on Mesh Computers*, rapp. tech., University of Buffalo, 1990.

[100] B. Monien et H. Sudborough, *Comparing interconnecting networks*, rapp. tech., Universität Gesamthochschule Paderborn, Oct. 1988.

[101] D. Nassimi et S. Sahni, *Data Broadcasting in SIMD Computers*, IEEE Transactions on Computers, 30 (1981), pp. 101–107.

[102] L. Ni et P. McKinley, *A Survey of Routing Techniques in Wormhole Networks*, Computer, (1993), pp. 62–76.

[103] D. Palermo, E. Su, J. Chandy et P. Banerjee, *Communication Optimizations used in the* Paradigm *Compiler for Distribued-Memory Multicomputers*, in Proceedings of ICPP'94, K. Tai, ed., vol. 2 - Software, CRC Press, Août 1994, pp. 1–10.

[104] J. Peters et M. Syska, *Circuit-Switched Broadcasting in Torus Networks*, rapp. tech., Simon Fraser University, 1993.

[105] B. Plateau, *APACHE: Algorithmique Parallèle et pArtage de CHargE*, Rapp. Tech. Apache 1, LMC - LGI - IMAG, Nov. 1994.

[106] B. Plateau, A. Rasse, J.-L. Roch et J.-P. Verjus, *Cours de Parallélisme*. Polycopié ENSIMAG, Nov. 1991.

[107] Y. Saad et M. H. Schultz, *Topological Properties of Hypercubes*, IEEE Transactions on Computers, 37 (1988), pp. 867–871.

[108] ——, *Data communication in parallel architectures*, Parallel Computing, 11 (1989), pp. 131–150.

[109] S. Saini et H. Simon, *Applications Performance Under OSF/1 AD and SUNMOS on Intel Paragon XP/S-15*, in Proceedings of Supercomputing'94, IEEE Society Press, Nov. 1994, pp. 580–589.

[110] M. Schwartz, *Telecommunication Networks - Protocols, Modeling and Analysis*, Addison Wesley Publishing Company, Nov. 1987.

[111] D. Scott, *Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies*, in Proceedings of sixth Distributed Memory Computing Conference, IEEE Computer Society Press, 1991, pp. 398–403.

[112] O. Sentieys, H.Dubois, J. Philippe et E. Martin, *A methodologic approach to configure architectures applied to an MIMD transputer based machine for image and signal processing.* Workshop on Massively Parallel Computing, Mars 1992.

[113] Q. Stout et B. Wagar, *Intensive hypercube communication, prearranged communication in link-bound machines*, Journal of Parallel and Distributed Computing, (1990), pp. 167–181.

[114] P. N. Swarztrauber, *Multiprocessors FFTs*, Parallel Computing, 5 (1987), pp. 197–210.

[115] A. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992.

[116] The MPI Forum, *MPI: A Message Passing Interface*, rapp. tech., University of Tenessee, Knoxville, 1993.

[117] A. Trew et G. Wilson, *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*, Springer - Verlag, 1991.

[118] C. Tron, *Modèles quantitatifs de machines parallèles : les réseaux d'interconnexion*, PhD thesis, Institut National Polytechnique de Grenoble, Déc. 1994.

[119] C. Tron et B. Plateau, *Modeling of Communication Contention in Multiprocessors*, in Computer Performance Evaluation, Modeling Techniques and Tools, G. Haring et G. Kotsis, eds., Springer Verlag, Mai 1994.

[120] Y. Tsai et P. McKinley, *A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers*, Rapp. Tech. MSU-CPS-94-35, Communications Research Group - Michigan State University, Juin 1994. ftp.cs.msu.edu/pub/crg.

[121] ——, *An Extended Dominating Node Approach to Collective Communication in All-Port Wormhole-Routed 2D Meshes*, in Proceedings of the SHPC Conference, IEEE, Mai 1994, pp. 199–206.

[122] ——, *A Broadcast Algorithms for All-Port Wormhole-Routed Torus Networks*, in Proceedings of FRONTIERS'95 Conference, Fév. 1995.

