N° d'ordre : 212
N° attribué par la bibliothèque : 01ENSL0212

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir le grade de

Docteur de l'École normale supérieure de Lyon

spécialité Informatique

au titre de la formation doctorale DIL

par Frédérique Silber-Chaussumier

Recouvrement des communications et des calculs, du matériel au logiciel

Présentée et soutenue publiquement le 21 Décembre 2001

Après avis de : Monsieur Pierre Manneback
Monsieur Serge Miguet

Devant la commission d'examen formée de :

Monsieur Frédéric Desprez (directeur de thèse)
Monsieur Pierre Manneback (rapporteur)
Monsieur Serge Miguet (rapporteur)
Monsieur Jean Roman
Monsieur Jean-Luc Basille
Monsieur Éric Théron

Thèse préparée à l'École normale supérieure de Lyon
au sein du Laboratoire de l'Informatique du Parallélisme.

Remerciements

Je souhaite remercier particulièrement Frédéric Desprez. Par son intermédiaire, je suis entrée dans le monde de la recherche et de l'enseignement. Tout au long de ma thèse, comme directeur de thèse, et même ensuite, il m'a, à la fois, laissée libre de faire des choix (pas toujours bons) et m'a repêchée ensuite quand c'était nécessaire. Disponible, serviable et toujours prêt à rire, Frédéric est un modèle de collègue.

Je souhaite aussi remercier l'ensemble des personnes contribuant à la bonne humeur et à l'efficacité du LIP où les conditions de travail sont absolument excellentes.

Je dois remercier Matra Systèmes & Information qui m'a financée et particulièrement Éric Théron qui m'a permis de terminer ma thèse dans de bonnes conditions.

Enfin, je veux remercier Georges-André Silber qui m'a non seulement supportée mais aussi sauvée de la noyade plus d'une fois : relectures, figures manquantes, installations, constitution de dossiers divers et toutes les tâches pour lesquelles le temps et l'envie manquent toujours.

Table des matières

Table des figures

Chapitre 1

Introduction

Dans le but d'accélérer l'exécution des programmes informatiques ou d'augmenter le volume de données traité, il est possible d'utiliser simultanément les ressources de plusieurs ordinateurs pour exécuter un même programme. Ainsi, l'exécution pourra bénéficier de plusieurs processeurs et espaces de stockage qui seront utilisés en parallèle. Afin de bénéficier de ce « *parallélisme* », il faut, à partir de l'algorithme du programme exécuté sur un seul processeur (le *programme séquentiel*) construire un *programme parallèle* qui pourra être exécuté sur plusieurs processeurs. Une telle construction (appelée *parallélisation*) est coûteuse en temps d'analyse et de développement. De plus, le gain obtenu en temps d'exécution varie énormément selon l'architecture et l'environnement logiciel de la machine cible. Nous présentons tous ces concepts dans le chapitre 2 de ce document.

L'objectif de cette thèse est de rendre plus efficace l'exécution des programmes parallèles sur une classe particulière de machines parallèles, les machines à mémoire distribuée. Parmi ces machines, les *grappes* de PC sont très populaires parce qu'elles sont peu coûteuses et équipées d'un matériel très performant. Notre travail a consisté à rechercher l'efficacité à travers deux méthodes : le recouvrement des communications par le calcul et le pipeline de calculs. Pour chacune de ces méthodes, nous avons cherché à évaluer leur gain et à faciliter leur application afin de réduire le coût de développement d'un programme parallèle efficace.

Lors de l'exécution d'un programme parallèle, plusieurs processeurs exécutent des calculs simultanément. Certains calculs opérent sur des données calculées précédemment par un autre processeur. Il sera donc nécessaire de communiquer ces données au processeur en charge du calcul. Ces communications sont une des principales sources de perte de performance des programmes parallèles. Le recouvrement des communications par le calcul permet à un processeur de continuer à calculer pendant que les données distantes sont placées dans sa mémoire locale. Cette technique a été utilisée dans de nombreuses applications sur différents types d'architectures. Dans cette thèse (chapitre 3), nous étudions l'efficacité de cette technique sur des architectures à mémoire distribuée actuelles comme les grappes de PC en utilisant la bibliothèque à passage de messages standard MPI. Pour cela, nous détaillons les moyens matériels mis à disposition du processeur pour effectuer une communication et un calcul en même temps. La principale contribution de cette thèse pour l'étude des recouvrements est la description détaillée d'une implantation de la bibliothèque de communication MPI sur une grappe de PC interconnectés par un réseau rapide Myrinet. Cette étude nous

a permis d'identifier les différents obstacles au recouvrement des communications. Nous verrons que, dans notre cas, l'obstacle principal n'est pas le matériel mais la bibliothèque de communication elle-même.

Dans le chapitre 4, nous présentons la méthode du pipeline de calculs utilisée pour extraire du parallélisme dans certains calculs dont les dépendances impliquent une exécution séquentielle. Le pipeline logiciel consiste à ne faire calculer à chaque processeur qu'une partie de ses données avant d'envoyer les données mises à jour aux autres processeurs. Cette stratégie permet aux autres processeurs de commencer leurs calculs plus tôt dans le but de terminer l'exécution de l'intégralité de l'application plus rapidement. Cette technique est souvent considérée comme du recouvrement de calculs puisqu'un processeur effectue des calculs sur les données précédemment mises à jour par un autre processeur pendant que ce dernier traite les données suivantes. Le gain en temps de cette technique n'est pas garanti car elle induit un surcoût de communication. De plus, comme toutes les optimisations de programmes parallèles, cette technique nécessite une modification importante du programme parallèle. Nous avons donc cherché dans cette thèse à évaluer le gain d'un pipeline de calculs. Dans un premier temps, nous avons validé sur l'application Sweep3D les équations de gain obtenues par Zory [75]. La principale contribution de cette thèse dans l'évaluation du pipeline logiciel est l'établissement d'équations donnant le temps d'exécution du pipeline sur des données tridimensionnelles pour des distributions mono, bi et tridimensionnelles. Ainsi, il est possible de déterminer par avance quelle distribution donnera le plus petit temps d'exécution en fonction des caractéristiques de la machine cible et de l'application, économisant dès lors du temps de développement. En effet, plus le nombre de dimensions de la distribution de données est élevé, plus le programme est difficile à établir. Il est donc intéressant de pouvoir déterminer a priori si une distribution mono ou bidimensionnelle est suffisante pour atteindre de bonnes performances.

Afin de valider ces résultats, nous avons appliqué ces deux techniques d'optimisation dans le cas de la parallélisation d'une application irrégulière de rendu volumique en imagerie médicale (chapitre 5). Nous avons cherché, par l'application de ces deux techniques, à obtenir une image en trois dimensions de bonne qualité en temps réel à partir de données médicales. Cette partie nous a également permis d'étudier les problèmes spécifiques posés par la parallélisation d'une application utilisant une structure de données creuse, source importante de l'irrégularité de l'application.

Pour finir, nous avons regroupé dans l'annexe A la description de notre participation dans les différents projets industriels avec Matra Systèmes & Information (MS&I)[1] s'inscrivant dans le cadre de cette thèse avec financement CIFRE.

[1] Cette société s'appelle maintenant Sycomore Aérospatiale Matra (SAM).

Chapitre 2

Préliminaires

2.1 Optimisations d'un programme parallèle

Dans cette partie, nous définissons les deux stratégies d'optimisation étudiées au cours de cette thèse : le recouvrement des communications par le calcul et le pipeline de calculs.

Recouvrement des calculs et des communications

Le recouvrement des communications par le calcul est un moyen d'améliorer les performances et l'extensibilité des applications parallèles. Quand les calculs et les communications sont indépendants, un processus peut exécuter les calculs pendant qu'il communique des données. Cela permet de réduire le surcoût de temps d'exécution lié aux communications.

Malheureusement, le recouvrement des communications par le calcul n'est pas toujours possible. Parfois, les architectures logicielles et matérielles utilisées ne permettent pas le recouvrement (voir le Chapitre 3). Comme nous l'avons vu au chapitre précédent, les dépendances de calcul peuvent impliquer une séquentialité qui empêche que les calculs et les communication soient effectués en même temps. Dans ce dernier cas, et suivant le type de dépendances, on peut parfois utiliser le pipeline de calculs.

Pipeline de calculs

Quand les dépendances de calcul impliquent la séquentialité (voir la figure 2.1), pipeliner les calculs permet d'améliorer l'efficacité parallèle. Le pipeline de calculs consiste à ne faire calculer à chaque processeur qu'une partie de ses données avant d'envoyer les données mises à jour aux processeurs voisins. Cette stratégie permet aux processeurs voisins de commencer leurs calculs plus tôt.

La figure 2.1 représente un calcul matriciel simple qui tirerait profit du pipeline des calculs. En effet, toutes les distributions mono- ou bi-dimensionnelles de la matrice A impliquent la séquentialité parce que les deux boucles portent des dépendances. Dans un premier temps, le domaine d'itérations correspondant est partitionné sur une ligne de quatre processeurs (voir la figure 2.2), chaque processeur possédant les données nécessaires au calcul de ses itérations. Sans pipeline, l'exécution s'effectue complètement séquentiellement. En pipelinant

```
for(i=1; i<6; i++)
  for(j=1; j<6; j++)
    A(i, j) = A(i, j-1) + A(i-1, j);
```

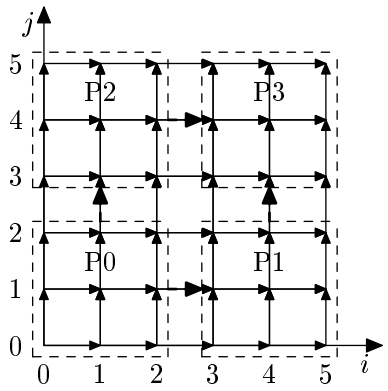
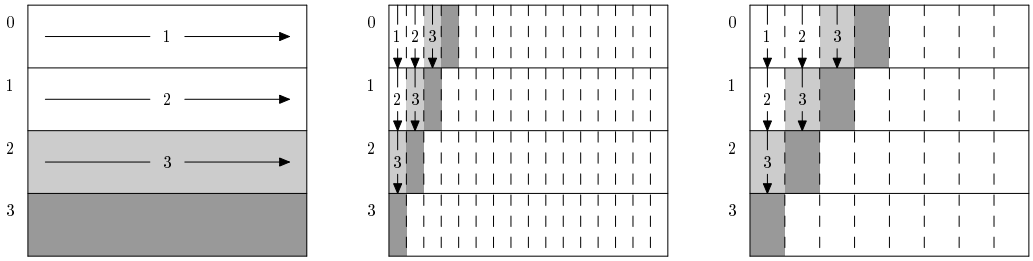


FIG. 2.1 – Code simple qui tirerait profit du pipeline des calculs.

l'exécution, $P0$ ne calcule qu'une partie de ses données avant de les envoyer à $P1$, qui peut alors aussitôt démarrer le calcul de ce bloc, dont il enverra les résultats à $P2$, etc.

Avec une distribution en deux dimensions, un parallélisme par vague apparaît. La figure 2.1 montre un partitionnement en grille sur quatre processeurs physiques. Sans pipeline, l'exécution s'effectue en effet par vagues pendant lesquelles les processeurs situés sur une même diagonale travaillent en parallèle. Ici, après avoir reçu les données de $P0$, $P1$ et $P2$ travailleront en parallèle puis enverront les données mises à jour à $P3$. En pipelinant l'exécution, $P0$ ne calcule qu'une partie de ses données avant de les envoyer à $P1$ et $P2$. Le nombre de vagues balayant l'espace d'itérations est égal au nombre de sous-blocs de pipeline. Ainsi, si le nombre de sous-blocs est supérieur au nombre de diagonales de la grille de processeurs, tous les processeurs travailleront à un moment donné en parallèle. La taille d'un de ces sous-blocs est appelé le *facteur bloquant* du pipeline. Plus ce facteur est petit, plus tôt est amorcé le pipeline mais plus le nombre de communications augmente. Il y a donc un compromis à trouver entre le temps de communication et la taille d'un grain de calcul afin de parvenir au plus petit temps d'exécution possible. On parle de *pipeline à grain fin* lorsque la taille du bloc de calculs est égale à un (voir la figure 2.2(b)). On parle de *pipeline à gros grain* lorsque la taille du bloc de calcul est bien supérieure à un (voir la figure 2.2(c)).



(a) Sans pipeline. (b) Grain fin. (c) Gros grain.

FIG. 2.2 – Pipelines à grain fin et à gros grain.

Les deux stratégies de recouvrement et pipeline peuvent néanmoins être combinées en recouvrant les communications des données précédemment mises à jour avec le calcul des données suivantes.

2.2 Parallélisation d'une application

Cette thèse présente une étude de deux techniques d'optimisation de programmes parallèles sur des machines à mémoire distribuée : le recouvrement des communications par le calcul et le pipeline de calculs. Paralléliser une application consiste à l'implanter sur une architecture multiprocesseurs, soit dans le but d'en accélérer le temps d'exécution soit dans le but de calculer d'importants volumes de données. Dans cette thèse, nous nous intéressons à l'accélération du temps d'exécution. La plupart des applications possèdent plusieurs algorithmes parallèles. Ces algorithmes ne sont pas tous équivalents ni en terme de temps d'exécution, ni en terme de volume de données traitées. Nous verrons plus loin qu'il y a des critères d'évaluation de ces algorithmes, le meilleur d'entre eux étant celui qui traitera le plus rapidement le plus gros volume de données. Pour aboutir à un algorithme parallèle, il est possible de décomposer le travail en plusieurs étapes.

Il faut tout d'abord partitionner les calculs et les données sur lesquelles sont effectués ces calculs. Partitionner les calculs consiste à déterminer les parties de calcul indépendantes les unes des autres. Il existe des cas simples d'applications complètement parallèles. Par exemple, un traitement comme le filtre inverse utilisé par le projet PARSMED-3D décrit dans la partie A.1.1 de ce document est complètement parallèle. Il consiste à calculer les valeurs inverses des pixels d'une image. Les calculs des valeurs inverses de chaque pixel constituant l'image peuvent donc être effectués indépendamment les uns des autres. De plus, le calcul de la nouvelle valeur d'un pixel n'implique aucune autre donnée que le pixel lui-même. On pourra donc partitionner l'image en la distribuant soit par rapport à l'une de ses dimensions, *distribution monodimensionnelle* soit selon ses deux dimensions, *distribution bidimensionnelle*. Malheureusement, il existe aussi des applications dont la parallélisation est moins évidente. Au chapitre 5, nous détaillons la parallélisation que nous avons effectuée d'une application de rendu volumique. Il s'agit d'une *application irrégulière*, c'est-à-dire qu'au cours de l'application, le volume de calculs et de données à traiter est très irrégulier et dépendant des données elles-mêmes.

La deuxième étape consiste à considérer les communications nécessaires pour coordonner l'exécution de ces différentes tâches. Par exemple, si l'on cherchait à afficher l'image inversée, il faudrait que chaque processus transmette la portion de données qu'il a calculée au processus en charge de l'affichage.

Enfin, la dernière étape est la répartition statique de ces calculs et de ces données sur des processeurs physiques. L'affectation des calculs et des données à un processeur cherche à réduire les communications et à accroître l'utilisation du processeur. Ainsi, la règle « Owner Computes Rule » instituant qu'un processeur effectue les calculs correspondant aux données qu'il stocke, s'applique. Pour accroître l'utilisation des processeurs, il faut que tous les processeurs calculent pendant toute la durée de l'exécution de l'application. On cherche donc à équilibrer la charge de chaque processeur dans le but que tous les processeurs terminent leur exécution en même temps. La notion d'*équilibrage de charge* est centrale dans le parallélisme.

Dans le cas d'une application irrégulière, on cherchera le plus souvent à équilibrer la charge dynamiquement au cours de l'exécution.

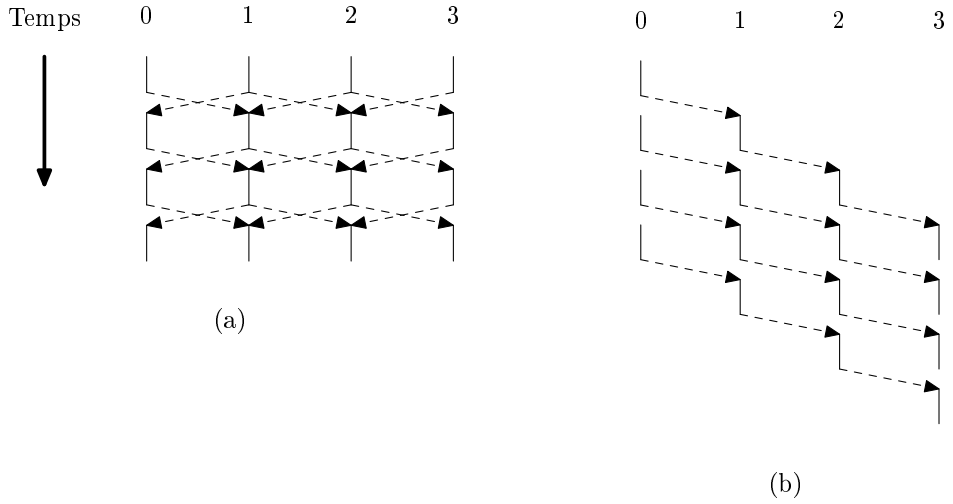


FIG. 2.3 – Exécutions parallèle et pipelinée.

Dans le but d'effectuer globalement le plus de calculs possibles sur le plus de données possibles, les calculs et les données ne sont, dans un premier temps, pas dupliqués. Cela a pour conséquence de générer beaucoup de communications inter-processeurs. Pour réduire le nombre de communications, certains calculs peu coûteux peuvent être exécutés par tous les processeurs. De même, il est souvent nécessaire pour calculer une valeur donnée de connaître ses valeurs voisines. Les différents filtres d'interpolation utilisés dans le projet PARSMED-3D (partie A.1.1) font partie de ce type d'applications. Dans ce cas, plutôt que de générer une communication à chaque fois que l'on a besoin d'une donnée ne nous appartenant pas, on pourra dupliquer les données frontières nécessaires. La figure 2.3(a) schématise une exécution parallèle régulière. Au cours de cette exécution, la charge des processeurs est parfaitement équilibrée. Les processeurs voisins échangent à chaque étape les données nécessaires à la suite de leur calcul par communication de messages. Dans le cas où les dépendances de calcul sont telles que, pour effectuer son calcul, un processeur attend les résultats des calculs du processeur précédent, il est possible de paralléliser cette application par le moyen du *pipeline de calculs*. Le processeur précédent calculera alors une sous-partie de ces données afin de transmettre les résultats nécessaires au processeur suivant pour qu'il commence à calculer au plus tôt. La figure 2.3(b) schématise une exécution pipelinée. Le processeur suivant attend les données du processeur pour commencer son calcul. On voit alors que, pendant un certain temps au cours de l'exécution, l'ensemble des processeurs travaillent en parallèle.

2.3 Performances d'un programme parallèle

Dans cette thèse, nous utilisons trois grandeurs permettant de mesurer les performances d'un programme parallèle. Dans un premier temps, nous nous intéressons au temps d'exécu-

tion du programme sur la machine cible. Des mesures comme l'accélération et l'extensibilité mesurent les qualités intrinsèques de l'algorithme parallèle et son adéquation à la machine cible.

Temps d'exécution

Le temps d'exécution d'un programme parallèle est la mesure la plus intuitive et souvent la finalité de la parallélisation d'une application. Ainsi, au Chapitre 5 ainsi qu'au cours du projet industriel PARSMED-3D (voir la partie A.1), nous cherchons à obtenir une image tridimensionnelle à partir de données médicales en temps réel sur une architecture parallèle définie au préalable. C'est pourquoi le temps d'exécution est seul juge de la réussite de ces applications parallèles.

Accélération

L'accélération (ou *speedup*) est le ratio $\frac{W}{T_p}$. W représente le plus petit temps d'exécution séquentiel de cette application. T_p est le temps d'exécution parallèle. Si l'application parallèle est exécutée sur P processeurs, une application parallèle possédant une bonne accélération a une accélération proche de P.

Extensibilité et passage à l'échelle

L'extensibilité (ou *scalability*) d'un algorithme fait référence à sa capacité à conserver de bonnes performances au fur et à mesure que le nombre de processeurs augmente. Une bonne extensibilité se caractérise donc par une bonne accélération lorsque le nombre de processeurs croît. Un algorithme extensible doit aussi permettre d'augmenter le volume de données traité au fur et à mesure que le nombre de processeurs croît. On parle souvent de passage à l'échelle ; il s'agit de dépasser le stade expérimental en introduisant les données réelles qu'une application aura à traiter. On augmente alors le volume de données dans une application scientifique ou par exemple le nombre d'utilisateurs dans le cas du projet CHARM de cache web parallèle (voir la partie A.3).

2.4 Aspects architecturaux

Les machines parallèles sont classées en deux grandes classes selon que les processeurs accèdent à une mémoire commune, dite « partagée », ou qu'ils possèdent chacun leur propre mémoire, on parle alors de « mémoire distribuée ». Dans le premier cas, le système devra garantir l'intégrité des données au sein de la mémoire commune, en particulier, en n'autorisant pas deux accès simultanés en écriture à la même adresse. Dans le cas de la mémoire distribuée, un processus souhaitant accéder à une donnée ne résidant pas dans la mémoire locale de son processeur devra accéder à la mémoire d'un autre processeur au moyen d'échange de messages. Dans la pratique, il existe aussi des machines à mémoire distribuée virtuellement partagée au sens où le système permet de concevoir la machine comme à mémoire partagée alors que les bancs mémoire sont physiquement distribués sur les processeurs. Les

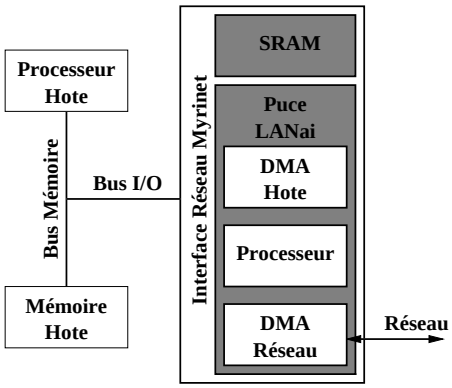


FIG. 2.4 – Interface réseau Myrinet.

paragraphes suivants décrivent brièvement les architectures des machines que nous avons souvent utilisées pour effectuer nos expériences.

Grappe de PC : PoPC

PoPC est une grappe de PC installée dans le cadre du LHPC (Laboratoire des Hautes Performances en Calcul). Cette grappe de PC est composée de douze stations de travail à base de microprocesseurs Pentium Pro 200 Mhz avec 128 Mo de mémoire exécutant Linux reliées par un réseau rapide Myrinet. Chaque station de travail est donc équipée d'une carte d'interface réseau Myricom/PCI installée sur le bus PCI[1]. Comme le montre la figure 2.4, la carte réseau contient un moteur DMA[2] hôte qui transfère les données de la mémoire principale du microprocesseur hôte à la mémoire SRAM[3] de la carte réseau et un moteur DMA réseau qui transfère les données de la mémoire SRAM au réseau. La carte réseau contient aussi 64 Ko de mémoire et un processeur LANai qui exécute le protocole de communication de messages et a la charge de coordonner les actions des moteurs DMA et de dialoguer avec le microprocesseur hôte.

Sur cette grappe de PC, nous utilisons la couche de communication rapide BIP[52][4]. Cette couche de communication est décrite dans la partie 3.3.1. Elle permet d'atteindre des débits de 1Gb/s.

IBM SP-2

Nous avons utilisé pour nos expériences deux SP-2 différentes, l'une installée au LaBRI (Laboratoire Bordelais de Recherche en Informatique) composée de seize nœuds, l'autre au CINES (Centre Informatique National de l'Enseignement Supérieur) composée de 128 nœuds. Chaque nœud est équipé d'un processeur RS/6000 et d'un adaptateur de communication. Les nœuds sont reliés par un réseau d'interconnexion à étages. L'adaptateur de communication du

[1]PCI : Peripheral Component Interconnect
[2]DMA : Direct Memory Access
[3]SRAM : Static Random Access Memory
[4]BIP : Basic Interface for Parallelism

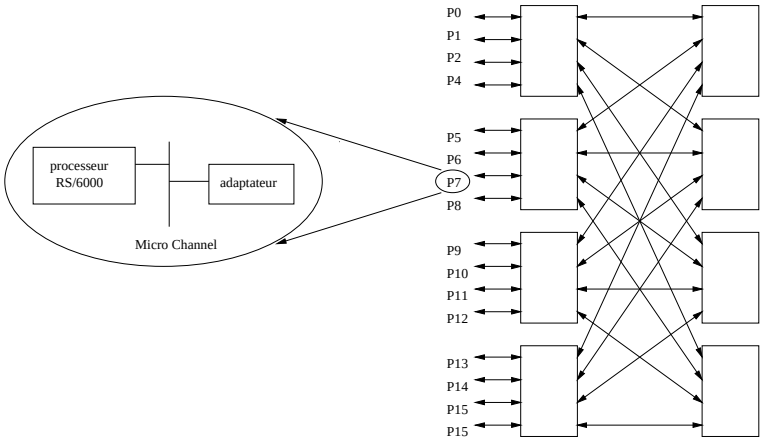


FIG. 2.5 – Architecture de l'IBM SP-2.

SP-2 est équipée d'un processeur Intel i860 utilisé comme co-processeur de communication. Les transferts de données utilisent deux bus, le bus principal Micro Channel et le bus i860 connecté au réseau. Ces bus sont connectés avec une FIFO bidirectionnelle de 4Ko et deux moteurs DMA. Le système d'exploitation est Unix Solaris. La figure 2.5 représente cette architecture.

SUN Enterprise 10000 : E10K

L'E10K est une machine parallèle à mémoire partagée installée dans le cadre du PSMN (Pôle Scientifique de Modélisation Numérique). Cette machine est un système multiprocesseur basé sur un bus. Elle est composée de seize processeurs UltraSparc 250Mhz avec 4Mo de mémoire cache, 2048Mo de mémoire principale et 30 Go d'espace disque. Le système de bus « Gigaplane » permet d'atteindre une bande passante maximale de 2,67 Go/s. Chaque carte d'entrée/sortie est équipée de deux bus indépendants d'entrée/sortie SBUS 64-bit × 25-Mhz, donc la bande passante augmente proportionnellement au nombre de cartes.

Les seize emplacements de la machine peuvent contenir aussi bien des cartes de calcul que des cartes d'entrée/sortie mais il faut qu'au moins une de chaque type soit présente. Chaque carte de calcul possède deux modules CPU et deux bancs de mémoire de 128 Mo chacun (2048/16), donc la capacité mémoire et la bande passante sont proportionnelles au nombre de cartes. Bien que la mémoire soit physiquement locale à une paire de processeurs, toute la mémoire est accessible par le système de bus qui assure un accès uniforme.

Le système d'exploitation est Unix Solaris. La figure 2.6 représente le système de la SUN Enterprise E10K.

Grappe de PC : icluster

Le icluster est une grappe de PC installée dans le cadre d'un projet HP/INRIA. À l'époque de nos expériences, cette grappe était composée de 100 nœuds équipés d'un processeur Pentium III 733 MHz avec 256 Mo de mémoire et 15 Go d'espace disque. L'ensemble de ces nœuds est divisé en trois sous-ensembles. Les nœuds à l'intérieur d'un sous-ensemble

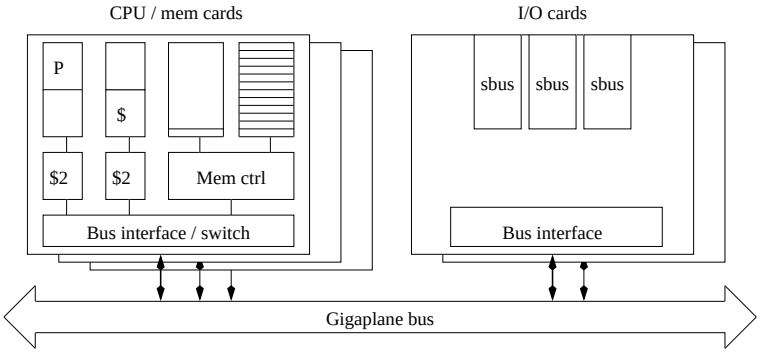


FIG. 2.6 – Le système de la Sun Enterprise 10000.

sont réliés par un réseau Ethernet 100. Chaque sous-ensemble est connecté à un autre par l'intermédiaire d'un pont Gigabit. Le système d'exploitation sur chacun des nœuds est Linux.

2.5 Bibliothèques de communication

Un des aspects les plus importants des applications parallèles, à la fois sur machines à mémoire distribuée et sur machines à mémoire partagée, est leur capacité à communiquer efficacement aux autres processus les données mises à jour.

Pour des machines à mémoire partagée, le plus gros effort est fourni sur les protocoles de mise à jour des données dans la mémoire partagée et la mémoire cache locale à un processeur. Les utilisateurs n'ont donc *a priori* pas de modifications à effectuer dans leurs programmes. Pour les machines à mémoire distribuée, le programmeur utilise directement des bibliothèques à passage de messages, ou un langage à directives de compilation permettant au compilateur de générer lui-même les appels à la couche de communication. Plus on travaille à un haut niveau d'abstraction, plus la programmation est simple mais il est souvent aussi plus difficile d'obtenir de bonnes performances. L'approche par passage de messages semble, pour l'instant, être un bon compromis entre difficulté de programmation et performances. Elle consiste à considérer la machine comme un ensemble de processus s'échangeant des messages par l'intermédiaire d'un réseau d'interconnexion.

Les bibliothèques de communication auxquelles nous nous intéressons sont des bibliothèques à passage de messages ou construites au-dessus de telles bibliothèques.

2.5.1 Bibliothèques et couches de communication

Les bibliothèques de communication comme PVM et MPI sont des bibliothèques à passage de messages qui se sont imposées comme standard de fait sur tous les types de machines parallèles.

Le projet PVM (*Parallel Virtual Machine*), démarré en 1990, est parti du constat que les stations de travail restaient inutilisées pendant la majeure partie du temps alors qu'elles auraient pu être utilisées comme machine parallèle bon marché. PVM se caractérise par la possibilité de faire communiquer des machines distribuées hétérogènes et d'avoir une plate-

forme complètement dynamique où les processus se créent et se détruisent au gré de l'exécution de l'application parallèle. MPI (*Message-Passing Interface*) est issu d'un forum démarré en 1992 réunissant les développeurs de bibliothèques, les utilisateurs et les constructeurs de machines. Le point clé de ce forum est la définition d'un standard de bibliothèque à passage de messages sur les calculateurs parallèles. En effet, chaque constructeur fournissait jusqu'ici une bibliothèque de communication spécifique au calculateur parallèle comme *MPL* pour IBM SP-2 ou encore *NX* sur une architecture Intel. Ces bibliothèques de communication, bien que spécifiques à une architecture, sont souvent très complètes et contiennent par exemple des fonctions de diffusion et des opérations globales comme la somme, le maximum ou le minimum. Néanmoins, une application implantée sur une architecture Intel avec *NX* doit être réécrite pour être exécutée sur IBM SP-2 ; d'où la nécessité de constituer les spécifications d'un standard de bibliothèques de communication.

L'implantation de ces bibliothèques standard ou spécifiques à une architecture est basée sur des fonctions de transfert de messages différentes pour chaque architecture de machine. Ces fonctions propres à une architecture définissent la *couche de communication de base*.

Les couches de communication sont, par exemple, *shmem* pour l'architecture Cray T3D. Comme le montre la partie 3.2.2, les couches de communication peuvent être très nombreuses. En effet, sur certaines architectures, grâce aux coprocesseurs de communication programmables, chaque utilisateur peut définir sa propre couche de communication. BIP (voir la partie 3.3.1), par exemple, est une couche de communication optimisée définie pour une grappe de PC interconnectés par un réseau rapide Myrinet.

La principale différence entre bibliothèques et couches de communication est le confort d'utilisation pour le programmeur. Une couche de communication est la plus proche possible du matériel et délègue ainsi au programmeur beaucoup de tâches, telles que la synchronisation des processus ou le format des messages. Les fonctions fournies par une couche de communication sont peu nombreuses et implantent des fonctionnalités simples : envoi et réception de données d'un processus vers un autre.

À l'inverse, une bibliothèque de communication décharge le programmeur des tâches de bas niveau. PVM, par exemple, fournit des fonctions d'empaquetage et de dépaquetage de messages permettant à deux machines hétérogènes de communiquer entre elles. De plus, les bibliothèques cherchent à fournir, en plus des fonctions de réception et d'envoi de données d'un processus vers un autre, des fonctions de communications globales comme la diffusion, l'échange total, le rassemblement des données, des opérations globales telles que la somme, le minimum, le maximum, etc.

La partie suivante détaille la sémantique de la bibliothèque à passage de messages MPI, ses implantations, performances et utilisations.

2.5.2 MPI : bibliothèque de communication à passage de messages

MPI [62] désigne une spécification de bibliothèque de communication à passage de messages qui se veut à la fois portable et efficace. La syntaxe et la sémantique d'un ensemble de fonctions ont été définies pour que ces fonctions puissent être utilisées dans beaucoup d'applications différentes et donner lieu à une implantation efficace sur tous les types d'architectures. De nombreuses implantations de MPI existent aussi bien commerciales que dans

le domaine public. Cette diffusion importante de MPI en a fait son principal atout. Une application écrite avec MPI est portable car toutes les architectures parallèles proposent une implantation de MPI. Ainsi, bien que le principe du passage de messages soit principalement utilisé sur des machines à mémoire distribuée, il existe aussi des implantations de MPI sur les machines à mémoire partagée. Le principe de base est la communication entre processus par l'intermédiaire de messages.

MPI a été découpée en deux parties MPI-1 et MPI-2 afin d'avoir assez rapidement des premières versions pour les différents types de machines. Les premières spécifications de MPI, MPI-1, contiennent des communications point-à-point où un processus envoie un message à un autre, des opérations de communications globales pendant lesquelles plusieurs processus prennent part à une même opération ainsi que la gestion de contextes de communication séparés, les communicateurs, permettant d'isoler des communications à l'intérieur d'un contexte, et de topologies de processus. Ce premier ensemble de spécifications a été largement implanté. Un deuxième ensemble de spécifications MPI, MPI-2 vient compléter MPI-1. Ces spécifications fournissent des fonctionnalités de gestion dynamique de processus comme le fournit PVM, des primitives de communication *one-sided* permettant à un unique processus de déposer ou récupérer des données dans une mémoire distante sans impliquer un autre processus et enfin des fonctions d'entrées-sorties parallèles.

Par la suite, nous nous focalisons sur les communications point-à-point définies dans MPI-1 afin d'en comprendre les performances. Ces fonctionnalités ont été largement étudiées et implantées. Leurs performances varient néanmoins beaucoup selon les implantations et leurs comportements sont parfois surprenants comme le montre l'exemple d'implantation des recouvrements de communication dans la partie 3.1.1.

Communications point-à-point

La forme la plus simple d'échange de messages est la communication point-à-point. Le principe d'une communication point-à-point est la transmission de données entre deux processus, l'un expéditeur, l'autre destinataire. Les paramètres sont classiquement : pour l'émetteur, la taille du message (en nombre d'éléments d'un type donné ou en octets), l'adresse d'une zone de mémoire tampon contenant le message, l'étiquette du message et l'adresse du processus destination ; pour le processus récepteur, l'adresse d'une zone de mémoire tampon de réception du message et l'adresse du processus expéditeur.

La bibliothèque MPI définit cependant plusieurs modes de communication pour effectuer cela. Le mode de communication *standard* est le mode le plus compliqué. La terminaison d'un envoi en mode standard peut dépendre de l'existence ou non de la réception correspondante. MPI ne spécifie pas si le message sera recopié dans une zone de mémoire tampon ou non. S'il n'y a pas de réception correspondante, la message peut être recopié dans le but de terminer l'envoi ou attendre jusqu'à ce que la réception correspondante ait lieu. Un envoi en mode avec *recopie* (*buffered*) peut être effectué même s'il n'existe aucune réception correspondante. Dans ce cas, la recopie du message peut être nécessaire. La terminaison d'un tel envoi ne dépend pas de l'éventuelle exécution de la réception correspondante. Le mode *synchrone* garantit qu'au moment où l'envoi se termine, le processus destinataire a exécuté la réception correspondante, c'est-à-dire qu'il a non seulement initialisé la réception mais aussi commencé

Standard	La communication s'effectue avec ou sans recopie, avec ou sans réception correspondante.
Recopie	Grâce à la recopie du message, l'envoi se termine indépendamment d'une éventuelle réception.
Synchrone	Au moment où l'envoi se termine, le processus destinataire a exécuté la réception correspondante.
Prêt	La réception correspondante doit être initialisée avant d'effectuer l'envoi.

FIG. 2.7 – Récapitulation des modes de communications en MPI.

à recevoir le message. Et enfin le mode *prêt* (*ready*) impose qu'un envoi ne soit initialisé que lorsque la réception correspondante a été postée. Le mode le plus utilisé par le programmeur est le mode standard. Il est néanmoins celui pour lequel on possède le moins d'informations sur l'efficacité de l'implantation. De plus, chacun de ces modes peut être utilisé avec des appels à des communications bloquantes ou non bloquantes. La figure 2.7 récapitule dans un tableau ces différents modes de communication.

Communication bloquante. Lors d'une communication bloquante, l'appel à un envoi bloquant ne se termine pas avant que la zone de mémoire contenant le message puisse être réutilisée ; de même, l'appel à une réception bloquante ne se termine pas avant que les données du message reçu se trouvent effectivement dans la zone de mémoire fournie à la fonction de réception.

Communication non bloquante. Une communication non bloquante est constituée de deux parties : l'initialisation et la terminaison. Un appel à un envoi non bloquant initialise l'envoi mais ne l'effectue pas. Pour pouvoir réutiliser la zone de mémoire fournie à la fonction d'envoi, il faut terminer l'envoi par un appel à une fonction testant la terminaison de l'envoi. De même, un appel à une réception non bloquante initialise la réception mais ne termine pas la réception. Pour savoir si la communication est terminée, il faut appeler une fonction testant la terminaison de la réception. L'appel à un envoi ou une réception non bloquante se termine donc immédiatement sans attendre que la communication soit en cours.

Ainsi, si le principe du passage de messages est très simple, sa mise en œuvre peut être assez compliquée, même dans le cas de simples communications point-à-point. Elle nécessite une bonne compréhension des différents modes de communications. De plus, les libertés laissées aux implantations, dans le but de garantir à la fois la portabilité et les performances sur toutes les architectures, sont à l'origine de variations importantes des performances d'une implantation à l'autre. Ainsi, dans le but de comparer les performances de deux solveurs creux sur deux architectures différentes, IBM SP-2 et Cray T3E, Amestoy *al.* dans [7] ont dû se pencher sur certains aspects des implantations de MPI sur ces deux architectures.

Implantations et performances

De nombreuses implantations de MPI existent, à la fois commerciales et gratuites. Néanmoins, à part la version de MPI annoncée par Fujitsu, aucune implantation de MPI n'implante complètement MPI-2. Le tableau suivant donne les performances d'implantations de MPI en comparant la *latence* (temps de préparation du message sur le nœud expéditeur) et la *bande passante* (nombre d'octets que l'on peut transférer sur le réseau par seconde).

	Latence (μs)	Bande passante (Mo/s)
Intel Paragon	40	70
IBM SP-2	35	35
T3D	21	100
SGI Power Challenge	47	55
Grappe de PC Myrinet (MPI-BIP)	9	125
Grappe de PC Fast Ethernet (MPI-TCP/IP)	121	11
Grappe de PC SCI (MPI-SISCI)	4.5	83

Devenir

Le développement de bibliothèques MPI suit celui des architectures de machines. On trouve des bibliothèques MPI construites au-dessus de bibliothèques de communication adaptées à des architectures SMP telles que BIP-SMP [26]. On distingue alors les communications à l'intérieur d'un nœud SMP et les communications entre deux nœuds SMP différents ; les premières bénéficiant alors de la localité des données et devenant, comme il se doit, moins coûteuses que les secondes.

L'avenir de MPI réside aussi dans le développement de bibliothèques pour architectures hétérogènes. Le méta-computing, mouvement visant à mettre en commun l'ensemble des ressources de calcul, nécessite, par exemple une telle évolution. Le développement de bibliothèques MPI multi-protocoles permet l'exécution d'un même programme sur plusieurs types d'architectures de machines et de réseaux différentes simultanément. MPICH/Madeleine [11], par exemple, permet par l'intermédiaire de la couche de communication multi-protocoles d'utiliser les réseaux Myrinet, Giganet ou encore SCI simultanément et donc donne accès à très grand nombre de machines cibles.

2.5.3 Bibliothèques spécifiques

Certaines bibliothèques ont été développées pour des applications particulières. Ces bibliothèques ne sont pas toujours des bibliothèques de communications à proprement parler mais prennent en charge les communications. Elles sont alors construites au-dessus de bibliothèques de communication, soit au dessus d'une bibliothèque comme PVM ou MPI à des fins de portabilité soit directement optimisées pour chaque architecture de machine.

Il existe notamment beaucoup de bibliothèques d'algèbre linéaire pour le calcul scientifique parallèle. ScaLAPACK, par exemple, est construite au dessus de la bibliothèque de communication BLACS (*Basic Linear Algebra Communication Subroutine*), elle-même implantée sur la plupart des architectures de machine, avec les couches de communication na-

tives puis sur PVM et MPI. PETSc (*Portable Extensible Toolkit for Scientific Computation*) est en revanche directement construite au dessus de MPI.

Chaque domaine professionnel utilisant le parallélisme comme la météorologie ou l'imagerie, a développé des bibliothèques appliquées à la parallélisation de leurs applications.

Il existe aussi des bibliothèques d'aide à une bonne parallélisation fournissant par exemple des fonctions de distribution de données, de redistribution de données, d'équilibrage de charges.

Chapitre 3

Recouvrement des calculs et des communications

Dans ce chapitre, nous présentons l'étude d'une première technique d'accélération d'applications : le recouvrement des communications par le calcul introduit dans la partie 2.1. Il s'agit, de manière générale, de permettre au processeur de calcul de continuer à calculer pendant que les données nécessaires pour la suite sont placées de manière asynchrone dans sa mémoire locale. Pour cela, il est tout d'abord nécessaire de pouvoir identifier, à l'intérieur de l'application, des blocs de calcul indépendants exploitant des données différentes. Dans un deuxième temps, il faut que le transfert des données puisse se faire indépendamment du processeur de calcul. Si le temps de communication que l'on cherche à recouvrir est inférieur au temps de calcul ne dépendant pas de cette communication alors le temps de communication est totalement recouvert et le temps d'exécution total est égal au temps de calcul. Dans le cas contraire, si le temps de calcul est inférieur au temps de communication, alors le temps d'exécution est égal au temps de communication.

Dans ce chapitre, nous allons tout d'abord donner des exemples de travaux existants utilisant le recouvrement des communications par le calcul dans le but d'accélérer des applications. Puis, nous décrirons les moyens matériels et logiciels mis à disposition du processeur de calcul pour récupérer des données se trouvant dans une mémoire distante. Enfin, nous présenterons l'étude de l'implantation de la bibliothèque de communication à passage de messages MPI sur une architecture Myrinet que nous avons menée dans le but d'évaluer la faisabilité des recouvrements des communications. Nous montrerons qu'à part les dépendances dans le code, le principal obstacle au recouvrement des communications par le calcul est bien souvent dû aux couches de communication logicielles.

3.1 Applications

Les principales sources de perte de performance sur des machines à mémoire distribuée sont les communications inter-nœuds permettant d'accéder aux données distantes. Afin de masquer ces coûts de communication, le recouvrement des communications par le calcul a été implanté dans différents types d'applications. Il s'agit d'une optimisation très classique pour des applications possédant des calculs et des communications indépendants les uns des

autres. En effet, du point de vue du programmeur, recouvrir les communications permet de diminuer le coût de ces communications sans ajouter de surcoût lié à cette technique. Dans [66], Strumpen montre que le gain maximal en temps d'exécution que l'on peut obtenir en recouvrant les communications est égal à deux. En effet, intuitivement, le cas le plus favorable est le cas où le temps de calcul est au moins égal au temps de communication. Dans ce cas, en recouvrant les communications par le calcul, on peut gagner 50% du temps. Strumpen a implanté la méthode du gradient conjugué sur un réseau de stations au-dessus de TCP/IP. Il obtient un gain de 1,3 et une accélération égale à 155 sur 229 processeurs. Quinn et Hatcher, dans [54] montrent aussi que le gain maximal que l'on pourrait obtenir en recouvrant les communications par le calcul est égal à deux. Ils appellent ce type de recouvrement le *recouvrement systolique*. Nous revenons plus longuement sur le modèle utilisé dans la partie 4.1.2.

Le recouvrement des communications par les calculs a été implanté dans de nombreuses applications, sur de nombreuses architectures cibles différentes et et dans des bibliothèques de communications différentes. Bien souvent, il a été nécessaire de réordonnancer les calculs et les communications.

Baden et Fink [12] ont implanté la relaxation de Gauss-Seidel sur une grappe de SMP et obtiennent un gain de 15 % en temps d'exécution grâce au recouvrement des communications sur une couche de communication non standard. Domas *et al.* [20] montrent que le recouvrement des communications dans la factorisation LU permet d'obtenir jusqu'à 20% de gain en temps d'exécution pour des tailles de matrices ne saturant pas la mémoire locale du processeur sur un Intel Paragon. Pour de grandes tailles de matrices, le gain obtenu est autour de 4%. Cet exemple est intéressant parce qu'il montre que même dans un cas défavorable, une mémoire saturée et des temps de communication supérieurs aux temps de calcul donc jamais intégralement recouverts, l'implantation du recouvrement des communications ne mène pas à une perte de performances. La même expérience a été effectuée sur un IBM SP-2. Elle donne les mêmes résultats pour des petites tailles de matrices. En revanche, pour de grandes tailles de matrices, la factorisation LU sur IBM SP-2 est moins performante avec recouvrement que sans recouvrement. Dans [21], Desprez *et al.* obtiennent le recouvrement total des communications pour la résolution d'un modèle de combustion sur une machine Intel Paragon ainsi que sur une iPSC/860. L'extensibilité de l'implantation parallèle est très bonne. McManus *et al.* [45] étudient les possibilités de recouvrement des communications dans la méthode du gradient conjugué. Ils ont testé l'implantation du gradient conjugué avec recouvrement des communications sur une machine Transtech Paramid dans laquelle les nœuds étaient équipés d'un processeur de calcul Intel i800 et d'un co-processeur de communication Inmos T800 utilisé dans l'implantation de la bibliothèque à passage de message. Ils ont obtenu une nette amélioration des performances avec l'implantation avec recouvrement sur une telle architecture. Abdelrahman [1] étudie les gains liés au recouvrement des communications sur une architecture COMA (*Cache Only Memory Access*) pour différentes applications : la factorisation LU, la FFT et la méthode de résolution de problème d'optimisation linéaires denses, le simplex. La machine cible est la KSR2. L'implantation gagne jusqu'à 40% de temps d'exécution avec implantation du recouvrement des communications. Pour la factorisation LU, le gain atteint 20% pour un grand nombre de processeurs (jusqu'à

48). En revanche, pour la FFT, l'impact des communications n'est pas assez grand pour que le gain soit important.

Le recouvrement des communications a donc été implanté sur de nombreuses machines différentes avec des couches de communication différentes et non standard.

Utilisant la couche de communication standard MPI, Crockett *et al.* [44] utilisent le recouvrement des communications dans une application de rendu volumique à base de projection de cellules sur une machine IBM SP-2. Si le gain en performance directement lié à cette optimisation n'est pas fourni, ils obtiennent néanmoins une très bonne accélération de 96 sur 128 processeurs et détaillent l'utilisation de MPI sur SP-2.

Dans la partie suivante, nous détaillons l'implantation parallèle en MPI d'une application simple avec et sans recouvrement.

3.1.1 Résolution de l'équation de Laplace avec Jacobi

Dans cette partie, nous détaillons une implantation parallèle de la résolution de l'équation de Laplace avec la méthode de Jacobi [27, 62]. Nous allons tout d'abord présenter un algorithme parallèle sans recouvrement des communications. Dans un deuxième temps, nous étudierons les possibilités de recouvrement des communications et les gains espérés. Enfin, nous montrerons les traces générées par l'implantation de ces algorithmes en MPI.

Algorithme séquentiel. Dans notre exemple, nous utilisons donc la méthode de Jacobi pour résoudre l'équation de Laplace $\nabla^2 A = 0$ à deux dimensions. L'équation de Laplace s'écrit aussi :

$$\frac{\partial^2 A(x,y)}{\partial x^2} + \frac{\partial^2 A(x,y)}{\partial y^2} = 0$$

Nous appliquons le problème sur une grille régulière à deux dimensions. Les points de la grille sont éloignés d'une distance h et se trouvent aux abscisses :

$$x_i = \frac{i-1}{h}$$

où i varie de 1 à n. Sur cette grille, on peut approcher la dérivée de premier ordre par

$$\frac{\partial A}{\partial x} \approx \frac{A(x_{i+1}) - A(x_i)}{h}$$

et la dérivée de deuxième ordre par

$$\frac{\partial^2 A}{\partial x^2} \approx \frac{A(x_{i+1}) - 2A(x_i) + A(x_{i-1})}{h^2}$$

En remplaçant cette expression dans l'équation de Laplace, on obtient :

$$\frac{A(x_{i+1}, y_j) - 2A(x_i, y_j) + A(x_{i-1}, y_j)}{h^2} + \frac{A(x_i, y_{j+1}) - 2A(x_i, y_j) + A(x_i, y_{j-1})}{h^2} \approx 0$$

En extrayant $A(x_i, y_j)$, on obtient :

$$A(x_i, y_j) \approx \frac{A(x_{i+1}, y_j) + A(x_{i-1}, y_j) + A(x_i, y_{j+1}) + A(x_i, y_{j-1})}{4}$$

La méthode de Jacobi consiste à calculer la solution à l'itération $t+1$ à partir de l'itération t de la manière suivante :

$$A^{t+1}(x_i, y_j) \approx \frac{A^t(x_{i+1}, y_j) + A^t(x_{i-1}, y_j) + A^t(x_i, y_{j+1}) + A^t(x_i, y_{j-1})}{4} \tag{3.1}$$

i et j varient de 1 à n et t est le numéro de l'itération. Nous avons précisé des conditions aux limites en affectant $A(i, 0)$, $A(i, n+1)$, $A(0, j)$, et $A(n+1, j)$. L'algorithme séquentiel de Jacobi pour résoudre l'équation de Laplace est alors le suivant :

Donner une valeur initiale à $A(i, j)$ pour $i, j = 1, n$
Pour t = 0, maxt **faire**
  Pour i = 1, n **faire**
    Pour j = 1, n **faire**

$$A^{t+1}(x_i, y_j) = \left(A^t(x_{i+1}, y_j) + A^t(x_{i-1}, y_j) + A^t(x_i, y_{j+1}) + A^t(x_i, y_{j-1})\right)/4$$

      Calcul de l'erreur
    Fin pour
  Fin pour
  Si Convergence **alors**
    Arrêt
Fin pour

Algorithme parallèle. À chaque itération, tous les calculs des $A^{t+1}(x_i, y_j)$ sont indépendants. Dans cet exemple, on utilise une distribution des données linéaire monodimensionnelle (voir la figure 3.1). On attribue à chaque processeur un bloc de lignes. À chaque itération, chaque processeur calcule ses solutions locales. Tous les processeurs testent la convergence. Pour cela, une opération de réduction globale permet à chaque processeur de récupérer les valeurs de l'erreur correspondant aux solutions locales des autres processeurs. S'il n'y a pas de convergence suffisante, les processeurs voisins échangent leurs lignes de solutions frontières et commencent l'itération suivante jusqu'au maximum d'itérations défini.

Les lignes frontières de blocs de données contigus sont dupliquées sur les processeurs voisins. Comme le montre la figure 3.1, sur une grille 16×16 par exemple, le processeur 0 doit calculer les lignes 0 à 3 mais a besoin de la ligne 4 pour son calcul : la ligne est dupliquée sur le processeur 0 et le processeur 1. Le processeur 1, quant à lui, calcule les lignes 4 à 7 mais a besoin de la ligne 3 pour faire son calcul. La ligne 3 est donc dupliquée sur le processeur 0 et sur le processeur 1. Au moment de l'échange, le processeur 0 envoie les nouvelles de la ligne 3 au processeur 1 et le processeur 1 envoie les nouvelles valeurs de la ligne 4 au processeur 0.

L'algorithme parallèle de Jacobi pour résoudre l'équation de Laplace est alors le suivant :

Donner une valeur initiale à $A(i, j)$ dans son bloc de lignes local
Pour t = 0, maxt **faire**

Échange des lignes frontières

Pour i dans le bloc local de lignes **faire**
Pour j = 1, n **faire**
Calcul de l'itération $t + 1$ (voir l'Équation 3.1)
Calcul de l'erreur
Fin pour
Fin pour
Réduction globale des valeurs d'erreur
Si Convergence **alors**
Arrêt
Fin pour

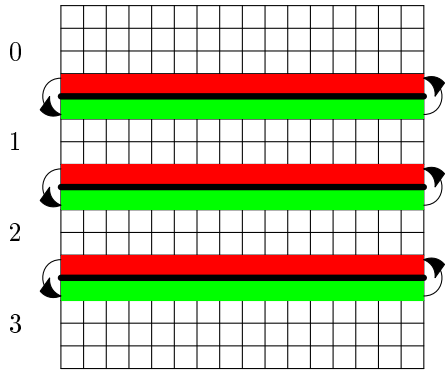


FIG. 3.1 – Distribution des calculs linéaire 1D.

Recouvrement des communications. Les calculs des lignes intérieures du bloc de lignes de chaque processeur ne dépendent pas des valeurs aux frontières et peuvent donc s'effectuer sans attendre que les lignes frontières soient échangées. Donc, à chaque itération, chaque processeur initialise l'échange des lignes frontières. Pendant que cet échange s'effectue, chaque processeur calcule les lignes intérieures non frontières au bloc de lignes local. À la fin du calcul, chaque processeur termine l'échange de lignes frontières et calcule les nouvelles valeurs des lignes frontières de son bloc de lignes local.

L'algorithme parallèle avec recouvrement des communications de Jacobi pour résoudre l'équation de Laplace est alors le suivant :

```
Donner une valeur initiale à A(i, j) dans son bloc de lignes local
Pour t = 0, maxt faire

  Démarrage de l'échange des lignes frontières

  Pour chaque ligne non frontière du bloc local faire
    Pour j = 1, n faire
      Calcul de l'itération t + 1 (voir l'Équation 3.1)
      Calcul de l'erreur
    Fin pour
  Fin pour

  Terminaison de l'échange des lignes frontières

  Pour chaque ligne frontière du bloc local faire
    Pour j = 1, n faire
      Calcul de l'itération t + 1 (voir l'Équation 3.1)
      Calcul de l'erreur
    Fin pour
  Fin pour
  Réduction globale des valeurs d'erreur
  Si Convergence alors
    Arrêt
Fin pour
```

Soit P le nombre de processeurs. Chaque processeur doit calculer $\frac{n}{P}$ lignes de n éléments. Nous espérons donc recouvrir le calcul de $(n-2)*\frac{n}{P}$ éléments avec l'échange de deux lignes de n éléments. La figure 3.2 donne un diagramme de Gantt de l'exécution sans recouvrement des communications de cinq itérations sans représenter l'initialisation de l'application ni la réduction globale de l'erreur. Les processeurs sont complètement synchronisés. La figure 3.3 donne cette même exécution avec recouvrement des communications. Dans le cas représenté, l'échange des deux lignes est un peu plus long que le calcul des $(n-2)*\frac{n}{P}$ éléments. Donc, avant de commencer le calcul des lignes frontières, chaque processeur doit attendre l'arrivée des données.

On voit que l'on espère un gain en temps d'exécution par itération égal au temps de calcul des $(n-2)*\frac{n}{P}$ éléments.

Les figures 3.4 et 3.5 représentent les mêmes exécutions que ci-dessus implantées en MPI (se reporter à la partie 2.5.2). Les traces ont été générées par l'outil `XMPI`. Ces traces représentent l'exécution complète de l'application. Elles incluent notamment les temps de réduction globale de l'erreur mais ces communications n'y sont pas représentées. La figure 3.5 montre que, bien que les communications des échanges se croisent, ce qui atteste d'une désynchronisation, le processeur 1, par exemple, attend que le processeur 0 soit prêt à recevoir pour effectuer réellement la communication. La latence en émission est très longue

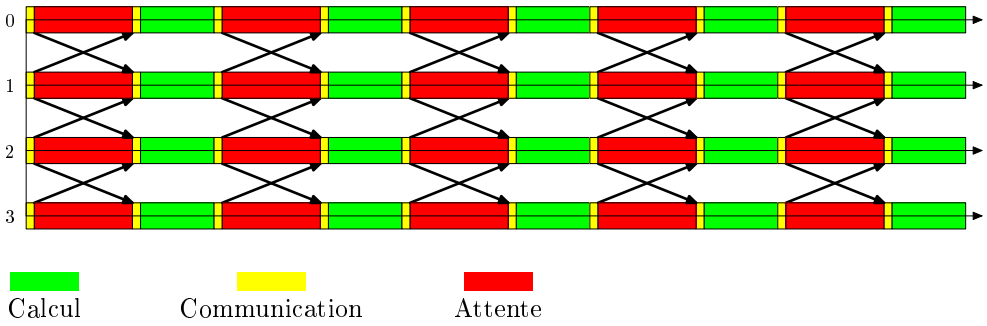


FIG. 3.2 – Diagramme de Gantt « théorique ». Version bloquante.

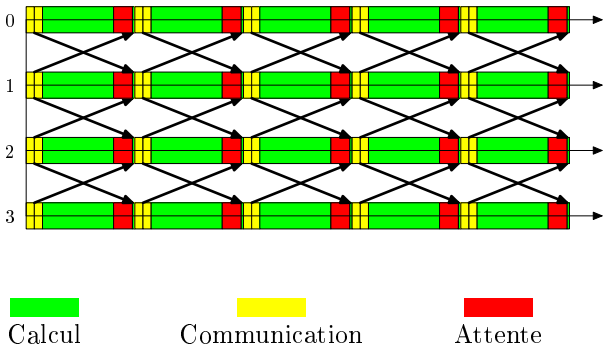


FIG. 3.3 – Diagramme de Gantt « théorique ». Version non bloquante.

sur le processeur 1. Le processeur 0 a pu, quant à lui, poster l'envoi, calculer puis recevoir et terminer son calcul. Le phénomène se répercute jusqu'au dernier processeur. À chaque itération, les processeurs sont synchronisés par la réduction globale. Tout se passe comme si on ne pouvait pas faire deux envois réellement asynchrones simultanément. Le résultat est que l'exécution avec recouvrement est plus longue que l'exécution sans recouvrement.

Les raisons de ce mauvais résultat ne sont pas facilement identifiables. Les implantations de MPI sont peu documentées et les processus étudiés sont compliqués. C'est pourquoi les parties suivantes sont consacrées à l'étude de l'implantation des communications dans une couche de communication de base et dans la bibliothèque standard MPI dans le but d'en évaluer les possibilités de recouvrement. La partie suivante présente les différents travaux déjà effectués sur les recouvrements et les moyens matériels et logiciels mis à disposition de la gestion des communications. Enfin, la partie 3.3 étudie en détail une implantation de MPI sur une grappe de PC.

3.2 Possibilités de recouvrements

De nombreux articles ont étudié la capacité de recouvrement des bibliothèques de communication et des différentes architectures. En effet, il faut non seulement que le matériel permette le recouvrement des communications mais aussi les bibliothèques de communication. Dans un premier temps, nous faisons un tour d'horizon des études déjà menées. La deuxième partie détaille les moyens matériels mis à disposition du processeur de calcul pour

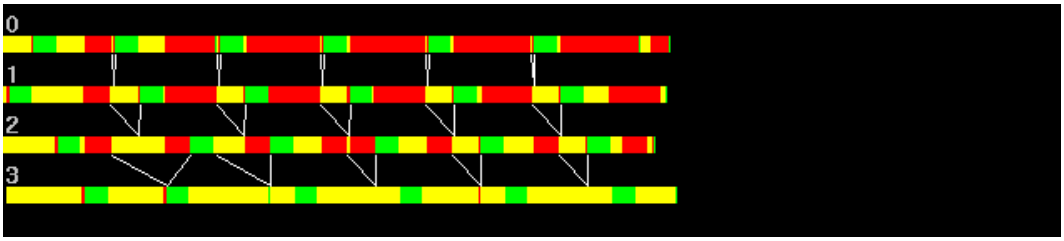


FIG. 3.4 – Diagramme de Gantt obtenu avec XMPI. Version bloquante.

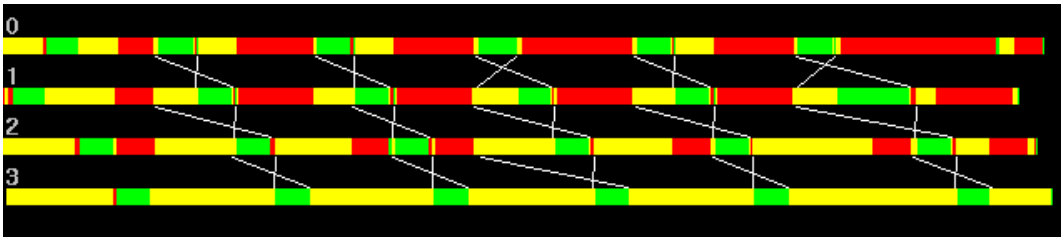


FIG. 3.5 – Diagramme de Gantt obtenu avec XMPI. Version non bloquante.

effectuer les communications. La troisième partie présente les moyens logiciels et notamment la bibliothèque standard MPI.

3.2.1 Mises en œuvre des recouvrements

Comme nous avons pu le constater dans la partie 3.1.1, l'utilisation de la bibliothèque MPI pour recouvrir les communications par du calcul ne donne pas toujours les résultats escomptés. En effet, les communications non bloquantes ne sont pas forcément asynchrones et coûtent plus cher que les communications bloquantes. C'est pourquoi l'utilisation d'une telle bibliothèque nécessite souvent malheureusement une bonne connaissance de l'implantation de MPI utilisée. Dans [51], Prieto *et al.* exécutent des programmes de test écrits avec MPI sur trois architectures différentes : IBM SP-2, Cray T3E et SGI Origin2000. Sur ces deux dernières architectures, ils obtiennent de plus mauvais résultats avec recouvrement que sans recouvrement. L'IBM SP-2 donne en revanche de bons résultats avec recouvrement. Leurs observations ne les mènent pas à mettre en cause les implantations de MPI mais plutôt un problème de localité de données. Pourtant, les différentes implantations de MPI donnent des résultats très différents en matière de recouvrement.

Plusieurs articles analysent les possibilités de recouvrement avec MPI. White et Bova [73] distinguent deux types de recouvrement : le recouvrement de synchronisation et le recouvrement de transfert de données. Le recouvrement de synchronisation permet à l'expéditeur d'envoyer son message sans avoir à se synchroniser avec le destinataire. Le recouvrement de transfert de données permet à un processeur d'effectuer un calcul pendant que le transfert physique des données a lieu. Nous nous sommes toujours placé dans le cadre du recouvrement de transfert de données. White et Bova montrent que toutes les implantations de MPI permettent le recouvrement de synchronisation mais que les possibilités de recouvrement de

transfert de données sont limitées. Ils étudient les implantations de MPI sur le Cray T3D, IBM SP-2 et sur SGI Origin2000. Sur ces deux dernières architectures, ils précisent comment positionner certaines variables globales pour que le recouvrement soit effectif. Sohn *et al.* [64] évaluent les capacités de recouvrement de l'implantation de MPI sur IBM SP-2 par l'intermédiaire d'une application de tri bitonique. Ils définissent une efficacité de recouvrement. Dans [63], la même analyse est effectuée pour la FFT sur une SGI PowerChallenge. Leur conclusion est que l'IBM SP-2 permet de recouvrir 30% de la communication pour un message inférieur à 1Ko. Sur l'application FFT, le SGI PowerChallenge permet un recouvrement de 20% des communications alors que l'IBM SP-2 permet d'en recouvrir 80%. L'IBM SP-2 a des comportements très différents selon les tailles de message à cause de l'architecture de son co-processeur de communication et notamment de la taille de sa mémoire. Le SGI PowerChallenge, quant à lui, ne possède pas de matériel dédié aux communications.

L'implantation de la bibliothèque de communication est très liée à l'architecture du matériel sur laquelle elle est implantée. Les possibilités de recouvrement des communications sont fortement liées au rôle attribué à chacun des composants matériels. Comme nous le verrons dans la partie 3.2.2, Scheiman et Schauser *et al.* [58, 40] étudient la nature et le rôle des co-processeurs de communication sur différentes architectures et en déduisent les possibilités de recouvrement. Il ressort de ces études que, le plus souvent, le matériel ne limite pas les possibilités de recouvrement des communications par le calcul.

Plusieurs articles décrivent les implantations de bibliothèques de communication non-standard en mettant l'accent sur les possibilités de recouvrement. Somani et Sansano [65] présentent l'implantation des recouvrements sur une architecture hiérarchique compliquée à base de grappes à mémoire partagée de processeurs. Cette architecture dédie un processeur à la communication pour quatre nœuds de calcul. Tanaka *et al.* [68] ont implanté des fonctions de communication sur des grappes de SMP interconnectées par Myrinet à base de primitives de communication NICAM. Ces fonctions permettent le recouvrement des communications inter-SMP. Ils obtiennent jusqu'à 25% de gain sur l'implantation d'un noyau de Gauss-Seidel. De la même manière, Baden et Fink [12] proposent un modèle de communication hiérarchique sur des grappes de SMP et obtiennent sur la relaxation de Gauss-Seidel un gain de 15%.

Le recouvrement des communications ne concerne pas uniquement les machines à mémoire distribuée. Sur des machines à mémoire partagée ou virtuellement partagée, il s'agit d'un mécanisme de chargement anticipé des données dans une zone de mémoire locale au processeur. Ce mécanisme est appelé mécanisme de *prefetch*. Abdelrahman et Liu [2] évaluent une transformation de code HPF effectuée à la compilation sur une grappe de PC à mémoire partagée. Cette transformation a pour but de recouvrir le temps de communication résultant des accès à des mémoires distantes avec le temps de calcul des boucles parallèles pour cacher la latence liée aux accès distants. Sur des exécutions d'itérations de Jacobi sur huit processeurs, ils obtiennent un gain d'environ trois.

3.2.2 Co-processeurs dédiés à la communication

Dans cette partie, nous étudions quels peuvent être les moyens matériels mis à la disposition du processeur de calcul afin d'effectuer un transfert de données.

La plupart des machines parallèles possèdent, à l'intérieur de chaque nœud, un co-

processeur de communication dédié afin d'accélérer les communications et de décharger le processeur de calcul.

Dans [58], Scheiman et Schauser répertorient les co-processeurs de communication de différentes machines et étudient les possibilités offertes par le co-processeur de communication sur l'architecture Meiko CS-2. Ils proposent de faire des catégories de co-processeurs de communications selon le matériel utilisé (contrôleur intelligent, processeur dédié ou processeur généraliste), ses capacités (possibilité d'exécuter un code en mode utilisateur ou noyau, ou un jeu restreint d'instructions), son utilisation (mémoire partagée ou mémoire distribuée) et l'interface le liant au processeur de calcul (bus mémoire, bus entrées/sorties ou un lien spécifique).

Par exemple, l'architecture Paragon possède deux à cinq processeurs Intel i860 dont un est le plus souvent utilisé comme co-processeur de communication ; le co-processeur dédié à la communication est donc identique au processeur de calcul. L'IBM SP-2 utilise un Intel i860 comme co-processeur de communication relié au bus Micro Channel (se reporter à la partie 2.4). Des co-processeurs de communication se trouvent aussi dans les cartes réseau reliant les nœuds de certaines grappes de PC. Par exemple, la carte FORE 200 ATM contient un processeur Intel i960 et la carte réseau Myrinet contient un processeur LANai. Nous reviendrons très en détail sur l'implantation d'une couche de communication de base et de la bibliothèque de communication MPI sur une grappe de PC reliée par un réseau Myrinet dans la partie 3.3. L'architecture Meiko possède un co-processeur de communication dédié relié au processeur de calcul Sparc par le bus mémoire. Donc ces deux architectures, la grappe de PC et la Meiko, ont un co-processeur intégrant l'interface réseau et ayant une puissance beaucoup plus limitée que le processeur de calcul. L'architecture Thinking Machines Corporation CM-5 ne possède pas de co-processeur de communication. Enfin, le Cray T3D fournit du matériel entièrement dédié aux lectures et écritures distantes. Il ne fournit donc pas de co-processeur programmable mais des fonctions de gestion de messages.

Dans l'article [40], Krishnamurthy *et al.* décrivent les différentes interactions possibles entre processeur de calcul et co-processeur lors d'un transfert de données sur cinq architectures différentes : l'architecture TMC CM-5, l'Intel Paragon, le Meiko CS-2, le Cray T3D et une grappe de PC reliées par un réseau Myrinet.

Un transfert de données est composé de trois étapes :
- le nœud expéditeur émet une requête,
- la requête est traitée sur le nœud destinataire par un accès à la mémoire, soit en mettant à jour une variable ou en générant un événement,
- la réponse ou une indication de fin est délivrée au nœud ayant émis la requête.

Au cours de toutes ces étapes, le flux d'information part du processeur de calcul, passe par le co-processeur de communication, les interfaces réseau et les systèmes gérant la mémoire.

Dans la plupart des cas, la communication de données appartenant à une application en cours implique la gestion de structures de données et un accès à la mémoire : par exemple, faire correspondre un message à un marqueur, insérer un élément dans une file, incrémenter une variable, stocker des données dans la mémoire. C'est pourquoi la gestion des communications sur le processeur principal est très coûteuse. De plus, le co-processeur peut gérer ces communications sans interrompre le processeur de calcul. Comme ce co-processeur est dédié aux communications, il peut gérer les messages en arrivée plus rapidement que le processeur

de calcul qui peut être occupé à autre chose. De plus, si tous les messages sont gérés par le co-processeur, le processeur de calcul n'a plus à être interrompu ou à scruter périodiquement l'arrivée de messages.

Cependant, les co-processeurs sont bien souvent des matériels dédiés qui ont une capacité de calcul limitée. Par exemple, certains co-processeurs ont une mémoire très limitée. C'est pourquoi le co-processeur de communication n'est pas toujours à même de gérer l'intégralité des communications.

Krishnamurthy *et al.* proposent quatre implantations différentes des communications :

Les requêtes sont gérées par le processeur de calcul. Le processeur de calcul injecte le message dans le réseau et le processeur de calcul distant reçoit le message, exécute la requête et répond. Cette même stratégie peut utiliser le co-processeur de communication, en l'utilisant comme une interface réseau évoluée. La communication entre le processeur de calcul et le co-processeur de communication se fera par l'intermédiaire d'une file d'attente se trouvant dans une zone de mémoire partagée. Le processeur de calcul écrit dans la file d'attente. Le co-processeur de communication, scrutant constamment la file d'attente, récupère les données en mémoire et les injecte dans le réseau. Sur le nœud distant, le message est reçu par le co-processeur de communication, stocké en mémoire partagée et finalement géré par le processeur de calcul. Dans ce cas, la tâche du co-processeur de communication est de déplacer les données entre la mémoire partagée et le réseau et contrôler le flux sortant et surtout entrant.

Les requêtes sont gérées par le co-processeur de communication. L'initialisation de la requête est faite comme précédemment. Le processeur de calcul utilise la mémoire partagée pour communiquer la requête au co-processeur de communication. Sur le nœud distant, le co-processeur de communication reçoit le message, exécute la requête et envoie la réponse sans impliquer le processeur de calcul.

Le processeur de calcul injecte directement les messages dans le réseau. Sur le nœud distant, le co-processeur de communication reçoit le message, exécute la requête et envoie la réponse qui ensuite est reçue et gérée par le co-processeur de communication sur le nœud ayant émis la requête.

La réception d'un message se fait par le processeur de calcul ou par le co-processeur de communication selon leur disponibilité. Le processeur de calcul injecte le message dans le réseau. Sur le nœud distant, la requête est gérée soit par le processeur de calcul soit par le co-processeur de communication selon leur disponibilité. De même, au retour de la réponse, celle-ci est gérée par l'un des deux processeurs sur le nœud émetteur.

Ces différents cas ne sont pas exhaustifs. Ils ne donnent que quelques grandes lignes des différents choix liés à l'implantation d'une couche de communication. Néanmoins, Krishnamurthy *et al.* ont testé ces quatre stratégies. Sur le CM-5, en raison de l'absence de co-processeur de communication, une seule stratégie a été testée : toutes les requêtes de communication sont gérées par le processeur de calcul. Sur le Meiko et la grappe de PC, comme l'interface

réseau est intégrée au co-processeur de communication, le processeur de calcul ne peut pas agir directement sur le réseau. Par conséquent, les deux premières stratégies ont été testées : gestion des requêtes par le processeur de calcul et gestion des requêtes par le co-processeur de communication. Enfin, sur le Paragon, il a été possible d'implanter toutes les stratégies.

Les performances obtenues dépendent des puissances relatives des deux processeurs et de leurs charges. On s'attend à avoir les meilleures performances lorsque les requêtes sont gérées par le co-processeur de communication. Pourtant, lorsque celui-ci est beaucoup plus lent que le processeur de calcul comme sur le Meiko, il peut être plus rapide de ne faire effectuer au co-processeur de communication que les opérations les plus simples et de passer les plus complexes au processeur de calcul. Les performances dépendent aussi des charges respectives des deux processeurs. Si le processeur de calcul est chargé, il peut être plus rapide de faire exécuter les opérations par le co-processeur de communication même si celui-ci est plus lent.

Le recouvrement d'une communication peut concerner le transfert des données sur le réseau, la mise en forme du message et enfin la recopie des données du message à partir et dans l'espace d'adressage du programme à l'envoi et à la réception. Selon la répartition des tâches entre processeur de calcul et co-processeur de communication, la mise en forme du message pourra se faire sans intervention du processeur de calcul. Si la plupart des architectures permettent que le transfert des données sur le réseau se fasse indépendamment du processeur de calcul, la recopie des données du message dépend fortement de l'architecture du co-processeur de communication. Dans [33], Ishizaki *et al.* montrent que l'architecture IBM SP ne permet pas le recouvrement des recopies entre zones de mémoire système et utilisateur avec le calcul. Hiranandani *et al.* affirment, en revanche, dans [28] que l'architecture Intel iPSC/860 permet le recouvrement des recopies par le calcul.

3.2.3 Moyens logiciels

Au point de vue du logiciel, il faut que la couche de communication ou le support d'exécution permette les recouvrements. Pour cela, il faut que la communication puisse s'effectuer de manière concurrente par rapport aux processus participant à la communication, c'est-à-dire que le transfert de données puisse s'effectuer pendant que les processus calculent.

Cela ne signifie pas que les communications doivent nécessairement être non bloquantes (nous avons donné la définition d'une communication non bloquante dans la partie 2.5.2). En effet, l'appel à un envoi bloquant de données peut se terminer en commençant le transfert immédiatement, donc libérer la zone de mémoire de l'envoi et rendre la main rapidement. Au moment où le processus distant réceptionne les données, celles-ci sont déjà dans la mémoire locale du processeur distant, le processus destinataire n'a donc pas à attendre. La communication est, dans ce cas, totalement asynchrone et pendant le temps de transfert, les deux processus ont pu effectuer des calculs. Bien sûr, cela suppose que les zones mémoires de réception ont été prévues à l'avance.

Dans le cas de communications point-à-point classiques où le récepteur précise au moment de la réception une zone de mémoire où stocker le message, il est possible de recouvrir le transfert des données en utilisant une réception non bloquante. Le principe est que le processus destinataire initialise la réception en avance pour qu'au moment où le processus expéditeur envoie les données, la zone de stockage du message sur le processeur destinataire

soit connue.

Ici encore, l'envoi peut être bloquant et la communication recouverte puisque, si au moment où le processus expéditeur envoie les données, le processus destinataire a déjà fourni l'information nécessaire, la fonction d'envoi peut transférer immédiatement les données et se terminer.

Néanmoins, dans la pratique, il est très difficile de savoir si la réception a lieu avant l'envoi. De plus, l'envoi des messages sur le réseau se fait parfois de manière fragmentée, donc il faudrait attendre effectivement l'envoi de tous les fragments, si le message n'est pas recopié dans une zone de mémoire tampon locale à l'expéditeur, en utilisant un appel bloquant. C'est pourquoi on préfère, dans la pratique, utiliser des appels non bloquants à la fois sur le processus expéditeur et sur le processus destinataire.

Tout au cours de cette partie, nous avons distingué communication asynchrone et communication non bloquante. Utiliser des fonctions non bloquantes et synchrones permet-il le recouvrement des communications par le calcul ? Si le terme synchrone signifie que le matériel ne permet pas le transfert de données pendant que les processus calculent, alors il n'y aura bien sûr aucun recouvrement. Si la communication est synchrone au sens du mode synchrone MPI défini dans la partie 2.5.2, l'appel à l'envoi non bloquant sur le processus expéditeur ne se terminera que lorsque le processus destinataire commencera à effectivement recevoir les données. Dans ce cas, si l'on considère que le processus destinataire peut commencer à effectivement recevoir des données dès l'initialisation de la réception, alors il peut y avoir recouvrement des communications. Si l'on considère que le processus destinataire ne reçoit effectivement les données qu'à la terminaison de la communication alors il n'y a aucun recouvrement.

Recouvrements en MPI. Nous avons présenté dans la partie 2.5.2 les différents modes de communication point-à-point de la bibliothèque MPI. Le mode standard est le mode de communication le plus général. S'il n'y a pas de réception correspondante, le mode standard sera équivalent au mode avec recopie. Si la réception correspondante a été postée, le mode standard n'effectuera probablement pas la communication aussi efficacement que le mode « prêt ». Enfin, le mode standard ne garantit pas que l'envoi ne se termine qu'au moment où le processus distant commence à recevoir les données comme le mode synchrone mais il est possible qu'en pratique la communication s'effectue de cette manière même dans le mode standard.

De plus, nous avons vu dans la partie précédente que selon l'implantation de MPI, l'utilisation de mode bloquant pouvait générer du recouvrement des communications dans des cas particuliers notamment lorsque la zone mémoire de réception est préalablement fournie et que le transfert de paquet ne s'effectue pas de manière fragmentée. Cependant, pour s'abstraire au maximum de l'implantation, nous utiliserons les primitives non bloquantes du mode standard : `MPI_Isend()` et `MPI_Irecv()` et des primitives de terminaison comme `MPI_Test()` et `MPI_Wait()`.

Le pseudo-code ci-dessous donne un exemple d'utilisation de ces primitives. L'échange d'un tableau est initialisé en postant une requête d'envoi et une requête de réception sur chaque processeur. Pendant le transfert des données, on souhaite effectuer un calcul indépendant des données transférées. Enfin, avant d'utiliser les données mises à jour, il faut

attendre la fin de la communication.

```
Tant que Non terminé Faire
  (* Initialisation des communications *)
  MPI_Isend(tableau, ..., sendRq) (* Requête d'envoi *)
  MPI_Irecv(tableau, ..., recvRq) (* Requête de réception *)

  Calcul_indépendant_de_tableau()

  (* Terminaison des communications *)
  MPI_Wait(sendRq, ...)
  MPI_Wait(recvRq, ...)

  Calcul(tableau)
Fin tant que
```

La capacité de la bibliothèque MPI de gestion des communications asynchrones est remise en cause notamment parce que les spécifications prévoient qu'un envoi non bloquant peut être posté que la réception correspondante ait été postée ou non et qu'une implantation de MPI doit pouvoir supporter un grand nombre d'opérations non bloquantes en suspens. Ceci pose des problèmes pratiques comme la congestion du réseau et la gestion des communications en suspens. Dans [24], Edwards propose une interface MPI étendue permettant de mieux gérer les communications asynchrones.

3.2.4 Synthèse

Nous avons étudié tout au long de cette partie les moyens matériels et logiciels de mise en œuvre des recouvrements des communications par le calcul. La plupart des architectures matérielles actuelles possèdent un co-processeur de communication capable de décharger le processus de calcul d'une partie de la charge liée aux communications. Un simple transfert de données peut s'effectuer sans solliciter le processeur de calcul de manière totalement asynchrone. De plus, les bibliothèques de communication comme MPI fournissent des primitives permettant le recouvrement des communications. Cependant, si beaucoup d'applications ont été optimisées grâce aux recouvrements des communications (voir la partie 3.1), leur mise en œuvre est très souvent effectuée par l'intermédiaire de bibliothèques de communication natives. La partie 3.2.1 montre que l'utilisation de MPI pour effectuer le recouvrement des communications est très souvent délicate. Elle nécessite une bonne connaissance de l'implantation de MPI comme dans les cas de l'IBM SP-2 et SGI Origin2000 où il faut modifier la valeur de certaines variables globales pour obtenir de bonnes performances [73]. Parfois, comme nous le montrons dans la partie suivante, l'implantation de MPI rend tout simplement impossible le recouvrement des communications.

3.3 Exemple d'implantation de MPI

Cette partie décrit l'implantation de la bibliothèque MPI présentée dans la partie 2.5.2 au-dessus de la couche de communication BIP reliant des PC connectés par le réseau ra-

pide Myrinet. Cette description a pour but d'étudier les possibilités de recouvrement des communications par le calcul fournies par MPI.

La première partie prouve que le matériel composant la plate-forme et la couche de communication de base BIP permettent de recouvrir les calculs par des communications. La deuxième partie explique pourquoi, malgré les possibilités offertes par à la fois le matériel et le logiciel sous-jacents, l'implantation de la bibliothèque de communication standard MPI [62] implantée au-dessus de BIP ne permet pas le recouvrement. La dernière partie propose des solutions en termes d'implantation et d'équilibrage de charge entre le processeur de calcul et le processeur de communication.

La plate-forme expérimentale est la grappe de PC PoPC installée dans le cadre du LHPC. Cette grappe de PC est décrite dans la partie 2.4.

3.3.1 La couche de communication de base : BIP

BIP [1] [52] est une couche de communication rapide disponible sur un réseau Myrinet. Elle permet d'atteindre 1Gb/s de débit. Dans la suite de cette partie, nous allons détailler l'implantation de cette couche de communication permettant d'atteindre de tels débits.

Nous considérons ici le problème simple consistant à envoyer un message à partir de la mémoire du microprocesseur hôte vers la mémoire du microprocesseur distant.

Sémantique

BIP implante des sémantiques différentes pour les messages longs (de taille supérieure à 4Ko) et pour les messages courts (de taille inférieure à 4 Ko).

Les envois et réceptions des longs messages ont une sémantique de rendez-vous, c'est-à-dire qu'une réception doit être postée avant que l'envoi correspondant n'ait commencé.

À l'opposé, les messages courts respectent la sémantique « eager » . Ils sont stockés dans une file d'attente circulaire afin que les envois n'aient pas à attendre que les réceptions correspondantes ait été postées.

De plus, on ne peut poster qu'un seul envoi ou une réception non bloquante à la fois.

Implantation

Les différentes étapes de la communication. Comme l'interface réseau fait partie du processeur réseau, le processeur de calcul ne peut pas envoyer directement les données dans le réseau. La figure 3.6 décrit les différentes étapes de la communication de données. La figure 3.7 décrit l'implantation de la communication à la fois sur le nœud expéditeur et sur le nœud destinataire.

Pendant la première étape indiquée sur la figure 3.6, le processeur de calcul communique avec le processeur réseau par l'intermédiaire d'une file d'attente circulaire se trouvant dans la mémoire locale du processeur réseau (voir figure 3.7). Cette file d'attente circulaire est accessible par le processeur de calcul. Le processeur de calcul remplit la file d'attente circulaire avec des messages qui ont la structure suivante :

– un en-tête physique composé d'octets de routage,

[1] BIP : Basic Interface for Parallelism.

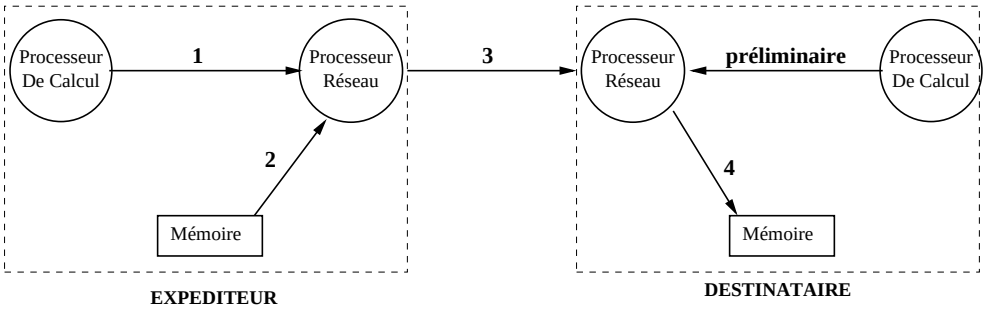


FIG. 3.6 – Les différentes étapes de la communication : le processeur de calcul fait une requête d'envoi auprès du processeur réseau (1). Le processeur réseau récupère les données avec un transfert DMA à partir de la mémoire vers la mémoire de la carte (2). Il injecte les données dans le réseau. Le processeur réseau distant récupère les données sur le réseau (3). Le processeur de calcul transmet au processeur réseau une liste d'adresses mémoire où stocker le message suivant (préliminaire). Ainsi le processeur réseau peut déposer les données dans la mémoire grâce à un transfert DMA (4).

- un en-tête logique composé des numéros de l'expéditeur et du destinataire, de la longueur du message et quelques autres séquences de numéros,
- et, si le message est court, les données du message ou, si le message est long, une liste d'adresses des blocs de mémoire contenant les données du message.

De plus, chaque message est identifié par un marqueur. Le nombre de marqueurs est limité.

Pendant ce temps, le processeur réseau scrute périodiquement la file d'attente circulaire afin de relever l'arrivée de messages.

La deuxième étape décrite par la figure 3.6 est le transfert des données à partir de la mémoire du microprocesseur hôte vers la mémoire de la carte réseau :

- si le message est court, cette étape n'a pas lieu parce que le processeur a directement copié les données dans la file d'attente circulaire située dans la mémoire de la carte réseau ;
- si le message est long, le processeur réseau initialise un transfert DMA entre la mémoire du microprocesseur hôte et la mémoire de la carte réseau afin de copier des données pointées par la liste d'adresses. Le transfert du message est fragmenté en plusieurs transferts pipelinés avec la troisième étape (voir figure 3.8).

La troisième étape décrite par la figure 3.6 est l'injection de données dans le réseau par le processeur réseau et la récupération de ces données par le processeur réseau distant. L'injection de données dans le réseau consiste en un transfert DMA entre la mémoire de la carte réseau et le réseau :

- si le message est court, le processeur réseau écrit les données directement dans le réseau à partir de la structure de données stockée dans la file d'attente circulaire ;
- si le message est long, le processeur réseau envoie tout d'abord les en-têtes et envoie par la suite les fragments de données de manière pipelinée avec les étapes précédentes.

Sur le processeur réseau distant, un tampon de mémoire (voir figure 3.7) est prêt à recevoir le nouveau message. Enfin, des files de réception circulaires se trouvent sur le processeur de

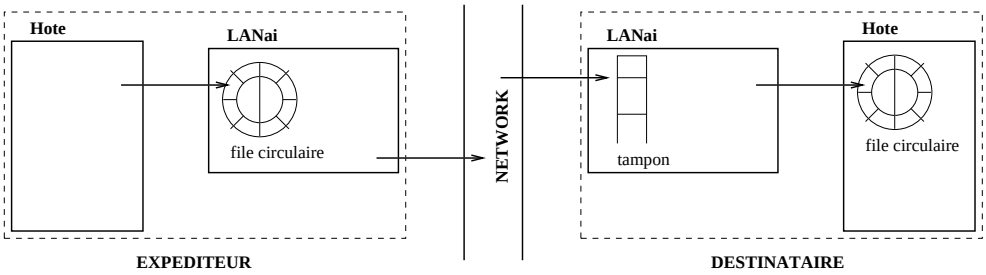


FIG. 3.7 – Implantation de BIP.

calcul distant (voir figure 3.7) pour chaque marqueur de message possible.

Le processeur de calcul distant maintient plusieurs listes à jour composées d'un pointeur sur le message courant dans la file de réception circulaire correspondante, le nombre de messages consommés et une liste d'adresses des blocs de mémoire constituant le message à recevoir pour chaque message en réception. Il y a autant de listes qu'il y a de marqueurs de messages. Ces listes sont stockées dans la mémoire du processeur réseau.

L'étape asynchrone notée `préliminaire` dans la figure 3.6 consiste à poster la réception correspondant à l'envoi. Elle met à jour le pointeur courant et la liste d'adresses de réception dans la file de réception circulaire.

Dès que l'en-tête du message est arrivé, le processeur réseau détermine si le message est long ou court :

- si le message est court, le processeur réseau attend jusqu'à ce que les données soient arrivées par l'intermédiaire du transfert DMA entre le réseau et la mémoire de la carte réseau et passe directement à la quatrième étape ;
- si le message est long, il vérifie que le processeur de calcul a bien fourni la liste d'adresses de réception. Ensuite, pour chaque fragment de message, il initialise le transfert DMA entre le réseau et la mémoire de la carte réseau de manière pipelinée (voir figure 3.8) avec la quatrième étape.

La quatrième et dernière étape (voir figure 3.6) consiste à placer les données dans la mémoire principale à partir de la mémoire de la carte :

- si le message est court, le processeur réseau place directement les données de la file de réception circulaire correspondant au marqueur de message dans la mémoire principale grâce à un transfert DMA entre la carte réseau et la mémoire principale ;
- si le message est long, pour chaque bloc de données reçu, il effectue le transfert DMA nécessaire entre la carte et la mémoire principale de manière pipelinée avec l'étape précédente (voir figure 3.8).

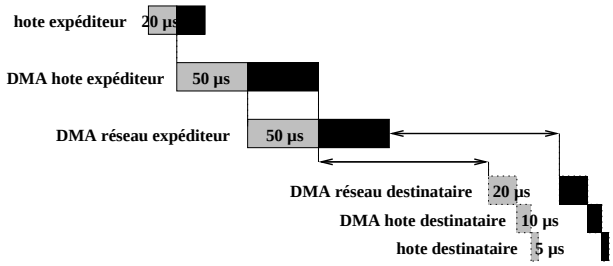


FIG. 3.8 – Flux de fragments de messages de 4Ko traversant le réseau.

Implantation sur le processeur réseau. L'implantation de BIP sur le processeur réseau gère la plus grande partie de la charge liée à la communication. Le programme principal est donné ci-après.

```
Procédure Processeur_Réseau()
   Initialisation d'un tampon de réception
   Tant que OK (* Scrute une zone mémoire *) faire
     Si requête d'envoi alors
       traite_requête_envoi()
     Fin si
     Si requête de réception alors
       traite_requête_réception()
     Fin si
   Fin tant que
Fin procédure (* Processeur_Réseau() *)
```

Tout d'abord, le processeur réseau initialise une zone mémoire tampon de réception, cette zone est de 128Ko dans l'implantation actuelle de BIP. Puis il scrute une zone de mémoire afin de vérifier si le processeur de calcul a posté une réception ou un envoi.

Si un envoi a été effectué, le processeur réseau lit l'en-tête du message de la file d'attente circulaire se trouvant dans sa propre mémoire (voir le pseudo-code ci-après). Deux cas peuvent se produire :

- si le message est court, les données se trouvent alors déjà dans la mémoire de la carte réseau et le processeur ne fait qu'un seul transfert DMA réseau ;
- si le message est long, les données se trouvent alors toujours dans la mémoire du microprocesseur hôte. Le processeur réseau découpe le message en fragments et pipeline les transferts DMA hôte avec les transferts DMA réseau.

Enfin, le processeur réseau met à jour la file d'attente circulaire se trouvant dans sa mémoire locale.

```
Procédure traite_requête_envoi()
  Lecture de l'en-tête du message
  Si message court alors
    Transfert DMA réseau du message
  Fin si (* Message court. *)
  Si message long alors
    Transfert DMA hôte du premier fragment du message
    Pour i allant de 1 au nombre de fragments composant le message faire
      Transfert DMA réseau du fragment de message i
      Transfert DMA hôte du fragment de message i + 1
    Fin pour
    Transfert DMA réseau du dernier fragment de message
  Fin si (* Message long. *)
  Mise à jour de la file d'attente circulaire
Fin procédure (* traite_requête_envoi() *)
```

Si une réception a été effectuée, alors le processeur réseau lit l'en-tête du message de la file circulaire (voir le pseudo-code ci-dessous). Deux cas peuvent se produire :

- si le message est court, les données sont directement transférées dans la mémoire du microprocesseur hôte dans une file d'attente circulaire au moyen d'un transfert DMA hôte (il n'est pas nécessaire de transférer les données à partir du réseau parce qu'elles sont intégralement contenues dans l'en-tête du message déjà lu) ;
- si le message est long, si la réception correspondante n'a pas été postée par le microprocesseur hôte alors il y a une erreur de protocole sinon deux transferts DMA sont nécessaires, du réseau vers la carte puis de la carte vers la mémoire hôte pour chaque fragment de message arrivant. Ces transferts DMA sont exécutés de manière pipelinée.

Enfin, le processeur réseau met à jour sa propre mémoire.

```
Procédure traite_requête_réception()
  Attente jusqu'à ce que l'en-tête du message de réception soit complet.
  Si message court alors
    Transfert DMA hôte vers la file d'attente circulaire correspondante sur l'hôte
  Fin si (* Message court. *)
  Si message long alors
    Si la réception correspondante a été postée alors
      Transfert DMA réseau du premier fragment du message
        avec la liste correspondante d'adresses de réception
      Pour i allant de 1 au nombre de fragments composant le message faire
        Transfert DMA hôte du fragment du message i
        Transfert DMA réseau du fragment du message i + 1
      Fin pour
      Transfert DMA hôte du dernier fragment du message
    Sinon
      Erreur de protocole
    Fin si
  Fin si (* Long message. *)
  Mise à jour du tampon de réception
Fin procédure (* traite_requête_réception() *)
```

Recouvrement des calculs et des communications avec BIP

La couche de communication de base BIP offre la possibilité de recouvrir les communications avec les calculs pour deux raisons. La première raison est, qu'après avoir initialisé la communication, le processeur de calcul n'intervient plus dans le processus de communication. La deuxième raison est que la charge liée à la communication sur le processeur de calcul est très faible.

Interaction entre le processeur de calcul et le processeur de communication. L'interaction entre les processeurs de calcul et de communication pour envoyer des données est minimale. En effet, si le message est court, une seule communication est nécessaire pour envoyer les données. Si le message est fragmenté alors le processeur de calcul donne au processeur de communication toutes les informations nécessaires (par exemple la liste des adresses des blocs de mémoire contenant les données du message) afin qu'il puisse gérer la communication de tous les fragments du message par lui-même. Enfin le processeur de calcul peut exécuter, pendant toute la durée de la communication, d'autres calculs indépendants sans qu'il soit nécessaire de les interrompre.

Charge liée à la communication sur le processeur de calcul. En plus d'une interaction minimale entre les processeurs de calcul et de communication pour effectuer une communication, la charge liée à la communication sur le processeur de calcul est très faible. En effet, comme nous l'avons vu dans la partie 3.3.1, le processeur réseau a la charge de la plus grande partie de la gestion de la communication. Quand une primitive d'envoi asyn-

chrone BIP est appelée, le processeur de calcul écrit uniquement dans la zone de mémoire du processeur réseau la structure de données nécessaire pour la communication du message dont notamment la liste des adresses des blocs de mémoire contenant les données du message. De manière symétrique, lorsqu'une primitive de réception asynchrone BIP est appelée, le processeur de calcul écrit uniquement dans la zone de mémoire du processeur réseau la liste des adresses de réception du message et met à jour le pointeur. Le processeur de calcul n'a aucun calcul à effectuer pour la gestion des communications. C'est pourquoi l'appel à une primitive d'envoi ou de réception asynchrone BIP se termine rapidement.

Expériences. En utilisant des primitives BIP natives, il est alors possible de recouvrir les communications par des calculs.

Nous avons exécuté un programme de test avec des communications bloquantes puis non bloquantes sur deux processeurs. Premièrement, le processeur 0 envoie les données au processeur 1, ils attendent la fin de la communication, exécutent un calcul et ensuite le processeur 1 renvoie les données au processeur 0, les deux attendent jusqu'à la fin de la communication et exécutent le même calcul (voir figure 3.9).

Les tailles des messages échangés varient de 0Ko à 1Mo. La figure 3.10 donne les résultats pour un temps de calcul égal au temps de communication d'un message de 8Ko. Dans la version bloquante, le temps total est égal à la somme du temps de calcul et du temps de communication. Dans la version non bloquante, le temps total est égal au temps de calcul ajouté au temps de communication non recouverte. Dans le cas non bloquant idéal, si le temps de communication est inférieur au temps de calcul alors le temps de communication devrait être entièrement recouvert et le temps total devrait être égal au temps de calcul.

Recouvrement total. Comme le montre la figure 3.10, le degré de recouvrement fourni par la couche de communication BIP est proche de l'idéal.

Contentions sur le bus mémoire. La figure 3.11 montre que si l'on augmente la taille de la matrice à calculer, les communications sont affectées par des contentions sur le bus mémoire.

3.3.2 L'implantation de MPI au dessus de BIP

Dans cette partie, nous détaillons l'implantation de MPI au dessus de BIP. Nous expliquons pourquoi, bien que BIP fournisse des possibilités de recouvrement, cette implantation de MPI ne le permet pas. Nous évoquons tout d'abord la sémantique que doivent respecter les fonctions de communication MPI puis leur implantation.

Sémantique

Une procédure est non bloquante si cette procédure peut se terminer avant que l'opération ne soit totalement effectuée et avant que l'utilisateur ne soit autorisé à réutiliser les ressources (par exemple les tampons de mémoire) spécifiées dans l'appel. L'appel à un envoi non bloquant initialise l'opération d'envoi mais ne la termine pas. L'appel à un envoi non

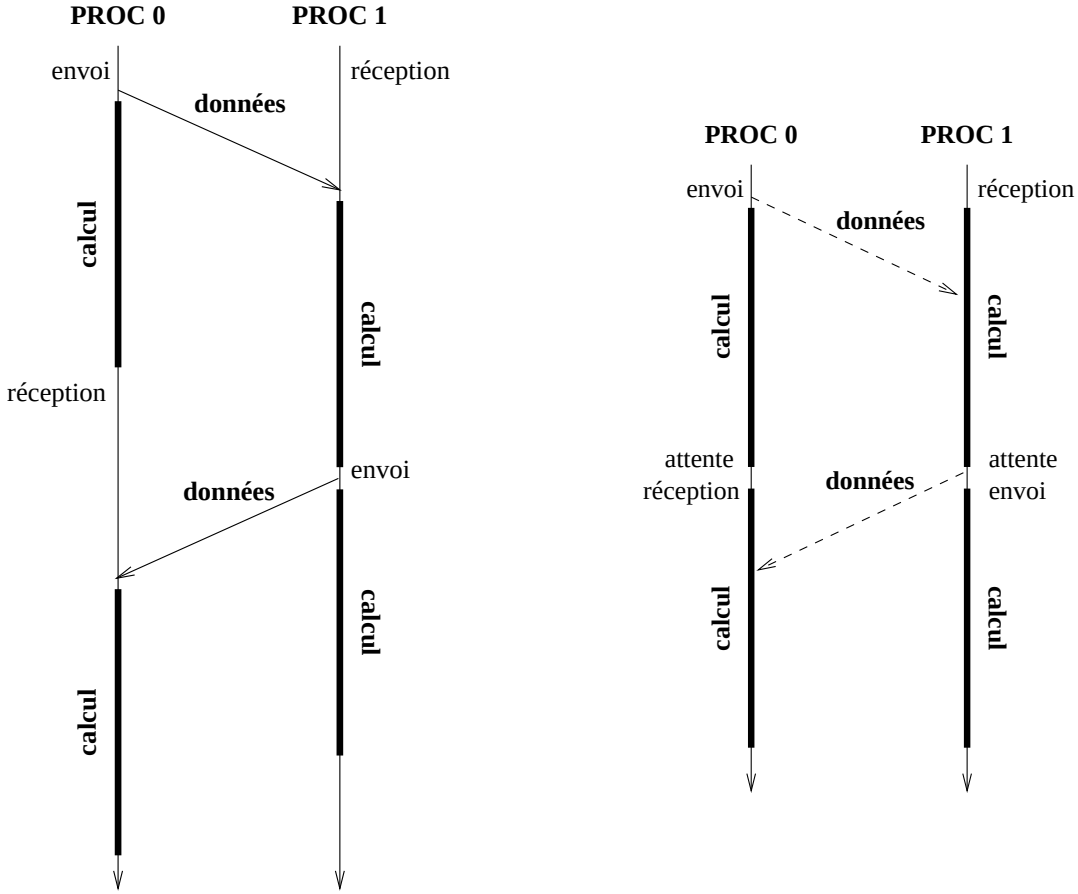
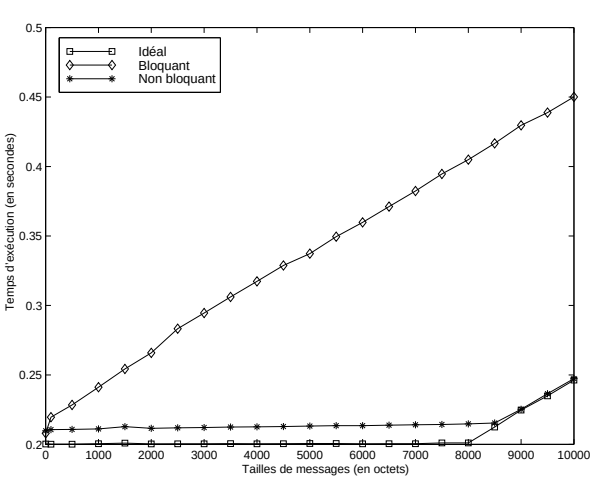
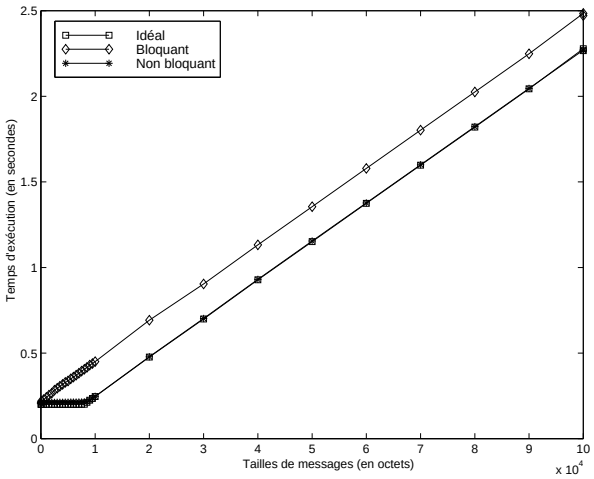


FIG. 3.9 – Exécutions d'un programme de test avec des communications respectivement bloquantes et non bloquantes.

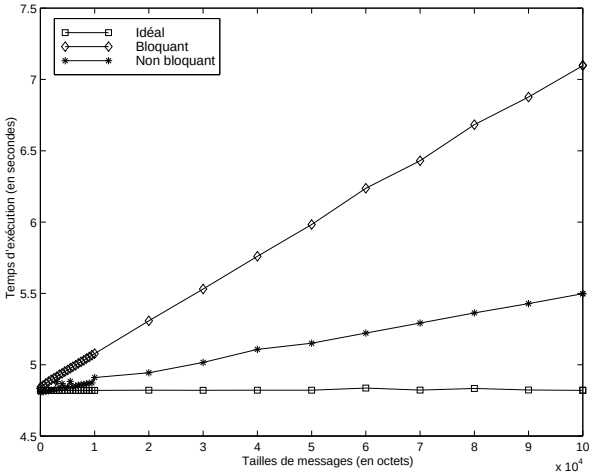


(a) Gros plan sur les petits messages.

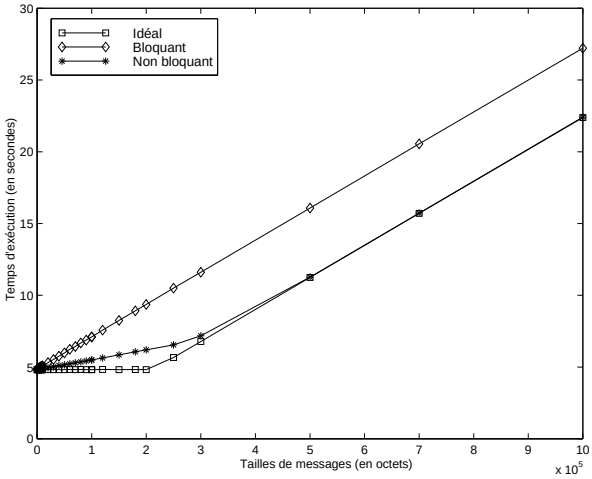


(b) Messages variant de 0 à 100 Ko.

FIG. 3.10 – Recouvrement des communications BIP natives avec des calculs tenant dans le cache.



(a) Messages variant de 0 à 100 Ko.



(b) Messages variant de 0 à 1 Mo.

FIG. 3.11 – Recouvrement des communications BIP natives avec des calculs plus importants.

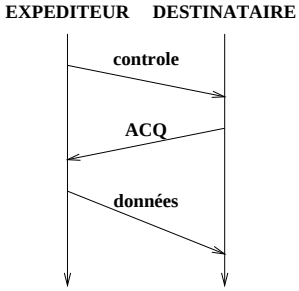


FIG. 3.12 – Les communications nécessaires à l'envoi non bloquant MPI-BIP d'un long message.

bloquant se terminera avant que le message ne soit copié en dehors du tampon mémoire d'envoi.

Un autre appel à une fonction d'envoi est nécessaire pour terminer la communication, c'est-à-dire pour vérifier que les données sont copiées en dehors du tampon mémoire d'envoi.

Avec le matériel qui convient, le transfert des données en dehors de la mémoire de l'expéditeur peut s'effectuer en même temps que des calculs entre l'initialisation de l'envoi et sa terminaison. L'utilisation de réceptions non bloquantes peut aussi éviter des recopies de la part du système et des recopies de mémoire à mémoire parce que les informations sur l'emplacement de la zone mémoire de réception sont fournies suffisamment tôt.

Un envoi non bloquant peut être posté que la réception correspondante ait été postée ou non. Une implantation de MPI doit pouvoir supporter un « *grand nombre* »[2] d'opérations non bloquantes en suspens.

Implantation

Dans l'implantation de MPI au dessus de BIP, trois communications sont nécessaires pour envoyer des données avec une primitive non bloquante (voir figure 3.12) : la requête d'envoi, l'acquittement, et la communication des données.

Ces trois communications sont nécessaires pour deux raisons. La première raison vient de la sémantique de MPI. En effet, l'implantation des trois communications est un moyen simple de supporter un grand nombre d'envois non bloquants en suspens. Cela permet de contrôler le flot de communications. En effet, la réception de l'acquittement assure que la réception pourra être effectuée par le processus destinataire. Dans le cas contraire, les données sont conservées localement. La deuxième raison vient de la sémantique de BIP : la réception doit être postée avant que l'envoi correspondant ait commencé. Grâce à ces trois communications, il est facile de garantir cette condition. Le processus destinataire ne renverra l'acquittement qu'après avoir initialisé la réception.

L'implantation MPI des primitives non bloquantes `MPI_Isend()` et `MPI_Irecv()` et des primitives de terminaison `MPI_Test()` et `MPI_Wait()` est construite sur un sous-ensemble de fonctions BIP qui sont :

[2]La spécification de MPI [62] précise : « an MPI implementation should be able to support a large number of pending nonblocking operations ».

- `SndControl()` qui envoie des messages de contrôle ; ceci correspond à l'envoi d'un petit message BIP ;
- `RcvControl()` qui reçoit des messages de contrôle ; ceci correspond à la réception d'un petit message BIP consistant en une lecture dans la mémoire principale ;
- `ISndData()` qui envoie les données de manière non bloquante, c'est un appel à une fonction d'envoi non bloquante BIP ;
- `IRcvData()` qui reçoit les données de manière non bloquante, c'est un appel à une fonction de réception non bloquante BIP ;
- `TSnd()` qui teste et termine une requête d'envoi ; ceci correspond au test d'envoi de BIP ;
- et `TRcv()` qui teste et termine une requête de réception ; ceci correspond au test de réception de BIP.

Structures de données. Comme le montre la figure 3.14, les structures de données nécessaires pour gérer une communication non bloquante se trouvent essentiellement sur le récepteur.

La structure de données stockant une requête contient l'enveloppe du message composée de la source, de la destination, du marqueur de message, du communicateur et de la taille du message. Il faut noter que pour une requête de réception le marqueur peut être `MPI_ANY_TAG`, c'est-à-dire prêt à recevoir tous les marqueurs.

Trois différents cas de requête de communication peuvent se produire (voir figure 3.14) :
- les requêtes d'envoi postées sur l'expéditeur (`SReqS`),
- les requêtes d'envoi postées sans aucune réception correspondante sur le récepteur (`RReqS`),
- et les requêtes de réception postées sans aucun envoi correspondant sur le récepteur (`RReqR`).

Sur le récepteur, deux tables sont aussi stockées :
- une table contenant des requêtes d'envoi reçues sans aucune requête de réception correspondante (la table `RSnd`) ; il y a une entrée par message et pour chaque message, la requête d'envoi et un pointeur sur la requête d'envoi courante sur l'expéditeur sont stockés ;
- une table contenant les requêtes de réception postées sans aucune requête d'envoi correspondante (la table `RRcv`) ; il y a aussi une entrée par message et pour chaque message, la requête de réception et une zone de mémoire tampon de réception sont stockées.

Ces tables sont stockées dans la mémoire principale du processeur de calcul. Elles ne se trouvent pas dans la mémoire du processeur de communication parce qu'il est nécessaire de pouvoir gérer un grand nombre de communications non bloquantes en suspens.