[123] C.-W. Tseng, *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*, PhD thesis, Dept. of Computer Science, Rice University, Jan. 1993.

[124] L. Turcotte, *A Survey of Software Environments for Exploiting Networked Computing Resources*, rapp. tech., Engineering Research Center for Computational Field Simulation, PO Box 6176 - Mississippi Sate, MS 39762, Juin 1993. ftp:unix.hensa.ac.uk:/pub/parallel/papers/survey/soft-env-net-report.ps.gz.

[125] R. van de Geijn, *Massively Parallel LINPACK Benchmark on the Intel Touchstone and iPSC/860 Systems*, Computer Science Technical Report TR-91-28, University of Texas, Août 1991.

[126] T. von Eicken, D. Culler, S. Goldstein et K. Schauser, *Active Message: a Mechanism for Integrated Communication and Computation*, in Proceedings of The 19th Symposium on Computer Architecture, A. I. C. Society, ed., ACM Press, Mai 1992, pp. 256–266.

[127] D. Walker, P. H. Worley et J. B. Drake, *Parallelizing the Spectral Transform Method - Part II*, rapp. tech., Oak Ridge National Laboratory, Juil. 1991.

[128] D. W. Walker, *Portable Programming within a Message-Passing Model: the FFT as an Example*, in The Third Conference On Hypercube Concurrent Computers and Applications, 1988.

[129] J. P. Zhu, *An Efficient FFT Algorithm on Multiprocessors with Distributed Memory*, in Proceedings of DMCC 5, vol. 1, 1990, pp. 385–363.

Résumé

Le but de ce mémoire est d'étudier certaines voies possibles pour minimiser le sur-coût des communications consécutif à la parallélisation d'algorithmes numériques.

La première voie explorée consiste à optimiser l'algorithme de communication globale. Nous proposons notamment de nouveaux algorithmes pour réaliser une transposition de matrices carrées allouées par blocs, sur différentes topologies de réseaux d'interconnexion. Nous avons également étudié le problème de l'échange total. Ce schéma de communication se retrouve fréquemment dans les versions parallèles d'algorithmes numériques (comme dans l'algorithme du gradient conjugué). Nous proposons des algorithmes efficaces d'échange total pour des topologies toriques.

La deuxième voie qui a été étudiée consiste à recouvrir les communications par du calcul. Nous avons étudié quelques principes algorithmiques de base permettant de masquer au mieux les communications. Ceux-ci sont basés notamment sur des techniques d'enchaînement de phases de calcul et de communication ainsi que sur le ré-ordonnancement local de tâches afin d'optimiser le recouvrement.

Ces techniques sont illustrées sur des algorithmes parallèles de calcul de transformée de Fourier. Les différentes implantations de ces algorithmes sur de nombreuses machines parallèles à mémoire distribuée (T3D de Cray, SP2 d'IBM, iPSC/860 et Paragon d'Intel) démontrent le gain en temps d'exécution apportées par ces méthodes.

Mots clefs : Parallélisation d'algorithmes numériques - Recouvrement des communications - Communications globales - Algorithmes de transformée de Fourier.

Abstract

The aim of this thesis is the study of different methods to minimize the communication overhead due to the parallelization of numerical kernels.

The first method consists in optimizing collective communication algorithms. We have proposed novel algorithms to achieve matrix transpose, for squared matrices distributed in a block fashion. We have also studied the total exchange problem. This communication scheme is useful in the parallelization of numerical kernels (as, for instance, the conjugate gradient algorithm). We have proposed efficient algorithms of total exchange for torus topologies.

The second method consists in overlapping communications by computations. We have studied some basic algorithmic principles which allow the overlap of communications. These ones are based on pipelining technics and local computational task reordering.

These technics have been illustrated in the parallelization of Fourier transform algorithms. The different implementations of these algorithms on various distributed memory parallel machines (Cray T3D, IBM SP2, Intel iPSC/860 and Paragon) hightlight the gain of efficiency induced using these technics.

Keywords: Parallelization of numerical kernels - Overlap of communications - Collective communications - Fourier transform algorithms.