La fonction Progress(). Comme le montre le pseudo-code ci-dessous, la fonction `Progress()` appelée par le processeur de calcul teste si un événement MPI s'est produit sur le processeur de communication ou teste si une communication s'est terminée. Cette fonction est appelée dans toutes les fonctions MPI : `MPI_Isend()`, `MPI_Irecv()`, `MPI_Test()`, `MPI_Wait()` et `MPI_Probe()`.

```
Procédure Progress()
  Tant que message de contrôle ou terminaison de communication faire
    Si message de contrôle dans une file d'attente circulaire spécifique alors
      Read_control()
    Fin si (* Message de contrôle disponible. *)
    Si terminaison de communication alors
      Complete_communication()
    Fin si (* Terminaison de la communication. *)
  Fin tant que (* Message de contrôle ou terminaison de la communication. *)
Fin procédure (* Progress() *)


Procédure Read_control()
  RcvControl() (* Lire une valeur dans la mémoire principale. *)
  Si requête d'envoi alors
    Si requête de réception correspondante alors
      IRcvData(données)
      SndControl(acquittement d'envoi)
    Sinon (* Aucune requête de réception correspondante. *)
      Si il est possible d'allouer une zone de mémoire temporaire pour la réception alors
        IRcvData(données)
        SndControl(acquittement d'envoi)
      Sinon (* Impossible d'allouer une zone de mémoire temporaire. *)
        Mémorise la requête d'envoi dans la table des requêtes d'envoi
      Fin si
    Fin si
  Fin si (* Requête d'envoi. *)
  Si acquittement d'envoi alors
    ISndData(données)
  Fin si (* Acquittement d'envoi. *)
```

```
Procédure Complete_communication()
  Si l'envoi courant de données est terminé alors
    Marquer la requête d'envoi comme terminée
  Fin si (* Envoi courant de données terminé. *)
  Si la réception courante de données est terminée alors
    Si le processeur de calcul a alloué une zone de mémoire temporaire alors
      Si la requête de réception correspondante a été exécutée alors
        Détruire la requête d'envoi dans la table RSnd
        Copier les données dans la zone mémoire de destination
      Fin si
    Fin si
    Marquer la requête de réception comme terminée
  Fin si (* Réception courante des données terminée. *)
```

Messages de contrôle. Seuls deux types d'événements MPI peuvent se produire : une requête d'envoi ou un acquittement d'envoi.

Le processeur de calcul a dans sa mémoire principale une file d'attente circulaire spécifique à l'intérieur de laquelle les messages de contrôle sont stockés.

À la réception d'une requête d'envoi, le processeur de calcul recherche tout d'abord dans la table des requêtes de réception si la réception correspondante a été postée :

- s'il y a une requête de réception correspondante, le processeur de calcul initialise la réception des données du message en appelant la fonction `IRcvData` puis renvoie l'acquittement de la requête d'envoi comme le décrit l'étape 5 dans la figure 3.13 avec la fonction `SndControl`. Ceci garantit que la réception correspondant à l'envoi asynchrone est postée avant cet envoi. L'acquittement renvoie un pointeur sur la structure de données de la requête d'envoi courante sur le processeur expéditeur et un marqueur de message BIP pour recevoir effectivement les données du message ;
- s'il n'y a aucune requête de réception correspondante et si le processeur de calcul peut allouer une zone de mémoire temporaire, alors il poste aussi la requête de réception `IRcvData` associée avant de renvoyer l'acquittement de la requête de réception avec la fonction `SndControl`. La requête d'envoi est mémorisée dans la table des requêtes d'envoi avec la zone de mémoire temporaire associée ;
- si le processeur de calcul ne peut pas allouer de zone de mémoire temporaire, la requête d'envoi est mémorisée dans la table des requêtes d'envoi et l'acquittement n'est pas envoyé.

À la réception d'un acquittement d'envoi, le processeur de calcul envoie les données du message.

Terminaison de communication. Deux types de terminaison peuvent se produire : la terminaison de l'envoi courant ou de la réception courante. Si l'envoi courant se termine, le processeur de calcul marque la requête d'envoi comme terminée.

Si la réception courante se termine, alors trois cas différents peuvent se produire :

- les données sont directement stockées dans la mémoire destination alors le processeur de calcul marque la requête de réception comme terminée ;

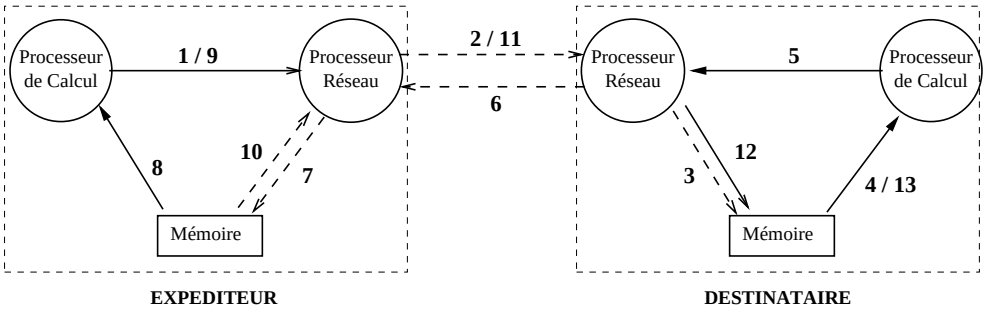


FIG. 3.13 – Les différentes étapes nécessaires à l'exécution totale d'un envoi non bloquant. Les lignes continues représentent les étapes spécifiques à l'interface MPI ; les lignes en pointillés représentent les étapes BIP natives.

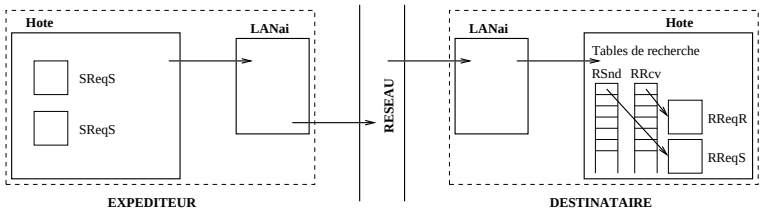


FIG. 3.14 – Implantation MPI-BIP.

- les données ont été transmises dans une zone de mémoire temporaire et l'exécution de la requête de réception correspondante n'a pas été effectuée, alors le processeur de calcul marque aussi la requête de réception comme terminée ;
- les données ont été transmises dans une zone de mémoire temporaire et l'exécution de la requête de réception correspondante a été effectuée, le processeur de calcul détruit alors la requête d'envoi dans la table `RSnd`, copie les données dans la mémoire destination et marque la requête de réception comme terminée.

Les différentes étapes de communication. La figure 3.13 montre les différentes étapes nécessaires pour terminer un envoi non bloquant. Les étapes 1 à 8 effectuent la communication de la requête d'envoi et du retour de l'acquittement.

La première étape commence par un appel à `MPI_Isend()` sur le processeur de calcul. Le pseudo-code ci-dessous montre que cette fonction commence par un appel à la fonction `Progress()`.

Procédure MPI_Isend(données, enveloppe du message)
  Progress()
  Création de la requête d'envoi
  SndControl(requête d'envoi, pointeur sur la requête d'envoi)
Fin procédure (* MPI_Isend *)

Par conséquent, avant de poster sa requête d'envoi, le processeur de calcul gère tous les messages de contrôle entrants. Ensuite, il effectue la communication associée à l'envoi non

bloquant. Le processeur de calcul crée une requête d'envoi et l'envoie au nœud destination avec un pointeur sur la structure de données associée à cette requête d'envoi dans la mémoire principale locale.

Ce message est envoyé avec la fonction `SndControl` qui le copie directement dans la mémoire du processeur réseau.

L'étape 2 et 3 sont des étapes BIP natives décrites dans la figure 3.6. Ces étapes transportent le message de contrôle de la mémoire principale locale à la mémoire principale distante.

À l'étape 4, le processeur de calcul traite le message grâce à un appel à la fonction `Progress()`.

À la réception de la requête d'envoi, le processeur de calcul renvoie dans la plupart des cas un acquittement de requête d'envoi comme le décrit l'étape 5 (voir la description de la fonction `Progress()` ci-dessus).

Les étapes 6 et 7 sont des étapes BIP natives comme le décrit la figure 3.6 aux étapes 3 et 4.

À l'étape 8, le processeur de calcul traite le message (comme à l'étape 4) grâce à un appel à la fonction `Progress()`.

À la réception de l'acquittement de la requête d'envoi, le processeur de calcul envoie les données du message. Selon la taille du message, l'appel à la fonction `ISndData` envoie un message BIP court ou un message BIP long.

Ensuite, les étapes 9 à 11 sont des étapes BIP natives correspondant au processus complet de communication d'un message décrit par la figure 3.6 sur le processeur expéditeur.

Dans la plupart des cas, le processeur réseau peut recevoir les données comme le décrit l'étape 12 (voir la description de la fonction `Progress()` ci-dessus).

Sur le nœud destinataire, l'étape asynchrone consiste à poster la réception correspondante avec un appel à `MPI_Irecv()` (voir le pseudo-code ci-dessous).

```
Procédure MPI_Irecv( données, enveloppe du message)
  Progress()
  Création de la requête de réception
  Tant qu' il n'y a pas de requête d'envoi correspondant à la requête de réception faire
    Teste l'entrée dans la table des requêtes d'envoi
    Si la requête d'envoi correspond à la requête de réception alors
      Si les données sont dans une zone de mémoire temporaire alors
        Copier les données dans la mémoire destination
      Sinon (* Données n'ont pas encore été intégralement transmises. *)
        Si les données ne sont pas en cours de transmission alors
          IRcvData(données)
          SndControl(acquittement d'envoi)
        Fin si
      Fin si
      Mise à jour de la table des requêtes d'envoi
    Fin si
    Aller à l'entrée suivante
  Fin tant que
  Si aucune requête d'envoi n'a été trouvée alors
    Mémoriser la requête de réception dans la table des requêtes de réception
  Fin si
Fin procédure (* MPI_Irecv *)
```

Comme le montre la description de la fonction `Progress()`, l'appel à `MPI_IRecv()` peut se produire avant ou bien après l'arrivée de la requête d'envoi. Comme le décrit le pseudo-code de la fonction `MPI_IRecv()`, le processeur de calcul traite tout d'abord la table des requêtes d'envoi afin de trouver une requête d'envoi associée.

S'il n'y a pas de requête d'envoi associée, les données correspondant à cette communication peuvent être soit dans une zone de mémoire temporaire soit toujours dans la mémoire du processeur de calcul distant ou encore en cours de transmission. Si les données sont dans une zone de mémoire temporaire, le processeur de calcul les copie dans la zone de mémoire destination. Si les données sont encore dans leur intégralité dans la mémoire de l'expéditeur, le processeur de calcul poste effectivement la réception par un appel à la fonction `IRcvData` afin de recevoir les données et envoie l'acquittement de la requête d'envoi. Si les données sont en cours de transmission, il ne fait rien. En revanche, si aucune requête d'envoi associée n'est trouvée, le processeur de calcul mémorise la requête de réception dans la table des requêtes de réception.

À l'étape 13, le processeur de calcul traite le message (comme aux étapes 4 et 8) avec un appel à la fonction `MPI_Test()` ou `MPI_Wait()`.

```
Fonction booléenne MPI_Test(requête)
  Progress()
  Si requête de réception alors
    Si la requête de réception est terminée alors
      TRcv(requête de réception)
      retourne vrai
    Sinon
      retourne faux
    Fin si
  Sinon (* C'est une requête d'envoi. *)
    Si la requête d'envoi est terminée alors
      TSnd(requête d'envoi)
      retourne vrai
    Sinon
      retourne faux
    Fin si
  Fin si (* type de requête. *)
Fin procédure (* MPI_Test *)

Procédure MPI_Wait(requête)
  Si requête de réceptionalors
    Tant que la requête de réception n'est pas terminée faire
      Progress()
    Fin tant que
    TRcv(requête de réception)
  Sinon (* C'est une requête d'envoi. *)
    Tant que la requête d'envoi n'est pas terminée faire
      Progress()
    Fin tant que
    TSnd(requête d'envoi)
  Fin si (* type de requête. *)
Fin procédure (* MPI_Wait *)
```

Après un appel à la fonction `Progress()`, la fonction `MPI_Test()` teste si la requête est terminée. Si la requête est terminée, elle retourne vrai et détruit la requête avec soit `TRcv()` ou `TSnd()` sinon elle retourne faux. La fonction `MPI_Wait()` appelle, quant à elle, la fonction `Progress()` jusqu'à ce que la requête soit terminée et détruit la requête avec soit `TRcv()` ou `TSnd()`.

Schéma de traitement des différentes requêtes de communication

Trois différentes requêtes de communication peuvent être postées :
- les requêtes de réception sans envoi posté correspondant sur le destinataire,
- les requêtes d'envoi postées sans réception correspondante postée sur le destinataire,
- les requêtes d'envoi postées sur l'expéditeur (`SReqS`).

La figure 3.15 montre les différents états d'une requête de réception sur le destinataire. L'appel à la fonction `MPI_Irecv()` par le processeur de calcul crée une structure de données représentant la requête de réception et recherche dans la table des requêtes d'envoi (table `RSnd`) la requête d'envoi associée. S'il n'y a pas de requête d'envoi associée dans la table `RSnd`, les données peuvent être locales, distantes ou en cours de transmission :

- si les données sont locales, elles sont copiées dans la zone de mémoire destination et la requête de réception est considérée comme terminée ;
- si les données sont en cours de transmission, la requête de réception est dans un état intermédiaire. À la fin de la transmission (détectée par la fonction `Progress()`), les données sont copiées dans la zone de mémoire destination, la requête de réception est alors considérée comme terminée ;
- si les données sont distantes, le processeur de calcul envoie l'acquittement d'envoi et la transmission des données commence. À la fin de la transmission (détectée par la fonction `Progress()`), la requête de réception est alors considérée terminée.

S'il n'y a pas de requête d'envoi associée dans la table `RSnd`, la requête de réception est ajoutée dans la table de requêtes de réception (la table `RRcv`). Ensuite, par un appel à la fonction `Progress()`, si la requête d'envoi est arrivée alors le processeur de calcul recherche dans la table `RSnd` une requête d'envoi associée. S'il y en a une, le processeur de calcul envoie un acquittement d'envoi à l'expéditeur et les données sont transmises. À la fin de la transmission (toujours détectée par un appel à la fonction `Progress()`), la requête de réception est considérée terminée.

Les appels à `MPI_Test()` ou `MPI_Wait()` sur la requête de réception détruisent cette requête.

La figure 3.16 montre les différents états d'une requête d'envoi sur le nœud destinataire. Sur le processeur de calcul, un appel à la fonction `Progress()` détecte un événement MPI sur le processeur réseau. Si une requête d'envoi a été reçue sans requête de réception associée postée alors une requête `RReqS` est créée. Le processeur de calcul essaie tout d'abord d'allouer une zone de mémoire temporaire pour stocker les données. Si cela a été possible, il envoie un acquittement d'envoi à l'expéditeur et la transmission des données commence. Pendant cette transmission, un appel à `MPI_Irecv()` peut être exécuté avec la requête de réception associée. Dans ce cas, à la fin de transmission, détectée par un appel à la fonction `Progress()`, `RReqS` est finalement détruite.

Si, durant la transmission, le processeur de calcul détecte la fin de la transmission, la requête d'envoi est considérée exécutée. En exécutant la requête de réception correspondante, la requête RReqS est finalement détruite.

La figure 3.17 décrit les différents états d'une requête d'envoi sur le nœud expéditeur. Sur le processeur de calcul, un appel à `MPI_Isend()` crée une structure de données représentant la requête d'envoi et envoie une requête d'envoi au processeur distant. À la réception de l'acquittement de la requête d'envoi, le processeur de calcul initialise le transfert de données. La requête d'envoi est alors terminée. La requête d'envoi est détruite à l'appel de `MPI_Wait()` ou `MPI_Test()` sur cette requête.

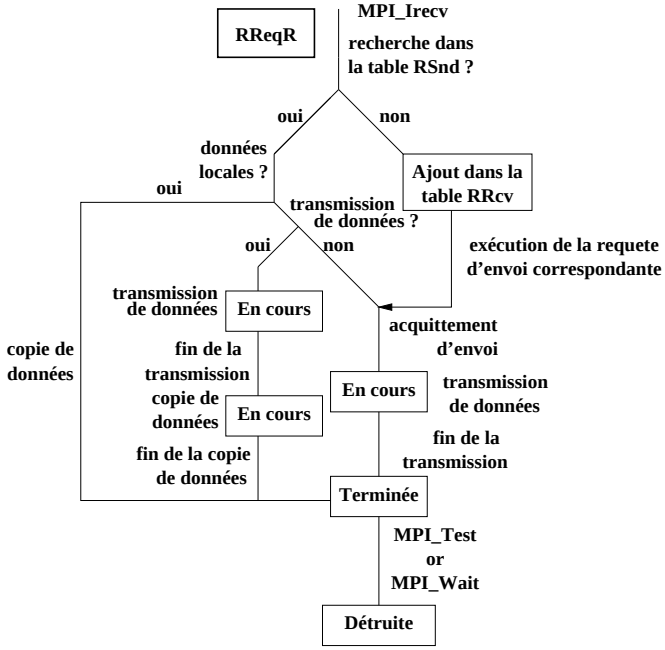


FIG. 3.15 – Requêtes de réception sur le destinataire.

Recouvrement des communications MPI-BIP

Cette implantation de MPI ne permet aucun recouvrement. La principal obstacle est l'interaction entre le processeur de calcul et le processeur de communication. De plus, la charge liée à la communication sur le processeur de calcul est élevée.

Interactions entre les processeurs de calcul et de communication. Comme la partie 3.3.2 le décrit, trois communications sont nécessaires pour envoyer des données avec la couche de communication MPI (sauf pour envoyer des petits messages). De plus, la plupart des structures de données sont gérées par le processeur de calcul et particulièrement les requêtes de communication (voir les parties 3.3.2 et 3.3.2). Ceci mène à un schéma d'interactions très dense dessiné figure 3.13 et à la conclusion que la plupart du processus de communication est géré par le processeur de calcul comme le montre le schéma de la figure 3.14.

C'est la raison pour laquelle les messages ne sont pas traités immédiatement lorsqu'ils arrivent si le processeur de calcul est en cours de calcul. La figure 3.18 décrit trois différents cas pathologiques en terme de recouvrement de déroulement de calcul et de communication.

Le premier schéma décrit le cas où le processeur de calcul sur le destinataire est occupé à calculer lorsque le message de contrôle contenant la requête d'envoi arrive. L'acquittement de la requête d'envoi ne sera par conséquent envoyé qu'à la fin du calcul.

Le deuxième schéma décrit un autre cas où le processeur de calcul sur l'expéditeur est occupé à calculer lorsque le message de contrôle contenant l'acquittement de la requête d'envoi arrive. Le processeur de calcul n'initialisera donc le transfert des données qu'à la fin de son calcul.

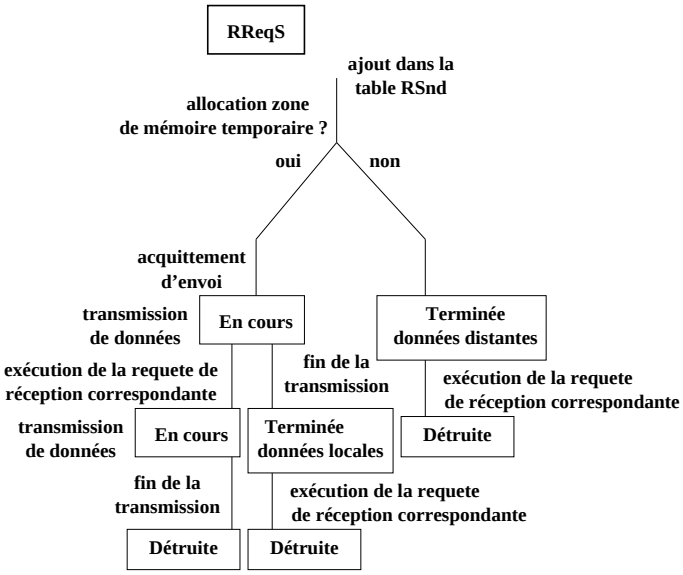


FIG. 3.16 – Requêtes d'envoi sur le destinataire.

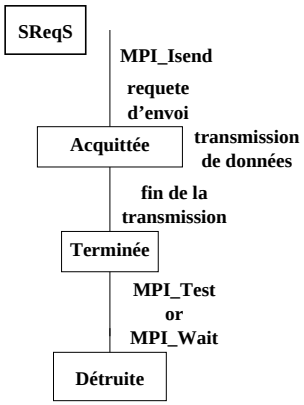


FIG. 3.17 – Requêtes d'envoi sur l'expéditeur.

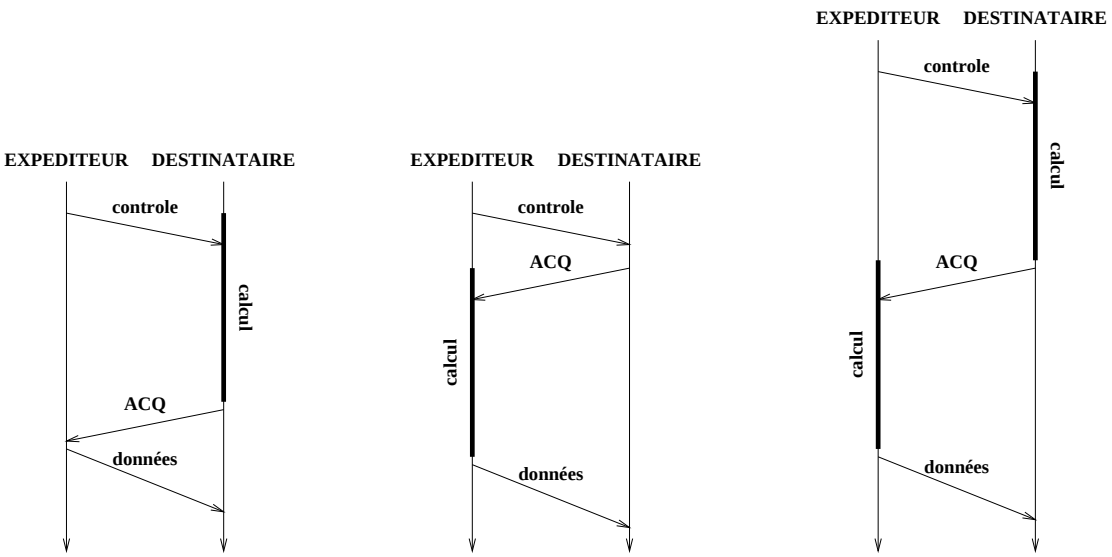


FIG. 3.18 – Trois cas pathologiques empêchant le recouvrement avec MPI-BIP.

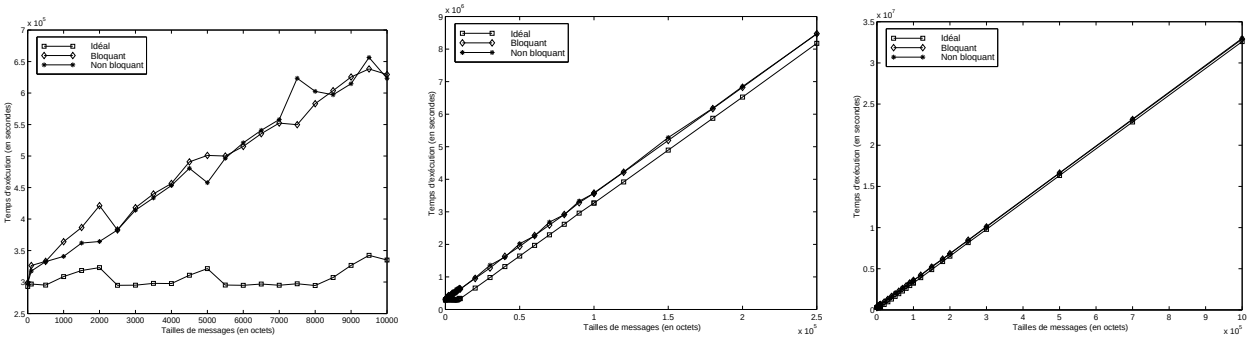
Le dernier schéma décrit à la figure 3.18 est l'aggrégation des deux cas précédents. Lorsque le premier message de contrôle arrive sur le destinataire, celui-ci est en train de calculer. Le destinataire n'envoie donc l'acquittement de la requête d'envoi qu'à la fin de son calcul. À la réception de l'acquittement, l'expéditeur est lui aussi occupé à calculer, il n'initialisera donc le transfert des données qu'à la fin de son calcul.

Charge liée à la communication sur le processeur de calcul. Comme le montrent la figure 3.14 et la partie 3.3.2, la gestion des communications se trouve principalement sur le processeur de calcul. Les deux tables sur le récepteur sont en effet gérées par le processeur de calcul et les différentes requêtes de communications sont aussi gérées par le processeur de calcul.

Expériences

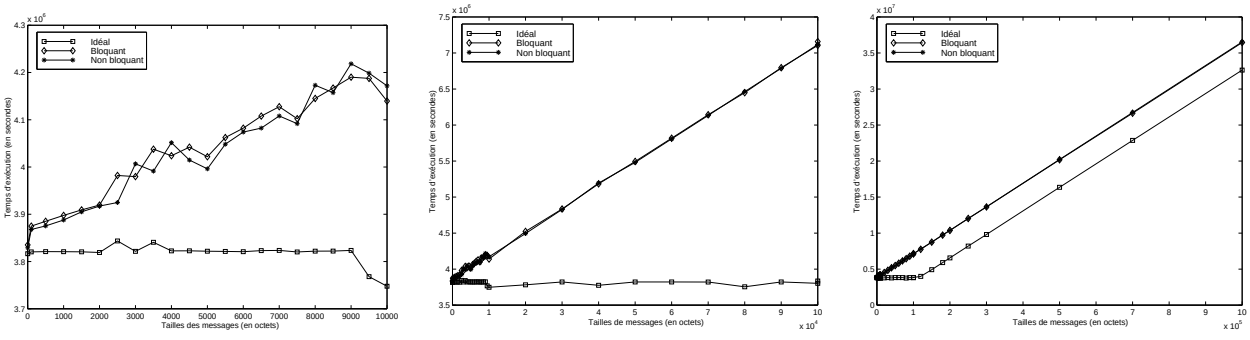
En utilisant les fonctions MPI implantées au dessus de BIP, on ne peut pas recouvrir les communications par les calculs. Nous avons exécuté le même programme de test que pour les primitives BIP natives (voir la partie 3.3.1). Les résultats sont données par la figure 3.19. Nous constatons que le temps total du test non bloquant est proche du temps total du test bloquant. Les résultats sont même moins bons avec le test non bloquant probablement à cause du surcoût lié à la gestion des structures de données.

Communications bidirectionnelles. Les communications asynchrones peuvent profiter des liens de communications bidirectionnels du réseau Myrinet. Le programme de test consiste tout d'abord à échanger des données puis à exécuter des calculs (voir figure 3.21). Le gain lié aux communications asynchrones est meilleur que le gain attendu avec le recouvrement avec des communications unidirectionnelles comme le montre la figure 3.22.



(a) 0 à 10 Ko. (b) 0 à 250 Ko. (c) 0 à 1 Mo.

FIG. 3.19 – Recouvrement des communications MPI-BIP avec des calculs tenant dans le cache.



(a) 0 à 10 Ko. (b) 0 à 100 Ko. (c) 0 à 1 Mo.

FIG. 3.20 – Recouvrement des communications MPI-BIP avec des calculs plus importants.

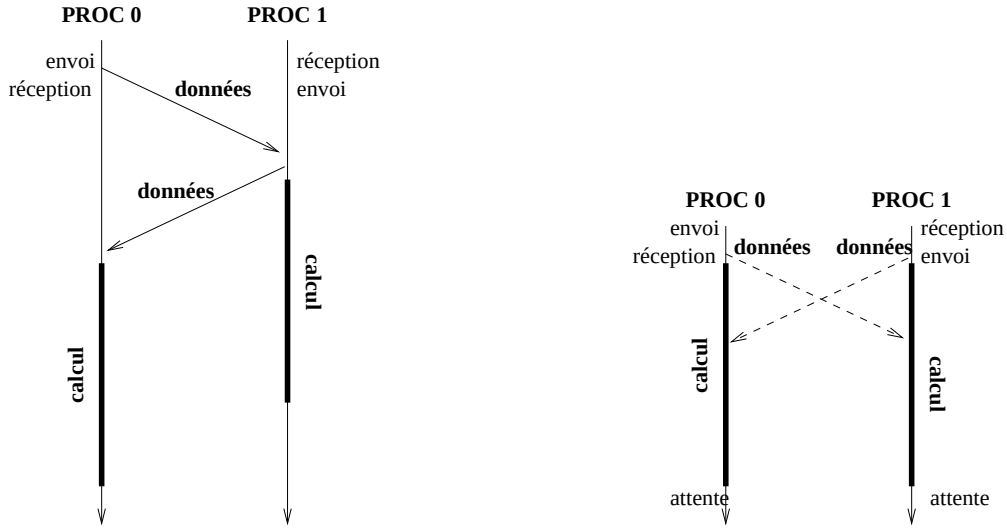


FIG. 3.21 – Programmes de test de communications bloquantes et non bloquantes utilisant un lien bidirectionnel.

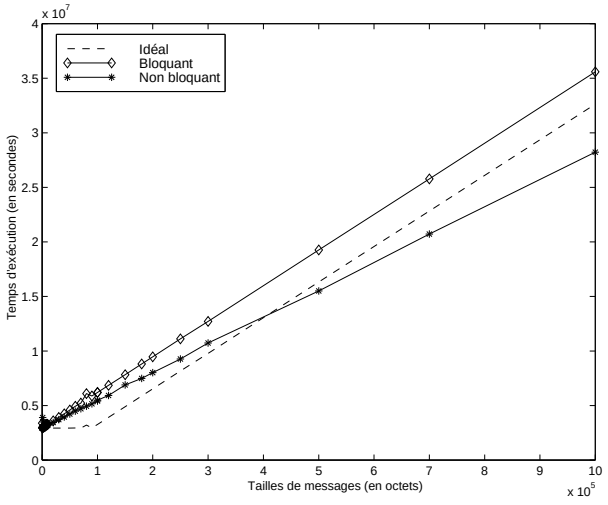


FIG. 3.22 – Recouvrement des communications en utilisant des liens bidirectionnels. Les lignes continues représentent les courbes bloquantes et non bloquantes alors que les lignes en pointillés représentent le la courbe idéale avec des communications unidirectionnelles.

Une solution

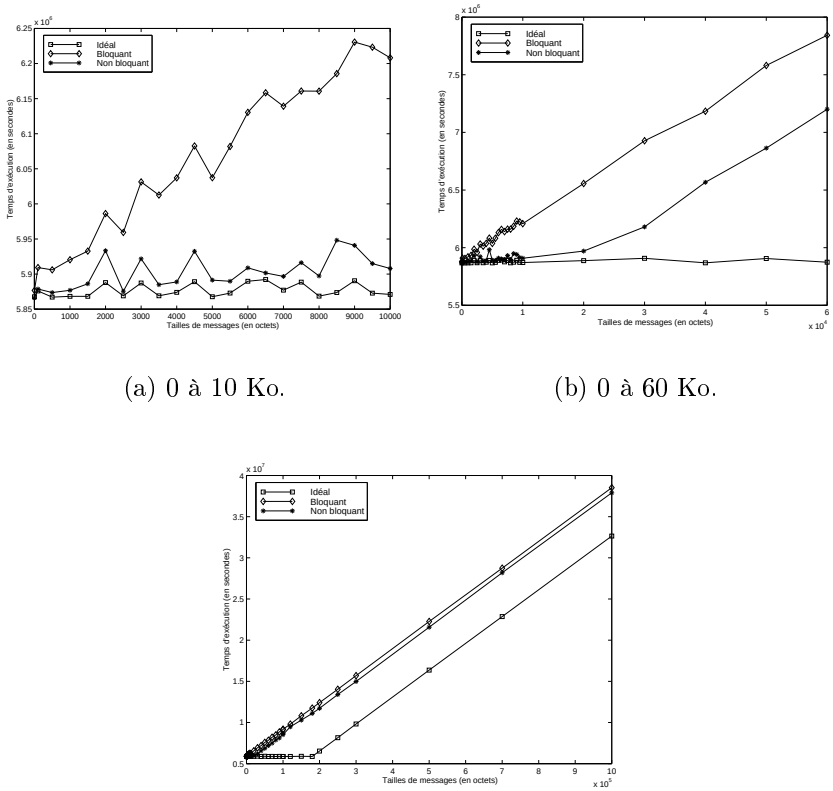
Afin d'obtenir du recouvrement avec MPI-BIP, une solution est d'interrompre le programme utilisateur afin de gérer les communications en attente. Le coût d'une interruption doit être suffisamment faible pour que le gain à recouvrir ne soit pas perdu. Ceci pourrait être réalisé par des interruptions matérielles et de signaux dans l'implantation MPI-BIP. Nous avons évalué cette stratégie en modifiant l'application pour qu'elle vérifie périodiquement l'état du réseau et lance l'étape suivante de la communication.

Pour cela, nous avons appelé périodiquement une fonction MPI dans le programme de calcul ; cet appel permet aux communications de continuer leurs exécutions en arrière plan. La fonction utilisée est `MPI_Iprobe()`. Cette fonction teste l'arrivée d'un message mais ne termine pas la communication dans le cas où un message est arrivé. Les figures 3.23 et 3.24 donnent le résultat en appelant `MPI_Iprobe()` respectivement dans la boucle externe et interne du calcul. Ces expériences montrent qu'appeler `MPI_Iprobe()` dans la boucle externe du calcul ne génère pas de recouvrement alors qu'en l'appelant dans la boucle interne, le recouvrement est total. Ici, le surcoût lié à l'appel de `MPI_Iprobe()` et à l'interruption du programme de calcul est négligeable.

3.3.3 Discussion

Temps de réponse. Une gestion rapide des messages entrants est nécessaire afin de minimiser la latence. Une manière de garantir que les messages sont traités dès qu'ils arrivent est que le message génère une interruption. Malheureusement, peu de processeurs du marché ont des interruptions rapides, c'est pourquoi l'observation périodique de l'interface réseau est utilisée. Si le processeur de calcul a la charge de scruter l'interface réseau et qu'il ne peut pas le faire parce qu'il est en train de calculer de manière intensive alors la latence effective peut augmenter de manière considérable. Si les opérations distantes sont gérées par le processeur réseau, celui-ci peut répondre rapidement à ces requêtes quelles que soient les activités du processeur de calcul.

Performance des processeurs de communication. Pour que le processeur de communication ait la charge du processus de communication, il faut que celui-ci ait de bonnes performances tant au niveau mémoire qu'au niveau rapidité d'exécution. Néanmoins, si, comme c'est le cas pour notre architecture, le processeur réseau a de faibles capacités à la fois en mémoire et rapidité, il peut être globalement plus rapide de ne lui confier que des opérations simples à exécuter comme l'envoi et la réception d'entiers et de transférer les requêtes complexes, l'envoi de structures de données élaborées par exemple, au processeur de calcul. Pourtant le meilleur choix ne dépend pas seulement de la comparaison des capacités du processeur de communication et de celles du processeur de calcul, il dépend aussi de leur charge respective. En effet si le processeur de calcul est occupé alors il peut être plus rapide de confier la gestion des requêtes complexes au processeur de communication même s'il est plus lent. Bien sûr si la requête nécessite des ressources par exemple en mémoire qui ne sont pas disponibles sur le processeur de communication, celle-ci devra dans tous les cas être traitée par le processeur de calcul. Par exemple, dans le cas de la grappe de PC PoPC,

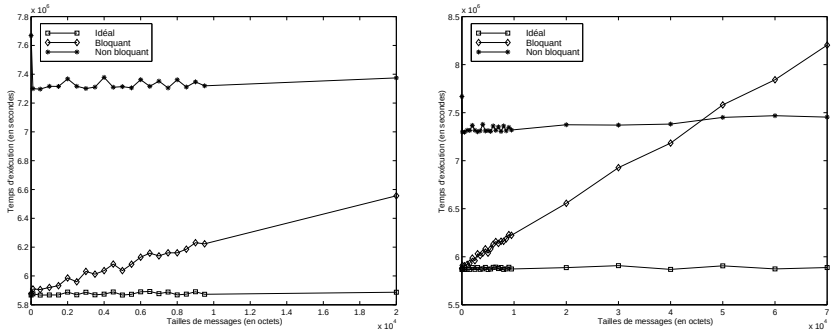


(a) 0 à 10 Ko.

(b) 0 à 60 Ko.

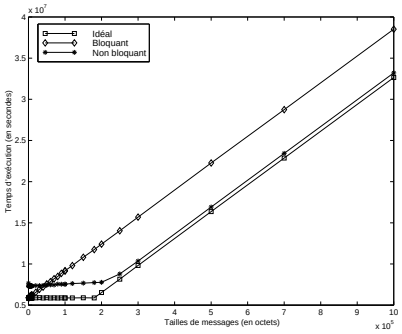
(c) 0 à 1 Mo.

FIG. 3.23 – Recouvrement des communications MPI-BIP avec des appels à `MPI_Iprobe()` dans le calcul.



(a) 0 à 20 Ko.

(b) 0 à 70 Ko.



(c) 0 à 1 Mo.

FIG. 3.24 – Recouvrement des communications MPI-BIP communication en appelant plus souvent `MPI_Iprobe()` dans le calcul.

le processeur de communication n'a pas d'unité de calcul flottant et possède seulement 256 Ko de mémoire vive.

Protocoles de communication. Le protocole de rendez-vous implique une synchronisation des processus communiquants. C'est pourquoi, lorsque ces processus ne sont pas interrompus dans le calcul en cours, la communication est inefficace. Tatebe *et al.* dans [69] proposent que la gestion de la communication soit entièrement gérée par le processus expéditeur. L'idée est qu'au moment de l'émission de la requête de réception, le destinataire envoie cette requête au processus expéditeur. Ainsi, lorsque celui-ci enverra ses données, il ne sera plus nécessaire d'interrompre le processus expéditeur ; le processus expéditeur pourra placer directement les données dans la zone de mémoire voulue sur le nœud distant.

Encombrement du bus mémoire. Nous avons vu au paragraphe 3.3.1, qu'à bas niveau, si la communication entre le processeur principal et le co-processeur s'effectue par l'intermédiaire du bus mémoire utilisé par le processeur principal, les calculs et les communications ne peuvent pas s'effectuer de manière indépendante parce qu'ils utilisent tous les deux intensément le bus mémoire.

3.4 Synthèse

Dans ce chapitre, nous avons présenté une étude des recouvrements des communications par le calcul. Dans une première partie, nous avons décrit les travaux existants utilisant cette technique dans le but d'améliorer le temps d'exécution des applications. Cette technique a été très utilisée pour tous les types d'applications parallèles, la plupart du temps en utilisant des couches de communication natives spécifiques aux architectures cibles. À l'heure actuelle, la parallélisation d'une application ne se fait plus avec ces couches de communication natives mais avec les standard de communication PVM ou MPI.

Le gain lié à cette optimisation n'est pas toujours satisfaisant. Nous avons pris pour exemple l'implantation d'itérations de Jacobi pour montrer que même si algorithmiquement les communications devraient être intégralement recouvertes, les expériences peuvent être décevantes et les traces d'exécutions difficilement compréhensibles. Ces constatations ont motivées notre étude approfondie des moyens matériels et surtout des moyens logiciels mis à notre disposition pour recouvrir les communications.

Dans un premier temps, nous nous sommes interrogés sur les possibilités matérielles. Le recouvrement de la communication peut se faire à plusieurs niveaux. Tout d'abord, le message transite sur le réseau indépendamment du processeur de calcul. Du point de vue du matériel, cette transmission peut donc être recouverte par du calcul. Deuxièmement, la gestion du message, c'est-à-dire la récupération des données, le format et l'envoi sur le réseau, que nous avons appelée *latence de communication*, peut elle-aussi être recouverte. En effet, nous avons vu que la plupart des architectures utilisent un coprocesseur de communication dans le but de décharger le processeur de calcul tout ou partie de ces tâches. Ceci nous permet de conclure que le matériel permet dans tous les cas le recouvrement de la transmission du message sur le réseau et parfois même le recouvrement de la latence de communication.

Du point de vue logiciel, le résultat est tout autre. Nous avons effectué des expériences en utilisant une implantation de MPI au dessus de BIP sur une grappe de PC. Malheureusement, les expériences ont montré un recouvrement quasiment inexistant voire même pénalisant avec MPI-BIP. Nous avons donc mené une étude approfondie de cette implantation pour comprendre les raisons de cette absence de recouvrement dans cette implantation par ailleurs très efficace.

Nous montrons tout d'abord que la couche de communication de base BIP permet de recouvrir la transmission des données ainsi qu'une partie de la latence de communication grâce au coprocesseur de communication du réseau Myrinet. Nous avons découvert que l'implantation de MPI au dessus de BIP transfère dans un premier temps la gestion du message sur le processeur de calcul ne permettant ainsi pas le recouvrement de la latence de communication. Cette étude montre en détail que l'exigence des spécifications du standard, laissant beaucoup de libertés à l'utilisateur, ne permet pas à un coprocesseur de communication moins puissant d'assurer les communications. De plus, l'utilisation du protocole de rendez-vous empêche dans certains cas le recouvrement de la transmission des données. Pour résoudre ce problème, l'implantation de MPI sur IBM SP-2 propose d'interrompre le processeur de calcul de manière périodique afin de faire progresser la communication. Nous avons vu, par l'intermédiaire d'une astuce consistant à appeler une fonction MPI dans le calcul, que les interruptions se font au détriment du temps de calcul. Ainsi, la fréquence d'interruption doit être soigneusement choisie pour permettre un recouvrement satisfaisant.

Chapitre 4

Pipeline de calculs

Les pipelines de calculs sont des méthodes très utilisées pour extraire du parallélisme dans certains calculs dont les dépendances impliquent une exécution séquentielle. Leurs performances dépendent du compromis entre le temps d'exécution du calcul effectué entre deux communications successives et le temps d'exécution de ces communications ; plus les communications sont courtes, plus le temps de remplissage du pipeline est long. Leurs performances dépendent aussi d'un éventuel recouvrement des communications et des calculs. La granularité optimale des tâches dépend à la fois de paramètres logiciels et matériels (taille des calculs, taille des communications, vitesse de calcul, latence et bande passante de la communication inter-processeur).

Notre but dans ce chapitre est de déterminer s'il est profitable de pipeliner une application en prenant en compte la machine cible et l'application. Nous avons cherché à évaluer le gain apporté par cette technique. Malgré la popularité des pipelines de calculs, il existe en effet peu de travaux cherchant à évaluer l'amélioration de performance que l'on peut attendre. De plus, dans le but de réduire encore le temps d'exécution, il est possible de recouvrir les calculs et les communications à l'intérieur du processus de pipeline de calcul.

Nous présentons tout d'abord les différentes études existantes sur le pipeline de calculs, les modèles utilisés et l'étude du gain d'un pipeline effectuée par Desprez et Zory [75]. Dans une deuxième partie, nous présentons les travaux que nous avons effectués sur le pipeline de calculs en nous appuyant sur l'application Sweep3D.

Dans un premier temps, notre travail a consisté à valider ces équations sur une application réelle : le Sweep3D. Nous présentons une modélisation du temps d'exécution du Sweep3D en fonction de la distribution des données. À partir des équations obtenues, nous validons, dans un premier temps, les équations de gains de la partie précédente. Des expérimentations sur grappes de PC et par l'intermédiaire d'un simulateur viennent appuyer ces équations. Dans un deuxième temps, nous avons cherché à déterminer en fonction des paramètres de la machine et de l'application la meilleure distribution des données en terme de temps d'exécution.

4.1 Modélisations des pipelines de calculs

Nous distinguons dans cette partie pipeline synchrone et pipeline asynchrone. La figure 4.1 représente l'exécution d'un pipeline monodimensionnel synchrone sur trois processeurs. Chaque processeur calcule puis communique son résultat sans aucun recouvrement de la communication.

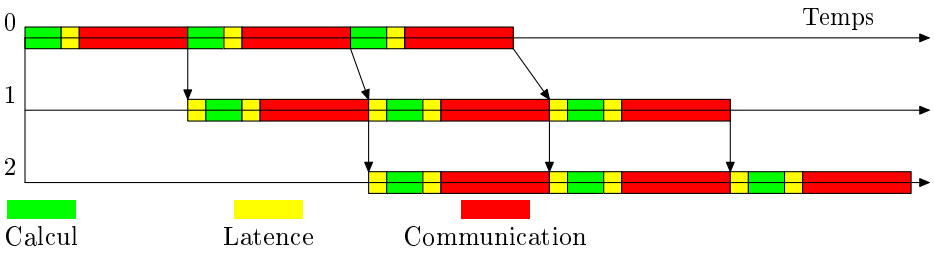


FIG. 4.1 – Pipeline synchrone.

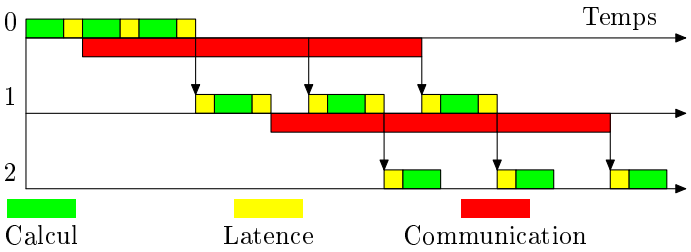


FIG. 4.2 – Pipeline asynchrone.

La figure 4.2 représente l'exécution asynchrone du calcul effectué à la figure 4.1. Les processeurs continuent à calculer pendant les communications. L'exécution asynchrone est plus difficile à modéliser mais beaucoup plus proche de la réalité. Nous verrons dans la partie suivante que certains travaux ont modélisé un pipeline synchrone, d'autres des pipelines asynchrones en prenant des hypothèses comme un temps de calcul bien supérieur au temps de communication afin de simplifier les équations.

Les figures 4.3 et 4.4 représentent respectivement le cas où le temps de calcul est supérieur au temps de communication et le cas contraire.

4.1.1 Applications

Les pipelines ont été utilisés dans de nombreuses applications. L'état de l'art sur les recouvrements dans la thèse de Ramet [55] montre que les pipelines sont très utilisés dans des applications de calcul numérique, pour l'ADI, les factorisations LU, le SOR, Cholesky, le gradient conjugué mais aussi pour accélérer des couches de communications comme BIP [53].

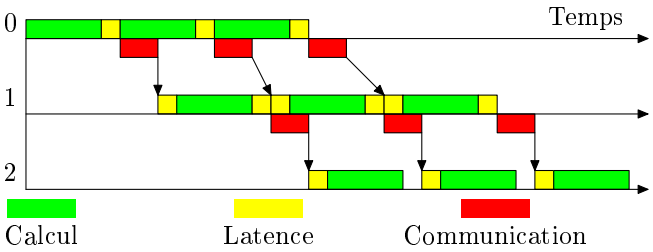


FIG. 4.3 – Pipeline asynchrone : temps de calcul supérieur au temps de communication.

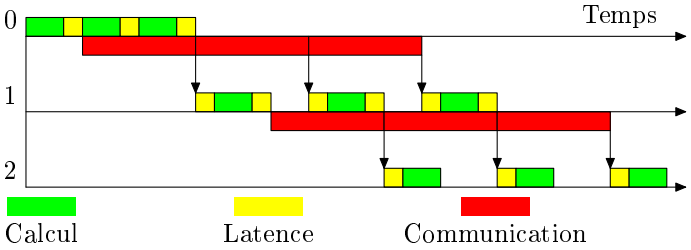


FIG. 4.4 – Pipeline asynchrone : temps de calcul inférieur au temps de communication.

L'amélioration des performances liée au pipeline n'est pas garantie. Nous verrons très en détail par la suite qu'elle dépend du compromis entre la taille des blocs de calcul que l'on exécute avant d'envoyer les données au processeur voisin et le coût de cette communication. L'investissement logiciel que représente le pipeline n'est pas négligeable. Le pipeline peut en effet générer des communications dans une ou plusieurs dimensions de l'espace des processeurs. Il ajoute de plus au programme des itérations liées au pipeline à l'intérieur des itérations de calcul. Un code de pipeline n'est donc ni facile à faire fonctionner ni facile à maintenir. C'est pourquoi les approches en compilation automatique sont très intéressantes. La mise en œuvre du pipeline est laissée au compilateur pendant que l'utilisateur peut manipuler un code de plus haut niveau. Dans cet esprit, Desprez [23, 19] a développé une bibliothèque fournissant au programmeur un code générique de boucles et de communications de pipeline de calculs. Celui-ci n'a plus qu'à fournir le code des blocs de calculs et à indiquer les données à échanger. Cette bibliothèque a, de plus, été intégrée à un compilateur HPF en vue d'une compilation automatique [14]. En complément à cette bibliothèque, Ramet [22] a développé une bibliothèque donnant le calcul de la taille de bloc optimale du pipeline. Nous revenons dans la partie suivante sur le modèle utilisé. Avec ces deux outils, le programmeur n'a plus à se soucier ni de la programmation ni de la détermination de la taille de bloc. Restent deux interrogations : ce pipeline va t-il lui apporter un gain significatif ? quelle distribution des données choisir ? Nous cherchons à répondre à ces deux questions tout au long de ce chapitre.

4.1.2 Travaux existants

De nombreux travaux tentent de modéliser les pipelines de calculs implantés sur des machines à mémoire distribuée.

Dans le domaine de la parallélisation automatique, Hiranandani *et al.* [28] établissent l'équation du temps d'exécution d'un pipeline synchrone pour des données mono- et bi-dimensionnelles distribuées sur une dimension. Ils en déduisent la taille optimale du bloc de calcul dans chacun des cas. Leur modèle prend en compte un coût de communication fixe, ne dépendant pas de la taille du message. Ils ont validé ce modèle sur une architecture Intel iPSC/860 avec trois applications différentes. Pour le noyau de calcul Livermore 23 (*Implicit Hydrodynamics*), la taille de bloc optimale obtenue est égale à huit ; pour SOR[1] et ADI[2], elle est égale à douze. Le meilleur temps d'exécution est obtenu avec ces tailles de bloc pour chacune de ces trois applications. Ils obtiennent alors une accélération de 13,9 pour le SOR et 13,6 pour l'ADI sur seize processeurs.

Hoisie *et al.* ont modélisé les exécutions d'applications par vagues dans le cas de pipelines synchrones. Nous revenons très en détail sur ce modèle dans la partie 4.2.1.

Siegell et Steenkiste [60] modélisent un pipeline synchrone en prenant comme paramètre le temps d'exécution séquentielle. Ainsi lorsque la charge des processeurs varie, en mesurant le temps d'exécution séquentielle sur un échantillon de code, ils peuvent adapter dynamiquement la taille de bloc optimale.

Palermo [49] modélise un pipeline asynchrone à deux niveaux, c'est-à-dire un pipeline ralenti après un certain nombre d'étapes par une synchronisation avec les données du processeur suivant. Le coût de communication d'un message est une fonction affine par rapport à la taille du message. Le terme constant est un surcoût lié à l'envoi ou à la réception du message par le processeur. Le temps de transit du message sur le réseau est proportionnel à sa taille. Néanmoins, sa modélisation suppose que le temps de communication du message est inférieur au temps de calcul. Les exécutions du SOR et du noyau de calcul Livermore 23 sur l'Intel Paragon et sur TMC CM-5 montrent que prendre en compte les communications remontantes par l'utilisation d'un modèle à deux niveaux permet de déterminer une taille de bloc plus proche de la meilleure taille de bloc expérimentale.

Dans [18], Wijngaart *et al.* modélisent un pipeline en prenant en compte le coût d'interruption du processeur de calcul à l'arrivée d'un message. Ils se placent dans le cas de pipeline à grain fin et considèrent alors que les messages ont une longueur nulle. Le coût d'une communication est donc représenté par les latences d'envoi et de réception ainsi que le coût d'interruption du processeur de calcul. Avec ce modèle, la résolution d'une équation de la chaleur tridimensionnelle en utilisant la méthode de l'ADI sur 512 nœuds d'Intel Paragon implantée avec la bibliothèque NX gagne 40% de temps d'exécution par rapport à un modèle ne prenant pas en compte les interruptions. Sur IBM SP-2, avec la bibliothèque MPI, le gain est de 67%.

Enfin, Ramet *et al.* [22] proposent une méthode générique de calcul de la taille optimale de bloc basée sur la complexité du calcul du bloc d'éléments. Ainsi, en représentant le coût de calcul par une fonction linéaire et celui d'une communication par une fonction affine, il est

[1] Successive Over Relaxation
[2] ADI : Alternating Direction Implicit

possible en n'utilisant que des paramètres de la machine de donner la taille de bloc optimale. Ramet *et al.* ont considéré un pipeline homogène monodimensionnel asynchrone.

Calcul du gain d'un pipeline

Les modèles de communication LogP et LogGP. Les deux principaux modèles de communication sur machine à mémoire distribuée sont les modèles LogP et LogGP.

Le modèle LogP[17] prend en compte la plupart des caractéristiques communes à toutes les machines à mémoire distribuée constituées d'un microprocesseur et d'un réseau d'interconnexion. La communication est modélisée par des messages point-à-point de petite taille fixée. Ce modèle utilise quatre paramètres. La latence (L) de communication est le délai de communication d'un message d'un mot du processeur source au processeur destinataire. Le surcoût (o comme *overhead*) est le temps passé par le processeur à envoyer ou à recevoir chaque message. La distance (g comme *gap*) est l'intervalle de temps minimum entre deux envois ou réceptions consécutifs sur un même processeur. Le dernier paramètre est le nombre de processeurs (P).

Le modèle LogGP est une extension du modèle LogP utilisant un modèle linéaire pour les messages longs [6] pour prédire précisément le temps de communication à la fois pour les messages courts comme le faisait LogP, mais aussi pour les messages longs. G représente la bande passante obtenue pour un message long. Les caractéristiques spécifiques à chaque machine comme la topologie du réseau, les algorithmes de routage ou la taille des caches ne sont pas prises en compte. Inspirés par ces modèles, nous allons détailler le modèle utilisé par Quinn et Hatcher dans [54], aussi utilisé dans [75].

Modélisation de Quinn et Hatcher. Le modèle de communication de Quinn et Hatcher [54] divise le temps de passage d'un message en trois composantes : le temps passé par le processeur à initialiser le message, que l'on appelle souvent la latence du message, le temps de transmission des données et le temps passé par le processeur à recevoir le message. La *latence du message* $L(n)$ est exprimée par la fonction $\lambda + \beta\ n$, où λ est un terme constant représentant le temps nécessaire pour gérer un appel d'envoi ou de réception, β est inversement proportionnel à la vitesse à laquelle le système peut recopier le message et n est la longueur du message. Le temps de transmission du message est le temps de déplacement du message à travers le réseau. La bande passante fixe signifie que le temps de *transmission du message* $T(n)$ est linéaire en fonction de la taille du message.

Le modèle de calcul sépare le temps de calcul qui peut être recouvert avec le temps de transmission du message $C_o(n)$ du temps de calcul qui ne peut pas l'être $C_r(n)$. Quinn et Hatcher distinguent alors trois cas possibles. Le cas où $C_o(n)$ et $C_r(n)$ sont tous les deux non nuls correspond au cas le plus général. Lorsque $C_r(n)$ est égal à zéro alors les calculs ne dépendent pas des communications et il est possible de recouvrir ces communications comme nous l'avons vu au chapitre précédent. Dans le cas de calcul strictement pipeliné où les dépendances de données impliquent la séquentialité, $C_o(n)$ est égal à zéro. Néanmoins, comme expliqué dans la partie 2.1, dans ce cas, les communications des données précédemment calculées peuvent être recouvertes par le calcul des données suivantes.

Quinn et Hatcher évaluent alors le gain d'un pipeline sur un processeur. Ce gain est le

rapport entre le temps d'exécution de l'application sans pipeline et le temps d'exécution de l'application avec pipeline. Le temps d'exécution sans pipeline sur un processeur est égal à la somme du temps de calcul et du temps de communication. Le temps d'exécution sans pipeline sur un processeur est donc égal à

$$C_r(n) + L(n) + T(n)$$

Le temps de l'application pipelinée est égal à la somme du temps de remplissage du pipeline, du temps d'exécution des blocs de calcul sans le premier et du temps de terminaison du pipeline. Pour un processeur, le temps de remplissage du pipeline est égal à la somme du temps de transmission du message et de la latence d'envoi. Le temps d'exécution d'un bloc de calcul est égal à la somme du temps de calcul du bloc et de l'attente du suivant. Il est donc égal au maximum entre le temps de calcul d'un bloc et le temps de transmission de la communication dont dépend le calcul d'un bloc. Enfin le temps de terminaison est égal au temps de calcul du dernier bloc de calcul. Sur un processeur, le temps d'exécution avec pipeline est donc égal à

$$L(n_0) + T(n_0) + (\frac{n}{n_0} - 1)\max(L(n_0) + C_r(n_0), T(n_0)) + C_r(n_0)$$

où n_0 désigne la taille de bloc comprise entre 0 et n.

Ainsi, si le temps de communication est négligeable par rapport au temps de calcul, le gain du pipeline, rapport entre le temps d'exécution sans pipeline et celui avec pipeline, tend vers un lorsque la taille des données tend vers l'infini.

4.1.3 Évaluation du gain d'un pipeline

Dans cette partie, nous développons le modèle de pipeline de calculs utilisé par Zory et Desprez dans [75] afin d'en déduire le gain en performance théorique obtenu grâce au pipeline de calculs sur une machine à mémoire distribuée. En effet, dans [54], Quinn et Hatcher démontrent que le gain maximum que l'on puisse espérer sur un processeur est égal à un. Cette étude est étendue à P processeurs par Zory et Desprez dans [76, 75]. Ils démontrent que pipeliner peut amener des gains importants en temps d'exécution. Nous allons tout d'abord décrire les modèles de calcul et de communication utilisés. Puis, en reprenant [75], nous considérerons successivement les matrices de données à une, deux et trois dimensions distribuées sur une, deux ou trois dimensions afin de déterminer le temps d'exécution d'une application en fonction des paramètres du modèle.

Paramètres de calcul et de communication. L'étude de Zory des gains d'un pipeline concerne les pipelines de calculs pendant lesquels les communications des données précédemment calculées sont recouvertes par les calculs des données suivantes.

Les dépendances de calcul sont telles que l'espace d'itération rectangulaire à deux dimensions a des dépendances verticales et horizontales comme le montre la figure 4.5. Les données sont distribuées par bloc.

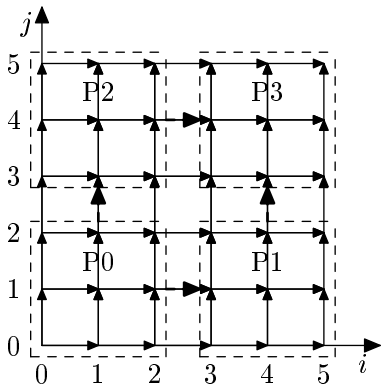


FIG. 4.5 – Code simple qui tirerait profit du pipeline de calculs.

Le modèle de communication. Le modèle utilisé est une extension du modèle de Quinn et Hatcher. Il suppose que le réseau est homogène. Ce modèle considère que les temps de latence et de transmission du message ont un terme constant et un terme proportionnel.

Le temps de latence du message est donné par :

$$L(n) = \lambda_L + \beta_L \times n \tag{4.1}$$

Le temps de transmission du message est donné par :

$$T(n) = \lambda_T + \beta_T \times n \tag{4.2}$$

où λ_L, β_L, λ_T et β_T sont des constantes et n la taille du message.

Le modèle de calcul. Le modèle de calcul suppose que tous les processeurs ont les mêmes vitesses de calcul, que le nombre de processeurs impliqués dans le calcul de l'application est constant et enfin que le charge en calcul est distribuée de manière équilibrée sur tous les processeurs.

Le modèle de calcul utilisé est classique et ne prend en compte que la vitesse de calcul du processeur et la complexité du calcul. Il ne tient notamment pas compte des effets de cache et de l'ensemble de la hiérarchie mémoire. Les paramètres du temps de calcul sont la taille du problème, le nombre de calculs flottants par élément de la grille, et la vitesse de calcul en opérations flottantes d'un unique processeur.

N_{flops} est le nombre d'opérations flottantes par point de la grille.

R_{flops} est le nombre d'opérations flottantes que peut calculer le processeur par seconde.

τ_c est le coût de calcul d'un élément de la matrice de données.

τ_c est donc donné par :

$$\tau_c = \frac{N_{flops}}{R_{flops}} \tag{4.3}$$

Si N_m représente le nombre de dimensions de la matrice de données et n la taille de chaque dimension, alors le temps de calcul de la matrice de données total $C(n)$ est donné par :

$$C(n) = \tau_c \times n^{N_m} \tag{4.4}$$

P est le nombre total de processeurs.
N_d représente le nombre de dimensions distribuées de la matrice de données.
N_p représente le nombre de dimensions pipelinées.
C_n est le temps de calcul d'une sous-matrice calculé par un processeur.
n_0 est le facteur bloquant.
C_{n_0} est le temps de calcul d'un étage de pipeline d'une sous-matrice du processeur.
$t_{np}^{N_d,N_m}$ est le temps total d'exécution de l'application complète sans pipeline.
$t_p^{N_d,N_m}$ est le temps total d'exécution de l'application complète avec pipeline.

Données bidimensionnelles

Dans cette partie, le gain à pipeliner sur une matrice de données à deux dimensions est étudié. Cette matrice peut être distribuée selon une ou deux dimensions. Les deux cas sont étudiés. n est le nombre d'éléments de la matrice de données à deux dimensions dans les directions I et J.

Distribution monodimensionnelle. La matrice de données bidimensionnelle est distribuée sur une dimension avec P processeurs en utilisant une distribution par blocs. Le temps de calcul d'une sous-matrice appartenant à un processeur est donné par :

$$C_n = \tau_c \times \frac{n^2}{P} \tag{4.5}$$

Le temps de calcul d'un étage du pipeline d'une sous-matrice d'un processeur est donné par :

$$C_{n_0} = \tau_c \times \frac{n}{P} \times n_0 \tag{4.6}$$

Lemme 1 *Soit $t_{np}^{1,2}(n)$ le temps total d'exécution d'une application 2-D de taille de problème n décomposée sur une dimension de P processeurs sans pipeline. Alors*

$$t_{np}^{1,2}(n) = PC_n + (P-1)[2L(n) + T(n)] = n^2\tau_c + (P-1)[2L(n) + T(n)] \tag{4.7}$$

Preuve. Il est assez facile de visualiser l'exécution d'une application dont les calculs sont distribués monodimensionnellement avec un diagramme de Gantt comme le montrent les figures 4.6 et 4.1.3.

La figure 4.6 donne un exemple d'exécution non pipelinée avec le temps de communication supérieur au temps de calcul. Pour en évaluer le temps total, il faut prendre en compte séquentiellement le calcul des P processeurs et les $P-1$ phases de communication.

Lemme 2 *Soit $t_p^{1,2}(n)$ le temps total d'exécution d'une application 2-D pipelinée ayant une taille de problème n et décomposée sur une dimension. Alors*

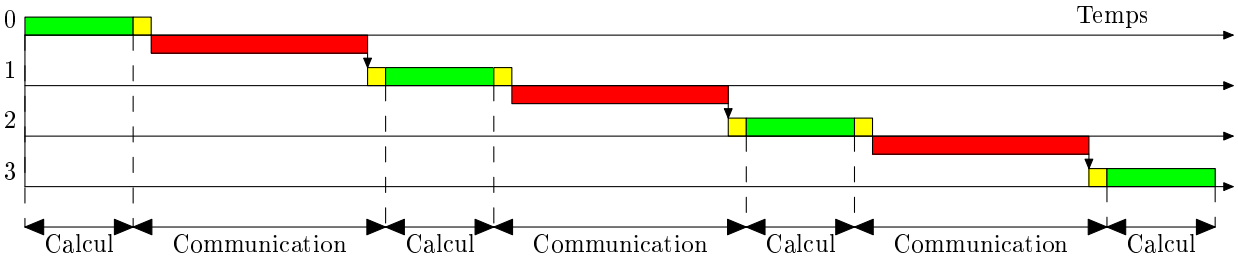


FIG. 4.6 – Exécution non pipelinée.

$$\begin{aligned} t_p^{1,2}(n) &= PC_{n_0} + (P-1)[2L(n_0) + T(n_0)] + (\frac{n}{n_0} - 1)[\max(L(n_0) + C_{n_0}, T(n_0)) + L(n_0)] \\ &= \tau_c \times n \times n_0 + (P-1)[2L(n_0) + T(n_0)] \\ &+ (\frac{n}{n_0} - 1)[\max(L(n_0) + (\tau_c \times \frac{n}{P} n_0), T(n_0)) + L(n_0)] \end{aligned} \tag{4.8}$$

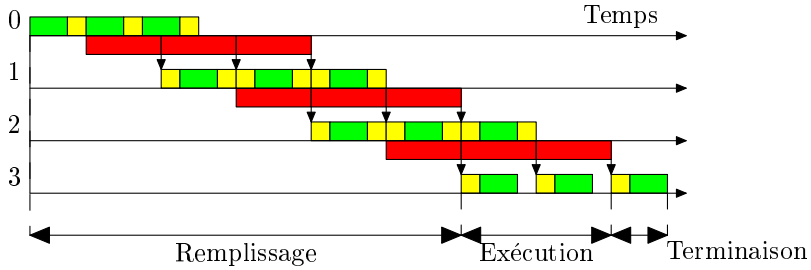


FIG. 4.7 – Exécution pipelinée.

Preuve. Comme nous l'avons vu dans la partie 4.1.2, le temps de l'application pipelinée est égal à la somme du temps de remplissage du pipeline, du temps d'exécution des blocs de calcul sans le premier et du temps de terminaison du pipeline. La figure 4.1.3 reprend l'exemple de la figure précédente en décomposant le calcul en trois blocs.

$$\begin{aligned} t_p^{1,2}(n) &= C_{n_0} + L(n_0) + T(n_0) + (P-2)[C_{n_0} + 2L(n_0) + T(n_0)] \\ &+ (\frac{n}{n_0} - 1)[\max(L(n_0) + C_{n_0}, T(n_0)) + L(n_0)] + L(n_0) + C_{n_0} \end{aligned} \tag{4.9}$$

Théorème 1 *Soit $G^{1,2}(n)$ le gain en performance obtenu en pipelinant une application 2-D décomposée sur une dimension de P processeurs et ayant une taille de problème n. Alors*

$$G^{1,2}(n) = \frac{n^2\tau_c + (P-1)[2L(n) + T(n)]}{D^{1,2}(n)} \tag{4.10}$$

avec

$$D^{1,2}(n) = \tau_c \times n \times n_0 + (P-1)[2L(n_0) + T(n_0)] \\ + (\frac{n}{n_0} - 1)[\max(L(n_0) + (\tau_c \times \frac{n}{P} \times n_0), T(n_0)) + L(n_0)] \quad (4.11)$$

Preuve. Nous déduisons des Équations 4.7 et 4.8 le gain en performance théorique obtenu en pipelinant une application 2-D décomposée sur une dimension. Ce gain est le rapport entre le temps d'exécution de l'application sans pipeline et le temps d'exécution pipeliné.

$$G^{1,2}(n) = \frac{t_{np}^{1,2}(n)}{t_p^{1,2}(n)} \quad (4.12)$$

Corollaire 1 *Si* $lim_{n\to\infty} \frac{T(n)}{C_n} = 0$ *et* $lim_{n\to\infty} \frac{L(n)}{C_n} = 0$, *alors* $G^{1,2} = P$.

Ce corollaire établit que si $T(n)$ et $L(n)$ ont une complexité inférieure à C_n, alors le gain en performance a pour ordre de grandeur P.

Preuve. Si $C_n >> T(n)$ alors

$$G^{1,2}(n) = \frac{n^2\tau_c + (P-1)[2L(n) + T(n)]}{\tau_c \times n \times n_0 + (P-1)[2L(n_0) + T(n_0)] + (\frac{n}{n_0} - 1)[L(n_0) + (\tau_c \times \frac{n}{P} \times n_0) + L(n_0)]}$$

$$G^{1,2}(n) = \frac{P\tau_c \times \mathbf{n^2} + (P-1)(2\beta_L + \beta_T) \times \mathbf{n} + (P-1)(2\lambda_L + \lambda_T)}{\tau_c \times \mathbf{n^2} + (P-1)n_0\tau_c \times \mathbf{n} + P \times [(P-1)(2L(n_0) + T(n_0)) + (2\frac{n}{n_0} - 1)L(n_0)]}$$

et

$$lim_{n\to\infty} G^{1,2}(n) = \frac{P\tau_c \times o(n^2) + o(n)}{\tau_c \times o(n^2) + o(n)}$$

$$lim_{n\to\infty} G^{1,2}(n) = P$$

Distribution bidimensionnelle. La matrice de données bidimensionnelle est distribuée sur une grille bidimensionnelle de taille $\sqrt{P} \times \sqrt{P}$ processeurs en utilisant une distribution par blocs. Le temps de calcul d'une sous-matrice appartenant à un processeur est donné par :

$$C_n = \tau_c \times \frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}} = \tau_c \times \frac{n^2}{P}$$

Le temps de calcul d'un étage du pipeline d'une sous-matrice d'un processeur est donné par :

$$C_{n_0} = \tau_c \times \frac{n}{\sqrt{P}} \times n_0 \; avec \; 0 < n_0 \leq \frac{n}{\sqrt{P}}.$$

Lemme 3 *Soit* $t_{np}^{2,2}(n)$ *le temps total d'exécution d'une application 2-D de taille de problème* $n \times n$ *décomposée sur une grille carrée de dimension* $\sqrt{P} \times \sqrt{P}$ *processeurs sans pipeline. Alors*

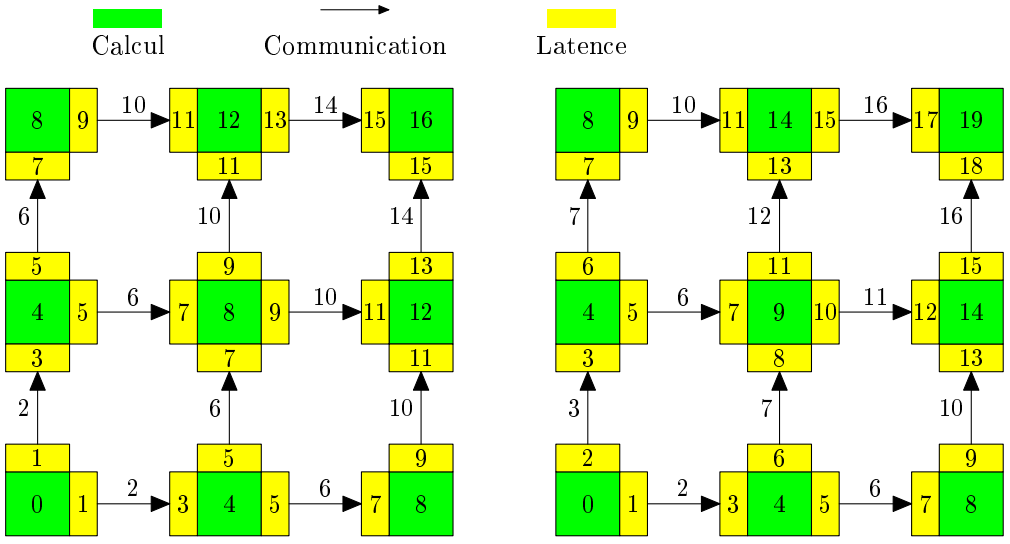
$$t_{np}^{2,2}(n) = (2\sqrt{P} - 1)C_n + (7\sqrt{P} - 8)L(n) + (2\sqrt{P} - 1)T(n) \quad (4.13)$$

Preuve. L'exécution se déroule par vagues transversales pendant lesquelles tous les processeurs appartenant à une même diagonale travaillent en parallèle. Nous revenons en détail sur les algorithmes par vague au Chapitre 4.2. Il y a plusieurs manières de modéliser les communications sur les deux dimensions. La figure 4.1.3 décompose les étapes de calcul et de communication où le coût de chaque action (calcul, latence et transmission) est égal à un. La figure de gauche (figure 4.8(a)) représente le cas où chaque processeur peut envoyer et recevoir deux messages simultanément. La figure de droite (figure 4.8(b)) représente le cas où un processeur ne peut envoyer et recevoir qu'un seul processeur à la fois. Un processeur envoie alors d'abord à son voisin Est puis à son voisin Sud et reçoit d'abord de son voisin Ouest puis de son voisin Nord.

Alors, si le modèle choisi permet à un processeur d'envoyer deux messages simultanément non seulement du point de vue de la latence mais aussi du point de vue du transfert comme le montre la figure 4.8(a), alors les étapes de calcul et celles de communication sont complètement synchronisées. On peut donc ajouter les $2\sqrt{P}-1$ étapes de calcul et les $2\sqrt{P}-2$ étapes de communication. Dans ce cas, le temps d'exécution non pipeliné est égal à

$$t_{np}^{2,2}(n) = (2\sqrt{P}-1)C_n + (2\sqrt{P}-2)[2L(n)+T(n)] \tag{4.14}$$

En pratique, si le nœud de calcul n'a qu'une seule entité réseau (coprocesseur de communication, carte et interface réseau), il est plus vraisemblable qu'un seul message ne pourra être envoyé à la fois. De plus, pour des raisons de simplicité de programmation, le programmeur privilégiera une direction plutôt qu'une autre. La figure 4.8(b) représente le cas où le processeur envoie d'abord à l'Est puis au Nord. Dans ce cas, Zory démontre donc les équations de temps d'exécution sans pipeline $t_{np}^{2,2}(n)$ donné à l'Équation 4.13.



(a) Envois et réceptions simultanés.

(b) Un envoi (ou réception) à la fois.

FIG. 4.8 – Exécutions d'une matrice 2D non pipelinées sur une grille 2D.

Dans ce même cas, il donne l'équation du temps d'exécution avec pipeline reprise ci-dessous, toujours en décomposant temps de démarrage, d'exécution courante et de terminaison.

Lemme 4 *Soit $t_p^{2,2}(n_0)$ le temps total d'exécution d'une application 2-D pipelinée ayant une taille de problème $n \times n$ et décomposée sur une grille carrée de dimension $\sqrt{P} \times \sqrt{P}$. Alors*

$$\begin{aligned} t_p^{2,2}(n) = & (\sqrt{P}-1) \times [L(n_0) + C_{n_0} + 2L(n_0) + T(n_0) \\ & + (\frac{n}{n_0} - 2)\max(T(n_0), L(n_0) + C_{n_0} + L(n_0))] \\ & + (\sqrt{P}-2) \times [2L(n_0) + C_{n_0} + L(n_0) + T(n_0) + \max(T(n_0), 2L(n_0) + C_{n_0} + L(n_0))] \\ & + \max(T(n_0), L(n_0) + C_{n_0} + L(n_0)) + L(n_0) + C_{n_0} + T(n_0) + C_{n_0} + L(n_0) \end{aligned} \tag{4.15}$$

Donc, en posant le gain du pipeline égal au rapport entre le temps d'exécution de l'application sans pipeline et le temps d'exécution pipeliné,

$$G^{2,2}(n) = \frac{t_{np}^{2,2}(n)}{t_p^{2,2}(n)}$$

il en déduit le corollaire suivant

Corollaire 2 *Si $lim_{n\to\infty}\frac{T(n)}{C_n} = 0$ et $lim_{n\to\infty}\frac{L(n)}{C_n} = 0$, alors $G^{2,2} = P$.*

Ce corollaire établit que si $T(n)$ et $L(n)$ ont une complexité inférieure à C_n, alors le gain en performance a pour ordre de grandeur P.

Preuve. Si $C_n >> T(n)$ alors

$$lim_{n\to\infty}G^{2,2}(n) = \frac{P\tau_c \times o(n^2) + o(n)}{\tau_c \times o(n^2) + o(n)} \tag{4.16}$$

$$lim_{n\to\infty}G^{2,2}(n) = P \tag{4.17}$$

Données tridimensionnelles

Zory et Desprez [76] étendent les formules de gain trouvées pour des données bidimensionnelles aux données tridimensionnelles et en déduisent les gains suivants.

Dans le cas d'une distribution bidimensionnelle, le gain d'un pipeline $G^{2,3}$ sera borné par $2 \times \sqrt{P}$.

Corollaire 3 *Si $lim_{n\to\infty}\frac{T(n)}{C_n} = 0$ et $lim_{n\to\infty}\frac{L(n)}{C_n} = 0$, alors $G^{2,3} = 2 \times \sqrt{P}$.*

Et enfin, dans le cas d'une distribution tridimensionnelle, le gain d'un pipeline $G^{3,3}$ sera borné par 3.

Corollaire 4 *Si $lim_{n\to\infty}\frac{T(n)}{C_n} = 0$ et $lim_{n\to\infty}\frac{L(n)}{C_n} = 0$, alors $G^{3,3} = 3$.*

Conclusion

Cette étude du gain à pipeliner montre qu'il est possible d'atteindre un gain significatif en pipelinant, jusqu'à un facteur P où P est un nombre de processeurs. Des expériences viennent appuyer ces résultats en utilisant l'application Sweep3D dans la partie suivante. Les meilleurs gains sont obtenus ici pour des distributions monodimensionnelles. En effet, dans ce cas, la séquentialité du code est importante et le fait de pipeliner augmente significativement le parallélisme.

Les plus mauvais cas correspondent aux cas où la matrice de données est distribuée sur toutes ses dimensions. Dans ces cas, sans pipeline, le parallélisme potentiel de l'application est déjà pleinement exploité. Néanmoins, le gain à pipeliner deux dimensions tend vers deux lorsque la taille des données augmente et celui à pipeliner trois dimensions vers trois.

4.2 Étude du Sweep3D

Dans cette partie, nous présentons l'étude de l'application Sweep3D. Dans cette étude, nous avons repris la modélisation développée par Hoisie *et al.* ; nous avons corrigé l'expression du temps de calcul et utilisé cette modélisation dans le but de valider les expressions de gains données par Desprez et Zory sur une application réelle. Le Sweep3D est utilisé pour faire des mesures de performance des machines du programme américain ASCI [3]. Les équations obtenues spécifiques à une application par vague comme le Sweep3D retrouvent les estimations de gain plus générales détaillées de la partie précédente. De plus, nous avons mené deux types d'expérimentations. Nous avons exécuté le Sweep3D sur des grappes de PC de manière pipelinée et non pipelinée afin de mesurer le gain du pipeline. Nous avons aussi programmé un simulateur de pipeline à deux dimensions afin de valider rapidement les équations obtenues par rapport au modèle choisi.

Dans un deuxième temps, nous avons déterminé en fonction des paramètres matériels et logiciels la meilleure distribution possible. Dans ce but, nous avons exprimé puis comparé les temps d'exécution de cette application pour les distributions de données mono-, bi- et tridimensionnelles. Ces temps d'exécution sont exprimés en fonction des caractéristiques de la machine et de l'application (temps de transfert sur le réseau, vitesse du processeur de calcul, temps de calcul élémentaire) dans le but de déterminer la meilleure distribution des données.

4.2.1 L'application Sweep3D

Description

Sweep3D est un solveur d'une équation de transport de particules en trois dimensions et indépendant du temps [13, 29, 39]. Sweep3D calcule le flux de particules traversant une région donnée de l'espace, flux qui pour chaque région est dépendant du flux venant des cellules voisines. L'espace 3D est discrétisé en cellules 3D. Le calcul s'exécute par vagues à partir des huit octants de l'espace 3D, chaque octant contenant six angles de direction de flux indépendants (un pour chacune des faces des cellules cubiques).

[3] ASCI :Accelerated Strategic Computing Initiative [48]

Cet espace 3D est alors distribué sur une grille à deux dimensions de processeurs dans les directions I et J (voir figure 4.9). Dans cette configuration, un processeur calcule un flux traversant une colonne de cellules et envoie ensuite les frontières nécessaires à ses deux processeurs voisins.

Pour améliorer les performances, les données suivant la dimension K et les angles de direction sont divisés en blocs, permettant ainsi à un processeur de ne calculer qu'une partie des valeurs selon la dimension K et qu'une partie des angles avant de les envoyer aux processeurs voisins.

Le Sweep3D exploite plusieurs types de parallélisme :
- du parallélisme inhérent à cette application sur les angles : chaque balayage pour un angle donné est indépendant ;
- du parallélisme par « vague » : décomposition spatiale en 2D sur une grille 2D de processeurs ;
- de plus, des blocs de calcul sur la dimension K et divisant le nombre d'angles de direction peuvent être pipelinés sur la grille de processeurs ;
- enfin, le démarrage du calcul d'un octant peut démarrer avant la fin du calcul de l'octant précédent, ceci se limite aux calculs de deux octants simultanément.

Nous nous intéressons ici au pipeline des calculs sur la dimension K.

Pipeline selon la dimension K

Pour valider les équations de Zory et Desprez [75], détaillées dans la partie précédente, nous nous intéressons au pipeline induit par le découpage en paquets de blocs de données selon la dimension K.

En effet, le découpage en paquets des blocs de données sur la dimension K induit un pipeline à deux dimensions sur les dimensions I et J.

Les caractéristiques du Sweep3D sont alors les suivantes :
- l'ensemble des données est une matrice 3D de cellules de taille n^3 ;
- les dépendances de calculs sont de la forme $(1,0)$ et $(0,1)$ (voir la figure 4.5) ;
- les données sont distribuées sur deux dimensions I et J sur une grille 2D de processeurs de taille $p \times q$.
- les calculs sont divisés en paquets sur la dimension K induisant un pipeline à deux dimensions sur les dimensions I et J.

Travaux existants

L'implantation du Sweep3D a été beaucoup étudiée. Tout d'abord, Koch, Baker et Alcouffe [39] ont développé une formule donnant l'efficacité parallèle ne prenant en compte que les calculs.

Pour déterminer les performances de systèmes de calcul de grande échelle, le projet POEMS [3, 4, 13, 5, 67, 70, 71] a modélisé le Sweep3D en utilisant le modèle de communication LogGP (détaillé dans la partie 4.1.2) dans le but de prévoir précisément le temps d'exécution de cette application et la meilleure configuration du Sweep3D sur une architecture donnée.

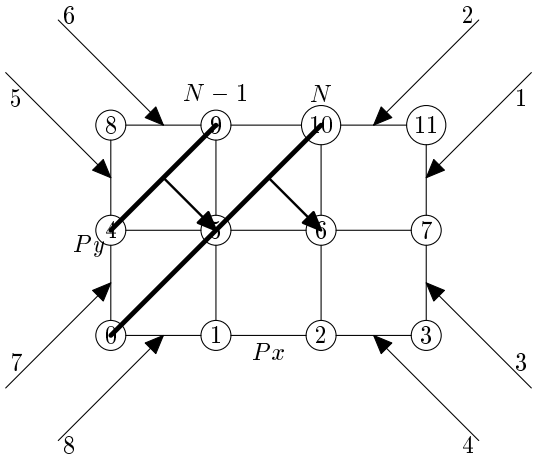


FIG. 4.9 – Deux vagues consécutives $N-1$ et N de balayages traversent la grille 4×3 de processeurs. L'espace 3D sera balayé plusieurs fois, à partir de chacun des sommets de l'espace 3D, appelé octant. Les directions des huit octants sont indiquées.

Récemment, Hoisie *et al.* [29, 31, 30] ont développé et validé un modèle donnant les performances parallèles d'applications en vagues d'exécutions bi-dimensionnelles sur des architectures à passage de messages.

Dans la partie suivante, nous détaillons le modèle donné par Hoisie *et al.*.

Le modèle donné par Hoisie *et al.*

Le modèle donné par Hoisie *et al.* [29, 31, 30] combine la contribution des vagues de calculs et celle des vagues de communications. Leur modèle est basé sur l'équation ci-dessous. Cette équation donne le nombre d'étapes nécessaires pour exécuter le calcul de N_{sweep} vagues. Chacune de ces vagues parcourt un pipeline de N_s étages avec un délai de répétition d.

$$Steps = N_s + d(N_{sweep} - 1) \tag{4.18}$$

La première vague sort du pipeline après N_s étapes et les vagues suivantes sortent à une fréquence $1/d$.

Le pipeline est constitué d'étapes de calculs et d'étapes de communications. Le nombre d'étapes de ces deux types et le délai de répétition de leurs vagues sont définis comme une fonction dépendant du nombre de processeurs et de la configuration de la grille de processeurs.

Étapes de calculs. Le nombre d'étapes de calculs est simplement le nombre de diagonales de la grille. En effet, un nombre différent de processeurs est utilisé à chacune des étapes du calculs mais toutes les étapes ont le même temps d'exécution dans la mesure où les processeurs situés sur une même diagonale calculent en parallèle.

L'Équation 4.19 donne le nombre d'étapes de calculs dans le pipeline. Le nombre d'étapes de calculs est égal au nombre de diagonales de la grille de processeurs.

$$N_s^{comp} = P_x + P_y - 1 \tag{4.19}$$

Le balayage suivant peut commencer dès que le premier processeur a terminé son calcul donc le délai de répétition d'une étape de calculs est :

$$d^{comp} = 1$$

Étapes de communications. Les communications sont bloquantes et synchrones. De plus, tel que le Sweep3D est implanté, pour un balayage vers le bas à droite, l'ordre des réceptions est tout d'abord en provenance de l'Ouest, puis du Nord et l'ordre des envois est tout d'abord vers l'Est puis le Sud.

Ainsi le nombre d'étapes de communications est égal à

$$N_s^{comm} = 2(P_y - 1) + 2(P_x - 1) \tag{4.20}$$

Comme le message que doit envoyer le processeur en haut à gauche (processeur 0) à son voisin Est (processeur 1) à la deuxième vague ne peut pas être initialisé avant que le processeur 1 n'ait terminé sa communication avec son voisin Sud à la première vague, le délai de répétition du pipeline de communications est

$$d^{comm} = 4$$

Combinaison des étapes de calculs et des étapes de communications. Dans le cas présent, les communications étant bloquantes et synchrones, cela assure qu'il n'y a aucun recouvrement donc nous pouvons ajouter les contributions des calculs et des communications.

Soit $T^{cpu}(n)$ le coût d'une étape de calculs et $T^{msg}(n)$ le coût d'une étape de communications (voir ci-dessous les Équations 4.21 et 4.22), le temps total d'exécution est donné par l'équation

$$T_{total}(n) = (N_s^{comp} + d^{comp}(N_{sweep} - 1)) \times T^{cpu}(n) + (N_s^{comm} + d^{comm}(N_{sweep} - 1)) \times T^{msg}(n)$$

4.2.2 Performances des pipelines

Coût du calcul distribué sur deux dimensions. Le coût $T^{cpu}(n)$ d'une étape de calcul dans le cas d'une distribution à deux dimensions est égal à

$$T^{cpu}(n) = \frac{N_x}{P_x} \times \frac{N_y}{P_y} \times n \times \tau_c \tag{4.21}$$

n est égal au nombre d'éléments calculés. On note N_x, N_y le nombre de points de la grille dans chacune des dimensions I et J. $T^{cpu}(n)$ représente le coût associé au calcul d'une sous-grille sur chaque processeur.

Coût de communication pour une distribution sur deux dimensions. Le coût d'une étape de communications pour une distribution à deux dimensions est égal à

$$T^{msg}(n) = L(n) + T(n) + L(n) \tag{4.22}$$

$T^{msg}(n)$ représente le temps nécessaire pour terminer un couple d'envoi/réception d'une taille n. $L(n)$ et $T(n)$ sont respectivement le temps de latence du message et le temps de transfert du message donnés respectivement par les Équations 4.2 et 4.1.

Donc, avec $\lambda = 2 \times \lambda_L + \lambda_T$ et $\beta = 2 \times \beta_L + \beta_T$, le coût d'une étape de communications est

$$\begin{aligned} T^{msgX}(n) &= \beta \times (\frac{N_y}{P_y} \times n) + \lambda \\ T^{msgY}(n) &= \beta \times (\frac{N_x}{P_x} \times n) + \lambda \end{aligned} \tag{4.23}$$

avec $T^{msgX}(n)$ étant le coût des communications dans la direction I et $T^{msgX}(n)$ étant le coût des communications dans la direction J.

Un octant du Sweep3D

Nous ne considérons dans cette partie qu'un seul octant dans l'application Sweep3D. En utilisant le modèle de Hoisie *et al.*, nous pouvons évaluer simplement le gain en performance apporté par le pipeline de la dimension K et des angles.

Pipeliner la dimension K génère un schéma de communications à deux dimensions dans la direction I et dans la direction J. En outre, si K_b est le nombre de blocs dans la direction K, K_b balayages sont nécessaires pour calculer un bloc d'angles.

De plus, pipeliner les angles a pour conséquence de multiplier par A_b le nombre de balayages nécessaires pour exécuter le calcul total avec A_b le nombre de blocs d'angles.

Soit n le nombre d'éléments à calculer dans les dimensions pipelinées, n est donné par

$$n = N_z \times N_a$$

où N_z est le nombre de points de la grille dans la direction K et N_a est le nombre d'angles.

Soit n_0 le nombre d'éléments de calcul dans chacun des blocs des dimensions pipelinées, n_0 est donné par

$$n_0 = \frac{N_z}{K_b}\frac{N_a}{A_b}$$

Donc, s'il n'y a pas de pipeline, $K_b = 1$ et $A_b = 1$ et $n_0 = n$. Donc pipeliner à la fois dans la dimension K et en angles implique que le nombre de balayages nécessaires pour exécuter du calcul total est égal à

$$N_{sweep} = \frac{n}{n_0} \tag{4.24}$$

Lemme 5 *Soit $t_{np}^{1O}(n)$ le temps d'exécution d'un octant du Sweep3D sans pipeliner la dimension K ni les angles. Alors*

$$t_{np}^{1O}(n) = (P_x+P_y-1)\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}n + 2\beta((P_x-1)\frac{N_y}{P_y}+(P_y-1)\frac{N_x}{P_x})\times n + 2(P_x+P_y-2)\lambda \quad (4.25)$$

Preuve. Sans pipeline, $N_{sweep} = 1$. Donc, en utilisant la définition du nombre d'étapes du pipeline donnée par l'Équation générale 4.18 et le nombre d'étapes de calculs et de communications donné par les Équations 4.19 and 4.20,

$$t_{np}^{1O}(n) = (P_x + P_y - 1) \times T^{cpu}(n) + 2(P_x - 1) \times T^{msgX}(n) + 2(P_y - 1) \times T^{msgY}(n)$$

avec $T^{msgX}(n)$ et $T^{msgY}(n)$ donnés par l'Équation 4.23.

$$t_{np}^{1O}(n) = (P_x + P_y - 1) \times \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y} \times n + 2(P_x - 1) \times (\frac{N_y}{P_y}n\beta + \lambda) + 2(P_y - 1) \times (\frac{N_x}{P_x}n\beta + \lambda)$$

Lemme 6 *Soit $t_p^{1O}(n_0)$ le temps d'exécution d'un octant du Sweep3D en pipelinant à la fois la dimension K et les angles.*

$$\begin{aligned} t_p^{1O}(n_0) = {} & \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}n + (P_x + P_y - 2)n_0\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y} + 2n_0\beta((P_x + \frac{n}{n_0} - 2)\frac{N_y}{P_y} \\ & + (P_y + \frac{n}{n_0} - 2)\frac{N_x}{P_x}) + 2(P_x + P_y + 2\frac{n}{n_0} - 4)\lambda \end{aligned} \quad (4.26)$$

Preuve. En pipelinant, N_{sweep} est donné par l'Équation 4.24. En utilisant le nombre d'étapes d'un pipeline donné par l'Équation générale 4.18, et le nombre d'étapes de calculs et le nombre d'étapes de communications donnés aux Équations 4.19 et 4.20 comme au Lemme 5

$$\begin{aligned} t_p^{1O}(n_0) = {} & (P_x + P_y - 1 + \frac{n}{n_0} - 1) \times T^{cpu}(n_0) + (2(P_x - 1) + 2(\frac{n}{n_0} - 1)) \times T^{msgX}(n_0) \\ & + (2(P_y - 1) + 2(\frac{n}{n_0} - 1)) \times T^{msgY}(n_0) \end{aligned}$$

Théorème 2 *Soit $G^{1O}(n)$ le gain en performance lié aux pipelines de la dimension K et des angles dans l'exécution d'un octant du Sweep3D. Alors*

$$G^{1O}(n) = \frac{t_{np}^{1O}(n)}{t_p^{1O}(n_0)} \quad (4.27)$$

Corollaire 5 *Si $lim_{n\to\infty}\frac{T(n)}{T^{cpu}(n)} = 0$ and $lim_{n\to\infty}\frac{L(n)}{T^{cpu}(n)} = 0$, alors $G^{1O}(n) = P_x + P_y - 1$.*

Preuve. En utilisant l'Équation 4.27,

$$lim_{n\to\infty}G^{1O}(n) = lim_{n\to\infty}\frac{(P_x+P_y-1)\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}n}{\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}n} = P_x+P_y-1$$

Corollaire 6 *Si $P_x = P_y = \sqrt{P}$ alors*

$$lim_{n\to\infty}G^{1O}(n) = 2\sqrt{P}-1$$

Preuve. En utilisant le Corollaire 5.

Remarque. Remarquons qu'évidemment $t_p^{1O}(n) = t_{np}^{1O}(n)$.

Pipeline des huit octants

Le nombre d'étapes de calculs et de communications décrites précédemment (respectivement N_s^{comp} et N_s^{comm}, Équations 4.19 et 4.20) représentent les vagues de calculs parallèles sans prendre en compte jusqu'à présent le recouvrement des calculs des octants. Comme nous l'avons vu à la figure 4.9, les calculs s'effectuent à partir des huit octants de l'espace. Le calcul de chaque octant dépend de conditions aux limites du calcul de l'octant précédent. Donc, un processeur ne commence le calcul de l'octant suivant que lorsqu'il a terminé le calcul de l'octant courant.

Soit $Q2$ le deuxième octant. Dès que $P3$ a terminé de calculer $Q2$ il peut commencer à calculer Q3 sans attendre que les autres processeurs aient terminé de calculer $Q2$.

Lemme 7 *Soit $t_{np}^{8O}(n)$ le temps d'exécution total du Sweep3D dans le cas où il n'y a pas de pipeline suivant la dimension K ni suivant les angles et où les huit octants sont pipelinés. Alors $t_{np}^{8O}(n)$ est donné par l'Équation 4.28*

$$\begin{aligned} t_{np}^{8O}(n) = {} & 2(P_x+2P_y+1)\tau_c \times \frac{N_xN_y}{P_xP_y}n + (2(3P_x+1)\beta+6\beta_L) \times \frac{N_y}{P_y}n \\ & + (2(3P_y+1)\beta+6\beta_L)\frac{N_x}{P_x}n + 2(3P_x+3P_y+2)\lambda + 12\lambda_L \end{aligned} \tag{4.28}$$

Preuve. Comme ni la dimension K ni les angles ne sont pipelinés, le calcul de chaque octant ne représente qu'un seul balayage. Pour chaque paire d'octants ($(Q1,Q2)$, $(Q3,Q4)$, $(Q5,Q6)$ et $(Q7,Q8)$, voir figure 4.9), le calcul du deuxième octant est pipeliné avec le calcul du premier. Donc,

$$N_{sweep} = 2 \tag{4.29}$$

De plus, dès qu'un processeur termine le calcul d'un octant, il commence le calcul de l'octant suivant. Par exemple, soit une grille de 4×3 processeurs comme le montre la figure

4.9. Dans ce cas, dès que $P3$ termine de calculer l'octant $Q2$, il peut commencer à calculer $Q3$, donc seulement après P_y étapes de calcul.

$$N_s^{comp} = P_y$$

$P3$ commencera à calculer $Q3$ après deux balayages (un balayage pour l'octant $Q1$ et un pour $Q2$) de longueur P_y étapes de calcul et $P_x - 1$ étapes de communication dans la direction I et $P_y - 1$ étapes de communication dans la direction J.

$$N_s^{commX} = P_x - 1$$

$$N_s^{commY} = P_y - 1$$

En plus de ces étapes de communication, il faut ajouter un temps de latence d'envoi dans la direction I afin que $P8$ initialise la communication avec $P9$ sans attendre la réception effective du message par $P9$ et de la même manière un temps de latence dans la direction J afin que $P3$ initialise la communication avec $P7$.

$$N_s^{sendX} = 1$$

$$N_s^{sendY} = 1$$

En utilisant l'Équation générale 4.18 donnant le nombre d'étape d'un pipeline, si $T_{start}^{P3}(n)$ est la date à laquelle $P3$ commence à calculer l'octant $Q3$, $T_{start}^{P3}(n)$ est alors donné par

$$T_{start}^{P3}(n) = (P_y+1)T^{cpu}(n)+(P_x+1)T^{msgX}(n)+(P_y+1)T^{msgY}(n)+3L^{sendX}(n)+3L^{sendY}(n)$$

En effet, le nombre d'étapes de calculs est donné par

$$N_s^{comp} + d^{comp}(N_{sweep} - 1) = P_y + 1 \times (2 - 1) = P_y + 1$$

avec $d^{comp} = 1$.

De la même manière, le nombre d'étapes de communication dans la direction I est donné par

$$N_s^{msgX} + d^{comm}(N_{sweep} - 1) = P_x - 1 + 2 \times (2 - 1) = P_x + 1$$

avec $d^{comm} = 2$.

Et le nombre d'étapes de communication dans la direction J est donné par

$$N_s^{msgY} + d^{comm}(N_{sweep} - 1) = P_y + 1$$

De plus, comme expliqué ci-dessus, il y a un temps de latence d'envoi dans la direction I et un dans la direction J. Donc il faut ajouter

$$\begin{aligned} & (1 + d^{comm}(N_{sweep} - 1)) \times L^{sendX}(n) + (1 + d^{comm}(N_{sweep} - 1)) \times L^{sendY}(n) \\ = \; & 3L^{sendX}(n) + 3L^{sendY}(n) \end{aligned}$$

À l'opposé, $P8$ doit attendre jusqu'à ce que les deux octants $Q3$ et $Q4$ aient été finis de calculer avec de commencer à calculer $Q5$. Donc on exécute deux balayages de longueur $P_x + P_y - 1$ étapes de calculs (voir l'Équation 4.19) et $2(P_x - 1)$ étapes de communication dans la direction I et $2(P_y - 1)$ dans la direction J.

En utilisant l'Équation 4.18, si $T^{P8}_{start}(n)$ est la date à laquelle $P8$ commence à calculer $Q5$, $T^{P8}_{start}(n)$ est donné par

$$T^{P8}_{start}(n) = T^{P3}_{start}(n) + (P_x + P_y)T^{cpu}(n) + 2P_xT^{msgX}(n) + 2P_yT^{msgY}(n)$$

En effet, dans ce cas, le nombre d'étapes de calculs est donné par

$$N^{comp}_s + d^{comp}(N_{sweep} - 1) = P_x + P_y - 1 + 1 \times (2 - 1) = P_x + P_y$$

avec $d^{comp} = 1$.

Le nombre d'étapes de communication dans la direction I est donné par

$$N^{msgX}_s + d^{comm}(N_{sweep} - 1) = 2(P_x - 1) + 2 \times (2 - 1) = 2P_x$$

avec $d^{comm} = 2$.

Et, de la même manière, le nombre d'étapes de communication dans la direction I est donné par

$$N^{msgY}_s + d^{comm}(N_{sweep} - 1) = 2P_y$$

Enfin l'intégralité de l'exécution se termine après la réexécution de cet ensemble d'opérations mais au départ de $P8$. En effet, les exécutions de $Q5$ et 6 sont recouvertes avec les exécutions de $Q7$ et $Q8$ de la même manière que les exécutions de $Q1$ et $Q2$ étaient recouvertes par celles de de $Q3$ et $Q4$. C'est pourquoi :

$$t^{8O}_{np}(n) = 2 \times T^{P8}_{start}(n) \tag{4.30}$$

$$\begin{aligned} t^{8O}_{np}(n) = {} & 2(P_x + 2P_y + 1)T^{cpu}(n) + 2(3P_x + 1)T^{msgX}(n) \\ & + 2(3P_y + 1)T^{msgY}(n) + 6L^{sendX}(n) + 6L^{sendY}(n) \end{aligned}$$

En utilisant $T^{cpu}(n)$ et $T^{msg}(n)$ donnés par les Équations 4.21 et 4.22, nous obtenons l'Équation 4.28.

Lemme 8 *Soit $t^{8O}_p(n_0)$ le temps d'exécution total du Sweep3D quand la dimension K et les angles sont pipelinés et les huit octants sont aussi pipelinés. Alors $t^{8O}_p(n_0)$ est donné par l'Équation 4.31*

$$t_p^{8O}(n_0) = 8\tau_c \times \frac{N_x N_y}{P_x P_y} n + (2P_x + 4P_y - 6)\tau_c \times \frac{N_x N_y}{P_x P_y} n_0 + 16\beta(\frac{N_y}{P_y} + \frac{N_x}{P_x}) \times n$$
$$+ 8\beta_L(\frac{N_y}{P_y} + \frac{N_x}{P_x}) \times n + ((6P_x - 14)\frac{N_y}{P_y} + (6P_y - 14)\frac{N_x}{P_x})n_0\beta - 2n_0\beta_L(\frac{N_y}{P_y} + \frac{N_x}{P_x})$$
$$+ 32\frac{n}{n_0}\lambda + 16\frac{n}{n_0}\lambda_L + (6(P_x + P_y) - 28)\lambda - 4\lambda_L \tag{4.31}$$

Preuve. De la même façon que pour le lemme précédent, Lemme 7, nous calculons le nombre d'étapes de calculs et le nombre d'étapes de communications. La seule différence avec le lemme précédent est le nombre de balayages donné ici par

$$N_{sweep} = 2 \times \frac{n}{n_0}$$

En effet, le nombre de balayages est égal au nombre de blocs de sous-domaines comme nous l'avons à l'Équation 4.24 multiplié par le nombre de balayages de deux octants pipelinés (voir Équation 4.29).

Donc $T_{start}^{P3}(n_0)$ est maintenant donné par

$$T_{start}^{P3}(n_0) = (P_y + 2\frac{n}{n_0} - 1)T^{cpu}(n_0) + (P_x + 4\frac{n}{n_0} - 3)T^{msgX}(n_0)$$
$$+ (P_y + 4\frac{n}{n_0} - 3)T^{msgY}(n_0) + (4\frac{n}{n_0} - 1)(L^{sendX}(n_0) + L^{sendY}(n_0))$$

Le nombre d'étapes de calculs est donné par

$$N_s^{comp} + d^{comp}(N_{sweep} - 1) = P_y + 1 \times (2\frac{n}{n_0} - 1) = P_y + 2\frac{n}{n_0} - 1$$

avec $d^{comp} = 1$.

Le nombre d'étapes de communications dans la direction I est donné par

$$N_s^{msgX} + d^{comm}(N_{sweep} - 1) = P_x - 1 + 2 \times (2\frac{n}{n_0} - 1) = P_x + 4\frac{n}{n_0} - 3$$

avec $d^{comm} = 2$.

Et, de la même manière, le nombre d'étapes de communications dans la direction I est donné par

$$N_s^{msgY} + d^{comm}(N_{sweep} - 1) = P_y + 4\frac{n}{n_0} - 3$$

Il faut aussi ajouter les latences d'envois dans les directions I et J

$$(1 + d^{comm}(N_{sweep} - 1)) \times (L^{sendX}(n_0) + L^{sendY}(n_0))$$
$$= (1 + 2(2\frac{n}{n_0} - 1))(L^{sendX}(n_0) + L^{sendY}(n_0))$$
$$= (4\frac{n}{n_0} - 1)(L^{sendX}(n_0) + L^{sendY}(n_0))$$

$T^{P8}_{start}(n_0)$ est maintenant donné par

$$T^{P8}_{start}(n_0) = T^{P3}_{start}(n_0) + (P_x + P_y + 2\frac{n}{n_0} - 2)T^{cpu}(n_0) + (2P_x + 4\frac{n}{n_0} - 4)T^{msgX}(n_0) + (2P_y + 4\frac{n}{n_0} - 4)T^{msgY}(n_0)$$

avec le nombre d'étapes de calculs égal à

$$N^{comp}_s + d^{comp}(N_{sweep} - 1) = P_x + P_y - 1 + 1 \times (2\frac{n}{n_0} - 1) = P_x + P_y + 2\frac{n}{n_0} - 1)$$

avec $d^{comp} = 1$.

Et les nombres d'étapes de communications respectivement dans les directions I et J donnés par

$$N^{msgX}_s + d^{comm}(N_{sweep} - 1) = 2(P_x - 1) + 2 \times (2\frac{n}{n_0} - 1) = 2P_x + 4\frac{n}{n_0} - 4$$

avec $d^{comm} = 2$.

$$N^{msgY}_s + d^{comm}(N_{sweep} - 1) = 2P_y + 4\frac{n}{n_0} - 4$$

Donc, comme le temps total est égal au double du temps déjà écoulé (voir les explications de l'Équation 4.30), le temps total de l'application pipelinée en octants, en la dimension K et en angles est égal à

$$t^{8O}_p(n_0) = 2(P_x + 2P_y + 4\frac{n}{n_0} - 3)T^{cpu}(n_0) + (6P_x + 16\frac{n}{n_0} - 14)T^{msgX}(n_0) + (6P_y + 16\frac{n}{n_0} - 14)T^{msgY}(n_0) + 2(4\frac{n}{n_0} - 1)(L^{sendX}(n_0) + L^{sendY}(n_0))$$

En utilisant $T^{cpu}(n)$ et $T^{msg}(n)$ donnés par les Équations 4.21 et 4.22, nous obtenons l'Équation 4.31.

Théorème 3 *Soit $G^{8O}(n)$ le gain en performance résultant du pipeline du Sweep3D en la dimension K et en angles quand les huit octants sont pipelinés.*

$$G^{8O}(n) = \frac{t^{8O}_{np}(n)}{t^{8O}_p(n_0)} \tag{4.32}$$

Corollaire 7 *Si $lim_{n\to\infty}\frac{T(n)}{T^{cpu}(n)} = 0$ et $lim_{n\to\infty}\frac{L(n)}{T^{cpu}(n)} = 0$, alors $G^{8O}(n) = \frac{P_x+2P_y+1}{4}$.*

Preuve. En utilisant l'Équation 4.32,

$$lim_{n\to\infty} G^{8O}(n) = lim_{n\to\infty} \frac{2(P_x + 2P_y + 1)\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}n}{8\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}n} = \frac{P_x + 2P_y + 1}{4}$$

Corollaire 8 *Si* $P_x = P_y = \sqrt{P}$ *alors*

$$lim_{n\to\infty} G^{8O}(n) = \frac{3\sqrt{P} + 1}{4}$$

Preuve. En utilisant le Corollaire 7.

Remarque. Remarquons qu'évidemment $t_p^{8O}(n) = t_{np}^{8O}(n)$.

Conclusion

Ces modèles d'exécution du Sweep3D montrent qu'il est possible d'écrire des équations précises décrivant une exécution pipelinée sur des distributions mono, bi et tridimensionnelles. Ce modèle est néanmoins un modèle d'exécution synchrone. L'exécution du Sweep3D, par sa régularité, ne gagnerait pas à utiliser des communications asynchrones, en revanche à chaque étape, l'on pourrait gagner une partie du temps de calcul en recouvrant les communications. Les études de gain montrent qu'asymptotiquement, les valeurs obtenues rejoignent celles obtenues par Zory, en utilisant un modèle de pipeline entièrement asynchrone. Ces modélisations représentent donc une première validation du modèle de Zory.

Résultats expérimentaux

Gain sur un octant. Nous avons exécuté le Sweep3D sur la grappe de PC PoPC décrite dans la partie 2.4 et comparé les résultats expérimentaux à ceux donnés par les Équations 4.25 et 4.26. Nous avons aussi programmé un simulateur d'application par vague distribuée sur une grille de processeurs à deux dimensions. Nous avons cherché à extrapoler les résultats obtenus sur PoPC au-delà de douze processeurs grâce à ce simulateur. Ce simulateur ne simule que le balayage d'un seul octant. Il prend les mêmes paramètres que les équations, c'est-à-dire un paramètre lié à l'application, le temps d'un calcul d'un élément, et des paramètres liés à la machine qui sont les paramètres de communication, dans notre modèle latence et transmission.

La figure 4.10 montre que la mesure des ces paramètres est délicate. Sur PoPC, nous avons mesuré :

$$\begin{aligned} \tau_c &= 1.46 \times 10^{-5} \\ \lambda_L &= 4.9 \times 10^{-5} \\ \beta_T &= 2.1 \times 10^{-8} \end{aligned}$$

Le temps de calcul élémentaire τ_c a été obtenu en exécutant l'application sur un seul processeur et en traçant une courbe en fonction de la taille des données. Les paramètres de

communication λ_L et β_T ont été obtenu en effectuant des ping-pong entre deux nœuds de la grappe.

Les premières courbes comparent les exécutions d'un octant avec une taille de bloc égale à un sur quatre processeurs effectuées sur PoPC, avec le simulateur et le résultat des équations. On voit que les résultats obtenus avec le simulateur et les équations sont proches. En revanche, les mesures effectuées sur PoPC donnent des temps d'exécution inférieurs d'environ 15%. La deuxième courbe montre qu'en ajustant le temps de calcul élémentaire d'un élément (de 1.46×10^{-5}, on passe à 1.26×10^{-5}) les courbes coïncident.

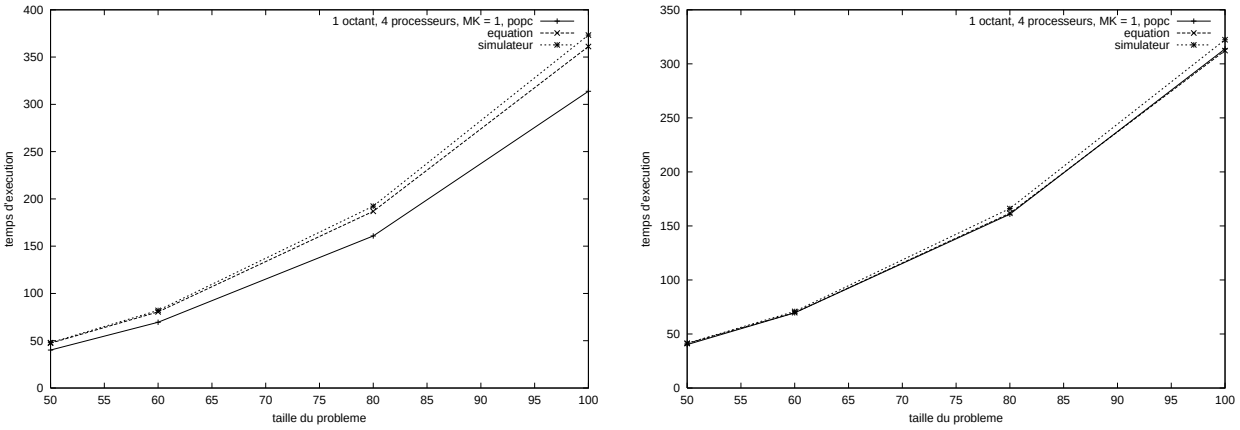


FIG. 4.10 – Comparaison entre le simulateur, l'équation et les exécutions sur PoPC sur une grille de quatre processeurs respectivement avec le coefficient de calcul mesuré et le coefficient de calcul ajusté.

Sur PoPC, il n'était possible d'effectuer des exécutions que pour des tailles de problème inférieures à $120 \times 120 \times 120$ sur douze processeurs. Avec le simulateur, nous avons effectué des exécutions avec le paramètre de calcul mesuré jusqu'à 100 processeurs et pour des tailles de problème jusqu'à $300 \times 300 \times 300$.

Pour des grandes tailles de données, le gain est alors proche de l'idéal égal, ici, à $2 \times \sqrt{P} - 1$ (voir les Corollaires 2 et 6) sur une grille de processeurs. Il n'est pas nécessaire de l'exécuter sur un grand nombre de processeurs.

Comme nous l'avons dit, la taille de bloc donnant le meilleur temps d'exécution dépend du rapport entre le temps de calcul d'un sous-bloc et le temps de communication. Nous avons cherché avec les paramètres de PoPC à faire apparaître une taille de bloc optimale différente de 1. Il a fallu pour cela déséquilibrer le temps de calcul en le rendant très petit et le temps de communication en l'augmentant énormément. La figure 4.12 montre que pour une petite taille de problème ($60 \times 60 \times 60$ éléments) et un très grand nombre de processeurs, 1024, la taille de bloc optimale se situe aux alentours de 3. Néanmoins, pour obtenir le meilleur temps de calcul, il faudra probablement réduire le nombre de processeurs.

Gain sur huit octants. Nous avons comparé l'adéquation entre les équations représentant le temps d'exécution de l'application Sweep3D sur huit octants sur une grille de processeurs

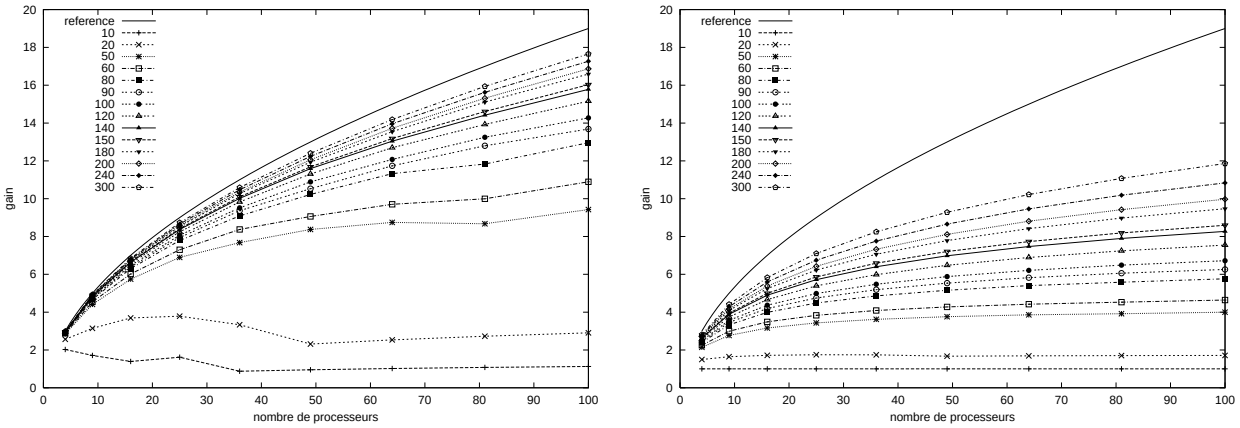


FIG. 4.11 – Comparaison des résultats obtenus avec le simulateur avec le coefficient de calcul mesuré et la courbe idéale de Zory sur une grille de processeurs pour respectivement MK = 1 et MK = 10. Chaque courbe est tracée pour une taille de données fixée.

(Équations 4.28 et 4.31) et les temps d'exécution obtenus sur PoPC. Là encore, la détermination des paramètres donnés aux équations est délicate.

La figure montre la différence entre le temps de calcul élémentaire mesuré (courbes de gauche) et celui ajusté (courbes de droite). Pour la plus grande taille de données, $120 \times 120 \times 120$, on obtient sur six processeurs avec une taille de bloc égale à un, un temps d'exécution sur PoPC de 20% inférieur sur six processeurs, jusqu'à 30% inférieur sur douze processeurs. Avec le temps de calcul élémentaire ajusté, les différences sont toutes les deux de l'ordre de 3%.

La figure 4.14 montre que pour des plus grandes tailles de bloc, ici cinq et dix, le temps de calcul élémentaire mesuré donne de bons résultats et que le temps d'exécution sur PoPC et celui donné par les équations sont proches. On obtient des différences entre 4 et 10% pour une taille de bloc égale à cinq et de 1 à 5% pour une taille de bloc égale à dix pour un problème de taille $120 \times 120 \times 120$ sur six et douze processeurs.

Expériences sur d'autres machines. Nous avons mesuré les performances du Sweep3D sur une autre grappe de PC, Icluster ainsi que sur l'architecture IBM SP-2 du CINES (voir la partie 2.4).

La figure 4.15 donne les mesures de gain du pipeline des huit octants sur la grappe de PC Icluster. Comme nous pouvons le voir, le gain tend moins vite vers l'idéal établi par Desprez et Zory que dans le cas de la grappe de PC PoPC. Des exécutions ont été effectuées jusqu'à 50 processeurs pour une taille de données maximale de $200 \times 200 \times 200$. Pour obtenir une meilleure convergence, il est probablement nécessaire d'augmenter la taille des données pour que la puissance des processeurs soit pleinement exploitée. Sur la grappe de PC PoPC, nous avions exécuté des mesures sur 12 processeurs pour des tailles de données $150 \times 150 \times 150$. La taille des données est limitée par la mémoire disponible sur chaque processeur. Les données sont des réels représentés en double précision ; $150 \times 150 \times 150$ représentent 27 Mo de données, soit 2 Mo par nœud sur douze nœuds.

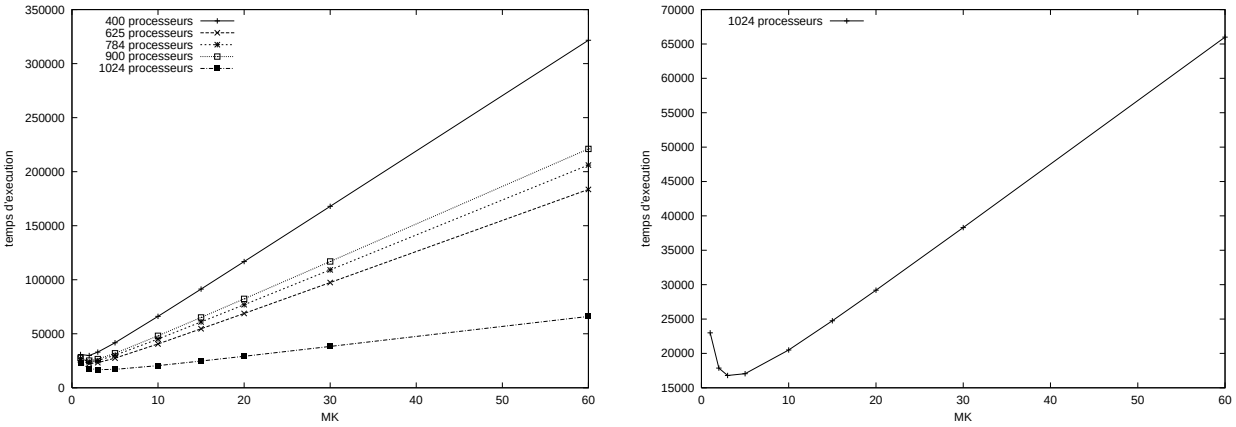


FIG. 4.12 – Recherche d'un minimum avec une petite taille de données ($60 \times 60 \times 60$ éléments) et une grille de 400 à 1024 processeurs ; gros plan sur l'exécution avec 1024 processeurs.

La figure 4.16 donne les mesures de gain des pipelines pour un et huit octants sur l'IBM SP-2 du CINES. Leur représentation met clairement en évidence le fait que ces mesures dépassent énormément le gain maximum que l'on pourrait obtenir alors que ce même gain négligeait déjà le coût des communications et représentait donc une borne infranchissable. Nous pensons que ces irrégularités peuvent avoir deux origines. La première est la mauvaise gestion de la mémoire par le compilateur Fortran 90 que nous avons utilisé. Ce phénomène s'était déjà produit sur la grappe de PC Icluster mais nous avions alors pu utiliser un autre compilateur. La deuxième cause possible est que cette mauvaise gestion de la mémoire a accru les coûts liés à la hiérarchie mémoire (effet de cache, accès au disque). Dans ce cas, le temps de calcul d'un bloc d'éléments dans le cas d'un pipeline à grain fin n'est plus proportionnel au temps de calcul d'un plus grand bloc d'éléments et ainsi, le modèle que nous avons choisi n'est plus adapté et le gain idéal ainsi calculé est largement dépassé.

Ces hypothèses sont corroborées par deux phénomènes. Le premier est que, sur moins de processeurs, ces distorsions apparaissent moins. Les tailles des blocs de calcul varient moins lorsque les processeurs sont moins nombreux. Ensuite, nous avons vu que le temps élémentaire de calcul était très difficile à évaluer, essentiellement parce que les coûts liés à la hiérarchie mémoire ne sont pas négligeables.

Les paramètres que nous avons mesurés sur les grappes de PC PoPC, Icluster et sur l'IBM SP-2 du CINES sont récapitulés dans le tableau suivant :

	Grappe de PC PoPC	Grappe de PC Icluster	IBM SP-2 CINES
τ_c	1.46×10^{-5}	3.9×10^{-5}	2.06×10^{-5}
λ_L	4.9×10^{-5}	2.2×10^{-5}	3.89×10^{-5}
β_T	2.1×10^{-8}	4.6×10^{-9}	5.01×10^{-9}

La figure 4.17 compare le temps d'exécution des huit octants du Sweep3D sur la grappe de PC Icluster et du temps donné par l'Équation 4.31 dans le cas de pipeline à grain fin. Avec le coefficient de calcul mesuré, on voit que l'équation donne des temps d'exécution supérieurs au temps d'exécution sur la grappe de PC. Il est alors envisageable que les mesures effectuées

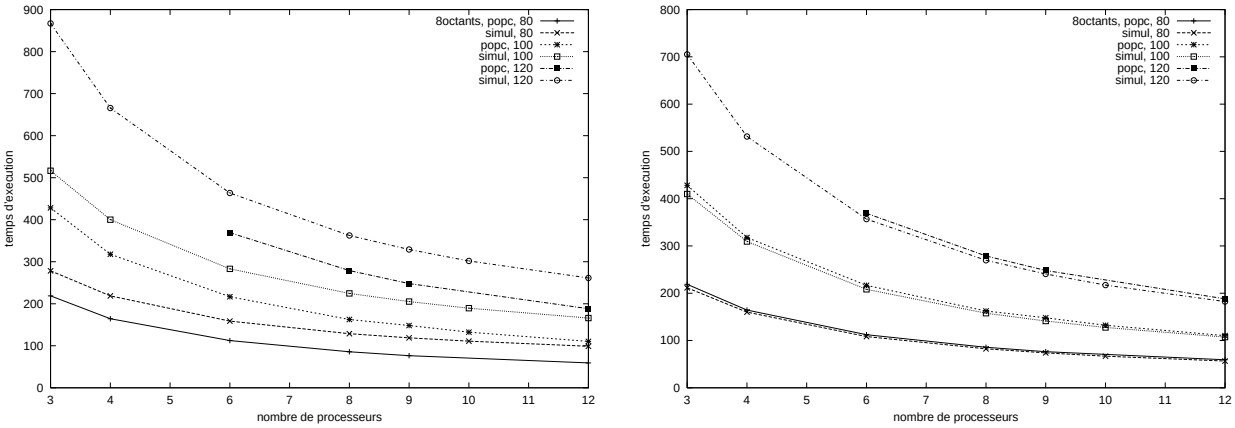


FIG. 4.13 – Comparaison des résultats de l'équation et des exécutions sur PoPC avec huit octants pour MK = 1 respectivement avec le coefficient de calcul mesuré puis le coefficient de calcul ajusté.

sur un processeur pour établir le temps de calcul élémentaire prennent en compte des effets de cache qui n'ont pas lieu lors d'exécution sur plus de processeurs dans le cas de pipeline à grain fin. Pour faire coïncider les deux courbes, exécution et équation, il faut diviser le temps de calcul élémentaire par sept !

4.2.3 Détermination de la meilleure distribution

Dans cette partie, nous cherchons à déterminer la meilleure distribution possible pour le Sweep3D entre une distribution mono-, bi- et tridimensionnelle. Dans ce but, nous établissons les temps d'exécution d'un octant du Sweep3D pour les distributions 1D et 3D du Sweep3D afin d'en déduire celle qui donnera le plus petit temps d'exécution sur la machine cible. La figure 4.18 illustre les trois différentes distributions considérées dans cette partie.

Distribution monodimensionnelle

La matrice de données est distribuée sur P processeurs selon une dimension.

Coûts de calcul et de communication. Le coût d'une étape de calcul pour une distribution monodimensionnelle dans la direction I est donné par

$$T^{cpu}(n) = \tau_c \times \frac{N_x}{P} \times N_y \times n$$

Le coût d'une étape de communication pour une distribution monodimensionnelle dans la direction I est donné par

$$T^{msgX}(n) = \beta \times N_y \times n + \lambda$$

où $n = N_z \times N_a$ sans pipeliner la dimension K ni les angles et $n = n_0 = \frac{N_z}{K_b}\frac{N_a}{A_b}$ avec ces pipelines.

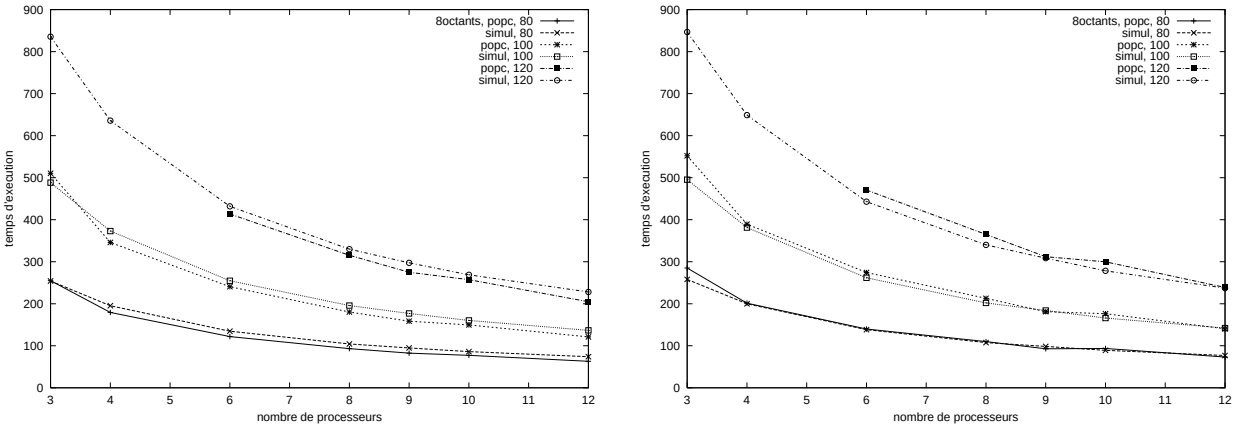


FIG. 4.14 – Simulateur d'équation avec le coefficient de calcul mesuré avec huit octants et exécutions sur PoPC pour respectivement MK = 5 et MK = 10.

Lemme 9 *Soit $t_{np}^{1D}(n)$ le temps d'exécution d'un octant du Sweep3D avec une distribution monodimensionnelle sans pipeliner la dimension K ni les angles. Alors*

$$t_{np}^{1D}(n) = \tau_c \times N_x N_y n + (P-1)\beta \times N_y n + (P-1)\lambda$$

Preuve. Sur une distribution monodimensionnelle, le nombre d'étapes de calculs est donné par

$$N_s^{comp} = P \tag{4.33}$$

Le délai de répétition d'une étape de calcul sur une distribution monodimensionnelle est

$$d^{comp} = 1 \tag{4.34}$$

En outre, sur une distribution monodimensionnelle, le nombre d'étapes de communication est donné par

$$N_s^{comm} = P - 1 \tag{4.35}$$

Et le délai de répétition d'une étape de communication sur une distribution monodimensionnelle est

$$d^{comm} = 2 \tag{4.36}$$

En outre, sans pipeline, $N_{sweep} = 1$. Donc en utilisant le nombre d'étapes d'un pipeline donné par l'Équation générale 4.18 et le nombre d'étapes de calculs et de communications et les délais donnés aux Équations 4.33, 4.35, 4.34, 4.36, le temps d'exécution est donné par

$$t_{np}^{1D}(n) = P \times T^{cpu}(n) + (P-1) \times T^{msgX}(n)$$

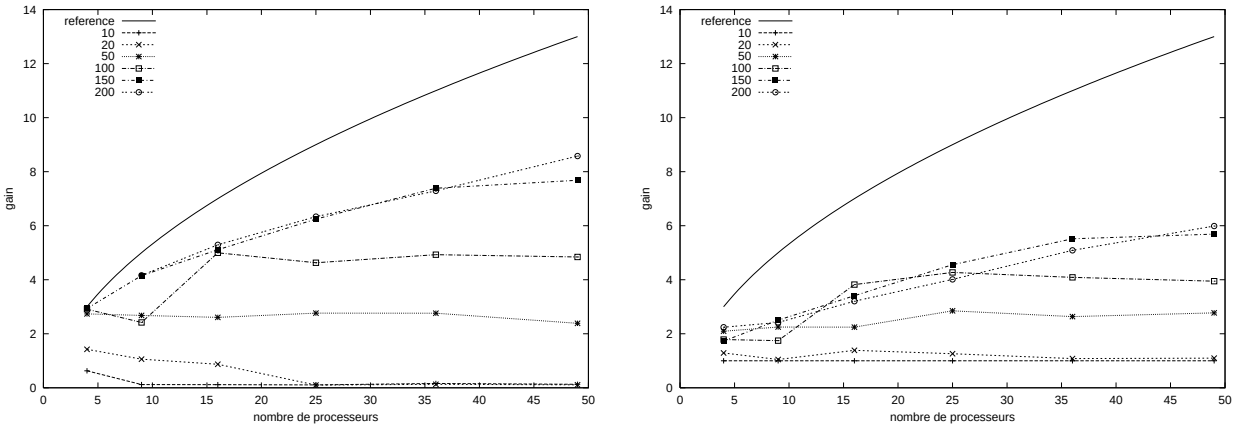


FIG. 4.15 – Gain du pipeline de huit octants du Sweep3D en fonction du nombre de processeurs sur la grappe de PC Icluster respectivement MK = 1 et MK = 10. Chaque courbe est tracée pour une taille de données fixée.

Lemme 10 *Soit $t_p^{1D}(n_0)$ le temps d'exécution d'un octant du Sweep3D sur une distribution monodimensionnelle en pipelinant la dimension K et les angles.*

$$t_p^{1D}(n_0) = \frac{\tau_c}{P} \times N_x N_y n + \frac{P-1}{P}\tau_c N_x N_y n_0 + 2\beta N_y n + (P-3)\beta N_y n_0 + 2\lambda\frac{n}{n_0} + (P-3)\lambda \quad (4.37)$$

Preuve. Comme au Lemme 9, en pipelinant la dimension K et les angles, le nombre de balayages est égal à $N_{sweep} = n/n_0$. Donc en utilisant l'Équation générale 4.18 et le nombre d'étapes de calculs et de communications et les délais donnés par les Équations 4.33, 4.35, 4.34, 4.36, le temps d'exécution est donné par

$$t_p^{1D}(n_0) = (P + \frac{n}{n_0} - 1) \times T^{cpu}(n_0) + (P + 2\frac{n}{n_0} - 3) \times T_{n_0}^{msg}$$

Remarque. Remarquons qu'évidemment $t_p^{1D}(n) = t_{np}^{1D}(n)$.

Théorème 4 *Soit $G^{1D}(n)$ le gain en performance résultant du pipeline d'un octant du Sweep3D sur une distribution monodimensionnelle de la dimension K et des angles. Alors*

$$G^{1D}(n) = \frac{\tau_c \times N_x N_y n + (P-1)\beta \times N_y n + (P-1)\lambda}{\frac{\tau_c}{P} \times N_x N_y n + \frac{P-1}{P}\tau_c N_x N_y n_0 + 2\beta N_y n + (P-3)\beta N_y n_0 + 2\lambda\frac{n}{n_0} + (P-3)\lambda} \quad (4.38)$$

Preuve. Comme nous l'avons vu au Théorème 1

$$G^{1D}(n) = \frac{t_{np}^{1D}(n)}{t_p^{1D}(n_0)}$$

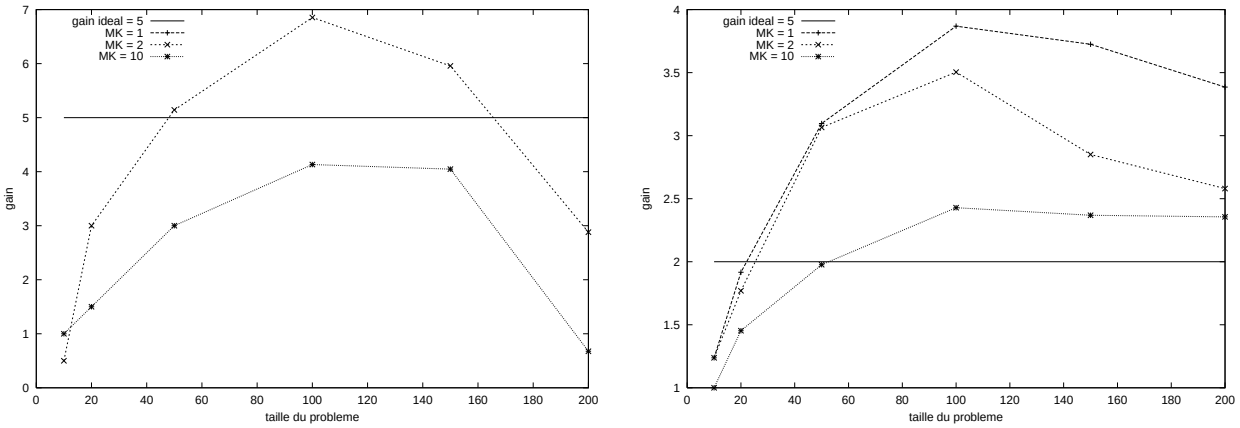


FIG. 4.16 – Gain des pipelines du Sweep3D en fonction de la taille des données sur l'IBM SP-2 du CINES sur 9 processeurs respectivement pour un puis huit octants.

Corollaire 9 *Si* $lim_{n\to\infty}\frac{T(n)}{T^{cpu}(n)} = 0$ *et* $lim_{n\to\infty}\frac{L(n)}{T^{cpu}(n)} = 0$, *alors* $G^{1D}(n) = P$.

Preuve. En utilisant l'Équation 4.38,

$$lim_{n\to\infty}G^{1D}(n) = lim_{n\to\infty}\frac{\tau_c \times N_x N_y n}{\frac{\tau_c}{P} \times N_x N_y n} = P$$

Distribution tridimensionnelle

La matrice de données est distribuée sur P_x, P_y et P_z processeurs respectivement sur les dimensions I, J et K.

Coûts des calculs et des communications. Le coût d'une étape de calcul pour une distribution tridimensionnelle est donné par

$$T^{cpu}(n) = \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y} \times n$$

Le coût d'une étape de communication respectivement dans les directions I, J et K pour une distribution tridimensionnelle est donné par les équations suivantes.

$$T^{msgX}(n) = \beta \times (\frac{N_y}{P_y} \times n) + \lambda$$

$$T^{msgY}(n) = \beta \times (\frac{N_x}{P_x} \times n) + \lambda$$

$$T^{msgZ}(n) = \beta \times (\frac{N_x}{P_x} \times \frac{N_y}{P_y}) + \lambda$$

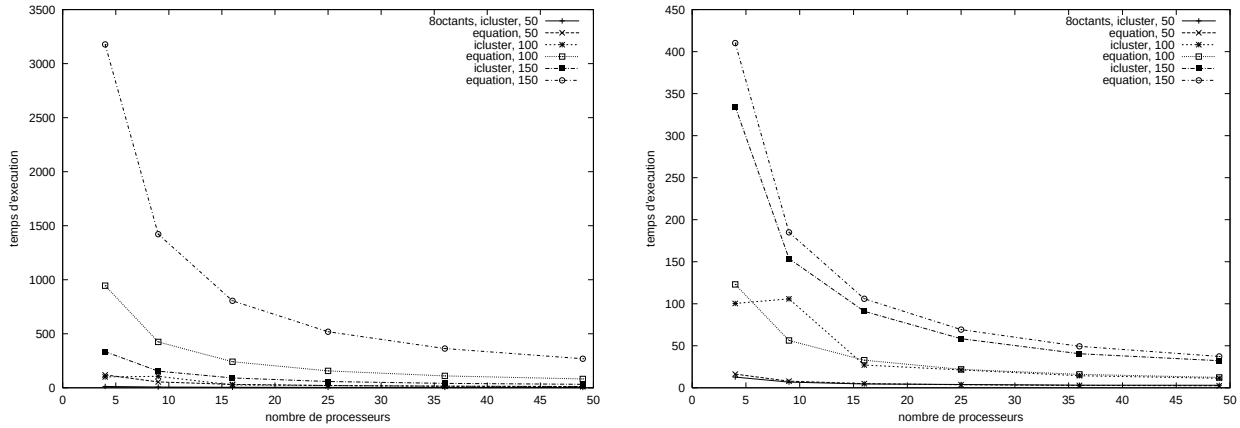


FIG. 4.17 – Comparaison du temps d'exécution des huit octants du Sweep3D sur la grappe de PC Icluster et du temps donné par l'Équation 4.31 avec les paramètres mesurés puis en modifiant le paramètre de calcul dans le cas de pipeline à grain fin.

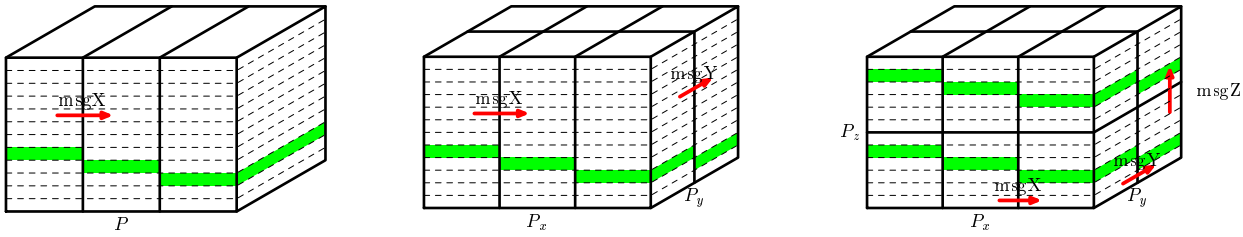


FIG. 4.18 – Distribution 1D, 2D et 3D pipelinée sur une dimension.

avec $n = n/P_z = \frac{N_z \times N_a}{P_z}$ sans pipeliner la dimension K et les angles et $n = n_0/P_z = \frac{N_z}{K_b}\frac{N_a}{A_b}\frac{1}{P_z}$ sinon.

Lemme 11 *Soit $t_{np}^{3D}(n)$ le temps d'exécution d'un octant du Sweep3D pour une distribution tridimensionnelle sans pipeliner la dimension K ni les angles. Alors*

$$\begin{aligned} t_{np}^{3D}(n) = (P_x + P_y + P_z - 2) \times \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{P_z} \\ + 2\beta(P_x - 1) \times \frac{N_y}{P_y} \times \frac{n}{P_z} + 2\beta(P_y - 1) \times \frac{N_x}{P_x} \times \frac{n}{P_z} \\ + 2\beta(P_z - 1) \times \frac{N_x}{P_x} \times \frac{N_y}{P_y} + 2(P_x + P_y + P_z - 3)\lambda \end{aligned}$$

Preuve. Pour une distribution tridimensionnelle, le nombre d'étapes de calculs est donné par

$$N_s^{comp} = P_x - 1 + P_y - 1 + P_z = P_x + P_y + P_z - 2 \tag{4.39}$$

Et le délai de répétition d'une étape de calcul pour une distribution tridimensionnelle est

$$d^{comp} = 1 \tag{4.40}$$

En outre, pour une distribution tridimensionnelle, le nombre d'étapes de communications est donné par

$$N_s^{comm} = 2(P_x - 1) + 2(P_y - 1) + 2(P_z - 1) \tag{4.41}$$

Et le délai de répétition d'une étape de communication pour une distribution tridimensionnelle est

$$d^{comm} = 6 \tag{4.42}$$

De plus, sans pipeliner, le nombre de balayages est égal à $N_{sweep} = 1$. Donc en utilisant l'Équation générale 4.18 et le nombre d'étapes de calculs et de communications et les délais donnés aux Équations 4.39, 4.41, 4.40, 4.42, le temps d'exécution est donné par

$$\begin{aligned} t_{np}^{3D}(n) = {} & (P_x + P_y + P_z - 2) \times T^{cpu}(n) \\ & + 2(P_x - 1) \times T^{msgX}(n) \\ & + 2(P_y - 1) \times T^{msgY}(n) \\ & + 2(P_z - 1) \times T^{msgZ}(n) \end{aligned}$$

Lemme 12 *Soit $t_p^{3D}(n_0)$ le temps d'exécution d'un octant du Sweep3D pour une distribution tridimensionnelle en pipelinant la dimension K et les angles.*

$$\begin{aligned} t_p^{3D}(n_0) = {} & \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{P_z} + (P_x + P_y + P_z - 3)\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n_0}{P_z} \\ & + 2\beta(\frac{N_x}{P_x} + \frac{N_y}{P_y})\frac{n}{P_z} + 2\beta\frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{n_0} + 2\beta((P_x - 2)\frac{N_x}{P_x} + (P_y - 2)\frac{N_y}{P_y})\frac{n_0}{P_z} \\ & + 2\beta(P_z - 2)\frac{N_x}{P_x}\frac{N_y}{P_y} + 6\lambda\frac{n}{n_0} + 2\lambda(P_x + P_y + P_z - 6) \end{aligned} \tag{4.43}$$

Preuve. Comme au Lemme 11, en pipelinant la dimension K et les angles, le nombre de balayages est égal à $N_{sweep} = n/n_0$. Donc en utilisant l'Équation générale 4.18 et le nombre d'étapes de calculs et de communications et les délais donnés aux Équations 4.39, 4.41, 4.40, 4.42, le temps d'exécution est donné par

$$\begin{aligned} t_p^{3D}(n_0) = {} & (P_x + P_y + P_z + \frac{n}{n_0} - 3) \times T^{cpu}(n_0) + (2(P_x - 1) + 2(\frac{n}{n_0} - 1)) \times T^{msgX}(n_0) \\ & + (2(P_y - 1) + 2(\frac{n}{n_0} - 1)) \times T^{msgY}(n_0) + (2(P_z - 1) + 2(\frac{n}{n_0} - 1)) \times T^{msgZ}(n_0) \end{aligned}$$

Remarque. Remarquons qu'évidemment $t_p^{3D}(n) = t_{np}^{3D}(n)$.

Théorème 5 *Soit $G^{3D}(n)$ le gain en performance résultant du pipeline de la dimension K et des angles pour un octant du Sweep3D pour une distribution tridimensionnelle. Alors*

$$G^{3D}(n) = \frac{N^{3D}(n)}{D^{3D}(n)} \tag{4.44}$$

avec

$$\begin{aligned} N^{3D}(n) = {} & (P_x + P_y + P_z - 2) \times \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{P_z} \\ & + 2\beta(P_x - 1) \times \frac{N_y}{P_y} \times \frac{n}{P_z} + 2\beta(P_y - 1) \times \frac{N_x}{P_x} \times \frac{n}{P_z} \\ & + 2\beta(P_z - 1) \times \frac{N_x}{P_x} \times \frac{N_y}{P_y} + 2(P_x + P_y + P_z - 3)\lambda \end{aligned} \tag{4.45}$$

et

$$\begin{aligned} D^{3D}(n) = {} & \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{P_z} + (P_x + P_y + P_z - 3)\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n_0}{P_z} \\ & + 2\beta(\frac{N_x}{P_x} + \frac{N_y}{P_y})\frac{n}{P_z} + 2\beta\frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{n_0} \\ & + 2\beta((P_x - 2)\frac{N_x}{P_x} + (P_y - 2)\frac{N_y}{P_y})\frac{n_0}{P_z} + 2\beta(P_z - 2)\frac{N_x}{P_x}\frac{N_y}{P_y} \\ & + 6\lambda\frac{n}{n_0} + 2\lambda(P_x + P_y + P_z - 6) \end{aligned} \tag{4.46}$$

Preuve. Comme nous l'avons vu au Théorème 1

$$G^{3D}(n) = \frac{t_{np}^{3D}(n)}{t_p^{3D}(n_0)}$$

Corollaire 10 *Si $lim_{n\to\infty}\frac{T(n)}{T^{cpu}(n)} = 0$ et $lim_{n\to\infty}\frac{L(n)}{T^{cpu}(n)} = 0$, alors $G^{3D}(n) = P_x + P_y + P_z - 2$.*

Preuve. En utilisant l'Équation 4.44,

$$lim_{n\to\infty}G^{3D}(n) = lim_{n\to\infty}\frac{(P_x + P_y + P_z - 2) \times \tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{P_z}}{\tau_c \times \frac{N_x}{P_x}\frac{N_y}{P_y}\frac{n}{P_z}} = P_x + P_y + P_z - 2$$

Corollaire 11 *Si $P_x = P_y = P_z = \sqrt[3]{P}$ alors*

$$lim_{n\to\infty}G^{3D}(n) = 3\sqrt[3]{P} - 2$$

Preuve. En utilisant le Corollaire 10.

Résultats expérimentaux

Gain sur un octant. La figure 4.19 montre la simulation d'exécutions sur PoPC sur une ligne de processeurs d'un octant du Sweep3D. Le gain idéal est égal au nombre de processeurs (voir les Corollaires 1 et 9). Nous voyons que, pour des grandes tailles de problème et pour une taille de bloc égale à un, les exécutions pourraient s'approcher du gain idéal.

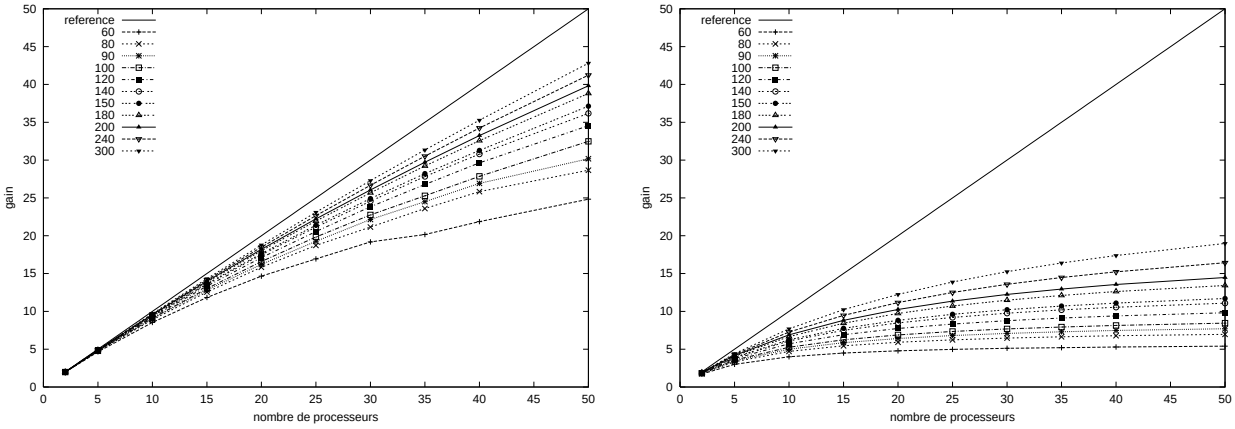


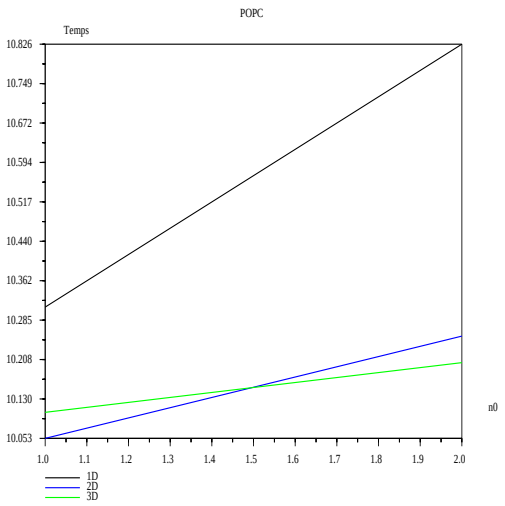
FIG. 4.19 – Comparaison des résultats obtenus avec le simulateur avec le coefficient de calcul mesuré et la courbe idéale sur une ligne de processeurs pour respectivement MK = 1 et MK = 10.

Détermination de la meilleure distribution. Dans cette partie, nous montrons que, grâce aux Équations 4.37, 4.26, et 4.43 donnant les temps d'exécutions d'un octant du Sweep3D pour les trois différentes distributions, il est possible de déterminer la meilleure distribution pour une machine et une application données.

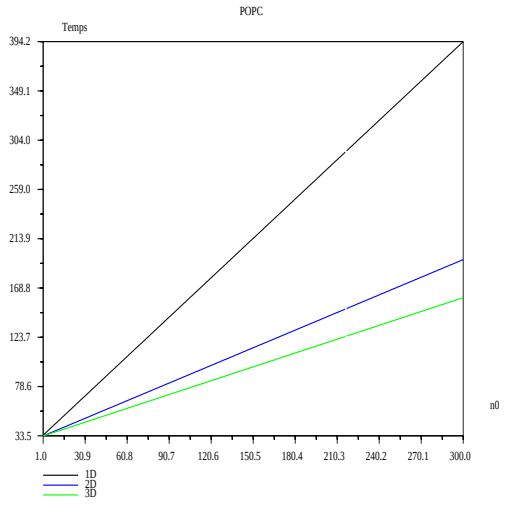
La figure 4.20 donne les courbes de ces trois équations pour les valeurs de paramètres obtenues sur la grappe de PC PoPC donnés dans la partie 4.2.2 page 82.

Ces courbes montrent que le meilleur temps d'exécution d'un octant du Sweep3D est obtenu pour une taille de blocs égale à un et une distribution bidimensionnelle. Les mesures ont été faites pour un nombre de processeurs égal au nombre de processeurs de la grappe et une taille de problème raisonnable qui pourra être exécutée sur cette machine. Il faut remarquer que pour toutes les tailles de blocs supérieures à un, la distribution tridimensionnelle donne un meilleur résultat.

La figure donne ces mêmes courbes pour les paramètres de la grappe de PC Icluster pour 100 processeurs et une taille de problème égale à 300. Ces courbes montrent que sur cette architecture, avec les paramètres que nous avons mesurés, la meilleure distribution serait en fait la distribution tridimensionnelle. Or, la modification du programme implantant une distribution bidimensionnelle est très importante. C'est pourquoi, nous n'avons pas de résultats expérimentaux venant appuyer ces courbes. En mettant ces équations au point au



(a) Tailles de blocs variant de 1 à 2.



(b) Tailles de blocs variant de 1 à 200.

FIG. 4.20 – Temps d'exécution en fonction de la taille de bloc pour les distributions 1D, 2D et 3D (P=12, N=200) sur PoPC.

préalable, il aurait été possible de directement programmer une distribution tridimensionnelle pour cette architecture.

Enfin, la figure 4.22 donne les courbes représentant les temps d'exécution pour les trois distributions différentes sur le SP-2 au CINES avec les paramètres que nous avons calculés. Elles montrent un comportement similaire à la grappe de PC Icluster avec le meilleur temps d'exécution obtenu avec une distribution tridimensionnelle pour toutes les tailles de blocs.

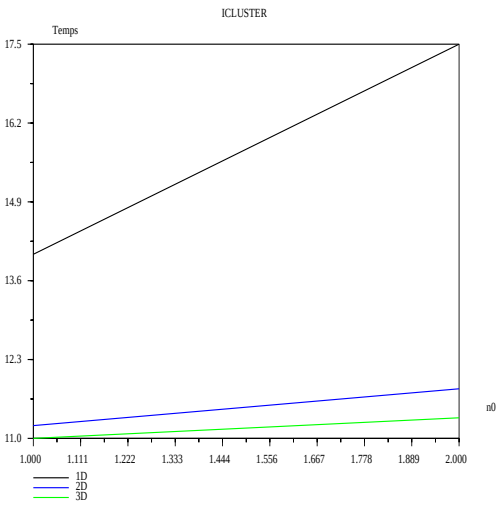
Nous avons cherché à visualiser le compromis entre latence de communication et taille de bloc de calcul. Nous avons donc tracé les courbes de temps d'exécution pour les trois différentes distributions avec des paramètres exagérés.

La figure 4.23 reprend les ordres de grandeur des paramètres de communication de la grappe de PC PoPC et donne les courbes pour des temps de calcul élémentaire très négligeable par rapport à la latence. Les paramètres de communication sont

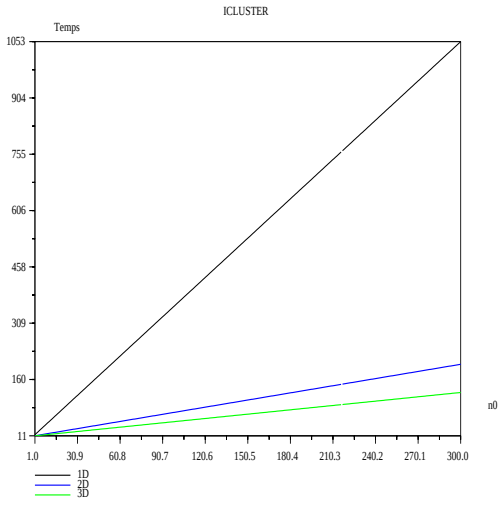
$$\begin{aligned} \lambda_L = &\quad 1.10^{-5} \\ \beta_T = &\quad 1.10^{-8} \end{aligned}$$

avec comme temps de calcul élémentaire $\tau_c = 1.10^{-7}$ pour les premières courbes et $\tau_c = 1.10^{-9}$ pour les secondes.

Dans ces deux cas, on peut observer que pour des petites tailles de blocs, les communications induisent un surcoût tel que la distribution tridimensionnelle donne des temps d'exécution très élevés pour un pipeline à grain fin. En revanche, pour un pipeline à gros grain, la distribution tridimensionnelle devient à nouveau la meilleure. Ce n'est que dans un cas extrême où le temps de calcul est dix mille fois inférieur au temps de latence de communication que la distribution bidimensionnelle reste meilleure. Il faut remarquer que la distribution monodimensionnelle n'est jamais la meilleure.



(a) Tailles de blocs variant de 1 à 2.



(b) Tailles de blocs variant de 1 à 300.

FIG. 4.21 – Temps d'exécution en fonction de la taille de bloc pour les distributions 1D, 2D et 3D (P=100, N=300) sur Icluster.

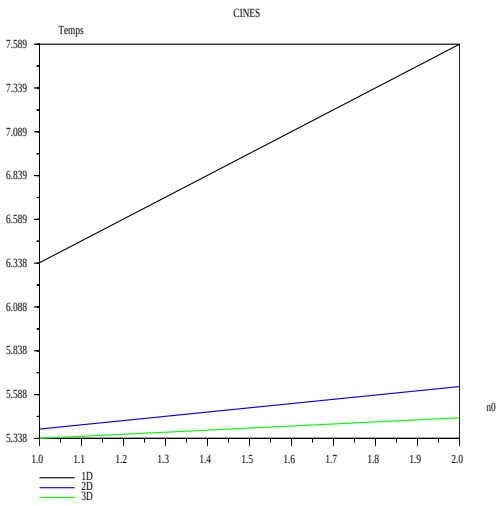
La figure 4.24 reprend quant à elle les ordres de grandeur du temps de calcul élémentaire du Sweep3D sur la PoPC et du temps de transfert sur le réseau de la grappe de PC PoPC et donne une latence de communication beaucoup plus élevée, cent mille fois supérieure $\lambda_L = 1$ et

$$\begin{aligned} \tau_c = &\quad 1.10^{-5} \\ \beta_T = &\quad 1.10^{-8} \end{aligned}$$

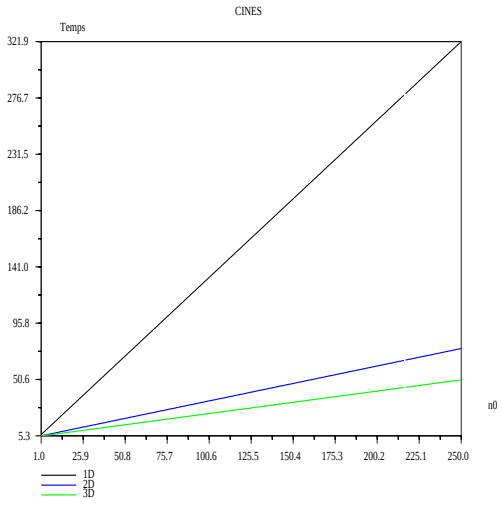
Ce cas est encore un cas très défavorable au pipeline puisque la latence est très supérieure au calcul. Néanmoins, la distribution monodimensionnelle ne donne toujours pas de bons résultats. Les distributions bi et tridimensionnelles seront celles qui donneront les plus petits temps d'exécution.

4.2.4 Conclusion

Cette étude du Sweep3D nous a tout d'abord permis de valider les équations de gain établies par Desprez et Zory sur une application réelle. Dans ce but, nous avons, à l'aide du modèle d'application par vagues donné par *Hoisie et al.*, modélisé tous les pipelines du Sweep3D, non seulement le pipeline des blocs de calcul en la dimension K et en angles mais aussi le pipeline des exécutions successives des différents octants. Puis, nous avons effectué des mesures de performances sur des grappes de PC dans le but de valider ces équations. Nous avons aussi programmé un simulateur d'application par vague pipelinée sur deux dimensions dans le but de s'abstraire des limitations du modèle notamment du temps de calcul élémentaire lié essentiellement au effet de cache.



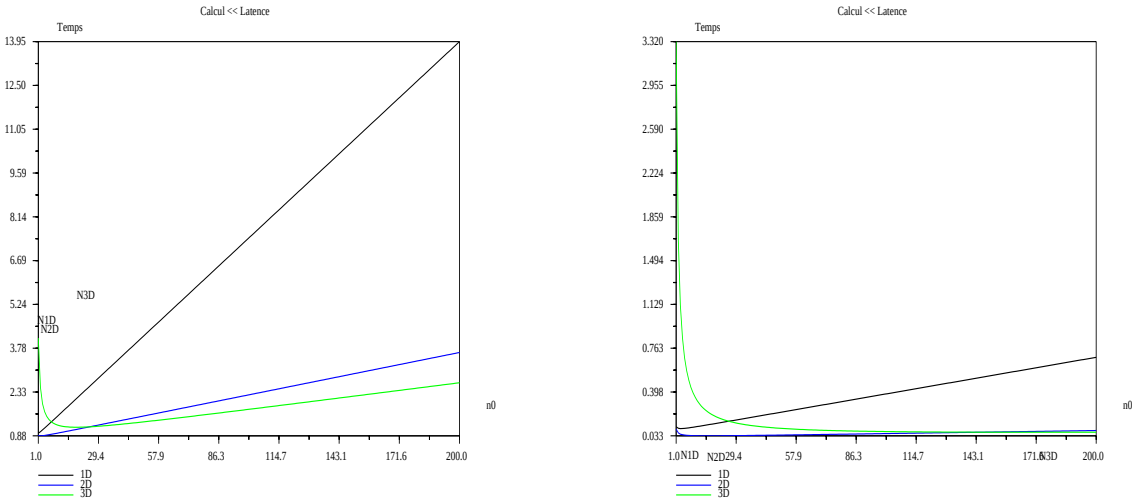
(a) Tailles de blocs variant de 1 à 2.



(b) Tailles de blocs variant de 1 à 300.

FIG. 4.22 – Temps d'exécution en fonction de la taille de bloc pour les distributions 1D, 2D et 3D (P=64, N=250) sur SP-2 au CINES.

Dans un deuxième temps, nous avons déterminé quelle était la meilleure distribution en fonction des paramètres de la machine cible et de l'application. En effet, plus la distribution a de dimensions, plus la programmation du pipeline est difficile. Ainsi, avant de s'investir dans ce codage, il est motivant de savoir que la distribution choisie donnera le meilleur temps d'exécution. Dans ce but, nous avons repris les équations pour les distributions mono et tridimensionnelles et tracé des courbes représentant ces temps d'exécution en fonction des paramètres des différentes machines disponibles. Sur les grappes de PC PoPC et Icluster ainsi que sur le SP-2 au CINES, le temps de calcul élémentaire est comparable au temps de latence de communication. Le temps de calcul d'un bloc d'éléments de la grille, même dans le cas d'un pipeline à grain fin est très grand comparé au temps de latence. C'est pourquoi la distribution tridimensionnelle est celle qui donne les temps d'exécution les plus petits.



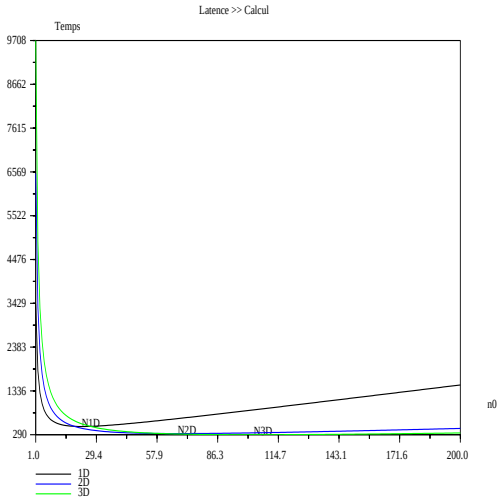
(a) $\tau_c = 1.10^{-7}$. (b) $\tau_c = 1.10^{-9}$.

FIG. 4.23 – Temps d'exécution en fonction de la taille de bloc pour les distributions 1D, 2D et 3D (P=64, N=800) avec la latence de communication de PoPC et un coefficient de calcul très petit.

4.3 Synthèse

Dans ce chapitre, nous avons tout d'abord clairement montré à l'aide d'équations mais aussi d'expérimentations qu'il est possible d'obtenir un gain significatif en pipelinant une application. Ces gains ont été évalués en s'appuyant sur les équations développées par Desprez et Zory. Notre travail a consisté à en démontrer la validité sur une application réelle. Ces expérimentations nous ont en fait conduit à écrire, à partir du modèle proposé par *Hoisie et al.*, d'autres équations qui au bout du compte viennent encore confirmer les estimations supérieures du gain d'un pipeline obtenues par Zory. Nous avons donc modélisé l'ensemble des trois différents pipelines de l'application Sweep3D, le pipeline sur la troisième dimension, le pipeline en angles et celui en octants.

Dans un deuxième temps, nous avons cherché à déterminer pour une application et une machine cible, la distribution donnant le plus petit temps d'exécution. En effet, jusqu'ici, le gain avait été envisagé sur trois distributions linéaires possibles : mono, bi- et tridimensionnelles. Il est maintenant intéressant de connaître quelle est celle qui donnera le meilleur temps d'exécution. Nous avons établi des équations pour les trois distributions et nous les avons comparées dans différents cas possibles. Nous avons vu que, dans les cas favorables pour le pipeline, lorsque le temps de latence est très petit par rapport au temps de calcul d'un bloc d'éléments, le pipeline maximum donne les meilleurs résultats et donc l'on choisira une distribution tridimensionnelle. En revanche, dans les cas moins favorables, une distribution bidimensionnelle donne de bons résultats. La distribution monodimensionnelle, quant à elle, n'extrait pas suffisamment de parallélisme pour pouvoir rivaliser avec les autres distributions.



(a) $\lambda_L = 1$.

FIG. 4.24 – Temps d'exécution en fonction de la taille de bloc pour les distributions 1D, 2D et 3D (P=64, N=800) avec le temps de calcul de PoPC et une latence de communication très grande.

Chapitre 5

Rendu volumique

5.1 Introduction

Dans le cadre de l'imagerie médicale, la visualisation 3D permet à un praticien, à partir d'un ensemble d'images (coupes) acquises par différentes modalités (scanner, résonance magnétique [56]), d'avoir une représentation en 3D de l'ensemble des informations acquises en 2D. La visualisation 3D peut se décomposer en deux types de méthodes [59] : le *rendu surfacique* et le *rendu volumique* ou encore rendu direct. Ce dernier opère sur l'ensemble des coupes vues comme un volume en trois dimensions composé de volumes élémentaires, les « voxels ». Alors que les algorithmes de rendu surfacique atteignent déjà des performances temps réel (10-30 images par seconde) grâce à des accélérateurs graphiques matériels, la lenteur d'un algorithme de rendu volumique est inacceptable dans la pratique. Celui-ci reste cependant intéressant parce qu'il permet d'obtenir des images d'une qualité supérieure du point de vue de l'aide au diagnostic.

Le rendu volumique met en œuvre un important volume de calculs parce qu'il travaille sur l'ensemble des données acquises. D'autre part, dans le cadre d'une utilisation interactive d'une application de rendu volumique, chaque changement de point de vue implique un recalcul complet à partir des données initiales de l'image. Afin d'accélérer ces calculs sans toutefois dégrader la qualité de l'image, des techniques séquentielles d'optimisation logicielle telles que l'exploitation de structures creuses ou encore la terminaison anticipée de rayon ont été développées. Cependant, la quantité de données manipulée par ce type de rendu reste importante. A l'heure actuelle, il permet de visualiser des données d'une taille 256^3. En revanche, les données de taille $512 \times 512 \times 200$, courantes dans le milieu médical, ne peuvent être visualisées par du rendu direct en séquentiel parce que le volume de données et la quantité de calculs sont trop importantes.

Nous avons choisi d'étudier l'algorithme de rendu volumique en *shear-warp*, développé par Lacroute [41]. Il est beaucoup plus rapide en séquentiel que tous les autres algorithmes de rendu volumique sans dégradation de la qualité de l'image. À partir de cet algorithme de rendu efficace, nous espérons arriver à une utilisation interactive (10 à 15 images par seconde) de cet algorithme, permettant à un utilisateur de changer rapidement de point de vue, en le parallélisant sur une grappe de PC.

Lacroute a présenté dans sa thèse le principe de l'algorithme du shear-warp. Cet algo-

rithme s'appuie sur la factorisation de la transformation liée au point de vue lui permettant de parcourir simplement le volume de données tout en permettant d'utiliser des optimisations séquentielles. Il se décompose en deux étapes : la première mettant en œuvre la transformation du volume en trois dimensions en une image intermédiaire en deux dimensions (*shear*) et la seconde passant de cette image intermédiaire à l'image finale (*warp*). L'essentiel des calculs se trouve dans l'étape de shear. C'est pourquoi nous allons nous focaliser sur cette étape.

Dans une première partie, nous nous sommes intéressés aux versions parallèles existantes du shear-warp. Ces travaux de parallélisation mettent en avant le rôle prépondérant de la machine cible. Les résultats obtenus en mémoire distribuée sont de 10 images par seconde sur 128 processeurs sur une TMC CM5 avec une accélération de 35 sur 128 processeurs. Ces résultats sont atteints grâce à des contraintes fortes sur l'application : les images ne peuvent être générées que de un degré en un degré d'angle de rotation de point de vue. Il est important de libérer l'algorithme d'une telle contrainte pour en conserver l'interactivité. La parallélisation du shear-warp devra donc pouvoir prendre en compte un changement de point de vue quelconque.

Ces hypothèses d'utilisation (changement de point de vue quelconque) et d'implantation (mémoire distribuée et encombrement mémoire minimal) nous ont amenés à proposer une nouvelle politique d'équilibrage de charge. Dans une deuxième partie, nous chercherons donc à implanter une nouvelle version parallèle du shear-warp en recouvrant les communications par des calculs. En outre, chaque nouveau point de vue nécessite un nouveau calcul de l'image intermédiaire à partir du volume de données initial. Le volume est alors, dans la parallélisation choisie, redistribué à chaque changement de point de vue. Ainsi, quelle que soit la distribution initiale des données, le surcoût dû aux communications ne pourra pas être évité. Ces travaux pourront permettre de définir un mécanisme générique de recouvrement des communications par les calculs appliqué à des algorithmes d'imagerie médicale.

5.2 Rendu volumique en shear-warp

Dans cette partie, nous allons mettre en place le contexte de l'étude en expliquant le principe séquentiel de l'algorithme du shear-warp et en exposant les différents travaux de parallélisation existants. Il est montré que l'implantation en mémoire distribuée est peu extensible. Jiang et al. [34] concluent que cette mauvaise extensibilité (l'accélération est égale à 35 sur 128 processeurs) est liée à l'architecture de la machine à mémoire distribuée utilisée (une Thinking Machines Corporation CM-5). Nous pensons que les performances en termes d'extensibilité et de temps d'exécution peuvent être améliorés sur une machine à mémoire distribuée par le recouvrement des communications par le calcul. Ces résultats peuvent encore être meilleurs en utilisant des techniques de macro-pipeline (voir la partie suivante). Ce chapitre exposera donc en dernière partie les trois parallélisations existantes du shear-warp afin d'en retenir les points positifs.

5.2.1 Les algorithmes de rendu volumique

Le rendu volumique direct met en jeu un ensemble de données brutes (des valeurs scalaires), une étape de classification permettant d'assigner à une valeur scalaire une opacité, soit grâce à une fonction de transfert, soit par des méthodes de segmentation en partitionnant le volume en des structures spécifiques (os, peau, muscle, veine, ...), l'application d'un algorithme d'ombrage dont le rôle est de simuler la lumière et de déterminer la couleur de chaque voxel et enfin l'extraction d'une vue.

Pour obtenir un rendu volumique, on utilise principalement deux techniques : une technique d'ordre image ou une technique d'ordre objet. Il existe, en outre, d'autres techniques moins courantes associées au rendu volumique explicitées dans un état de l'art sur la visualisation 3D de Kaufman en 1996 [38]. Les algorithmes d'ordre objet consistent à projeter les échantillons de données volumiques (les voxels) sur le plan de l'image (les pixels). A l'inverse, les techniques d'ordre image cherchent pour chaque pixel quels sont les échantillons de données qui contribuent à le déterminer.

L'algorithme le plus célèbre d'ordre image est le lancer de rayons. Ce dernier consiste à émettre des rayons à partir du point de vue jusqu'au plan de l'image en traversant le volume. Lorsqu'un rayon traverse le volume, le volume est échantillonné selon un certain nombre d'intervalles, en général, un ou deux par voxels. La valeur à chaque point d'échantillonnage est obtenue par une interpolation trilinéaire des valeurs voisines de ce point. Ces valeurs sont translatées vers des couleurs et des opacités grâce, par exemple, à une fonction de transfert. Les contributions de ces échantillons sont accumulées jusqu'à ce que le rayon ait quitté le volume. La couleur finale est alors calculée et placée sur le pixel que le rayon atteint.

Une méthode classique des algorithmes d'ordre objet est la méthode du *Splatting*[1] [72] ([35] analyse cet algorithme sur le Cray T3D). Cette méthode consiste à projeter voxel par voxel le volume sur le plan de l'image en le traversant du plan de l'image à l'observateur. En projection parallèle, l'empreinte d'un voxel sur ce plan de vue est constante pour tous les voxels. Comme un voxel ne se projettera pas exactement sur un pixel, un filtre est utilisé pour calculer une contribution en couleur et en transparence aux pixels voisins (d'où le terme

[1] *to splat* : éclabousser.

de *splatting*).

Les algorithmes d'ordre image et ceux d'ordre objet se distinguent par l'ordre dans lequel on parcourt le volume de données et l'image. La boucle externe des algorithmes d'ordre image itèrent sur les pixels dans l'image. En revanche, les algorithmes d'ordre objet itèrent sur les voxels dans l'objet à rendre : ils parcourent alors le volume dans l'ordre de stockage. Chacun de ces types d'algorithmes présente des inconvénients non négligeables. Le parcours de l'objet dans les algorithmes d'ordre image et le calcul du filtre dans les algorithmes d'ordre objet ralentissent considérablement le rendu. C'est pourquoi, on a cherché des techniques permettant d'accélérer encore le calcul s'appuyant sur leurs caractéristiques respectives : l'exploitation de la cohérence de données et la terminaison anticipée de rayon. Dans une coupe de voxels d'origine médicale de type scanner ou résonance magnétique, deux données proches ont souvent la même valeur et la plupart des voxels sont transparents : la coupe possède une certaine cohérence des données. Cette cohérence peut être exploitée si l'on parcourt le volume dans l'ordre de stockage. De plus, la terminaison anticipée des rayons consistant à arrêter le calcul de l'opacité accumulée le long du rayon lorsque celle-ci atteint une valeur proche de l'opacité totale peut être exploitée lorsque l'on calcule l'image d'avant en arrière (du plan image vers l'observateur).

Il existe des algorithmes mixtes : il suffit que la boucle externe parcoure en même temps l'image et l'objet pour que l'algorithme soit à la fois d'ordre image et d'ordre objet. L'algorithme du shear-warp fait partie de ces algorithmes mixtes.

5.2.2 Principe séquentiel du shear-warp

Les algorithmes de rendu volumique en shear-warp, développés par Lacroute [41], sont basés sur la factorisation de la transformation liée au point de vue pour simplifier la projection du volume vers l'image.

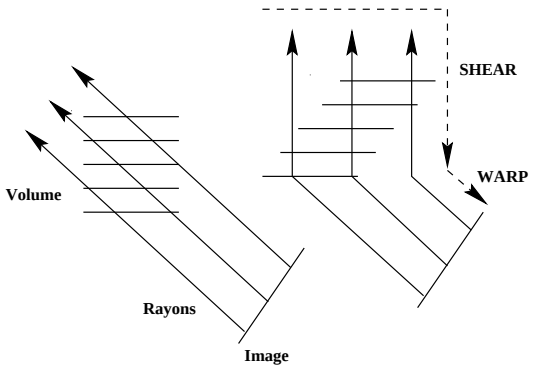


FIG. 5.1 – Factorisation de la transformation liée au point de vue.

La factorisation shear-warp consiste à considérer un système de coordonnées intermédiaires. Dans cet espace objet transformé, les rayons liés au point de vue sont parallèles au troisième axe de coordonnées (voir figure 5.1). Un algorithme basé sur cette transformation dans l'espace objet intermédiaire a deux étapes :

1. une première étape de composition appelée *shear*[2], composée de la transformation des données du volume dans l'espace objet transformé et de l'accumulation de ces données sur le plan image d'avant en arrière (du plan de l'image vers l'observateur). Cette étape crée une image intermédiaire distordue correspondant à l'espace objet transformé ;
2. une deuxième étape appelée *warp*[3], transformant l'image intermédiaire en image finale.

Passer par une image intermédiaire permet aux lignes de balayage des données du volume d'être alignées avec les lignes de balayage de l'image intermédiaire. On balaye alors l'objet et l'image simultanément dans leur ordre de stockage. C'est pourquoi cet algorithme combine les avantages des algorithmes d'ordre objet et des algorithmes d'ordre image.

Les conditions d'utilisation des techniques d'optimisation séquentielle telles que l'exploitation de la cohérence de données et la terminaison anticipée de rayon sont réunies grâce à ce parcours simultané des deux structures de données (objet et image).

Analyse de l'application

L'implantation de l'algorithme du shear-warp peut être décomposé en trois unités fonctionnelles : calcul d'une table de translation (*lookup table*) qui contient des informations nécessaires à l'ombrage, l'étape de composition et l'étape de *warp* de l'image intermédiaire :

```
Procedure Rendu()
   Pour chaque point de vue faire
      Calcul de la table de translation d'ombrage
      Pour chaque coupe de voxels d'avant en arrière faire
         Composition(coupe, tmp_image)
      FinPour
      image = Warp(tmp_image)
      Affiche(image)
   FinPour
Fin
```

Le calcul de la table de translation d'ombrage est un précalcul d'opacités. Le calcul des opacités est dépendant du point de vue. Il dépend de l'intervalle d'échantillonnage. Le volume de données contient des valeurs d'opacité associées à un intervalle Δx_0. On calcule à chaque nouveau point de vue la correction nécessaire à chacune de ces valeurs pour le nouvel intervalle Δx, d'où le calcul d'une table de translation d'ombrage.

La composition décrite ici est la combinaison des étapes de transformation du volume et d'accumulation. On doit échantillonner chaque coupe de voxels lorsqu'elle est translatée dans l'espace objet déformé. Le filtre utilisé est un filtre bilinéaire (à la différence de la plupart des algorithmes de rendu volumique qui utilisent des filtres trilinéaires) utilisant deux lignes de balayage d'une coupe pour produire une ligne résultat dans l'image intermédiaire.

Une manière simple d'accumuler entre elles les couleurs et opacités obtenues le long du rayon consiste à les additionner. Dans le shear-warp, elle se fait selon l'opérateur *over* : soient

[2] *shear* : élongation.
[3] *warp* : distorsion.

C_i la couleur et α_i l'opacité de l'échantillon i et c_i tel que $c_i = C_i\alpha_i$, la valeur finale composée est égale à $c_0 + c_1(1-\alpha_0) + c_2(1-\alpha_0)(1-\alpha_1) + \cdots + c_{n-1}(1-\alpha_0)\ldots(1-\alpha_{n-2})$ ou encore c_0 *over* c_1 *over* c_2 *over* ... *over* c_{n-1}.

Amin et al. [8] font un modèle analytique de ces trois étapes :

- **calcul de la table de translation d'ombrage** : les entrées de la table peuvent être calculées entièrement en parallèle. Les processeurs ont besoin de l'intégralité de la table pour la composition (heureusement celle-ci n'est pas très grande, environ 8K) ;
- **composition** : la phase de composition dans l'espace objet déformé peut s'effectuer indépendamment sur chacun des processeurs. Si n est une des dimensions du volume de données, il y a un total de $O(n^3)$ voxels dans le volume. Une grande partie d'entre eux est transparente et la terminaison anticipée des rayons fait qu'on ne les parcoure pas tous. Lacroute [41] a estimé statistiquement que la complexité en calcul est en $O(n^2)$ en séquentiel ;
- **warp** : dans cette phase, pour chaque pixel de l'image finale, quatre pixels voisins de l'image intermédiaire sont localisés et interpolés. A chaque pixel est associé un même calcul et il y a n^2 pixels donc la complexité en calcul est en $O(n^2)$.

Le *warp* et le calcul de la table de translation d'ombrage représentent pour des volumes de données de taille moyenne moins de 20% du temps total de rendu ([41] constate que plus le volume de données augmente, plus le temps de *warp* et de calcul de la table de translation d'ombrage deviennent négligeables devant la composition). L'essentiel du temps de calcul est donc consommé par la composition.

Description des structures de données

Le shear-warp encode l'objet avec une structure creuse en *run length encoding* (RLE). Ce codage est utilisé pour encoder des objets présentant une cohérence de données notamment des images pour lesquelles on obtient un taux de compression intéressant. Ceci est le cas pour les images issues de modalités médicales telles que le scanner ou la résonance magnétique que nous considérons. L'objet dans l'algorithme du shear-warp est un empilement de coupes 2D. On définit un axe principal tel que les coupes choisies soient perpendiculaires à cet axe. Chaque coupe est un tableau de voxels déterminés par une valeur de transparence (donc le volume est obtenu après l'étape de classification). On définit un seuil de tranparence pour lequel un voxel est considéré transparent ou non. On cherche, par l'intermédiaire du RLE, à ne pas stocker les voxels transparents (dont on néglige l'information). Le RLE est alors constitué de trois tableaux. Prenons par exemple, soit le tableau de données [123, 13, 33, 2, 1, 27]. Si on définit un seuil égal à 10, alors le premier tableau contenant le nombre de voxels transparents puis le nombre de voxels non transparents consécutifs sera égal à [3, 2, 1] et le deuxième tableau contenant les valeurs des données pour les voxels non transparents sera égal à [123, 13, 33, 27]. Le dernier tableau permet d'avoir un accès aléatoire (par lignes ou par coupe) à ces deux tableaux.

Analyse des performances

Les performances du shear-warp sont liées à la méthode de composition le long d'un rayon. La factorisation de la transformation liée au point de vue permet de parcourir en

même temps le volume de données et l'image composée (l'image intermédiaire). Elle élimine donc les inconvénients de parcours de l'objet des algorithmes d'ordre image. Le pourcentage de voxels transparents est, en général, pour des images issues d'un scanner ou de résonance magnétique de 70 à 95% ([56] décrit les principales modalités d'acquisition d'images médicales et le type de résultat obtenu). Les structures de données en RLE permettent d'optimiser l'algorithme en évitant les voxels transparents. On réduit donc le volume de calculs et la taille des données à parcourir. D'autre part, l'image intermédiaire est elle aussi encodée en RLE pour éviter les pixels opaques. Ainsi, lorsque l'on veut accumuler une valeur dans l'image intermédiaire, la structure RLE permet de ne pas effectuer le calcul en testant si le pixel a atteint le seuil d'opacité supérieure. Comme le volume et l'image sont parcourus simultanément, aucun calcul n'est effectué lorsqu'un pixel est déclaré opaque ; c'est l'implantation de la terminaison anticipée des rayons. Ces deux techniques d'optimisations combinées, l'algorithme du shear-warp est accéléré d'au moins un facteur 10 par rapport à l'algorithme brut [41]. C'est pourquoi ces optimisations sont considérées comme essentielles.

Grâce à toutes ces optimisations, le shear-warp rend une image de taille $256 \times 256 \times 225$ (taille moyenne) en moins d'une seconde sur une station de travail R4400 150MHz [41]. Un exemple de rendu est donné à la figure 5.2.

FIG. 5.2 – Exemple de rendu.

Les limitations. L'inconvénient de l'algorithme du shear-warp est qu'il contient deux passes de rééchantillonnage ce qui peut éventuellement causer un effet de flou. D'autre part, le filtre de reconstruction qui rééchantillonne le volume de données est un filtre 2D (correspondant à l'empilement des coupes décrit par la structure de données) et non pas un filtre

3D. Il peut en résulter un effet d'*aliasing*[4]. La troisième limitation de l'algorithme est que la résolution de l'image est fixée par le taux d'échantillonnage du volume en entrée (un volume de taille 256^3 donnera une image de résolution maximum 256^2).

5.2.3 Parallélisation du shear-warp

Malgré les bonnes performances séquentielles de cet algorithme relativement aux autres algorithmes de rendu volumique, le volume de calculs est trop important pour que cette technique de visualisation directe soit compatible avec une utilisation en temps réel. De plus, tout changement de point de vue de l'utilisateur implique un recalcul complet de l'image à partir du volume de données. C'est pour cela que plusieurs travaux de parallélisation du shear-warp existent afin de réduire le temps de calcul et d'augmenter le volume de données.

Ces travaux mettent en avant le rôle prépondérant de la machine cible et exploitent le parallélisme de données inhérent à cette application. L'algorithme du shear-warp a été parallélisé sur une machine à mémoire partagée par Lacroute [42] et ensuite par Jiang et al. [34] en utilisant un autre partage des tâches et une analyse des performances approfondie de l'algorithme sur la machine. La communication et la redistribution des données sont alors considérées comme négligeables du fait de l'utilisation de la mémoire partagée.

La parallélisation de l'algorithme du shear-warp sur une machine à mémoire distribuée a été effectuée par Amin et al. [8]. Cette parallélisation insiste sur la communication et la redistribution des données à chaque changement de point de vue. Nous pensons que la mauvaise extensibilité (accélération égale à 35 sur 128 processeurs) et les mauvais temps d'exécution (au plus 10 images par seconde dans des conditions d'utilisation de l'application très restreintes) de cet algorithme peuvent être améliorés. La partie suivante est alors consacrée à l'étude des parallélisations existantes du shear-warp aussi bien en mémoire partagée qu'en mémoire distribuée.

Parallélisations existantes

Partitionnement de tâches. Le partitionnement de tâches consiste à allouer à chaque processeur un travail que l'on ne cherchera pas à paralléliser.

Les deux types de partitionnement de tâches que l'on trouve dans les algorithmes de rendu volumique sont : le partitionnement de l'image ou le partitionnement de l'objet. Dans un partitionnement de l'objet, chaque processeur a une partie de l'objet à échantillonner et à composer. Chaque processeur contribuera donc potentiellement à la totalité de l'image à composer. Les pixels ainsi calculés sont des résultats partiels et doivent alors être composés entre eux pour former l'image finale. Dans un partitionnement de l'image, on assigne à chaque processeur une partie de l'image à calculer. Chaque pixel de l'image est calculé par un unique processeur mais le volume de données doit être déplacé sur les différents processeurs à chaque changement de point de vue. Quel que soit le type de machine visé, si l'on veut conserver la terminaison anticipée des rayons, on ne peut pas partager les tâches selon le volume de données à moins de faire une exécution spéculative. En effet, si l'on partage le calcul du rayon, chaque processeur calculera sa partie de rayon sans savoir si celui-ci est déjà terminé

[4] *aliasing* : marches d'escalier

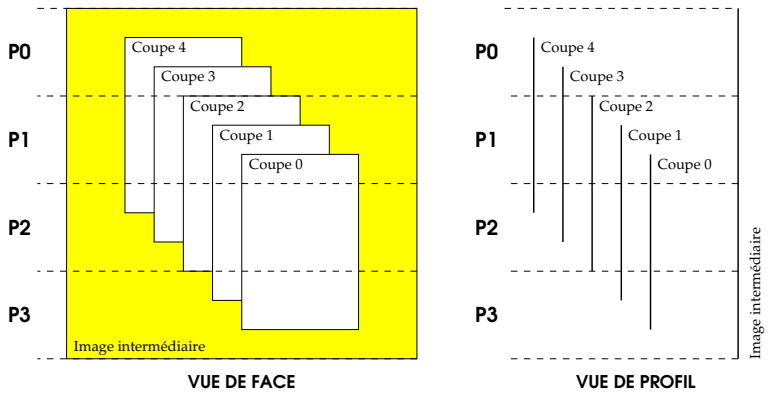


FIG. 5.3 – Distribution des données par rapport au découpage de l'image intermédiaire.

par un autre processeur. La partie précédente a montré qu'un des points clés de l'algorithme en shear-warp développé par Lacroute est l'utilisation des deux techniques d'accélération séquentielle : la terminaison anticipée de rayon et l'exploitation de la cohérence de données. Par conséquent, un partage de tâches qui ne permettrait pas la terminaison anticipée de rayon est exclu.

On choisit donc, quel que soit le type de machine visé un partitionnement selon l'image intermédiaire. Il faut remarquer que ce partitionnement est plus coûteux en communications sur une machine à mémoire distribuée que le partitionnement de l'objet. Il permet, en outre, sur machine à mémoire partagée d'éviter des synchronisations au niveau de l'image. L'image étant partitionnée, la granularité des tâches peut être le pixel, un bloc de lignes de pixels ou encore un pavé rectangulaire de pixels. La granularité est dans la pratique imposée par le RLE. Il est difficile de trouver avec cette structure de données les voxels associés à une partie de l'image. Dans un partitionnement basé sur les lignes, on précalcule des pointeurs associés à chaque début de lignes de voxels pour éviter un surcoût de décodage. En outre, la granularité en lignes permet d'améliorer la localité spatiale. Si le volume de données est fortement compressé alors plusieurs lignes de données peuvent tenir dans une ligne de cache. Ces arguments favorisent donc un découpage des tâches en blocs de lignes.

Distribution des données. Le but de la distribution des données est d'introduire à terme dans l'application des données de taille au moins 512^3 (taille courante dans la pratique). Pour une taille moyenne (256^3), les données manipulées, en couleurs, sont déjà de l'ordre de 30 Moctets. Donc des données en couleur de taille 512^3 atteindront 240 Moctets. C'est pourquoi, nous avons cherché à distribuer les données avec le moins de réplication possible.

Une telle distribution des données induite par ce partage de l'image intermédiaire n'est pas triviale puisqu'elle est dépendante du point de vue et morcelle chaque coupe de données sur plusieurs processeurs. Sur la figure 5.3, l'image intermédiaire est partitionnée de manière simple sur quatre processeurs. Le volume déformé correspondant, comprenant cinq coupes vues de profil, est alors partitionné comme le montre la figure, le processeur 3 possède quelques lignes de la première coupe et de la deuxième, le processeur 2 possède des lignes appartenant à toutes les coupes, etc...

Equilibrage de charges. Un équilibrage de charge simple consistant à entrelacer des blocs de lignes de l'image intermédiaire n'est pas suffisant pour réaliser un bon équilibrage de charges [42]. En revanche, dans le cadre d'une application devant calculer plusieurs points de vue, c'est une distribution initiale acceptable. Pour les machines à mémoire partagée, l'équilibrage de charges consiste à faire du vol de tâches (*task stealing*) [34, 42]. Lorsqu'un processus n'a plus de travail, il s'adresse à un processus maître qui lui assigne une nouvelle file d'attente. Cet équilibrage de charges n'est pas adapté à une machine à mémoire distribuée puisque le travail d'un processeur, partagé selon l'image intermédiaire, est directement lié à la distribution des données et nécessiterait donc une redistribution des données en cours de rendu. En mémoire distribuée, la politique d'équilibrage de charges proposée par Amin et al. [8] se base sur une utilisation particulière de l'application qui consiste à considérer que l'on cherche à exécuter une animation avec des changements de point de vue consécutifs petits (1 degré d'angle de rotation). L'équilibrage de charges est donc adaptatif et se base sur le rendu précédent pour équilibrer le rendu suivant. Pour une utilisation plus générale de l'application, notamment si l'on veut changer de point de vue de manière arbitraire ou choisir un angle de rotation simplement plus grand, cet équilibrage de charges n'est plus adapté.

Les politiques d'équilibrage de charges existantes sont donc directement liées à la machine cible et au comportement choisi de l'application.

Etude des communications

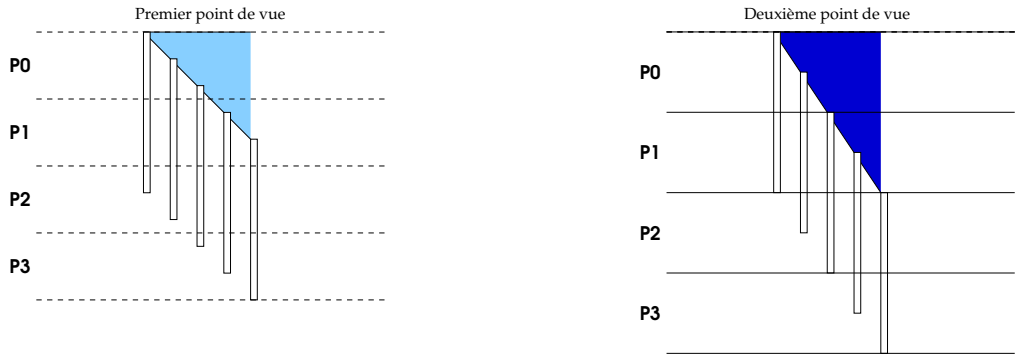


FIG. 5.4 – Déformations du volume selon deux points de vue.

Pour conserver l'optimisation liée à la terminaison anticipée de rayon, Amin et al. [8] proposent, bien qu'en mémoire distribuée, un partage des tâches selon l'image intermédiaire. Ce schéma de partitionnement induit que le volume doit être tout d'abord, déformé, puis découpé orthogonalement aux coupes du volume (voir figure 5.3). Chaque processeur peut maintenant lancer ses rayons à travers la partie du volume qui lui est assignée. Les images intermédiaires résultantes sont disjointes et peuvent être indépendamment déformées en images finales.

Les communications engendrées par cette distribution des données ont lieu lorsque le volume est déformé (à chaque changement de point de vue). Cette communication est illustrée

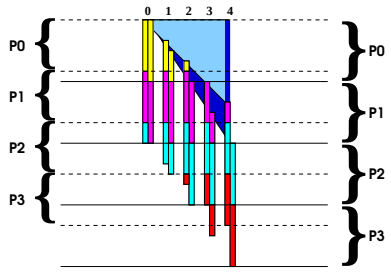


FIG. 5.5 – Comparaison des deux points de vue.

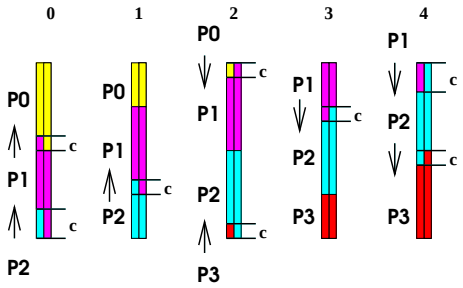


FIG. 5.6 – Communications engendrées.

aux figures 5.4, 5.5 et 5.6. La figure 5.4 montre la déformation du volume pour un premier puis pour un deuxième point de vue. Nous cherchons alors les communications engendrées par ce changement de point de vue. La comparaison de ces points de vue est effectuée à la figure 5.5. Nous avons colorié d'une même couleur les données devant appartenir à un même processeur d'un point de vue à l'autre. En faisant la différence entre ce que l'on a et ce que l'on voudrait obtenir (figure 5.6), on déduit les communications engendrées entre les processeurs pour ce changement de point de vue avec ce partitionnement de tâches. Sur la figure 5.6, **c** indique une communication entre les processeurs indiqués par la flèche.

Cette communication est une multidistribution de morceaux de structures creuses (chaque processeur envoie des données différentes à tous les autres processeurs). Une autre communication intervient lors de l'assemblage de l'image pour effectuer un affichage, par exemple. Cette communication est un rassemblement (*gather*) contenant des parties d'image finale (tous les processeurs envoient leurs données sur un seul processeur de rendu).

Pseudo-code parallèle. L'algorithme parallèle se déroule comme le montre le pseudo-code du processeur p ci-dessous. Dans un premier temps, chaque processeur calcule une partie de la table de translation d'ombrage, et la diffuse à tous les autres processeurs (multidiffusion : chaque processeur envoie ses données à tous les autres). Entre chaque point de vue, les données sont redistribuées (multidistribution) entre chaque processeur. Chaque processeur compose alors sa partie d'image intermédiaire (*shear*) et d'image finale (*warp*). L'image finale ainsi créée est assemblée sur un processeur puis affichée.

```
Procedure Render()
  DistributionInitiale()
  Pour chaque point de vue faire
    Calcul d'une partie de table de translation d'ombrage
    Multidiffusion de ces parties
    (* Maintenant chaque processeur connaît l'intégralité de la table. *)
    (* Chaque processeur possède les morceaux de coupe *)
    (* qui lui sont nécessaires pour son calcul. *)
    Pour chaque coupe de voxels d'avant en arrière faire
      Si je possède des données
        Composition(données, part_image)
    FinPour
    image = Warp(part_image)
    Rassemblement(image, cible)
    Si p == cible
      Affiche(image)
    Multidistribution(volume)
  FinPour
Fin
```

Performances

En mémoire partagée, Lacroute annonce, pour un volume de données de 256^3, 17 images par seconde sur une SGI Challenge avec 32 processeurs R4400. Les performances de la solution parallèle [8] en mémoire distribuée ne sont pas bonnes. Dans des conditions d'application restreintes, Amin et al. arrivent à fournir 10 images par seconde pour un volume de données de 256^3 et pour une variation d'angle de un degré. Ces résultats n'incluent pas de mesures de performances pour des angles de rotation supérieurs à un. Cependant, il est important d'avoir une politique d'équilibrage de charge qui ne s'appuie pas sur un angle de rotation faible. En effet, pour générer une image liée à une rotation de point de vue de quarante degrés, il serait impensable de générer 40 images car cela prendrait donc 4 secondes. Le défaut majeur de cet algorithme est qu'il est, en outre, très peu extensible. L'accélération est égale à 35 sur 128 processeurs et à 10 sur 20 processeurs. Cette efficacité faible est dûe à l'absence de recouvrement de communications par des calculs.

Pour améliorer cet algorithme, nous cherchons donc à mettre en place une méthode de recouvrement des communications par les calculs lorsque les calculs sont indépendants des données communiquées. Lorsque les calculs et les communications sont dépendants, il est parfois possible de faire des communications à un grain plus fin afin de faire démarrer le premier calcul plus tôt. Pour ce faire, il faut que l'algorithme choisi possède certaines caractéristiques. Par exemple, si le volume de communications est très faible comparé aux calculs, on ne pourra pas gagner un temps significatif. Dans la suite de ce rapport, nous étudions une nouvelle implémentation du shear-warp dans cette optique.

5.3 Optimisation de l'algorithme parallèle

5.3.1 Hypothèses et objectif

Les algorithmes de rendu volumique sont bien adaptés au parallélisme de données à cause de l'important volume de données traité. Dans ce cas, les phases de calculs sont suivies par des phases de communications dont dépendent d'autres phases de calcul, etc. Les parallélisations existantes du shear-warp suivent ce schéma. Lors de l'exécution de plusieurs rendus, les phases de composition et de *warp* sont suivies par une phase de communication transférant les données dont dépend la composition du rendu suivant. De plus, ces phases sont bien souvent synchrones, c'est-à-dire que tous les processeurs exécutent au même moment la phase de communication et attendent que tous aient terminé pour commencer la phase de calcul suivante. Le surcoût de la parallélisation est alors majoritairement dû aux communications et le programme est peu extensible. Pour réduire ce surcoût, on doit soigneusement choisir une distribution de données limitant les communications. Cette distribution étant appelée à changer au cours du temps, une solution supplémentaire consiste à recouvrir les communications par du calcul. Ceci n'est possible que dans le cas de calculs indépendants des communications à recouvrir. Le cas de calculs dépendants de données à communiquer empêche d'effectuer un simple recouvrement ; dans ce cas, l'idée est d'exécuter, si possible, l'algorithme à parallélisme de données de manière pipelinée (macro-pipeline à gros grain) [19]. Dans cette partie, nous étudions la faisabilité de telles techniques sur un algorithme de rendu volumique en shear-warp. Ce chapitre montre les différents choix de parallélisation effectués pour paralléliser le shear-warp sur une grappe de PC reliés par un réseau rapide, les mesures effectuées pour démontrer la pertinence du recouvrement des communications par le calcul et du macro-pipeline et les performances obtenues.

Dans cette partie, nous allons chercher à améliorer les parallélisations existantes du shear-warp. Pour ce faire, nous allons nous appuyer sur des hypothèses d'utilisation et d'implantation.

Une première amélioration des versions parallèles existantes consiste à relâcher les contraintes sur les hypothèses d'utilisation. En effet, comme nous l'avons justifié précédemment, nous cherchons à pouvoir générer une image correspondant à un point de vue quelconque par rapport à l'image précédente. Nous effectuerons donc une parallélisation tenant du changement de point de vue.

Les hypothèses d'implantation sont tout d'abord matérielles. Notre machine cible est une machine à mémoire distribuée avec un réseau rapide et une couche de communication optimisée. Ensuite, elles sont logicielles. Comme nous l'avons vu dans la partie précédente, nous n'allons paralléliser que l'étape de composition. Une autre hypothèse est que nous cherchons à réduire au maximum l'encombrement mémoire afin de pouvoir utiliser à l'avenir des jeux de données plus grands. En outre, après avoir montré l'importance des optimisations séquentielles dans la première partie, nous tâcherons de les implanter dans la version parallèle, dans un premier temps sans exécution spéculative (pour la terminaison anticipée de rayon, chaque processeur pourrait calculer une partie du rayon au cas où celle-ci serait nécessaire dans une exécution spéculative).

Etape	Durée en ms
Ombrage	34
Composition	755
Warp	42
Total	851

FIG. 5.7 – Temps d'exécution séquentiels.

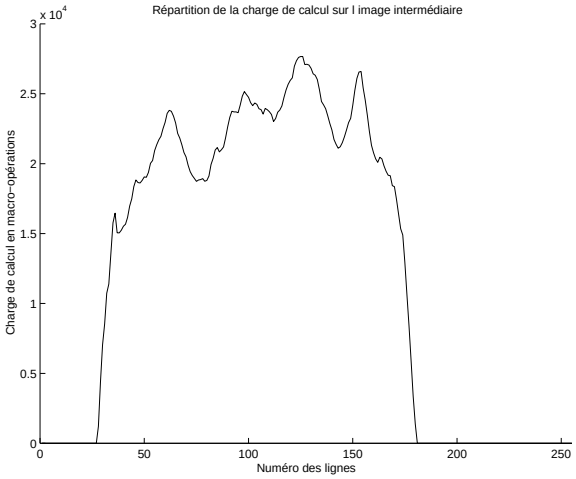


FIG. 5.8 – Répartition des calculs sur l'image intermédiaire.

5.3.2 Analyse quantitative

Comportement séquentiel

Temps d'exécution. Les temps d'exécution de l'algorithme séquentiel fourni par Lacroute décrit dans le chapitre précédent sur un PC (Intel P6-200 avec 64 Mo de DRAM) sont ceux de la figure 5.7. Il s'agit d'un rendu d'une image de taille 256^2 à partir d'un jeu de données médicales issues d'un scanner d'une taille $256 \times 256 \times 225$.

Ces résultats corroborent ceux de Lacroute [41] qui donnaient moins d'une seconde pour ce même type de rendu. La quantité de données est d'environ 15M. Les temps de calculs de la table d'ombrage et de *warp* représentent moins de 10% du temps total. Ce point justifie donc que dans la suite de cette étude, nous nous focalisions sur la composition.

La composition. La composition est la projection du volume de données sur l'image intermédiaire. Elle consiste à rééchantillonner le volume et à accumuler une opacité le long d'un rayon du plan de l'image vers l'observateur. Il s'agit d'une boucle composant chaque voxel de chaque coupe. Les données du volume étant encodée en RLE, l'algorithme séquentiel parcourt simplement l'ensemble du RLE pour composer le volume.

La charge de calcul de cette composition est intrinsèquement mal répartie sur l'image à composer. La charge en calcul pour une partie de l'image intermédiaire, calculée en macro-opérations correspondant à l'ensemble des opérations effectuées sur un voxel, n'est donc pas proportionnelle au nombre de lignes (voir figure 5.8).

Les techniques d'optimisation de l'algorithme apportent deux sources d'irrégularités supplémentaires. Si l'on cherche à délimiter le RLE en un ensemble de coupes, une première source d'irrégularité apparaît. Chaque coupe représente un ensemble de données ayant une quantité fortement différent d'une coupe à l'autre. En général, les premières coupes sont vides. En effet, lors de l'acquisition d'un crâne on commence légèrement au dessus du crâne, la

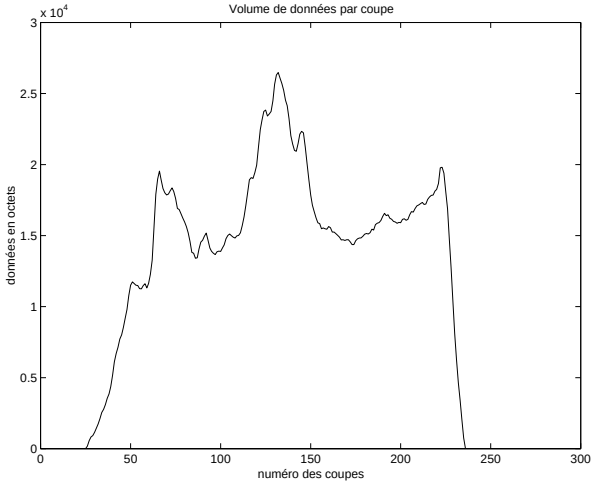


FIG. 5.9 – Quantités de données par coupe.

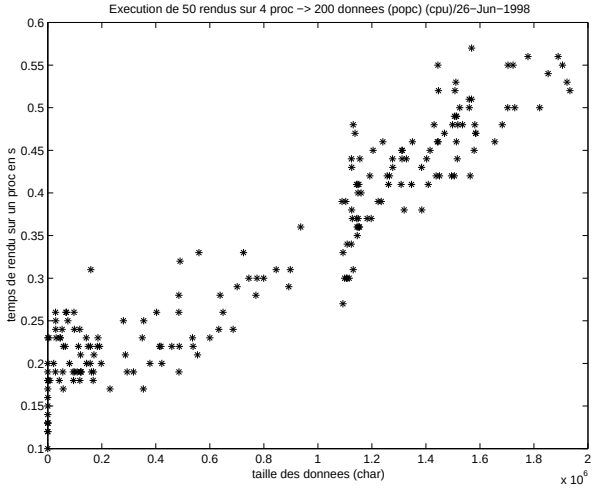


FIG. 5.10 – Temps de calcul par rapport à la quantité de données.

première coupe d'acquisition est vide. En revanche, les parties centrales de la tête contiennent le maximum de données. La structure de données (RLE) utilisée pour coder ces données est telle que sa taille est proportionnelle à la quantité de données possédant une information significative (voxels non transparents). Par conséquent, la première coupe pourra représenter quelques octets alors qu'une coupe centrale aura une taille proche du méga-octet. La figure 5.9 montre la quantité de données sur chacune des coupes. Ces quantités sont, bien sûr, extrêmement dépendantes du volume et du type de données en entrée. Les nombres cités se réfèrent à un volume de taille $256 \times 256 \times 225$. Ceci est une première source d'irrégularité, en effet, le nombre de coupes à traiter n'est pas proportionnel à la quantité de données à traiter.

La figure 5.10 montre que le temps de composition est proportionnel à la quantité de données à composer. Nous en concluons donc que le nombre de coupes à composer ne donne pas d'informations sur le temps de calcul de cet ensemble de coupes.

Dans l'algorithme séquentiel, l'irrégularité de la répartition du temps de calcul pour chaque ligne d'image intermédiaire est spécifiquement liée aux données ; en revanche la répartition du temps de calcul et des données par coupe est due à la structure en RLE liant calculs et données possédant de l'information. L'algorithme parallèle répercute naturellement ces irrégularités sur chaque processeur, rendant critique l'utilisation d'un mécanisme d'équilibrage de charge.

Comportement parallèle

Nous attachons une grande importance à paralléliser l'algorithme en conservant les sources d'accélération. Le partage des tâches, conservant la terminaison anticipée de rayon, défini au chapitre précédent est repris ici. En outre, le RLE permettant d'exploiter la cohérence de

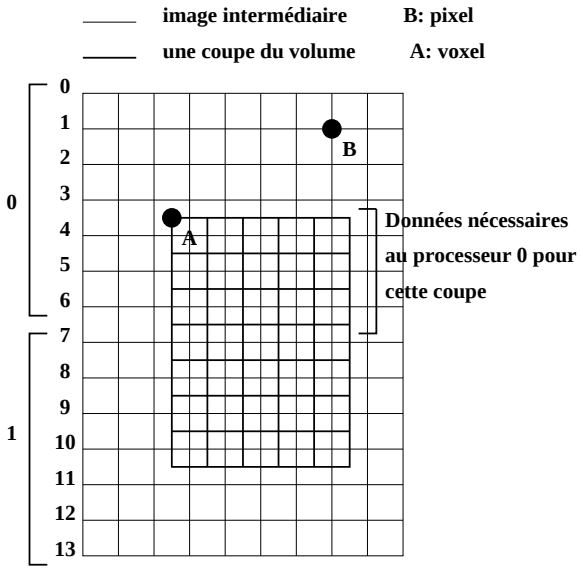


FIG. 5.11 – Coupe dans l'espace objet déformé.

données est conservé. On étudie, dans cette partie, quelles sont les implications du partage des tâches par rapport à l'irrégularité de l'application, la politique d'équilibrage de charges choisie et l'implication de la distribution des données sur les communications.

Nous partageons le calcul de l'image intermédiaire entre les processeurs. La granularité en blocs de lignes de ce partage a été justifiée au chapitre précédent. Pour commencer, on attribue des blocs de taille égale de lignes contigües à chaque processeur et on cherche à déterminer quelles sont les données du volume nécessaires pour effectuer ce calcul. Nous pouvons, à partir d'une coupe dans l'espace objet, trouver à quelles lignes de l'image intermédiaire elle contribue. La figure 5.11 montre une coupe du volume dans l'espace objet déformé. Le calcul de l'image intermédiaire est partagé entre deux processeurs avec une distribution en blocs de même taille. Les lignes de pixels de l'image intermédiaire sont parallèles aux lignes de voxels de la coupe. Les lignes de coupe du volume contribuant au calcul d'une ligne de l'image intermédiaire sont les lignes de voxels l'entourant (ceci est dû à la nature bilinéaire du filtre). Nous voyons donc sur la figure 5.11 que le processeur 0 a besoin des quatre premières lignes de la coupe courante.

Chaque coupe est ainsi morcelée en blocs de lignes entre les processeurs. L'irrégularité est alors double. En effet, pour calculer la première ligne de l'image intermédiaire, n'interviennent souvent (ceci est dépendant de la déformation du volume donc du point de vue) que les lignes de la première coupe. En revanche, le calcul du centre de l'image fait souvent appel à des lignes de voxels situées sur plusieurs coupes (voir figure 5.3). La deuxième source d'irrégularité est que, comme on l'a vu à la figure 5.3, selon la déformation du volume, le nombre de lignes par coupe attribué à chaque processeur est différent ; notamment il est courant qu'un processeur ait besoin de l'intégralité des données de la première coupe et d'aucune donnée de la dernière coupe. Ceci induit une autre irrégularité : le temps de calcul par coupe est différent pour chaque processeur même si toutes les coupes étaient

identiques du point de vue de leurs données. Les figures 5.12 et 5.13 montrent, pour quatre processeurs, quel est le nombre de lignes (respectivement le nombre de données) que chaque processeur possède par coupe. Nous constatons, de manière flagrante sur la figure 5.12, que les processeurs 0 et 3 possèdent un nombre de lignes linéairement décroissant (respectivement croissant) en fonction des coupes. Les deux processeurs possédant les parties centrales de l'image intermédiaire sont équilibrés en nombre de lignes. Ceci n'est pas, comme nous l'avons vu précedemment, proportionnel aux données. La figure 5.13 montre que les processeurs centraux possèdent la plupart des données, le processeur 0 en a beaucoup moins et le dernier processeur n'a aucune donnée sur aucune coupe. Ces résultats sont obtenus sans équilibrage de charges avec un découpage de l'image intermédiaire par blocs de tailles égales de lignes contigües.

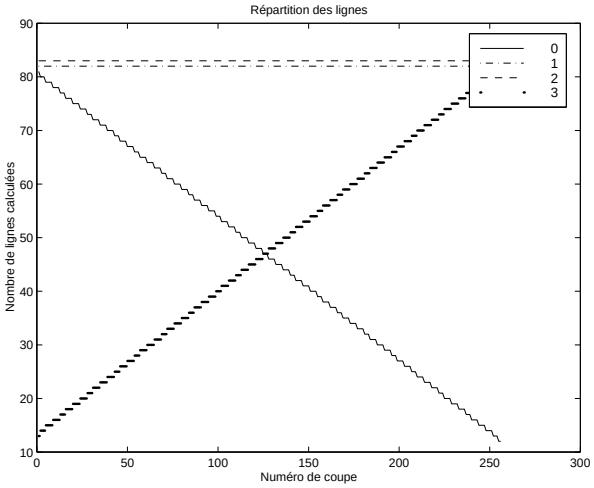


FIG. 5.12 – Répartition des lignes.

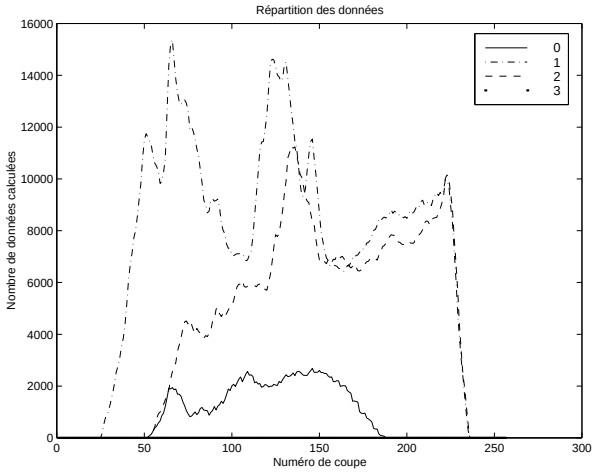


FIG. 5.13 – Répartition des données.

Comme le montre la figure 5.14, la répartition du temps de calcul par processeur suit cette répartition des données. Seuls les processeurs 1 et 2 travaillent. Il faut donc réfléchir à une méthode d'équilibrage de charge appropriée.

5.3.3 Équilibrage de charges

Principe

Comme on l'a vu au chapitre précédent, les méthodes d'équilibrage de charges qui ont été appliquées à l'algorithme du shear-warp par Lacroute et Amin et al. sont adaptées soit à la machine cible (machine à mémoire partagée) soit à des hypothèses fortes sur l'utilisation de l'algorithme. Sur une machine à mémoire distribuée, on ne peut pas utiliser le vol de tâches parce qu'il nécessite une redistribution massive des données au cours d'un même rendu. L'équilibrage de charges adaptatif proposé par Amin et al. n'est valable que dans le cas de l'utilisation particulière de l'application où l'on ne changerait le point de vue que par des rotations de degré en degré. La figure 5.16 montre cette répartition en fonction du point de vue. La courbe représente la charge pour un premier point de vue, puis après une rotation

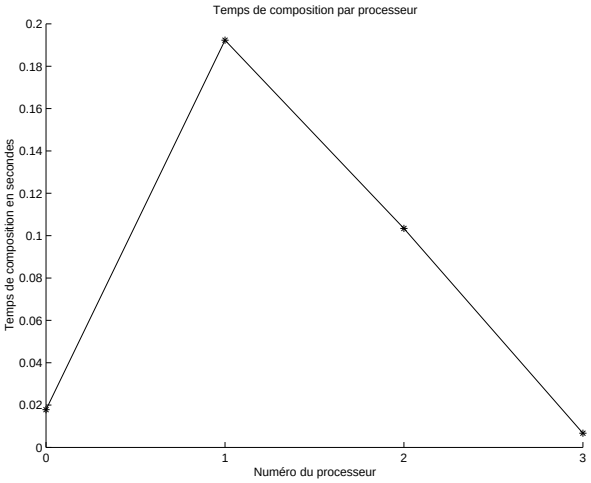


FIG. 5.14 – Répartition du temps de calcul.

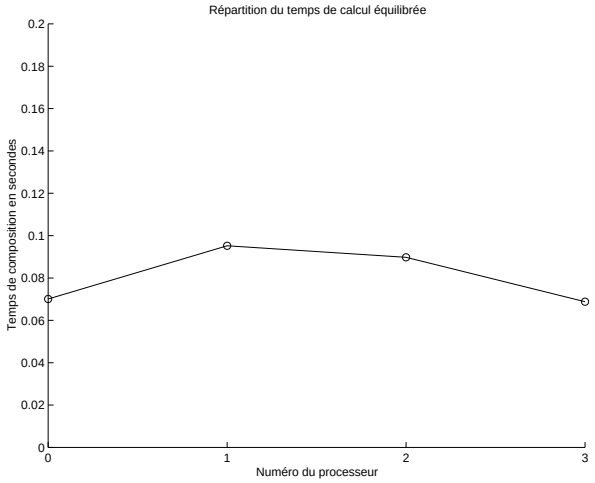


FIG. 5.15 – Répartition du temps de calcul équilibrée.

de 1 degré par rapport à l'un des axes, ou 5, 10 et enfin 20 degrés. L'allure de chaque courbe est identique à celle de la figure 5.8. La courbe est translatée sur l'axe des abscisses vers les numéros de lignes croissants. La figure montre alors qu'effectivement lorsqu'on fait varier l'angle de 1 degré, la variation de charge est petite mais elle est strictement croissante pour chaque ligne de l'image en fonction de l'angle.

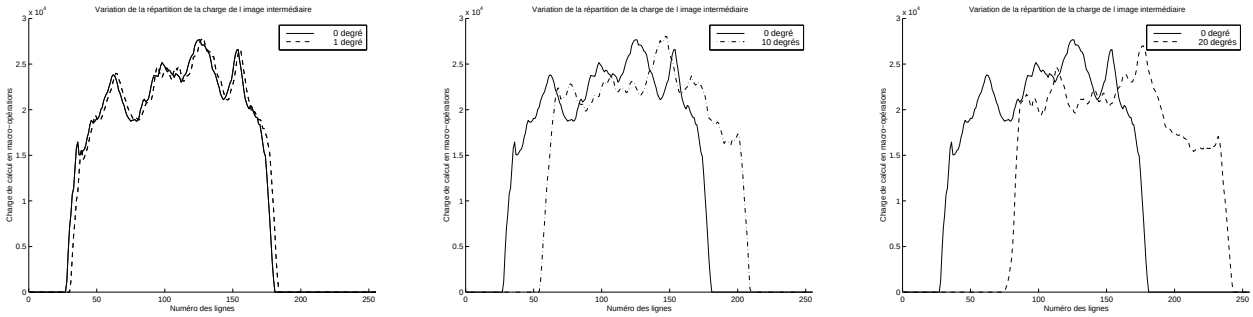


FIG. 5.16 – Répartition de la charge de l'image intermédiaire en fonction de l'angle de rotation.

L'équilibrage de charges de Amin et Al. valable pour des petits angles de rotation n'est donc plus valable pour un angle quelconque. Le temps de rendu devient proportionnel à l'angle de rotation. C'est pourquoi il serait intéressant de libérer l'application des contraintes d'utilisation liées à cette politique d'équilibrage de charges afin d'obtenir une image pour un point de vue quelconque en un meilleur temps.

Un nouvel équilibrage de charges

L'équilibrage de charges est lié au partitionnement de l'image intermédiaire. Nous avons vu, dans la partie précédente, que la granularité du partitionnement est la ligne. Afin d'équilibrer la charge, on peut donc partager l'image intermédiaire de manière bloc cyclique. Expérimentalement, on s'aperçoit que plus il y a de cycles, plus la charge est équilibrée, en revanche, pour une taille d'image moyenne (correspondant à un volume de données de $256 \times 256 \times 225$), les meilleurs temps de rendu sont obtenus pour un nombre de blocs par processeur égal à un. Nous sommes donc ramenés à la répartition précédente simple du temps de calcul (figure 5.14), particulièrement déséquilibrée. Il faut des blocs de lignes de l'image intermédiaire de taille variable. En effet, les partitionnements en deux dimensions ont été écartés puisqu'on a vu que la granularité du découpage devait être la ligne.

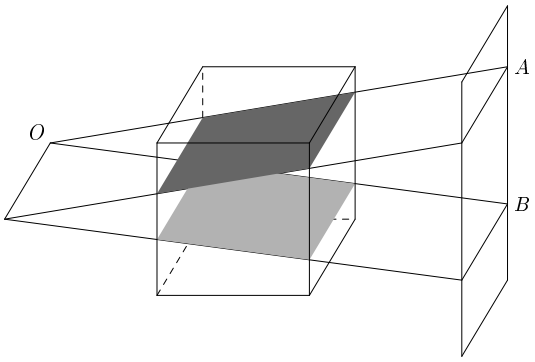


FIG. 5.17 – Plans découpant les données d'un volume.

Un algorithme d'équilibrage de charges adapté à un changement d'angle de vue quelconque doit être capable d'évaluer la charge en calcul d'un processeur par rapport à un nombre de lignes de calcul de l'image intermédiaire à attribuer. Nous avons montré (figure 5.10) que le temps de calcul d'une ligne de l'image intermédiaire est proportionnel à la quantité de données à parcourir dans le volume pour calculer cette ligne. À la différence d'un algorithme de rendu volumique ne manipulant pas de structure creuse, on ne peut pas, en connaissant la quantité de données totale et l'angle de vue, en déduire la répartition de la charge de l'image à composer. En effet, pour un tel algorithme, il suffit de connaître uniquement les deux plans grisés sur la figure 5.17 pour en déduire la charge associée à un bloc de l'image. Li présente dans sa thèse [43] l'algorithme élastique développé par Miguet et Robert [46] appliqué à un algorithme de rendu volumique n'utilisant pas de structure creuse pour encoder les données de la scène. L'algorithme élastique consiste à calculer une charge partielle locale à chaque processeur puis à cumuler toutes ces charges pour en déduire une charge élémentaire attribuée à chaque processeur par le biais d'une redistribution de données.

Dans l'algorithme du shear-warp, la structure creuse implique, sur une machine à mémoire distribuée, que chaque processeur ne connaît pas la quantité de données possédée par un autre processeur parce que la quantité de données n'est pas proportionnelle au nombre de lignes de coupe possédées (voir figure 5.9). Cette problématique est similaire à une problématique d'algorithme de rendu surfacique qui, en stockant un ensemble de surfaces, gère une

structure creuse. Dans le Z-buffer parallèle [43], par exemple, les structures à gérer sont un ensemble de sommets et un ensemble de surfaces composées par trois sommets. Les surfaces sont distribuées sur chaque processeur pour effectuer un calcul correspondant à un point de vue donc un processeur ne connaît pas les positions des surfaces possédées par les autres processeurs pour le prochain point de vue et n'est donc pas capable de calculer la charge en calcul du Z-buffer. Dans l'algorithme du shear-warp, les données sont distribuées pour le calcul d'une image correspondant à un point de vue particulier. Elles correspondent aux données nécessaires pour le calcul d'un bloc de lignes de l'image intermédiaire. Pour le point de vue suivant (quelconque), les données locales à un processeur auront une contribution à l'image intermédiaire qui ne correspondra plus à un bloc de lignes mais à des lignes éparpillées dans l'image intermédiaire. Pour évaluer la charge d'un bloc de lignes de l'image intermédiaire correspondant au nouveau point de vue, il faudrait pouvoir être capable d'estimer la quantité de données correspondant aux lignes à parcourir dans le volume pour calculer cette portion d'image intermédiaire. Or, on a vu que seul un processeur possédant une ligne donnée peut en connaître la taille. Un processeur ne peut donc pas calculer la charge de calcul globale de l'image intermédiaire. Nous appliquons alors la stratégie de l'algorithme élastique appliqué à un algorithme de rendu surfacique, le Z-buffer parallèle [43], pour équilibrer la charge dans l'algorithme de rendu volumique en shear-warp. L'algorithme élastique consiste alors à calculer un tableau contenant pour chaque ligne de l'image à composer sa contribution personnelle en terme de charge. Chaque processeur envoie ensuite ce tableau à chaque autre processeur. L'ensemble des processeurs somme ligne à ligne les contributions des autres processeurs. Le tableau résultant contient une répartition de la charge en temps de calcul par ligne de l'image intermédiaire. Ces charges ligne à ligne sont sommées pour obtenir la charge globale. La charge globale divisée par le nombre de processeurs donne une charge élémentaire d'équilibre. Ainsi, le premier processeur saura que pour atteindre cette charge, il lui faut $n1$ lignes, le processeur 2 les lignes $n1$ à $n2$, etc.

L'avantage de cet algorithme est que la charge est bien équilibrée. La comparaison des figures 5.14 et 5.15 montre la différence des répartitions de calcul entre l'algorithme sans équilibrage de charges et l'algorithme avec équilibrage de charges sur quatre processeurs. Qu'en est-il sur la répartition des lignes et des données pour chaque coupe ?

L'équilibrage de charges étudié dans les parties précédentes n'implique pas que le volume de données de chaque coupe attribué à chaque processeur soit régulier. En effet, il garantit simplement que le volume de données global sur chaque processeur soit équilibré. Les mesures effectuées avec équilibrage de charges quant aux volumes de lignes et de données de coupe par coupe sur chaque processeur sont données respectivement aux figures 5.18 et 5.19. La comparaison entre les figures 5.12 et 5.18 (sans et avec équilibrage de charges) montre par exemple, que les processeur centraux étaient surchargés puisque le nombre de lignes moyen qu'ils possédaient a diminué de plus de 50%. Les processeurs 0 et 3 étaient, quant à eux, sous chargés. Le processeur 0 par exemple, qui avait en moyenne 45 lignes par coupe dans la version non équilibrée se retrouve avec 110 lignes par coupe en moyenne dans la version équilibrée. Les courbes de données de la figure 5.19 montrent que tous les processeurs ont des données, ce qui n'était pas le cas dans la version non équilibrée. Le nombre moyen de données par processeur est dur à estimer mais on peut remarquer qu'aucune courbe n'est vraiment ni haute ni écrasée par rapport aux autres et qu'on ne peut pas distinguer de tendance globale.

L'irrégularité reste donc complète en ce qui concerne le volume de données par coupe sur chaque processeur.

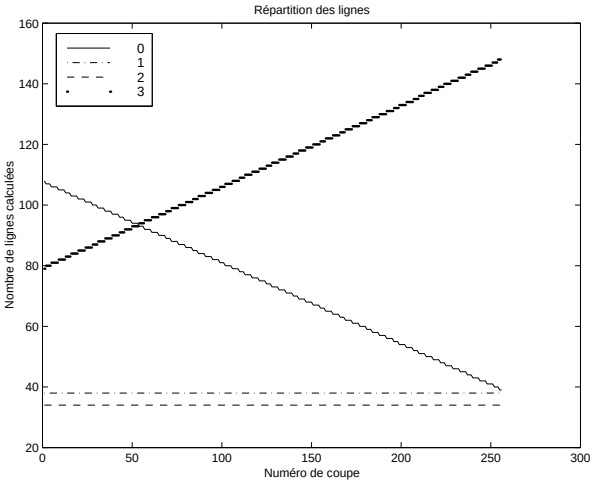


FIG. 5.18 – Répartition des lignes équilibrée.

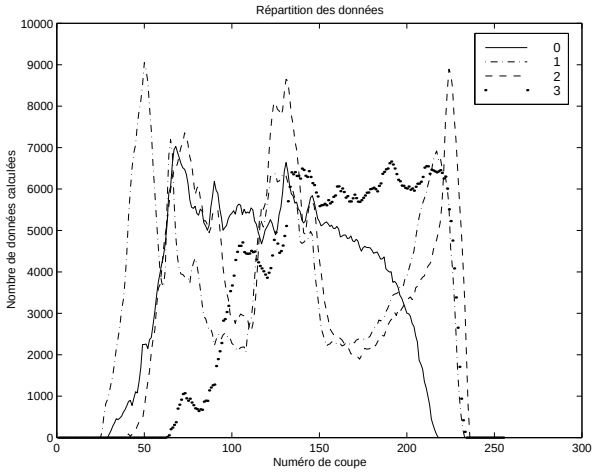


FIG. 5.19 – Répartition des données équilibrée.

L'inconvénient de cet algorithme d'équilibrage de charges est qu'il coûte cher du point de vue des performances en temps de rendu. En effet, il coûte une communication globale (multi-réduction) et engendre donc un point de synchronisation.

Une solution envisageable est de combiner l'équilibrage de charges adaptatif d'Amin et al. et l'algorithme élastique en exécutant une opération de redistribution des données équilibrée si l'angle de rotation relatif a dépassé un certain seuil en dessous duquel on considère que les charges sont suffisamment équilibrées.

Communications. Dans un algorithme à parallélisme de données, après avoir équilibré les calculs, il faut optimiser les communications. Dans l'algorithme du shear-warp parallèle sont présentés deux types de communications :

- un rassemblement des images finales partielles en une image,
- et une multidistribution au cours de la redistribution des données à chaque changement de point de vue.

Sur un graphe complet, un rassemblement, chaque processeur envoie ses données à un processeur source, ne peut être exécuté que d'une manière simple. De même, sur un tel graphe, une multidistribution, chaque processeur envoie des données différentes à chaque autre processeur, peut être principalement effectuée de deux façons. La manière la plus simple est que chaque processeur, chacun son tour, envoie ses données à tous les autres processeurs. Cette communication est donc, si p est le nombre de processeurs, en $O(p^2)$ étapes. Une manière plus efficace, en $O(p)$ étapes, est qu'à chaque étape, chaque processeur envoie ses données à un processeur à chaque étape plus éloigné dans les numéros de processeurs croissants et reçoive ses données d'un processeur plus éloigné à chaque étape dans les numéros décroissants. La figure 5.20 montre ce schéma de communications pour quatre processeurs. Il est donc composé de trois étapes.

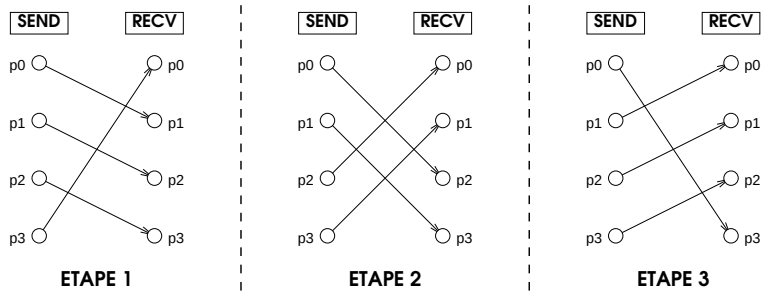


FIG. 5.20 – Multidistribution en $p-1$ étapes.

Nous nous intéressons particulièrement à la multidistribution. En effet, celle-ci est plus coûteuse puisqu'elle met en jeu les données du volume et non celle de l'image comme le rassemblement. De plus, on peut dédier un processeur à la génération de l'image pendant que les autres entameront les communications et les calculs liés au point de vue suivant.

Dans l'algorithme du shear-warp, la multidistribution est très irrégulière. Tous les processeurs n'ont pas nécessairement de données à échanger avec tout le monde et chaque volume de données échangé est très différent. Prenons un exemple. Nous exécutons le shear-warp sur quatre processeurs. Pour le premier point de vue, on établit une distribution des données propre à ce point de vue. Chaque processeur possède donc pour chaque coupe un bloc de lignes de coupe. Sur une coupe, le bloc de lignes nécessaires au calcul de l'image correspondant au prochain point de vue va être modifié en taille et en position. Le processeur n'aura par exemple plus besoin des lignes 1 à 10 mais des lignes 8 à 11. Il conservera donc des lignes, enverra aux processeurs qui en ont besoin les autres lignes et recevra les lignes dont il a besoin des autres processeurs, ceci pour toutes les coupes. Nous avons vu, dans la partie précédente, que le nombre de lignes échangé n'est pas proportionnel au volume de données échangé.

La figure 5.21 prend comme exemple le processeur 2. Pour chaque coupe du volume, on regarde avec combien de processeurs il doit échanger des données, soit en émission soit en réception soit les deux. Nous effectuons cette mesure sans changer de point de vue, le processeur ne communique alors avec personne comme on s'y attend. On tourne de 20 degrés selon l'axe des X. On voit alors que la plupart du temps le processeur 2 ne communique qu'avec un processeur et enfin pour un angle de rotation de 40 degrés, le processeur 2 communique souvent avec tous les autres processeurs. Ceci montre que le nombre de communications est vraiment dépendant du point de vue. Ce nombre de communications n'étant pas proportionnel au volume de communications, la figure 5.22 montre le volume des communications faisant intervenir le processeur 2.

Implantation. Nous avons implanté l'algorithme du shear-warp sur une grappe de PC avec un réseau rapide (Myrinet) et la couche de communication optimisée *Basic Interface for Parallelism* [52] (BIP). L'essentiel de l'effort a porté sur la composition (volume principal de calcul) et sur la redistribution des données nécessaires à chaque changement de point de vue. Un processeur a été dédié à la reconstruction et au warp de l'image finale. L'ensemble des processeurs restants parallélise le calcul de la table de translation d'ombrage et la com-

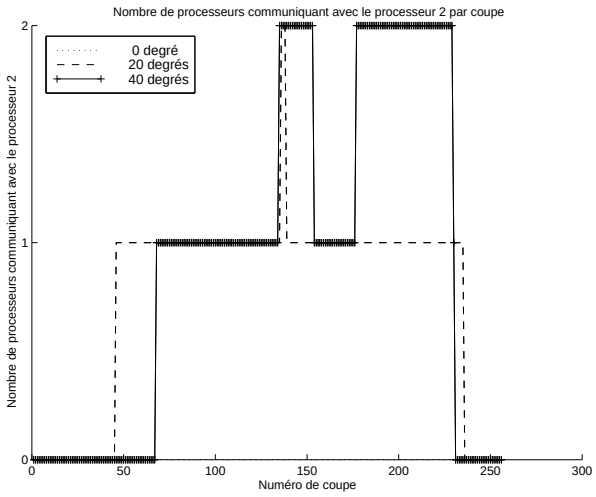


FIG. 5.21 – Nombre d'échanges entre processeurs.

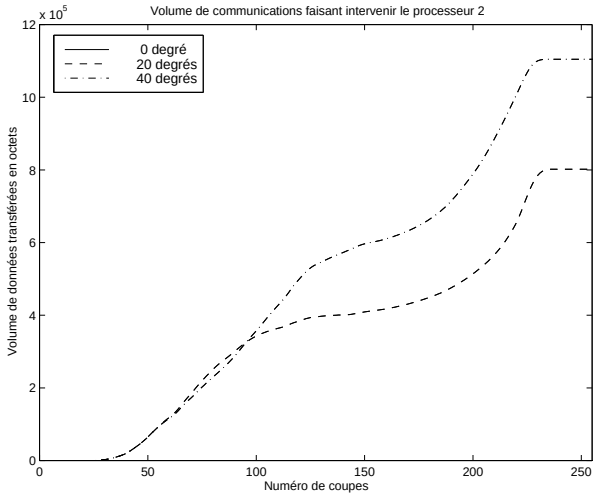


FIG. 5.22 – Volumes échangés entre processeurs.

position, possède donc localement les données du volume et effectue la redistribution des données nécessaires à la composition de la scène suivante. La figure 5.23 montre dans un diagramme de Gantt l'ordonnancement des tâches de l'ensemble des processeurs avec le processeur de rendu. Les points de synchronisation se situent à la reconstruction de l'image pour tous les processeurs et le processeur central et à la multidiffusion par les processeurs de la table de translation d'ombrage avant la composition. Pendant que les processeurs calculent la table d'ombrage, la nouvelle distribution et effectuent la redistribution, le processeur central reconstruit l'image intermédiaire. Après la diffusion de cette table, Les processeurs peuvent effectuer la composition alors que le processeur central attend la reconstruction. Enfin, après l'envoi de leur image au processeur central, les processeurs passent au point de vue suivant.

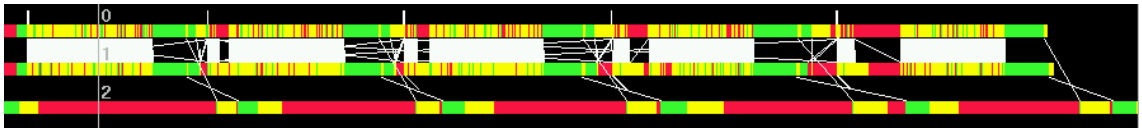


FIG. 5.23 – 5 rendus sur 3 processeurs.

Mesures

Nous avons vu dans la partie précedente que l'équilibrage de charge proposé équilibre effectivement la charge. L'objectif de cette partie est de mesurer le comportement de l'application avec ce nouvel équilibrage de charge. Par rapport à l'existant, la nouvelle implantation prend en compte un nouvel algorithme d'équilibrage de charge mais ne présente pas d'optimisations visant à améliorer les temps globaux de rendus. Donc, en plus de communications

liées à la redistribution des données, nous avons ajouté un algorithme d'équilibrage de charge potentiellement coûteux en temps d'exécution afin d'élargir les possibilités d'utilisation de l'application. Cette éventuelle perte de temps devrait être rattrapée par la suite par le recouvrement des communications par des calculs. Nous effectuons des mesures du temps pris par l'équilibrage de charge, du temps de redistribution des données ainsi que du temps de calcul pour un ensemble de données pour une rotation d'angle de 15 degrés. Ce temps de rendu est minimal pour six processeurs pour lesquels il est égal à $400ms$ (environ 2 images par seconde). L'objectif est d'atteindre au moins 10 images par seconde pour un angle de rotation quelconque. Le nombre de $400ms$ est donc trop élevé pour un rendu. Le temps pris par l'équilibrage est aussi extrêmement élevé, en effet, nous voyons que pour six processeurs, il prend $100ms$, durée que l'on souhaiterait obtenir pour l'ensemble du processus de rendu.

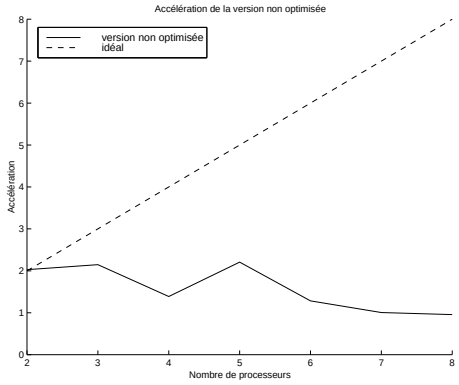


FIG. 5.24 – Accélération pour un rendu.

La figure 5.24 montre l'accélération de la version parallèle non optimisée. Comme nous l'avions remarqué avec la courbe des temps de rendu, cette accélération est très mauvaise. Elle l'est d'autant plus que l'intégralité du code n'ayant pas été parallélisée (notamment le *warp*), le processeur de rendu n'a quasiment aucune charge, il ne fait que le *warp* et les entrées-sorties éventuelles liées à la parallélisation de l'image. Nous pensons donc pouvoir améliorer ces performances en implantant du recouvrement des communications par les calculs.

5.3.4 Recouvrement et macro-pipeline

Jusqu'à présent, la structure des précédentes implantations parallèles était conservée, alternant phases de calculs et phases de communications. Les différences apparaissent au niveau de la technique d'équilibrage de charges et de la couche de communications utilisée. Pour améliorer l'algorithme, on cherche à identifier quelles sont les possibilités de recouvrement des calculs par les communications et de macro-pipeline.

Les possibilités de recouvrement

Jusqu'à présent, toutes les communications liées à la redistribution des données sont effectuées avant les calculs de composition (voir le pseudo-code ci-dessous).

```
Pour k=0, nbCoupes faire
  Pour etape=0, nbProcesseurs faire
    Si doit-envoyer(k, moi+etape)
      envoi(bloc_lignes, moi+etape)
    Si doit-recevoir(k, moi-etape)
      reception(bloc_lignes, moi+etape)
    FinPour
FinPour

Pour k=0, nbCoupes faire
  Composition(bloc_lignes(k))
FinPour
```

On peut donc recouvrir les communications liées à la coupe $k+1$ avec le calcul de la coupe k. Chaque processeur n'attend plus que toutes les communications soient effectuées mais seulement les communications nécessaires à chaque coupe. La figure 5.25 montre les étapes de communications initiales sur exemple de trois coupes. Le processeur attend que tous les $RECV(PI, CI)$ concernant les trois coupes soient arrivés. La première étape de recouvrement consiste à n'attendre que les (PI, CI) utiles au calcul d'une coupe ; le processeur commence alors le calcul de cette coupe pendant qu'il attend les données relatives aux coupes suivantes. Sur cette même figure, nous voyons que les calculs relatifs à la coupe 0 étant plus longs que les communications des données de la coupe 1, le processeur n'attend donc jamais ces données. Au final, nous aurons gagné $A \geq 0$. Ceci revient à fusionner les deux boucles en k.

Le recouvrement peut encore être effectué à un grain plus fin en composant un bloc de lignes à chaque étape de communications. La deuxième étape de recouvrement sur la figure 5.25 montre que l'on attend d'une première communication avec le processeur $P1$ pour commencer les calculs. Pendant ces calculs, le processeur attend de manière asynchrone les autres blocs de données provenant d'autres processeurs pour cette même coupe. Sur cette figure, nous voyons que le calcul du du premier bloc de données est plus court que la communication du deuxième bloc, le processeur attend donc un petit peu le deuxième bloc de données. Au final, nous auront gagné $B \geq A$.

Quels que soient les volumes relatifs des communications et des calculs, l'identification de blocs de calculs indépendants des communications permet de recouvrir celles-ci d'une durée au plus égale au temps de calcul. Ainsi, si les calculs sont beaucoup plus longs que les communications, à l'exception des communications initiales, l'ensemble des communications peuvent être recouvertes. En revanche, si les calculs sont négligeables par rapport aux communications, on ne gagnera pas grand chose en recouvrant les communications par ces calculs.

Il s'agit de mesurer le temps de communication par rapport au temps de calcul. Le choix de la couche de communication utilisée est alors important. Sur une couche de communication lente, le surcoût de la parallélisation de l'algorithme sera lié de manière évidente aux communications. Sur une couche de communication rapide, les ordres de grandeur des temps de calcul et de communication sont proches ou bien encore les temps de communication peuvent éventuellement être négligeables par rapport aux temps de calcul. Il est alors im-

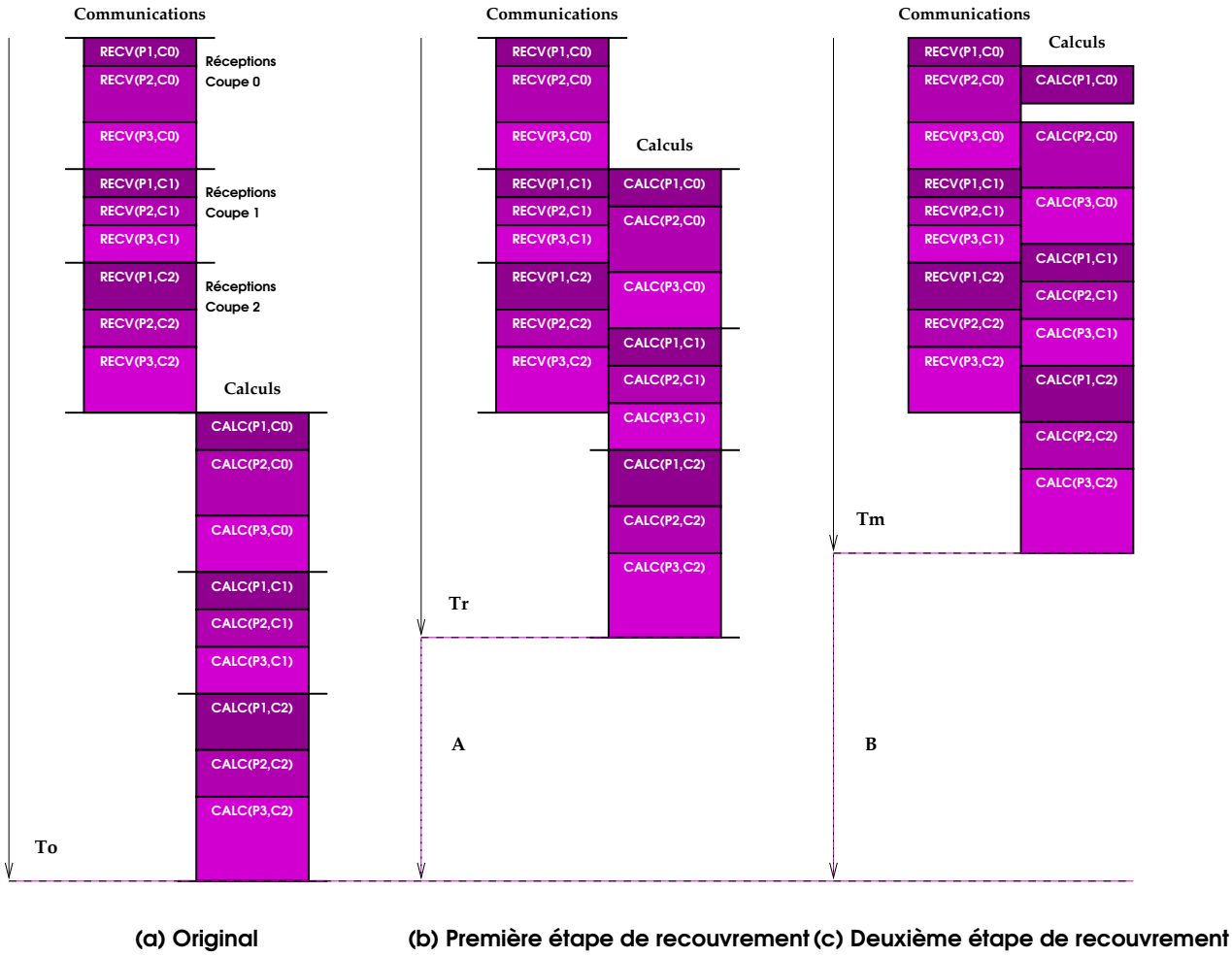


FIG. 5.25 – Recouvrement.

portant de déterminer si le recouvrement des communications par les calculs peut optimiser les performances globales.

Les temps de calcul de la composition sont de l'ordre 70 ms sur quatre processeurs. La composition parallèle équilibrée est parfaitement extensible (car elle n'implique de communications). Les temps de communication sont dépendants du point de vue puisqu'elles varient selon le point de vue en nombre et en volume (voir les figures 5.21 et 5.22).

La figure 5.26 représente les temps de calcul et de communications (mesures effectuées au dessus de BIP) en fonction de l'angle de rotation relatif au point de vue initial pour une exécution sur quatre processeurs. On remarque que le temps de calcul ne varie pas en fonction de l'angle de vue. En effet, quel que soit l'angle, le volume de calcul global est le même. En revanche, le temps de communication est strictement croissant en fonction de l'angle de vue. Cette courbe est rapidement linéaire avec une pente élévée de 2 ms par degré de rotation. On peut donc attendre un recouvrement total des communications par le calcul jusqu'au point d'intersection des deux courbes, avec un angle de rotation relatif de 17 degrés.

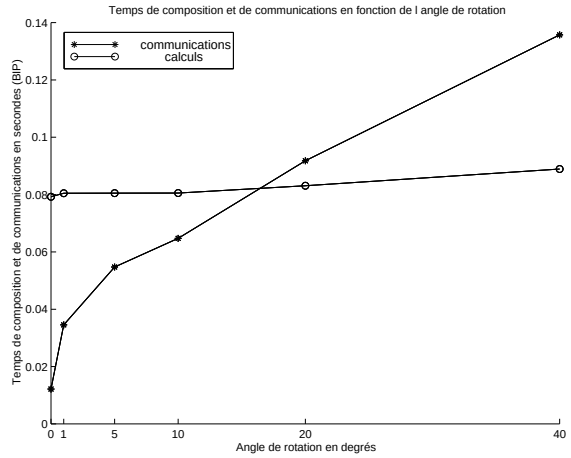


FIG. 5.26 – Temps de calculs et de communications en fonction de l'angle de rotation.

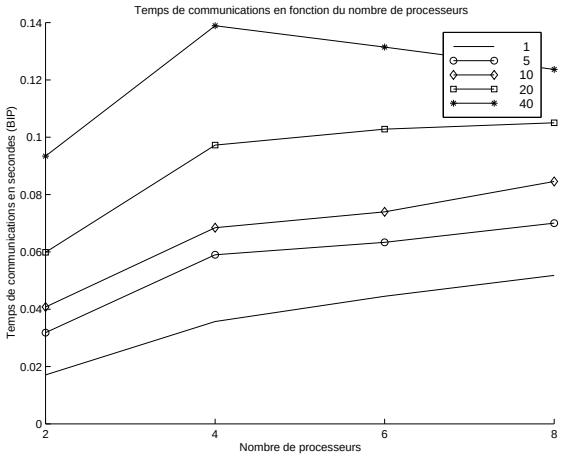


FIG. 5.27 – Temps de communications en fonction du nombre de processeurs.

Au dessus de cette valeur, les communications en seront pas entièrement recouvertes. On ne gagnera sur chaque communication que le temps de calcul qui ne varie pas selon l'angle.

On peut comparer les temps de communication et de calculs en fonction du nombre de processeurs. On a vu que le temps de calcul était parfaitement extensible mais on peut s'attendre à ce que le temps de communications soit fortement croissant avec le nombre de processeurs. Le nombre de communications doit augmenter puiqu'il s'agit d'une multidistribution mais leur volume doit diminuer. La figure 5.27 montre l'évolution du temps de communications en fonction du nombre de processeurs pour plusieurs angles de rotation.

On constate donc que dans certains cas, notamment lorsque les angles de rotation sont grands, le recouvrement des communications par le calcul peut ne pas être suffisant pour améliorer les performances de l'application. C'est pourquoi on cherche à réorganiser davantage les calculs et les communications en utilisant le macro-pipeline.

Le macro-pipeline

Lorsque les calculs sont dépendants des données à recevoir, une technique d'optimisation bien expérimentée pour le calcul dense [19], s'appelle le macro-pipeline. Cette technique consiste à réduire la granularité des communications afin de commencer plus tôt le calcul dépendant (dans un modèle en $\beta + L\tau$, on augmente donc le nombre de β). Dans la partie précédente, pour faire du recouvrement, nous nous contentions de recouvrir au maximum sans diviser les communications. Le pipeline consiste à faire un compromis entrele temps gagné à commencer plus tôt les calculs et le nombre d'initialisation de communications ajouté.

Dans une application régulière, on cherche à modéliser :

- le volume des communications, par exemple en $\beta + L\tau$ pour en déduire un temps de communications t_{comm},
- et le volume de calculs, à partir du volume de données et de la complexité du calcul, pour en déduire le temps de calcul t_{calc}.

Le temps total $t_{tot} = t_{calc} + t_{comm}$ peut donc s'écrire pour μ paquets de taille ν, $t_{tot} = \beta + \nu\tau + (\mu - 1)\max(\beta + \nu\tau, t_{calc}(\nu)) + t_{calc}(\nu)$. On commence par initialiser le processus par une communication de taille ν. Le calcul sur ce paquet peut alors commencer. Le temps total prend en compte le maximum entre le temps que prend la communication et le calcul d'un paquet de taille ν. En dérivant l'expression de t_{tot} en fonction de ν, on trouve donc une taille ν optimale.

Macro-pipeline dans le shear-warp. Dans une application irrégulière, on ne peut pas modéliser ce pipeline car L est très irrégulier. Par conséquent, pour une étape de calcul et de communications, nous ne pouvons pas modéliser sa date de commencement en fonction des étapes précédentes ni prévoir ses conséquences sur l'ordonnancement des calculs et des communications à venir. C'est pourquoi, nous ne pouvons calculer la taille de bloc optimale que dynamiquement.

Implantation. L'implantation des communications a été effectuée avec la bibliothèque de communications MPI[5] au dessus de la couche de communication optimisée BIP. L'interface avec MPI sur la grappe de PC utilisée n'implante pas encore les communications asynchrones. En outre, l'implantation de MPI au dessus de la couche IP[6] ne permet pas d'effectuer des communications asynchrones.

Nous nous sommes focalisés sur la parallélisation de la composition. Dans un premier temps, nous l'avons parallélisée en utilisant des fonctions de communication MPI bloquantes. La mauvaise extensibilité de cette implantation, comme le montre la figure 5.28, est lié au surcoût de la redistribution des données. C'est pourquoi nous avons choisi de recouvrir ces communications. La deuxième implantation de la composition est bien extensible, comme le montre la figure, parce que BIP permet un recouvrement presque parfait des communications et parce que l'indépendance des différents calculs effectués dans l'algorithme était suffisante.

Ces courbes sont obtenues pour un ensemble de données couleurs comprenant 512^3 voxels. Pour de telles données, le temps d'exécution est de 1.5 secondes sur 4 processeurs.

[5] *Message Passing Interface.*
[6] *Internet Protocol.*

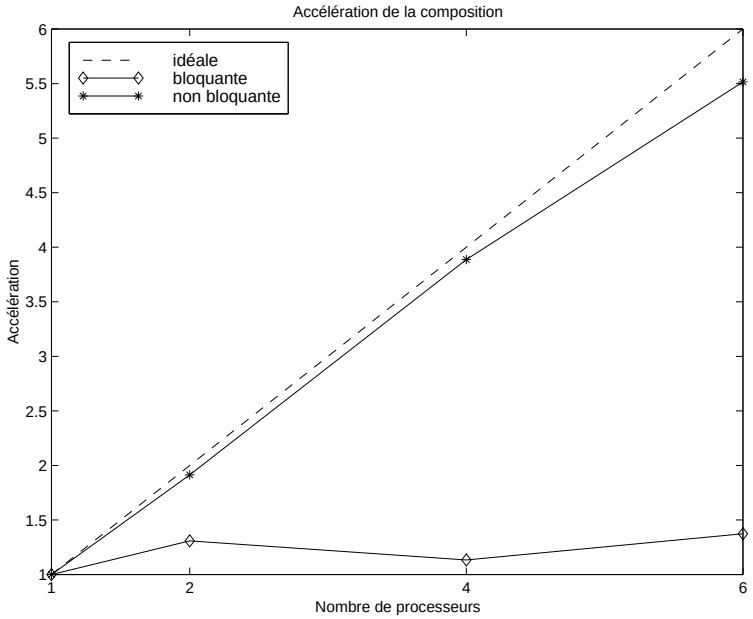


FIG. 5.28 – Extensibilité de la phase de composition.

5.4 Conclusion

Les algorithmes de rendu volumique sont intéressants parce qu'ils permettent d'obtenir des images utiles à l'aide au diagnostic. En effet, ces images conservent l'intégralité des informations contenues dans le volume de données initial. L'inconvénient de ces algorithmes est qu'ils manipulent un grand volume de données et de calculs limitant les performances et les tailles de données traitables.

Dans ce rapport, nous avons proposé une nouvelle parallélisation de l'algorithme de rendu volumique en shear-warp sur une machine à mémoire distribuée. Cet algorithme est en effet le plus efficace des algorithmes séquentiels de rendu volumique. Notre objectif était d'augmenter l'interactivité du shear-warp (permettre à l'utilisateur d'obtenir un point de vue quelconque sur l'objet en temps réel) tout en conservant un encombrement mémoire minimal.

Dans ce but, nous avons tout d'abord étudié les versions parallèles existantes du shear-warp pour en retenir les points forts. Cette étude nous a également permis de voir quels étaient les points faibles à améliorer.

Nous avons donc proposé une solution d'équilibrage de charge adaptée aux hypothèses d'utilisation interactive. Il s'est avéré que cette politique efficace pour équilibrer la charge était très coûteuse. Au surcoût lié aux communications venait s'ajouter un surcoût d'équilibrage de charge. Le changement de point de vue impliquant une redistribution des données, le surcoût lié aux communications, ne pouvait pas être résolu par une distribution fine des données. C'est pourquoi, nous avons étudié les possibilités de recouvrement des communications par les calculs. Cette étude n'a pu être effectuée qu'après une étude quantitative précise du shear-warp. Celle-ci révélant le caractère profondément irrégulier de cette implantation du shear-warp, le recouvrement des communications par les calculs s'est avéré plus ou moins

efficace selon l'angle de vue. La dernière optimisation que nous avons étudiée consiste à utiliser des techniques de macro-pipeline lorsque le recouvrement des communications par les calculs ne s'avère pas suffisant.

Nous proposons alors une stratégie adaptative de parallélisation. En effet, nous avons vu que pour les changements de point de vue inférieur à un degré, la politique d'équilibrage de charge adaptative d'Amin et al était suffisante. Pour des angles supérieurs, en revanche, nous avons implanté un algorithme d'équilibrage de charge adapté à une solution plus interactive. De même pour le recouvrement des communications par le calcul, nous avons montré qu'il existe un angle pour lequel ce recouvrement n'est pas suffisant pour diminuer le surcoût des communications. Pour ces grands angles, nous avons alors utilisé des techniques de macro-pipeline.

L'implantation paralléle de l'algorithme de rendu volumique en shear-warp sur une machine à mémoire distribuée que nous avons proposée a permis d'augmenter son interactivité, c'est-à-dire qu'elle permet à l'utilisateur d'obtenir une image en temps réel pour un point de vue quelconque pour un encombrement mémoire minimal. En effet, pour rendre un image correspondant à 512^3 voxels, il faut 1.5 secondes sur 4 processeurs.

Une première utilisation de ce travail est d'obtenir une application permettant à un praticien d'avoir une visualisation, de bonne qualité, en trois dimensions en temps réel de données acquises par des modalités médicales, ceci à l'aide d'une machine parallèle à mémoire distribuée. À terme, ce travail devra nous permettre d'obtenir un mécanisme générique de recouvrement des communications par les calculs appliqué à des algorithmes d'imagerie médicale de visualisation en trois dimensions.

Chapitre 6

Conclusion et perspectives

Cette thèse a présenté une étude des recouvrements des communications en trois parties : moyens logiciels, algorithmiques et applicatifs. La première partie a étudié les possibilités de recouvrement des communications sur une machine à mémoire distribuée. Nous avons montré que, même sur une architecture de machine relativement simple, la complexité des mécanismes mis en jeu ne permet pas à un utilisateur de contrôler la façon dont les calculs et les communications sont synchronisés entre eux. Si les performances intrinsèques des implantations actuelles de la bibliothèque MPI se rapprochent des performances que peut fournir le matériel, les coûts de synchronisation dans un programme réel ne sont pas maîtrisés. Or, les grappes de PC sont maintenant remplacées par des grappes de SMP et par des grappes de grappes de PC dans le cadre du métacomputing. C'est pourquoi les techniques d'optimisation de programmes parallèles par le recouvrement des communications ne peuvent plus être employées aussi directement qu'auparavant. La standardisation des bibliothèques de communication et la complexité des machines font que le comportement réel du programme échappe au programmeur. Celui-ci doit auparavant s'assurer qu'elles seront efficaces en fonction de l'architecture cible.

La deuxième partie a abordé les recouvrements de calculs en étudiant le pipeline de calculs. Nous avons montré qu'il est possible d'obtenir une diminution significative du temps d'exécution d'une application par cette technique. Dans cette partie, nous avons tout d'abord modélisé le gain obtenu par un pipeline en utilisant la modélisation d'une application par vagues. Nous avons, dans un premier temps, validé les équations de gain obtenu par Zory avec l'application Sweep3D. Puis, en modélisant cette application, nous avons obtenu des équations de gain plus précises pour des données tridimensionnelles dans le cas d'un pipeline synchrone. Le pipeline de calculs reste néanmoins une technique coûteuse en temps d'analyse et de développement. De plus, pour chaque application, il existe plusieurs possibilités de configuration de l'espace des processeurs et de dimensions de pipeline. La solution la plus efficace dépend des performances matérielles de la machine cible et de l'application. Nous avons établi les équations de temps d'exécution pour chacune de ces configurations possibles en fonction de paramètres caractérisant la machine cible et l'application. Ceci nous permet ainsi de prévoir la solution la plus efficace avant l'implantation du pipeline.

La troisième partie décrit la parallélisation d'une application irrégulière et étudie les possibilités de recouvrement simple et de pipeline dans une telle application. Cette partie montre

sur un cas concret complexe les gains obtenus grâce aux différents types de recouvrements et la difficulté de leur mise en œuvre. Nous avons parallélisé une application irrégulière utilisant une structure de données creuse de rendu volumique. Les optimisations par les recouvrements permettent d'obtenir une image tridimensionnelle obtenu par rendu volumique à partir d'un ensemble d'images médicales en temps réel.

Pour terminer, les annexes présentent mes travaux dans différents projets industriels effectués avec MS&I. Ces projets industriels ont tous pour point commun l'utilisation de grappes de PC fortement couplées pour améliorer le temps de réponse d'applications qu'elles soient dans le domaine de l'imagerie médicale ou des technologies Web.

Travaux futurs

Les modélisations effectuées au cours de cette thèse sont adaptées à des machines homogènes, tous les nœuds composant cette machines sont identiques et un même réseau d'interconnexion relie tous ces nœuds. Les nouvelles générations de machines sont des grappes de SMP ou même des grappes de grappes. Il y aura donc des nœuds reliés entre eux par un réseau local et ils seront connectés à d'autres nœuds de la même architecture par un réseau externe, c'est-à-dire moins performant. Dans ces cas là, notre modèle n'est plus adapté, comme le montrent les expérimentations de la partie 4.2.2 page 84. Il serait donc intéressant d'appliquer ces méthodes à des machines hétérogènes. Dans un premier temps, on pourra considérer une hétérogénéité simple, comme des grappes de SMP qui n'ont qu'un seul type de processeur et un réseau d'interconnexion à deux niveaux : accès à la mémoire partagée locale et aux mémoires distantes. On cherchera à modéliser l'exécution d'un pipeline dans ce cas là et à en déduire les tailles de bloc optimales. Enfin, on essaiera d'étendre ces résultats à une hétérogénéité plus générale.

Annexe A

Projets industriels

Cette partie présente les projets industriels auxquels j'ai participé en tant que doctorante chez Matra Systèmes & Information. Le premier projet, PARSMED-3D, est un projet d'imagerie médicale au cours duquel nous avons parallélisé des applications de reconstruction d'images en trois dimensions sur une grappe de PC et optimisé le processus d'acquisition des données en pipelinant l'acquisition des données et le début des traitements.

Le projet Medistar consiste en l'installation d'une grappe de PC interconnectée par un réseau rapide Myrinet et avec des bibliothèques de communication rapides telles que MPI/BIP. Cette grappe de PC était destinée à un service de l'hôpital de la Pitié-Salpétrière spécialisé dans l'étude du cerveau.

Enfin, le projet CHARM consiste en la parallélisation d'un cache web sur une grappe de PC. Nous avons étudié les fonctionnalités d'un cache web et plus particulièrement, le cache web du domaine public *Squid* afin d'évaluer les possibilités de parallélisation. Nous avons contribué à mettre au point l'architecture du système final.

A.1 PARSMED-3D : imagerie médicale

PARSMED-3D est un projet européen ayant pour but de reconstruire en 3D et en temps réel des images issu d'appareils de sonographie dans la pratique quotidienne. Les partenaires de ce projet sont : une équipe médicale (le service hospitalier de Gynécologie Obstétrique de l'Hôpital de Bologne), une entreprise développant des outils fournissant des images et des reconstructions 3D médicales : IôDP et enfin MS&I.

IôDP a fourni les technologies permettant d'acquérir les données issues de la sonde à ultrason et de les traiter. MS&I a fourni les bibliothèques parallèles nécessaires à la mise en œuvre des traitements fourni par IôDP. Enfin, l'Hôpital de Bologne a défini les spécifications d'un tel système et a effectué les expérimentations.

Le but du projet PARSMED-3D est de mettre en œuvre un environnement opérationnel à base d'un client spécifique et d'une architecture serveur parallèle permettant à des praticiens d'obtenir des images en trois dimensions (3D) en temps réel. Actuellement, l'imagerie 3D est la modalité la plus innovante de l'imagerie médicale ; elle modifiera profondément le diagnostic médical en gynécologie et en cardiologie dans les années à venir.

Les reconstructions 3D sont devenues très courantes dans le milieu médical. La plupart

des systèmes implantant ces reconstructions sont basés soit sur des stations de travail soit sur des composants spécifiques intégrés à la sonde à ultrason. Aucune de ces solutions ne permet d'obtenir des images en temps réel, c'est pourquoi la reconstruction 3D n'est utilisée par les praticiens dans leur exercice quotidien qui nécessite un système rapide et simple à utiliser. Aujourd'hui, il faut à peu près 10 min à une station de travail pour exécuter une reconstruction 3D qu'il faut ajouter aux 30 min d'examen échographique en moyenne. C'est pourquoi la plupart des praticiens ne demandent une reconstruction 3D que pour un nombre très limité d'examens à la fin de la journée lorsque les patients sont partis.

Le but de ce projet est de montrer comment les images 3D peuvent être utilisées par des praticiens en temps réel. Cela rend possible l'utilisation d'un tel système dans des conditions cliniques particulières pour lesquelles des images 3D peuvent élargir les possibilités de diagnostic des praticiens.

Ce but a été atteint par l'intégration de systèmes existants et par le développement d'une solution basée sur des composants standard. Il n'y a pas de développement matériel dans PARSMED-3D. Les principaux travaux qui ont été effectués dans ce projet sont : une bonne interactivité obtenue grâce à un calcul parallèle efficace et une interface temps-réel connectant les clients au serveur.

L'architecture du système est la suivante :

1. l'utilisateur de la sonde à ultrason fait l'acquisition de données analogiques dans un ordinateur standard (Macintosh),
2. la machine parallèle est composée de plusieurs éléments de calcul ; un des ces nœuds est un hôte : c'est le lien entre le Macintosh et le serveur parallèle,
3. le Macintosh transmet par l'intermédiaire d'une connexion TCP-IP les données au nœud hôte puis le nœud hôte les diffuse dans le serveur parallèle,
4. à la fin du calcul, le serveur parallèle renvoie la représentation 3D au Macintosh afin qu'elle soit affichée.

Pour atteindre des performances temps-réel, le projet PARSMED-3D utilise un serveur parallèle basé sur une machine à mémoire distribuée. Le serveur parallèle est une grappe de Pentium Pro 200 MHz standard. Il est composé de quatre nœuds interconnectés par un réseau rapide Myrinet. Pour être plus efficace, une couche de communication optimisée est utilisée : BIP. BIP est une bibliothèque à passage de messages au dessus de Myrinet. Elle atteint 1 Gb/s en débit et moins de 5 μs de latence.

Nous allons tout d'abord décrire le processus complet depuis la sonde à ultrason à la représentation 3D à la fois d'un point de vue fonctionnel et d'un point de vue technique. La deuxième partie donne les mesures de performances de l'installation. La dernière partie décrit les évolutions possibles logicielles et matérielles.

A.1.1 Traitements et représentations

IôDP nous a fourni deux types d'algorithmes d'imagerie que nous devons paralléliser : des traitements (filtres sur les données en entrée) et les représentations bi-dimensionnelles de données tridimensionnelles. Les données en entrée sont une matrice 3D contenant des niveaux de gris que l'on appelle « voxels ».

Les traitements

Les traitements sont appliqués aux données échographiques afin d'améliorer la qualité de la future image résultante, par exemple pour réhausser le contraste ou lisser l'image.

Nous avons parallélisé douze traitements différents :

- le filtre de lissage : filtre de convolution $3 \times 3 \times 3$,
- le filtre de lissage fort : filtre de convolution $5 \times 5 \times 5$,
- le filtre gaussien $5 \times 5 \times 5$,
- le filtre gaussien fort : filtre gaussien $7 \times 7 \times 7$,
- le filtre médian $3 \times 3 \times 3$,
- le filtre Sobel 2D : il calcule le contour de l'image en utilisant l'algorithme de Sobel,
- le filtre Kirsh 2D : calcule le contour de l'image en utilisant l'algorithme de Kirsh,
- le filtre d'érosion $3 \times 3 \times 3$,
- le filtre de dilatation $3 \times 3 \times 3$,
- le filtre inverse,
- le filtre anti-aliasing,
- le filtre de réhaussement utilisant une table de correspondance.

Pour chacun de ces filtres, nous avons redéfini en collaboration avec IôDP leur interface afin qu'elle puisse être utilisée aussi bien en séquentiel qu'en parallèle. La principale modification par rapport à l'interface séquentielle est d'ajouter en paramètres l'indice de début de traitement sur la matrice de données et celui de fin.

Les représentations

Les algorithmes de représentation produisent une représentation en deux dimensions de données en trois dimensions. IôDP fournit les algorithmes suivants :

- la représentation Xray consiste en une projection simple des voxels en additionnant leurs valeurs,
- la représentation avec ombrage est un algorithme Z-buffer suivi d'un algorithme d'ombrage sur l'image obtenue par le Z-buffer,
- la représentation en transparence est un lancer de rayon utilisant certaines valeurs de transparence,
- la représentation par lancer de rayon consiste à lancer un rayon à partir du pixel d'une image à travers le volume de données en trois dimensions. Au long du rayon, les valeurs des voxels sont accumulées pour calculer la valeur du pixel.

A.1.2 Description du processus de rendu volumique

Le processus de rendu volumique met en œuvre le client Macintosh et le serveur parallèle. Le client Macintosh est utilisé pour acquérir les donnés échographiques par l'intermédiaire d'une interface IôDP simplifiée. Les données échographiques sont ensuite transmises au serveur parallèle. Sur le serveur parallèle, à la fois les traitements et les représentations ont été parallélisés. Dès que les données arrivent, les calculs commencent.

Le processus se décompose en quatre étapes :

- T1 : transmission des données échographiques acquises par la sonde à ultrason dans le Macintosh,
- T2 : transmission des données échographiques du Macintosh vers le nœud hôte du serveur parallèle,
- C3 : calcul du filtre et d'une représentation sélectionnés exécutés par plusieurs nœuds de calcul dans le serveur parallèle,
- T4 : transmission de l'image 3D du nœud hôte du serveur parallèle vers le Macintosh.

Les données échographiques sont stockées sous forme d'une matrice tridimensionnelle découpée en coupes bidimensionnelles. On peut distinguer deux types d'étapes : trois étapes de transmission de données et une étape de calcul.

Description des étapes de transmission de données

Cette partie décrit plus en détail les trois étapes de transmission de données décrites ci-dessus dans le but d'évaluer leur temps d'exécution.

T1 : Données échographiques de la sonde vers le Macintosh. Cette étape existait déjà dans le système IôDP original. Comme la sonde à ultrason n'est pas directement connectée au Macintosh, il faut déplacer la matrice de données de la machine à ultrason au Macintosh. La carte d'acquisition sur le Macintosh limite le débit de la transmission. Elle peut tenir un débit de 25 coupes par seconde pour des formats de coupe courant.

T2 : Données échographiques du Macintosh vers le nœud hôte du serveur parallèle. Comme l'interface utilisateur n'est pas directement reliée au serveur parallèle, il faut transmettre la matrice de données au nœud hôte du serveur parallèle pour exécuter les calculs. La connexion entre le Macintosh et le nœud hôte du serveur parallèle est un lien Ethernet bidirectionnel à 100 Mb/s de bande passante. La bande passante réelle se situe autour de 6 Mo/s. Cette transmission utilise l'un des deux canaux bidirectionnels.

T4 : Données de l'image 3D du nœud hôte du serveur parallèle vers le Macintosh Enfin, comme l'image résultat est générée sur le nœud hôte du serveur parallèle, il faut la transmettre au Macintosh en utilisant le même lien Ethernet qu'à l'étape précédente. On utilise donc le deuxième canal bidirectionnel du lien Ethernet de bande passante 6Mo/s.

Description de l'étape de calcul

Le calcul est composé de traitements et représentations. Pour obtenir une image 3D à partir de la matrice de données, le processus exécute toujours un ou plusieurs algorithmes de traitements puis un algorithme de représentation sur la matrice de données. Le choix des algorithmes de traitement dépend du type de l'examen ; ces algorithmes sont sélectionnés à l'initiation du processus.

Parallélisation des calculs. Comme le serveur parallèle est une machine à mémoire distribuée, il faut déterminer la distribution des données et des calculs en fonction du type des calculs.

D'un côté, les traitements sont parfaitement parallèles : chaque nœud peut calculer indépendamment un traitement sur une partie de la matrice de données. En utilisant une distribution linéaire monodimensionnelle des calculs de la matrice de données, chaque nœud ne devra partager qu'une frontière avec le nœud voisin. La profondeur de cette frontière dépend du type du filtre calculé. Pour un filtre $3 \times 3 \times 3$, la profondeur du calcul est trois donc la profondeur de la frontière sera de deux coupes de données.

De l'autre côté, il y a deux catégories de représentations :

- les représentations *Xray* et *ombrage* peuvent être calculées en utilisant une distribution simple de la matrice de données comme une distribution linéaire monodimensionnelle pour générer sur chaque nœud une image résultat partielle
- les représentations *par transparence* et *par lancer de rayons* impliquent une distribution de données sophistiquée. C'est pourquoi nous avons choisi dans un premier temps de dupliquer l'intégralité de la matrice de données sur chacun des nœuds afin qu'aucune représentation ne génère de communication pendant le calcul.

Nous avons donc choisi de partitionner les calculs de la matrice de données suivant une distribution linéaire monodimensionnelle. Par exemple, si la matrice de données est composée de quatre coupes et le serveur parallèle de deux nœuds alors le premier nœud traitera les deux premières coupes et le deuxième nœud les deux dernières coupes.

Si les calculs sont partitionnés suivant une distribution linéaire monodimensionnelle, nous avons choisi de dupliquer l'intégralité de la matrice de données sur chacun des nœuds afin que les représentations *par transparence* et *par lancer de rayons* qui ont des parallélisations délicates ne génèrent de communication pendant le calcul. Nous espérons obtenir de cette façon de meilleures performances qu'avec une stricte distribution de la matrice de données. Il faut néanmoins remarquer que le volume de données standard d'un examen médical standard se situe autour de 12 Mo ce qui permet la duplication des données sur tous les nœuds. Pour un volume de données plus grand, il faudra envisager une distribution différente.

Les précédents choix de parallélisation impliquent trois phases de communications à l'intérieur du serveur parallèle :

- une phase d'acquisition de la matrice de données par chacun des nœuds ; chaque coupe de données est diffusée par le nœud hôte du serveur parallèle vers les autres nœuds du serveur parallèle ;
- une phase de redistribution de données pendant laquelle chaque nœud échange les données qu'il a calculées avec les autres nœuds ;
- une phase de concaténation des images partielles générées par l'algorithme de représentation sur chacun des nœuds afin de générer l'image finale sur le nœud hôte.

Toutes ces phases ne mettent pas en jeu le même volume de données transférées ; elles n'ont donc pas toutes les mêmes temps d'exécution. La phase d'acquisition est conditionnée par la fréquence d'acquisition des coupes par l'utilisateur (25 coupes par seconde). Si la diffusion des coupes dans le serveur parallèle est plus rapide que 25 coupes par seconde, il n'y a aucun surcoût lié à l'acquisition des coupes par les nœuds du serveur parallèle. La phase de redistribution met en jeu la matrice de données entière. En effet, chaque processeur

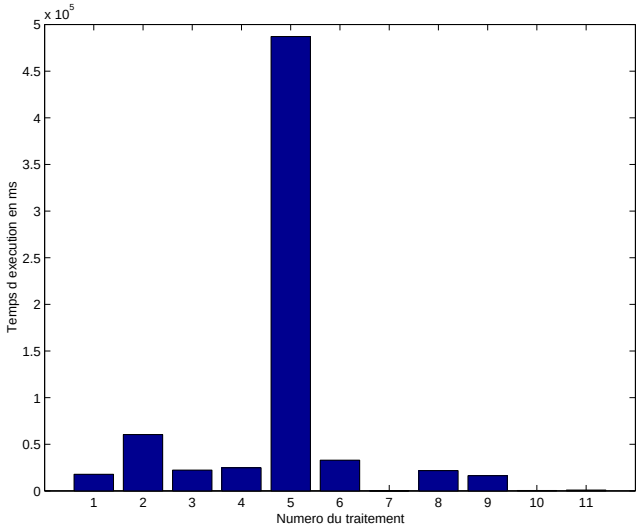


FIG. A.1 – Temps d'exécution séquentiels des différents traitements.

envoie à chaque processeur ses données nouvellement calculées. Cette phase est la phase la plus coûteuse. La troisième phase de communication ne transmet qu'une image 2D ; donc par rapport à la phase précédente, le surcoût lié à cette dernière phase est négligeable.

Exécution des traitements. Les traitements sont des filtres d'interpolation (voir la partie A.1.1) appliqués sur la matrice de données entière.

Le temps d'exécution séquentielle d'un traitement varie énormément d'un traitement à l'autre. Par exemple, le filtre inverse qui consiste simplement à calculer l'inverse de la valeur de chaque voxel est très rapide (400 ms) alors que le filtre médian qui effectue un tri de 27 points pour chaque voxel est très long (487000 ms). L'histogramme à la figure A.1 donne les différents temps d'exécution de chacun des filtres appliqués à un volume de données standard (environ 12 Mo). La représentation par ombrage a un temps d'exécution séquentiel de 1400 ms. Ce temps d'exécution est assez court comparé aux temps d'exécution des traitements.

Dans l'histogramme A.1, les filtres sont représentés par un numéro :

1	Lissage	7	Kirsh
2	Lissage fort	8	Érosion
3	Gaussien	9	Dilatation
4	Gaussien fort	10	Inverse
5	Médian	11	Anti-aliasing
6	Sobel		

Ces temps d'exécutions séquentiels très différents vont influencer énormément le temps d'exécution global du processus. Certains représenteront la plus grande partie du temps d'exécution du processus global.

Exécution des représentations. Nous n'avons parallélisé qu'un seul algorithme de représentation, la représentation par ombrage. Cet algorithme est basé sur la technique du Z-buffer. Il rend la surface du fœtus. Le principal avantage de cet algorithme est que l'interface du code séquentiel se prête bien à une transformation en interface parallèle. Son principal inconvénient est qu'il est peu coûteux en calcul donc son extensibilité sur une machine parallèle n'est pas bonne. La transformation des autres algorithmes de transformation était très compliquée ; c'est pourquoi ces algorithmes n'apparaissent pas dans le système actuel.

A.1.3 Mesures de performances

Nous avons mesuré le temps effectif écoulé depuis l'acquisition des données par la sonde jusqu'à l'affichage de l'image 3D résultat.

Description des expérimentations

Nous avons utilisé pour faire les mesures une configuration matérielle proche de la configuration réelle. Néanmoins un nœud Pentium Pro simule le client Macintosh. Le serveur parallèle est composé de quatre nœuds Pentium Pro. Le client et le serveur sont reliés par un lien Ethernet à canaux bidirectionnels 100-Megabit comme la configuration réelle. À l'intérieur du serveur parallèle, les nœuds sont connectés par un réseau rapide Myrinet comme la configuration réelle.

Un fichier de données comprenant un film d'une échographie de fœtus simule l'acquisition par la sonde effectuée par le praticien. Ce film fait à peu près 12 Mo. L'image résultat est de taille 256×256 pixels.

Transmission des données

La première étape de transmission de données T1 est incompressible.
Le temps d'exécution de la deuxième étape T2 est égal à

$$\frac{12Mo}{6Mo/s} = 2s$$

Le temps d'exécution de l'étape T4 est égal à

$$\frac{256 \times 256o}{6Mo/s} = 0.01s$$

L'étape T4 est négligeable comparée à l'étape T2.

Le processus complet

Nous avons mesuré le temps d'exécution du processus complet incluant la communication liée à l'acquisition des données, un traitement et l'algorithme de représentation et la communication liée à la reconstruction des données 3D. Nous avons fait des mesures pour chacun des traitements.

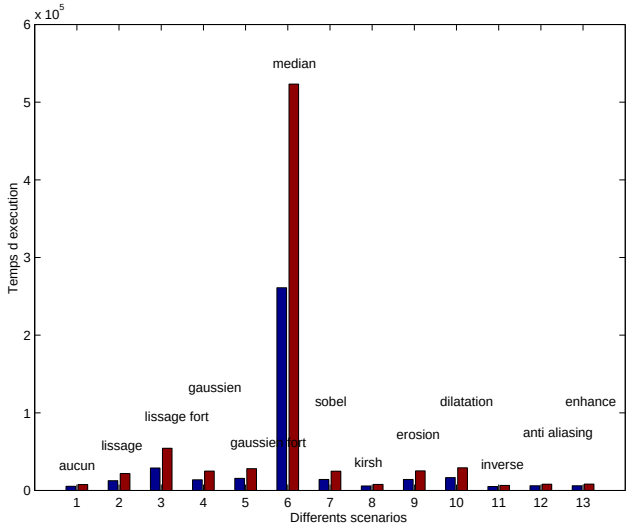


FIG. A.2 – Comparaison des temps d'exécution des différents traitements sur un et sur quatre nœuds.

La figure A.2 donne les temps d'exécution pour chacun des traitements et sans aucun traitement. Comme le montre cette figure, le filtre médian est beaucoup plus long que n'importe quel autre filtre. Le pire des cas est donc le traitement médian. Dans ce cas, le temps d'exécution est à peu près égal à 250s. Les traitements les plus employés sont le lissage et le gaussien. Pour ces traitements, les temps d'exécution sont respectivement 6s et 29s.

Par exemple, exécuter le processus complet en utilisant le traitement de lissage et la représentation par ombrage sur quatre nœuds en incluant le temps d'acquisition initial prend à peu près 12.5s. Ce temps inclut la transmission des données échographiques du Macintosh au serveur parallèle (environ 2s), le calcul du traitement (7.5s) et le calcul de la représentation (3s) et la transmission de l'image résultat du serveur parallèle au Macintosh (environ 0.1s). Nous montrerons par la suite que l'utilisateur ne perçoit qu'à peu près 3s (3s + 0.1s).

Comparaison entre le système séquentiel et le système parallèle

Dans cette partie, nous tentons d'évaluer le gain lié à l'utilisation du système parallèle par rapport au système séquentiel.

Pour faire une telle comparaison, nous avons considéré un dispositif séquentiel composé des mêmes éléments que le système parallèle, c'est-à-dire une sonde à ultrason, un Macintosh et un serveur de calcul mais à la place du serveur parallèle à quatre nœuds, nous n'avons utilisé qu'un seul nœud Pentium Pro.

La figure A.2 donne les performances du nœud de calcul (en rouge) et des quatre nœuds de calcul (en bleu). On peut remarquer que le temps d'exécution de chacun des traitements est plus court sur le système parallèle que sur le système séquentiel. Les performances sont meilleures pour des traitements ayant des temps d'exécution intrinsèquement longs comme le traitement médian et aussi pour les traitements couramment utilisés comme le traitement de lissage et le gaussien.

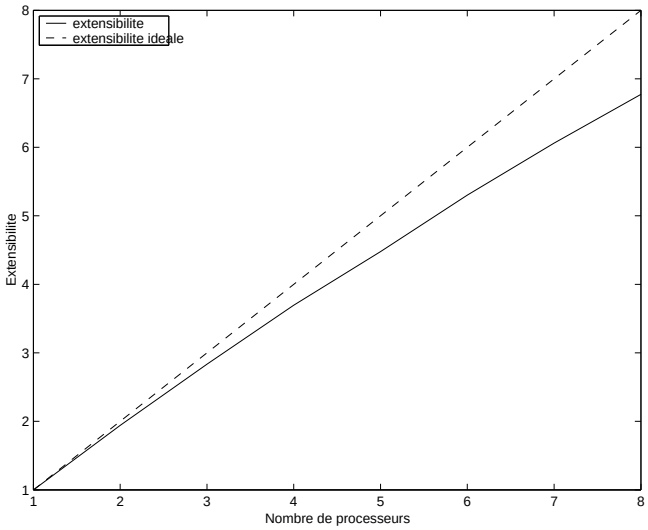


FIG. A.3 – Extensibilité du processus global avec un traitement médian et une représentation par ombrage.

Extensibilité

Nous avons fait trois types de mesures d'extensibilité.

La première mesure prend en compte le processus complet : acquisition des données, calcul d'un traitement puis d'une représentation. La figure A.3 donne l'extensibilité en utilisant le traitement médian avec une représentation par ombrage. Comme le montre cette figure, ce scénario (médian + ombrage) est bien extensible. Son extensibilité est égale à 7 sur 8 processeurs. Ce résultat est bon si l'on considère que l'extensibilité idéale sur huit processeurs est égal à huit. Ce bon résultat est dû au traitement médian qui est très coûteux en calcul et donc intéressant à paralléliser.

La figure A.4 mesure l'extensibilité de la représentation par ombrage.

Bien que la représentation ne soit pas très coûteuse en calcul, l'extensibilité reste assez bonne (elle est égale à 6 sur 8 processeurs). Ceci est dû au fait que cette représentation génère très peu de communications grâce à la duplication des données (seule la phase de concaténation de l'image résultat génère des communications).

La troisième courbe (figure A.5) d'extensibilité compare l'extensibilité de chacun des traitements. Nous avons mesuré le temps d'exécution du traitement et le temps d'exécution de la représentation. Nous n'avons pas pris en compte la phase d'acquisition parce que nous avons l'intention de recouvrir ce temps avec l'acquisition des données échographiques faite par le praticien.

Changement de point de vue

Nous avons cherché à mesurer le temps de génération d'une image lorsque le point de vue change. L'exécution sur quatre nœuds de dix images 3D en changeant le point de vue de cinq degrés à chaque fois en incluant le temps d'acquisition initial prend 34s. Ce temps

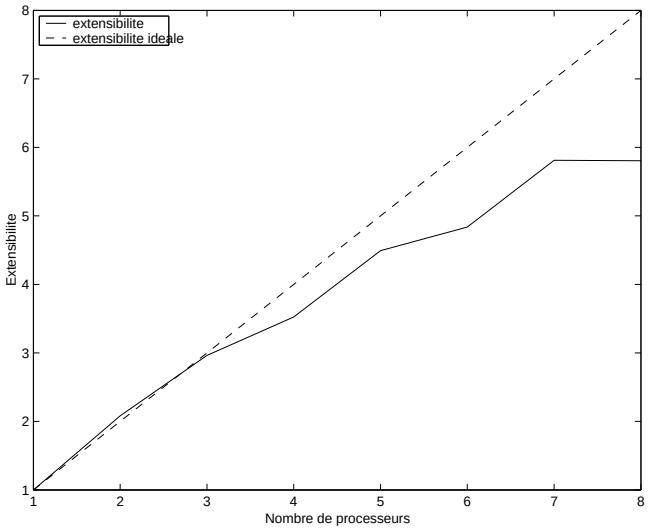


FIG. A.4 – Extensibilité de la représentation par ombrage.

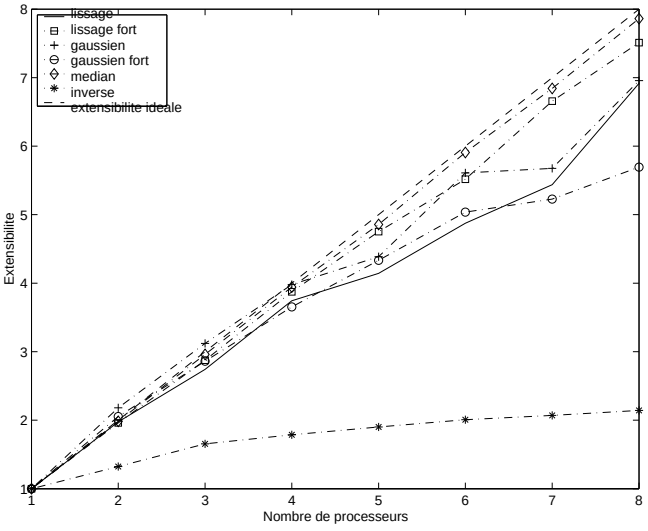


FIG. A.5 – Extensibilité de chacun des traitements et de la représentation par ombrage.

inclut la communication des informations concernant le point de vue du Macintosh vers le nœud hôte du serveur parallèle et la communication de l'image résultat du nœud hôte du serveur parallèle vers le Macintosh. Cela donne en moyenne 3.4s pour une image.

Conclusion sur l'évaluation des performances

Grâce à cette configuration, nous avons pu obtenir une représentation en temps réel de données échographiques tout juste acquises. Nous avons aussi montré dans la partie précédent que même en saturant l'utilisation du système en générant en boucle des représentations 3D, le système fonctionne bien.

Cette nouvelle approche accélère le temps de calcul mais aussi pipeline toutes les étapes nécessaires à l'obtention d'une représentation 3D. En effet, les données en cours d'acquisition sont transmises coupe à coupe au nœud hôte du serveur parallèle. Le nœud hôte du serveur parallèle n'attend pas la fin de la transmission pour commencer à diffuser ces données aux autres nœuds du serveur. Chacun des nœuds commence à calculer dès qu'il a reçu ses données sans attendre que l'ensemble des autres nœuds ait terminé.

L'amélioration des performances en temps d'exécution est visible. L'application séquentielle applique un traitement de lissage en 24s et une représentation par ombrage en 11s ; ce qui nous donne au total 35s de temps d'exécution. Ces 35s sont à comparer au 3.4 s obtenues avec le système parallèle à quatre nœuds.

Ces améliorations sont dues non seulement à la parallélisation mais aussi aux modifications faites dans les codes de traitement et de représentations. La transformation du code nécessaire à la parallélisation a été mise à profit aussi pour accélérer le code séquentiel. Actuellement, le code séquentiel est beaucoup plus rapide que le code original.

A.1.4 Évolutions

Dans cette partie, nous évoquons les évolutions possibles du système PARSMED-3D à la fois au niveau logiciel que matériel.

Évolution du logiciel

Les algorithmes de représentations 3D pourraient être améliorés de deux façons : du point de vue de la qualité de l'image et du point de vue de la rapidité.

L'algorithme de représentation. L'algorithme de représentation par ombrage utilisé dans le système PARSMED n'utilise que des informations surfaciques. Les algorithmes de rendu volumique donnent de meilleurs résultats du point de vue de la qualité de l'image et surtout donnent des images plus riches en information notamment au niveau de la transparence. De plus l'algorithme implanté utilise essentiellement des opérateurs de comparaison. Les algorithmes de rendu volumique plus coûteux en calculs sont eux plus extensibles et donc plus adaptés à la parallélisation.

Amélioration de la qualité de l'image. Le rendu volumique est le processus par lequel une image 2D est créée directement à partir de données volumétriques 3D de telle manière qu'aucune information contenue dans la matrice de données n'est perdue pendant le processus. Par exemple, les données issues de tomographie ne contiennent pas uniquement des informations utiles à la reconstruction de la surface mais aussi des informations sur le contenu des données. Il est donc intéressant d'utiliser le rendu volumique pour obtenir des images possédant plus d'informations.

Nous avons testé deux algorithmes de rendu volumique. Le premier, appelé Voxview, a été développé au LIP par J.J. Li [43] et est aussi basé sur du lancer de rayon. Il a été parallélisé sur une grappe de PC similaire à celle utilisée dans le système PARSMED. Il donne une image de très bonne qualité. Et grâce au rendu volumique, il serait possible de voir en plus e la surface du fœtus, son cœur ou son cerveau, ... En effet, les données ont des informations liées à l'opacité et à la transparence permettant de reconstruire les données et d'observer un élément par transparence. Le principal inconvénient de cet algorithme est qu'il est assez lent.

Le deuxième algorithme de rendu volumique testé est le shear-warp. Cet algorithme est décrit en détail au chapitre 5. Il est non seulement volumique mais aussi rapide grâce à une implantation séquentielle déjà très efficace. De plus, il opère sur des données compressées ce qui permettrait d'utiliser des données à de plus hautes résolutions. La qualité de l'image est aussi très bonne.

Amélioration de l'extensibilité. L'extensibilité d'un algorithme de représentation est essentiellement basée sur la quantité de parallélisme intrinsèque que possède cet algorithme. Par exemple, un algorithme de lancer de rayons n'est pas trivial à paralléliser. C'est pourquoi, bien qu'assez coûteux en calculs, il n'est pas nécessairement très extensible. L'algorithme implanté dans le système PARSMED est parfaitement parallèle mais très peu coûteux en calculs donc son extensibilité n'est pas très bonne non plus.

Par exemple, l'algorithme de rendu volumique « splatting » développé par Johnson et Genetti [35] et parallélisé sur un Cray T3D a une très bonne extensibilité.

Le tableau suivant résume chaque type d'algorithmes et leurs caractéristiques.

Représentation	Catégories	Qualité	Rapidité
Z-buffer	Surfacique	Moyenne	Rapide
Voxview	Volumique	Bonne	Lent
Shear-Warp	Volumique	Bonne	Lent
Splatting	Volumique	Bonne	Lent

D'autres fonctionnalités. L'acquisition des données est extrêmement contrainte par l'algorithme de représentation. En effet, l'algorithme de représentation implique que les images acquises soient parallèles les unes aux autres. Ceci impose deux contraintes. Tout d'abord, le praticien faisant l'échographie doit avoir un mouvement parfait afin que les images soient parallèles. Ensuite, le fœtus scanné ne doit faire aucun mouvement pendant l'échographie ce qui est très rare.

Ces deux contraintes pourraient être résolues en ajoutant au système un algorithme corrigeant les mouvements du praticien et du fœtus. Dans ce cas, même en cas de mouvement

du fœtus, on pourrait obtenir une représentation 3D. Ce type d'algorithmes est bien connu pour d'autres domaines d'applications.

Évolution du matériel

Évolution du serveur parallèle. Le principal avantage du serveur parallèle est sa puissance de calcul mais aussi par exemple une certaine tolérance aux pannes puisque quand un nœud tombe en panne les autres peuvent quand même effectuer le calcul. Cette architecture a un coût de base difficile à justifier pour un seul appareil à ultrasons. Avec ce serveur parallèle dans le système PARSMED, il est possible de connecter plusieurs appareils à ultrasons au même serveur.

Dans beaucoup d'hôpitaux ou cliniques privées, cette configuration en serveur ne peut néanmoins pas être acceptable. Une évolution possible est de diminuer le nombre de nœuds. Mais dans une configuration plus petite, ce serveur peut être remplacé par un nœud SMP.

Un SMP est une architecture multiprocesseurs dans laquelle les processeurs sont tous sur une même carte et partage la même mémoire. Les systèmes SMP fournissent une grande puissance de calcul et une bonne extensibilité.

Évolution du lien entre le serveur parallèle et le Macintosh. Le système PARSMED utilise un lien de communication standard entre le client Macintosh chargé de l'interface graphique et le serveur parallèle de calcul. Par conséquent, beaucoup de temps est perdu dans le dialogue de ces deux entités.

Ce temps n'est pas perçu par l'utilisateur durant l'acquisition de données réelles. Il n'en est pas de même pour quelqu'un souhaitant une représentation de données déjà anciennes stockées sur le disque dur.

Une autre interface réseau standard pourrait être utilisée pour obtenir une accélération notable. Par exemple, le Gigabit Ethernet offre théoriquement une accélération de 100. Beaucoup d'autres technologies peuvent être utilisées comme : FDDI, ATM, Fibre channel, Myrinet.

A.1.5 Conclusion

La partie technique du projet PARSMED-3D consistait à paralléliser des algorithmes de rendus séquentiels fournis par IôDP dans un système parallèle adapté permettant aux médecins d'obtnir des représentations 3D en temps réel. Le système parallèle a permis de faire en sorte que le médecin perçoive une grande amélioration de la qualité d'utilisation. La perception du temps d'exécution est en effet passée de 30 s à 3 s.

Pour atteindre ces performances, nous avons utilisé des technologies de réseau rapide et une couche de communication rapide ainsi que des stratégies de parallélisation optimisées.

L'évolution de ce système PARSMED pourrait être de deux types :

- évolution du couple performance/prix en utilisant un SMP voire une grappe de SMP,
- évolution de la qualité de l'image grâce à la puissance de calcul et de stockage du serveur parallèle.

A.2 Medistar : installation d'une grappe de PC

Le projet Medistar consiste en l'installation d'une grappe de PC pour l'Hôpital de la Pitié-Salpétrière. Cette grappe de PC sera utilisée pour accélérer des applications d'imagerie médicale.

Cette plate-forme doit fournir à l'utilisateur tous les services nécessaires à l'exécution des applications (la possibilité de se loger et accéder à son compte sur tous les nœuds, des compilateurs, des bibliothèques de communication, un réseau rapide de communication destiné aux applications) mais aussi un système d'administration permettant que les interventions de maintenance soient minimales et qu'elles puissent être effectuées à distance par MS&I.

La plate-forme est composée d'un nœud serveur COMPAQ, de 11 nœuds connectés entre eux par un réseau Ethernet et un réseau Myrinet.

Le nœud serveur est un COMPAQ PROLIANT contenant un disque SCSI, une carte Ethernet intégrée sur la carte mère et Linux. Un nœud non serveur est composé d'un PII 350Mhz, une mémoire vive de 128Mo, un disque IDE de 6.4 Go, un lecteur de CDROM, une carte graphique ATI Rage Pro Turbo, mémoire vidéo de 4 Mo, une carte Ethernet eepro100, une carte Myrinet.

L'utilisateur accède à la grappe de PC par le nœud serveur *med0*. Seul ce nœud conserve un clavier, un écran et une souris. Il est aussi accessible par le réseau local du service de l'hôpital. Tous les nœuds clients (*med1* à *med11*) sont accessibles par le nœud serveur.

A.2.1 Installation de base

L'installation de base comprend l'installation du système d'exploitation Linux sur les nœuds clients, l'intégration de ces nœuds au sous-réseau Ethernet, l'accès aux comptes utilisateurs et la possibilité de se loger sur tous les nœuds.

Installation de Linux sur les nœuds clients.

L'ensemble des nœuds serveur (*med0*) et clients (*med1* à *med11*) ont le même noyau Linux. Seuls certains fichiers de configuration diffèrent entre les nœuds clients et le nœud serveur. Tous les nœuds clients sont identiques.

Nous avons créé 3 partitions sur chacun des nœuds : une partition racine, une partition swap et une partition contenant les répertoires utilisateurs.

Àprès avoir procédé à une installation standard, nous avons supprimé les services inutiles au fonctionnement des nœuds comme *sendmail*, *pcmcia*, *lpd*, *smb*, ...

De plus, nous avons supprimé le clavier, l'écran et la souris des nœuds clients. Pour configurer un nœud pour qu'il démarre sans écran ni clavier ni souris, il faut aller modifier des paramètres dans le BIOS.

Intégration des nœuds au sous-réseau Ethernet.

Pour faire communiquer les nœuds sur le sous-réseau Ethernet, il faut configurer la carte Ethernet de chacun de ces nœuds en donnant l'adresse IP du nœud et le masque du réseau,

inclure dans le fichier de configuration de routage et les noms de tous les nœuds du sous-réseau.

Nous avons aussi configuré le nœud serveur *med0* comme passerelle entre le sous-réseau local de la grappe de PC et le sous-réseau local du service de l'hôpital. La configuration en passerelle consiste à faire connaître le nœud dans les deux sous-réseaux (il ne s'agit pas ici de faire du routage entre les deux sous-réseaux).

Pour cela, il faut faire fonctionner les deux cartes Ethernet. Il faut configurer la deuxième carte Ethernet comme appartenant au réseau local du service de l'hôpital. Ainsi, un utilisateur sur le réseau local de l'hôpital pourra se loger sur le nœud serveur et se reloger sur un des nœuds clients à partir de ce nœud. Enfin, les pilotes des deux cartes Ethernet doivent être installés et démarrer automatiquement au démarrage du nœud.

Accès au comptes utilisateurs sur tous les nœuds.

Pour que les comptes utilisateurs soient vus depuis tous les nœuds clients, on effectue le montage des disques du nœud serveur *med 0* stockant ces comptes sur les nœuds clients.

Le disque 0 de *med0* contient les partitions racine et comptes utilisateurs.

Les partitions contenant l'ensemble des applications, les comptes utilisateurs et la partition nécessaire au démarrage à distance des nœuds clients sont exportées vers les nœuds clients. Les cinq autres disques de *med0* sont exportés vers les nœuds clients.

Les disques locaux des nœuds clients sont montés sur la partition racine et sont vus par le serveur mais ils sont détruits à chaque redémarrage. Cela peut être utilisé pour stocker des données temporairement.

Au démarrage de la machine, toutes les opérations effectuées avant l'exportation des partitions devront démarrer sans les partitions montées.

Possibilité de se loger sur tous les nœuds.

Les utilisateurs doivent pouvoir effectuer des *rlogin*, *telnet*, *ftp* et *rsh* en tant qu'utilisateur et en tant que super utilisateur.

Il faut donc ajouter les droits nécessaires pour que le super-utilisateur puisse se loger à distance (*rlogin*, ajouter les droits nécessaires pour que les utilisateurs puissent faire un *rsh* et enfin ajouter les droits nécessaires pour que le super-utilisateur puisse faire un *rsh*.

Et enfin, il faut que les utilisateurs soient connus par tous les nœuds. Ceci est assuré par un système de maintenance centralisé des fichiers d'administration appelé NIS (Network Information Service).

Côté nœud serveur, il faut installer les démons qui permettront de créer la base de données et de répondre aux requêtes des clients.

Côté nœuds clients, il faut installer les démons qui iront interroger la base de données et mettre à jour la base de données locale.

Donc après création d'un utilisateur, il faut maintenant penser à mette à jour la base de données NIS.

A.2.2 Démarrage de la grappe de PC

Le démarrage du nœud serveur s'effectue à partir de son disque local alors que les nœuds clients démarrent à partir du réseau. Ce mécanisme était fait au départ pour que quel que soit l'état du disque, les nœuds clients puissent démarrer et donc être maintenus.

Pour démarrer les nœuds clients à partir du réseau, on utilise les protocole DHCP et TFTP.

Cette procédure utilise le fait que la carte réseau du nœud client a une BOOTPROM. La BOOTPROM est une ROM qui implante le protocole BOOTP. Il faut configurer le nœud client pour qu'il démarre en premier lieu sur le réseau. Au moment du démarrage, le BIOS interroge la BOOTPROM qui commence par diffuser une requête BOOTP sur le réseau. Les machines ayant le démon bootp qui tourne et le bon fichier de configuration répondront. Pour vérifier si le client a une bonne BOOTPROM, on peut débrancher le disque et voir si le client essaye de démarrer en faisant des appels à BOOTP.

TFTP n'étant pas extensible et ne permettant pas de démarrer douze nœuds à la fois à partir d'un même serveur sur un même réseau, on a mis au point un schéma de démarrage en arbre :

med0 démarre 4 nœuds, ensuite *med1* démarre *med5* et *med6* ensuite *med5* démarre *med7* et *med8*, *med7* démarre *med9* et *med10* et *med9* démarre *med11*.

Il faut à peu près 15 minutes pour démarrer toutes les machines.

Pour organiser cela, on utilise le protocole DHCP au-dessus de BOOTP, la partie cliente étant contenue dans la PROM des machines clientes.

Donc il faut écrire le fichier de configuration de DHCP propre à chaque nœud et lancer le démon dhcpd sur le serveur et sur les clients qui deviennent par cette architecture en arbre potentiellement serveur à leur tour.

Il faut penser à démarrer les démons des protocoles DHCP et TFTP sur le nœud serveur et sur les nœuds clients au démarrage de chacun des nœuds.

Le nœud serveur stocke dans le répertoire /tftpboot l'ensemble des fichiers nécessaires aux nœuds clients pour démarrer : une image du noyau Linux que l'on veut installer sur les nœuds distants, une image du système de fichiers, et un ensemble de fichiers permettant à TFTP de savoir quel noyau et quel système de fichiers installer.

Pour des raisons de simplicité, nous avons installé le même noyau sur les nœuds serveur et clients.

On ne peut pas utiliser ce mécanisme sur des nœuds clients sans disque. Il utilise nécessairement de l'espace disque pour fonctionner.

D'autre part, le contenu du disque des nœuds clients sera perdu à chaque redémarrage.

A.2.3 Maintenance minimale

La plupart des actions de maintenance consistent à aller redémarrer un nœud afin de redémarrer les services, réinitialiser des interfaces, ...

Afin d'éviter le plus d'interventions possibles, nous avons mis en place un processus de redémarrage automatique.

On cherche à faire redémarrer automatiquement une nœud client quand le serveur est non accessible. Donc, il y a deux cas :

1. si le nœud client est en train de tourner, c'est le « watchdog » qui fera automatiquement redémarrer ce nœud ;

2. si le nœud client cherche à démarrer alors il redémarrera en boucle jusqu'à ce que le nœud serveur soit accessible.

Redémarrage automatique au démarrage du nœud client. On s'est servi du initrd (Ramdisk d'initialisation). Il s'agit un micro-système permettant d'installer par la suite le système complet. L'idée est d'introduire à l'intérieur de ce micro-système, la commande de redémarrage sur les nœuds clients.

Redémarrage automatique pendant l'exécution du nœud client. Le « watchdog » est un démon qui vérifie périodiquement l'existence d'un fichier, si celui-ci existe alors il le détruit sinon il effectue une action. Dans notre cas, le nœud serveur crée un fichier local au nœud client. Le nœud client détruit périodiquement ce fichier. Si le nœud client n'est plus accessible par le réseau, le fichier n'est pas crée et le nœud client lancera une commande de redémarrage.

Nous avons installé le watchdog sur les machines clientes et sur le serveur afin de conserver des noyaux identiques. Pour installer le watchdog, il faut modifier les paramètres du noyau Linux.

Par défaut, une fois le fichier watchdog ouvert, le démon watchdog redémarrera la nœud client au bout de 60 secondes s'il n'y a pas eu d'écriture dans le fichier. Il faut donc écrire les scripts d'ouverture et d'écriture du démon watchdog. Comme on veut que ce démon watchdog détecte si le serveur a toujours accès au client par le réseau, au démarrage du nœud, le client exécutera un programme C qui ouvre un fichier local watchdog et lance sur le serveur un script en tâche de fond.

L'ouverture et l'écriture du watchdog doivent se faire à l'intérieur du même processus.

Donc, le principe implanté ici est de passer par l'intermédiaire d'un fichier watchdog. Le serveur crée toutes les 30 secondes ce fichier sur le client. Le client, toutes les 30 secondes, vérifie l'existence de ce fichier et s'il existe, le détruit.

A.2.4 Maintenance à distance

Le système a été configuré pour pouvoir être maintenu à distance par MS&I.

Configuration en vue d'une télémaintenance. Pour permettre une télémaintenance, nous avons redirigé la console sur le port série, installé une carte Specialix sur le nœud serveur, émulé un terminal et récupéré la sortie par modem.

Sur le serveur et sur les clients, nous avons redirigé la console sur le port série. Pour cela, il faut modifier le noyau Linux.

Sur le serveur, il y a donc deux démarrages du nœud possibles. Un démarrage renvoyant la sortie standard sur l'écran qui est le mode par défaut et un démarrage envoyant la sortie standard sur la série.

Tous les ports série sur lesquels la console a été redirigée sont connectés à la carte Specialix.

L'installation de la carte Specialix nécessite une modification du noyau Linux du nœud serveur et le lancement de son module au démarrage du nœud serveur.

Sur le serveur et sur les clients, Il faut ensuite créer les fichiers spéciaux des terminaux Specialix.

Sur le serveur et sur les clients, pour se loger à partir de la console, on recrée un fichier spécial console.

Sur le serveur, on a installé deux émulateurs de terminal : seyon et kermit.

Grâce à l'émulateur de terminal, on peut se connecter sur un nœud donné, par exemple, pour se connecter sur *med5* à partir du nœud *med0*, on utilise le fichier spécial X5.

Il faut maintenant récupération la liaison série distante au travers des deux modems : le modem distant branché sur le serveur et notre modem local.

La première chose à faire est de configurer le modem distant durablement en mode de réponse automatique. Il faut ensuite le configurer comme terminal pour qu'il accepte notamment les login.

En local, pour se connecter au modem distant, il n'y a qu'à brancher le modem et se connecter au port série local avec kermit.

Se mettre en mode maintenance. En cas de problème nécessitant l'intervention de MS&I en télémaintenance, l'utilisateur doit se mettre en mode maintenance. Pour se mettre en mode maintenance, il faut vérifier que le modem est allumé puis redémarrer le serveur en mode modem.

A.2.5 Installations matérielles et logicielles spécifiques au parallélisme

Pour accélérer les applications, nous avons installé un réseau rapide Myrinet et une couche de communication rapide : BIP.

Pour intégrer le nœud non serveur au sous-réseau Myrinet, il faut associer à la carte Myricom une interruption non partagée, installer le pilote de la carte, en l'occurence BIP. Ceci consiste en l'installation des binaires de BIP et de ceux relatifs à Myrinet. Il faut alors charger le module correspondant à ce pilote.

Nous avons installé des bibliothèques de communication utiles à la parallélisation : PVM, MPI, BIP, BIP/IP, MPI/BIP et des exemples de test des différentes bibliothèques.

Les exemples servent à utiliser les différentes couches de communication et bibliothèques de communication présentes sur le réseau rapide Myrinet de la machine. Sur le réseau rapide Myrinet, il est possible d'utiliser deux couches de communication différentes : BIP (Basic Interface for Parallelism) et BIP-IP (Internet Protocol). BIP-IP ajoute donc un protocole supplémentaire à BIP, il a donc un temps d'exécution plus long mais est plus fiable. D'autre part, nous avons installé deux bibliothèques de communication différentes : PVM (Parallel Virtual Machine) et MPI (Message Passing Interface).

A.3 CHARM : cache web parallèle

Le projet CHARM consiste en la conception et mise en œuvre d'un système de cache haut-débit fondé sur une architecture en grappe de PC.

Dans le cadre de ce projet, nous allons implanter un système de cache en miroir, un Miroir Haut Débit (MHD), et un système de cache à proprement parler, le Cache Parallèle Optimisé (CPO). Ses deux systèmes sont en effet complémentaires. Le MHD stockera les pages Web pour lesquelles on souhaite un accès rapide comme par exemple un contenu éducatif destiné au milieu scolaire ou encore les pages Web populaires. Le CPO stockera quant à lui les pages les plus accédées par l'ensemble des clients qui ne font pas partie du contenu du MHD.

La première partie a pour but de spécifier le CPO. La deuxième partie est une étude spécifique des méthodes de redistribution de requêtes HTTP sur une grappe de PC. Enfin, la troisième partie est une étude des différents outils de mesures de performances de caches et serveurs Web.

A.3.1 Spécifications

Le but de cette partie est de spécifier le cache parallèle optimisé (CPO). L'introduction du World Wide Web (ou Web) a changé la vision qu'avaient les utilisateurs du réseau en ce qui concerne son accessibilité et sa convivialité. Avec le Web, l'utilisateur est maintenant capable de rechercher et de récupérer toute sorte d'informations à partir du réseau sans avoir aucune connaissance de ce réseau. Du point de vue de l'utilisateur, il importe peu que l'information qu'il recherche, par exemple un séquence vidéo avec du son, soit sur l'ordinateur dans la pièce à côté ou de l'autre côté de la planète. Cela conduit à une croissance énorme du trafic sur le réseau local, national et international. Parce que l'utilisation du Web croît de manière exponentielle, nous pouvons prévoir que le trafic sur ces réseaux croîtra de même de manière exponentielle avec une latence grandissante. Quoi qu'il en soit, l'utilisateur attend une bonne qualité de service avec des temps de réponse modestes. La qualité de service et les temps de réponse peuvent être améliorés en diminuant la charge du réseau. Une manière d'accomplir cela est d'installer un service de cache Web. En effet, cacher signifie migrer des copies d'objets populaires contenus sur un serveur Web vers des endroits plus proches des clients. En général, les utilisateurs clients observent donc des délais plus courts lorsqu'ils font la requête d'un objet, les gestionnaires du réseau observent moins de trafic et les serveurs Web voient leur taux de requêtes baisser.

L'utilisation de caches pose deux problèmes principaux : comment garantir que les caches retournent de bonnes valeurs et comment obtenir la meilleure performance possible. En général, il y a un compromis entre la justesse des valeurs retournées et la performance du cache.

Le CPO est un système optimisé cachant des objets sur le Web. Le CPO permet de fournir une mémoire vive et un espace de stockage extensible afin d'augmenter le nombre de requêtes par seconde ainsi que les performances globales du cache telles que le temps de retrait d'un objet et la diminution du trafic sur le réseau extérieur. Pour accomplir cela, le CPO est constitué d'une grappe de PC possédant chacun un disque et reliés par un réseau rapide fournissant un module de cache et un module de stockage distribué optimisés.

Terminologie

Dans cette partie, un certain nombre de termes sont utilisés pour référer aux rôles joués par les participants et aux objets du service de cache. Les définitions suivantes sont utilisées :

- World Wide Web ou Web : le World Wide Web ou Web relie des serveurs Web qui envoient des pages HTML à des postes dotés d'un navigateur. Le protocole de communication entre les navigateurs et les serveurs est basé sur le principe des hypertextes et est appelé HTTP (HyperText Transfer Protocol). Le langage permettant de décrire les pages Web est le HTML (HyperText Markup Language) ;
- Client et client Web : application établissant des connexions dans le but d'envoyer des requêtes. Ce sont souvent des navigateurs, des éditeurs ou d'autres outils utilisateurs. Un client Web est donc un navigateur capable d'interpréter du HTML et d'envoyer des requêtes en utilisant le protocole HTTP. Serveur et serveur Web Un serveur est une application acceptant des connexions dans le but de répondre à des requêtes en envoyant en retour des réponses. Un serveur Web accepte des connexions HTTP et répond à des requêtes en utilisant HTTP ;
- Serveur origine : serveur Web possédant l'objet original.
- Proxy : programme intermédiaire qui joue le rôle à la fois du serveur et du client dans le but d'effectuer des requêtes à la place d'autres clients. Un proxy doit interpréter et si nécessaire réécrire un message de requête pour l'envoyer. Les proxies sont souvent utilisés comme des portails situés côté client faisant office de pare-feu réseau ou d'application complémentaire ;
- Cache et serveur de cache : programme stockant localement des messages de réponse à des requêtes ainsi qu'au sous-système contrôlant le stockage, la récupération et l'élimination des messages. Un cache stocke les réponses cachables afin de réduire le temps de réponse et la consommation de bande passante réseau dans le futur. Un serveur de cache ou proxy cache est un système récupérant des messages de requêtes clientes dans le but d'y répondre à la place du serveur Web destinataire ;
- Objet : terme générique pour n'importe quel document, image ou n'importe quel type de données sur le Web. Les URL (Uniform Resource Locator) identifient ces objets de manière non ambiguë et peuvent référer à des données disponibles à partir de serveurs Web, FTP, Gopher ou tout autre type de serveurs. Parce que la plupart du contenu actuel du Web est constitué d'images, audio, vidéo ou de fichiers binaires, il est préférable de dire que le cache stocke des objets plutôt que des documents ou des pages ;
- Succès et défaut de cache : un succès de cache signifie qu'une copie valide de l'objet demandé existe dans le cache. Un défaut de cache signifie que l'objet n'existe pas ou n'est plus valide à l'intérieur du cache. Le cache doit alors envoyer la requête résultant en un défaut de cache vers le serveur origine ;

Web et autres protocoles.

Le protocole HTTP est le protocole utilisé pour communiquer entre client Web, serveur Web et serveur de cache mais il n'est pas le seul existant sur l'Internet. Nous évoquerons donc les autres protocoles de récupération d'information sur l'Internet et celui utilisé par le CPO. L'Internet désigne un réseau sur lequel les objets sont accessibles par des adresses Internet incluant dresses IP et URL. Les principaux protocoles d'accès à l'information sur l'Internet sont FTP, Gopher, WAIS et HTTP. La plupart des navigateurs Web sont capables d'émettre des requêtes non seulement vers des serveurs Web en utilisant le protocole HTTP

mais aussi vers des serveurs FTP, Gopher, ...

Le protocole HTTP. Le protocole HTTP est le protocole le plus utilisé sur l'Internet. HTTP (HyperText Transfer Protocol) définit ce qu'on appelle le Web. La plupart des clients sur le Web utilise encore le protocole HTTP version 1.0.

Une requête HTTP est construite en trois parties principales : une méthode de requête, une URL (Uniform Resource Locator) et un ensemble d'en-têtes. La méthode permettant de télécharger un objet identifié par une URL est la méthode GET. Une réponse HTTP est composée d'un code de réponse numérique, d'un ensemble d'en-têtes de réponse et d'un corps de réponse optionnel. Certains champs optionnels des requêtes permettent d'en modifier la sémantique. Parmi ceux-ci, nous nous intéressons particulièrement à ceux qui spécifient des exigences de cohérence sur les objets délivrés :

- le champ 'If-Modified-Since' est souvent utilisé par les caches lorsqu'ils possèdent une version d'un objet et souhaitent s'assurer qu'ils en possèdent la version correcte. Le champ 'If-Modified-Since', associé à la date de dernière modification de l'objet dans le cache, indiquera au serveur de ne transférer l'objet demandé que si une nouvelle version est disponible. Sinon, il renverra uniquement un code de réponse indiquant que l'objet en cache est encore à jour. Ce champ est également utilisé par les navigateurs lorsque l'utilisateur clique sur le bouton 'reload'. Cela permet de vérifier la cohérence de l'objet en économisant son transfert dans le cas où il n'a pas été modifié.
- le champ 'Pragma' permet d'indiquer des options à la requête. L'option 'no-cache' permet d'indiquer que l'on refuse d'accepter un objet venant de l'espace de stockage d'un cache. Lorsqu'un cache reçoit une telle requête, il doit impérativement la transmettre au serveur Web. Il peut cependant mettre en cache l'objet retourné pour usage ultérieur.

Plusieurs des champs optionnels des réponses HTTP sont directement utiles aux caches :

- le champ 'Last-Modified' indique la date de dernière modification de l'objet délivré. Il sert souvent d'indicateur de version ;
- le champ 'Content-Length' indique la taille de l'objet délivré ;
- le champ 'Expires' indique la date à laquelle l'objet devra être considéré comme périmé. Peu de serveurs délivrent des requêtes comprenant ce champ : sauf dans quelques cas particuliers, il est difficile de prévoir à l'avance la date de la prochaine modification d'un document.

La version 1.1 du protocole HTTP fournit le support à des en-têtes de contrôle du cache permettant au client mais aussi au serveur de ne pas se soumettre à l'algorithme du cache par défaut. Une directive intéressante est la directive 'Max-age' (âge maximum) qui permet au client de placer une limite supérieure sur l'âge d'un document, l'âge d'un document étant le temps écoulé depuis que le document a été récupéré sur le serveur origine. Si le document caché est plus vieux que ce que demande le client alors la requête doit être adressée au serveur origine. Cette version permet aussi la gestion de connexions persistantes, du pipeline de requêtes, ...

Il faut cependant remarquer qu'à l'heure actuelle il n'existe que deux navigateurs utilisant HTTP/1.1 (Internet Explorer et Amaya) et aucun des deux n'utilise ces fonctionnalités.

Autres protocoles sur le Web.

- FTP (File Transfer Protocol) : protocole d'échange de fichiers. Comparé à HTTP, la phase d'initialisation d'une session FTP est beaucoup plus lente. Les objets FTP peuvent être cachés de la même manière que des objets HTTP. Néanmoins, le protocole FTP n'a pas de mécanisme facilitant une vérification performante de la validité des objets. Il ne contient aucune information sur les données tels que les en-têtes HTTP contenant le type de fichier ou encore sa dernière date de modification ;
- Gopher : largement supplanté par HTTP, Gopher est un protocole très simple pour le transfert de fichiers textes et de menus et était populaire avant l'apparition du Web. De plus, de la même manière que FTP, il ne possède aucune information sur les données transmises ;
- SSL : d'autres protocoles sont simplement convoyés par le proxy qui ne fait que relayer les données entre le client Web et le serveur Web. Transitant les données, le proxy ne comprend pas nécessairement le protocole proprement dit et ne peut donc pas filtrer les données, effectuer un contrôle d'accès,... Des exemples de protocoles généralement convoyés par les proxies sont les protocoles SSL (Secure Sockets Layer). SSL et les protocoles construits au dessus tels que HTTPS et SNEWS sont gérés complètement différemment de HTTP par les proxies. SSL est un protocole fournissant une session sécurisée du début à la fin entre le client Web et le serveur origine. C'est pourquoi le serveur de cache ne peut pas recevoir une requête du client et agir à la place du client en recevant une réponse et en l'envoyant au client. A la place, le proxy va simplement recevoir une requête lui demandant d'établir un « tunnel » entre le client Web et le serveur origine et ensuite il fera simplement transiter les octets dans les deux sens. Les seules informations connues du serveur de cache sont l'adresse hôte du serveur origine et le numéro de port.
- WAIS : les URL WAIS (Wide Area Information Servers) sont maintenant complètement obsolètes. En pratique, les applications Web sont maintenant routées à travers le Web en utilisant HTTP. Actuellement presque toutes les bases WAIS ont un accès Web avec des menus, permettant de poser une requête.

Protocoles divers.

- SNMP : protocole de gestion d'informations que l'on n'utilisera pas ;
- IP Multicast : beaucoup d'applications récentes de l'Internet concernent des communications où une ou plusieurs sources envoient des données à plusieurs récepteurs. Ces applications vont de l'envoi d'un message concernant la vie d'une entreprise à tous les employés à la conférence audio et vidéo pour des réunions en distant et à la duplication des bases de données et de l'information contenue sur un serveur Web. IP Multicast supporte efficacement ce type de transmission en autorisant plusieurs sources à envoyer une copie unique d'un message à plusieurs destinataires qui veulent explicitement recevoir cette information. IP Multicast est un concept basé sur le récepteur : le récepteur doit se joindre à un groupe particulier d'une session 'multicast' et les messages destinés aux membres de ce groupe leur seront délivrés par l'infrastructure réseau. IP Multicast est une extension du protocole niveau réseau IP standard.
  Pour pouvoir utiliser l'IP Multicast , les nœuds expéditeurs et récepteurs et l'infra-

structure réseau entre eux doivent être positionnés en mode 'multicast', en incluant les routeurs intermédiaires. Les contraintes pour utiliser du IP Multicast natif sur les nœuds finaux sont :
  - support pour la transmission IP Multicast dans la pile de protocole TCP/IP,
  - support logiciel d'IGMP pour communiquer les requêtes pour rejoindre un groupe 'multicast' et recevoir du trafic 'multicast',
  - une carte d'interface réseau capable de filtrer correctement des adresses mappées de la couche réseau IP Multicast ;
- IGMP (Internet Group Management Protocol) : utilisé par les routeurs 'multicast' pour connaître l'existence de membres d'un groupe 'multicast' dans le sous-réseau qui lui est directement rattaché. Ceci est effectué en envoyant des requêtes IGMP auxquelles les hôtes IP appartenant à des groupes répondent. IGMP est implanté au-dessus d'IP et les messages IGMP sont encapsulés dans des datagrammes IP.

Protocoles temps réel multimédia.
- RSVP (Resource ReSerVation Protocol) : protocole réseau qui permet au récepteur de données de demander une qualité de service particulière du début à la fin pour son flux de données. RSVP est utilisé pour installer des réservations sur les ressources réseau. Lorsqu'une application sur une machine hôte demande une qualité de service particulière pour son flux de données, elle utilise RSVP pour délivrer sa requête aux routeurs tout au long des chemins du flux de données. RSVP négocie les paramètres de connexion avec ces routeurs. Si la réservation est mise en place, RSVP maintient les états du hôte et des routeurs pour fournir le service demandé ;
- RTP : protocole de transport temps réel RTP (Realtime Transport Protocol) est un protocole basé sur IP pour fournir du support au transport de données temps réel telles que des flux audio et vidéo. Les services fournis par RTP incluent la reconstruction dans le temps, la détection de perte, sécurité et identification de contenu. RTP a été tout d'abord conçu pour la diffusion personnalisée (multicast) de données temps réel. Il peut être utilisé pour du transport en sens unique tel que pour la vidéo à la demande mais aussi pour des services interactifs comme la téléphonie sur le Web ;
- RTSP : À la place de stocker les grands fichiers de données multimédia, les données multimédia sont transmises sur le réseau en flux. La transmission en flux découpe les données en paquets avec une taille convenable pour la transmission entre le client et le serveur. Les données sont transportées en temps réel au cours de la transmission, décompressées et jouées en pipeline. Le client peut alors jouer le premier paquet pendant qu'il décompresse le second tout en recevant le troisième. Le protocole RTSP (Real Time Streaming Protocol) est un protocole multimédia niveau présentation, basé sur une architecture client/serveur qui permet d'effectuer du contrôle sur les données transmises par flux au dessus du réseau IP. Ce protocole fournit un contrôle à distance pour les flux vidéo et audio comme pause, avance rapide, retour rapide, et positionnement absolu. RTSP a été conçu pour travailler avec des protocoles de plus bas niveau comme RTP et RSVP pour fournir un service de flux complet sur l'Internet ;
- ICMP ;
- SNMP (Simple Network Management Protocol) : standard Internet pour échanger de

l'information concernant la gestion du réseau. Le modèle de gestion réseau de la première version de SNMP est composé de deux acteurs : un agent et un gestionnaire. Une machine agent répond aux requêtes de demande d'information provenant d'une machine gestionnaire. Le gestionnaire peut demander des informations à l'agent ou encore positionner des informations en demandant à l'agent de mettre à jour une partie de l'information contenue dans sa base de données avec la valeur fournie par le gestionnaire. La machine agent fait continuellement tourner un démon appelé snmpd pour écouter les différentes requêtes venant du gestionnaire. La machine gestionnaire utilise souvent une interface graphique sophistiquée qui émet les requêtes SNMP correspondantes. Les informations de gestion fournies par les agents aux gestionnaires sont stockées dans des MIB (Management Information Base).
En ce qui concerne les serveurs Web, le but à terme est de remplacer les nombreux fichiers de configuration qui permettent leur gestion par une 'HTTP-MIB' standard permettant à une même personne d'administrer plusieurs serveurs Web différents.
On pourrait en faire de même avec les serveurs de cache et remplacer notamment le fichier de configuration du CPO par une MIB.

Service de cache

Les acteurs d'un service de cache sont le client Web, le serveur Web et le serveur de cache. Un serveur de cache s'intercale entre un client et un serveur Web afin de stocker des objets près du client. Le serveur de cache joue donc le rôle de serveur vis-à-vis du client et le rôle du client vis-à-vis du serveur. Le serveur de cache peut s'intercaler entre le client Web et le serveur Web de manière transparente ou non. Les différentes interactions possibles entre ces trois acteurs permettront de définir les fonctionnalités nécessaires au serveur de cache.

Transparence du serveur de cache. Du point de vue du client Web, lorsque le serveur de cache est transparent pour le client Web, le client envoie directement ses requêtes aux serveurs Web propriétaires des objets qu'il cherche à récupérer. Le serveur de cache, quant à lui, se trouve sur le chemin des données. Grâce à un dispositif de routage capable de rediriger vers lui tous les paquets HTTP, il examine tous ces paquets. Si le serveur de cache possède l'objet recherché par le client Web, il ne transmet pas au serveur Web la requête du client et envoie l'objet au client à la place du serveur Web.

L'avantage de cette solution est qu'elle ne nécessite aucun investissement de la part des clients. Si l'on dispose d'un tel dispositif de routage, elle est quasiment immédiate. Si l'on décide de cacher de la même manière des données transitant par un autre protocole, il faudra prévoir un dispositif de routage adéquat. En cas de panne du serveur de cache, le dispositif de routage ne redirige plus les paquets HTTP vers ce serveur afin qu'ils puissent néanmoins transiter sur le Web.

Dans le cas où le cache n'est pas transparent pour le client, le client Web est configuré pour s'adresser directement au cache. Le client envoie donc ses requêtes directement au serveur de cache à la place du serveur Web propriétaire de l'objet. Le serveur de cache agira alors comme un client Web vis-à-vis du serveur Web.

L'inconvénient de cette solution est qu'elle nécessite de configurer chacun des clients. Il

existe néanmoins sur les navigateurs des procédures globales de configuration pour l'ensemble des clients administrés sur le même réseau. De plus, il existe des possibilités de fonctionnalités supplémentaires pour les clients ayant connaissance du serveur de cache. Par exemple, le client pourrait préciser l'âge maximum de l'objet qu'il souhaite récupérer ou encore faire de l'équilibrage de charge sur plusieurs caches de son choix en s'adressant à celui momentanément le plus disponible. Cependant, à l'heure actuelle, la plupart des navigateurs existants n'implantent pas d'opérations spécifiques aux caches mise à part la configuration. En cas de panne du cache, il suffit à l'utilisateur de reconfigurer son navigateur pour avoir accès au Web.

Du point de vue du serveur Web, s'il a connaissance du serveur de cache, serveur de cache et serveur Web peuvent interagir afin de gérer le contenu du serveur de cache. On peut en effet choisir un contenu pour le serveur de cache et dupliquer les objets populaires d'un serveur Web. Il s'agit dans ce cas d'un procédé que l'on appelle miroir qui consiste à dupliquer des objets pour qu'ils soient plus proches du client.

L'avantage d'un tel dispositif est de garantir la fraîcheur des objets se trouvant sur le serveur de cache et de pouvoir mettre en oeuvre des mécanismes puissants d'indexation et de gestion de contenu permettant de l'adapter en fonction de la demande de la clientèle.

Les bonnes propriétés d'un client Web. Le client Web doit posséder les propriétés suivantes :

- convivialité : la configuration du client Web précisant quel serveur de cache utiliser doit être facile. Le navigateur Netscape et Internet Explorer dans sa version 4.0 proposent, par exemple, une configuration du cache manuelle avancée permettant d'utiliser un serveur de cache différent pour chaque protocole ou encore de ne pas utiliser de cache pour certaines adresses. De plus, ils proposent une configuration automatique ('Automatic Proxy Configuration') permettant de configurer le cache à partir d'un fichier fourni par l'administrateur système.
- disponibilité : le client Web doit continuer à fonctionner même en cas de panne du serveur Web. Par exemple, si le serveur Web auquel le client Web est connecté tombe en panne, le client Web devrait pouvoir se connecter à un autre serveur de cache précisé dans le fichier de configuration.
- cohérence des données : le client Web devrait pouvoir adapter lui-même le degré de cohérence des données qu'il cherche à récupérer. Par exemple, le bouton 'actualiser' du navigateur Internet Explorer permet au client de préciser qu'il veut la version à jour de l'objet.
- équilibrage de charge : le client Web devrait pouvoir se connecter à plusieurs serveurs de cache.

Les bonnes propriétés d'un serveur Web. Le serveur Web doit posséder les propriétés suivantes :

- cohérence des données : le serveur Web doit être capable de dire si un objet est valide ou non ;
- information des serveurs de cache : un serveur Web devrait être capable d'informer un serveur de cache de la modification d'un document ;

Les bonnes propriétés d'un serveur de cache. Enfin un serveur de cache doit posséder les propriétés suivantes :

- convivialité : le serveur de cache doit générer des messages d'erreur compréhensibles. En outre, il doit y avoir une différence entre les messages d'erreur générés par le serveur de cache et ceux générés par le serveur Web afin d'identifier facilement la provenance de l'erreur.
- disponibilité : le cache ne doit pas être le maillon le plus faible de la chaîne de communication. Si celui-ci est en panne, il faut néanmoins assurer la communication.
- cohérence des données : un utilisateur Web doit pouvoir être capable de faire un compromis entre rapidité d'obtention et validité des données. Le serveur de cache doit donc pouvoir vérifier la validité des données avant de les retourner au client Web.
- temps de réponse : le temps d'accès à un objet pris aléatoirement doit être faible ;
- stratégie de cache : tout ce qui peut être caché doit être caché. Ne pas cacher les objets tels que les objets textes ou les petits objets afin de conserver de la place mémoire n'améliore pas les performances globales du cache et fait diminuer le taux de succès dans le cache. De plus, la duplication et la recherche anticipée de documents ajoutées au cache amélioreraient nettement les performances. La duplication (miroir) de sites populaires améliorerait significativement le taux de succès dans le cache. Rechercher de manière anticipée des objets populaires aurait la même conséquence ;
- limitations de stockage : il est évident qu'un serveur de cache a une capacité de stockage limitée pour gérer l'ensemble des objets cachés et des méta-informations. Lorsque la borne supérieure de la capacité de stockage est atteinte, le système doit continuer à fonctionner correctement. Bien sûr, le serveur de cache ne doit pas imposer de taille maximum à un objet, l'utilisateur ne doit pas voir de différence fonctionnelle entre un client Web avec serveur de cache et un client Web sans serveur de cache ;
- interopérabilité : le serveur de cache doit communiquer avec le client Web et avec le serveur Web en utilisant au moins HTTP sur le port TCP par défaut. De plus, le serveur de cache doit être au moins capable de communiquer avec les autres serveurs de cache en utilisant HTTP.
- équilibrage de charge : un service de cache devrait être capable d'équilibrer la charge entre différents serveurs de cache. Un serveur de cache est un goulot d'étranglement : le nombre de clients qui peuvent s'y connecter est limité, le nombre de requêtes qu'il peut gérer simultanément est limité, ...
- durée de vie des objets : la durée de vie des objets dans le cache doit être adaptée à chaque type d'objet en fonction de sa fréquence de modification.

Les différents types d'objets sur le Web

Sur le Web, à l'heure actuelle on peut trouver une multitude d'objets, des fichiers textes aux vidéos avec sons en passant par l'interrogation de base de données et la réservation de billets à distance. Nous allons donc préciser quels sont les types de d'objets qui peuvent être cachés par le CPO. Les serveurs Web fournissent deux types d'objets : les objets statiques provenant de fichiers stockés sur le serveur Web et des objets dynamiques qui sont construits par des programmes exécutés sur le serveur Web au moment où celui-ci reçoit la requête.

Les objets dynamiques. Il y a plusieurs manières de créer du contenu dynamique dans un document HTML :

- en utilisant de l'HTML parsé par le serveur,
- en utilisant des scripts CGI,
- en utilisant des applications dont les API sont spécifiques au serveur,
- en utilisant des serveurs spécialisés.

Dans la plupart des pages Web contenant des objets dynamiques tels que les pages fournies par des moteurs de recherche, une part significative des données est statique. En effet, considérons, par exemple, un document généré en réponse à une requête concernant l'état d'un stock. Il contient la bannière identifiant le fournisseur de contenu, des en-têtes, de l'information spécifiant formats et fontes et enfin seulement le nom du stock et le prix du stock pour l'entreprise choisie. La bannière, les en-têtes et le format restent les mêmes et la partie dynamique est constituée par le nom et le prix.

D'une part, le goulot d'étranglement n'est plus, en ce qui concerne la récupération de contenu dynamique sur un client, le réseau mais l'exécution de l'application (par exemple du script CGI) sur le serveur Web. En effet, si l'on peut sur un serveur Web récupérer plusieurs centaines de fichiers statiques, en revanche l'exécution du script peut prendre plusieurs secondes de temps CPU. Donc, si le script prend plusieurs secondes pour s'exécuter, le client ne verra pas le temps de réponse s'améliorer en ayant simplement le script exécuté sur un serveur plus proche de lui que le serveur origine. D'autre part, la plupart de ces scripts doit toujours être exécutée sur le serveur origine soit parce que l'application est propriétaire du serveur origine soit parce que l'accès au code source du script par le proxy pourrait constituer une faille de sécurité.

De plus, la plupart des objets dynamiques est contenue à l'intérieur de documents sécurisés authentifiés qui ne doivent pas être pris en compte par les serveurs caches.

Les caches essayent d'éviter au maximum de conserver ces objets Web, qui sont souvent générés d'après le contenu de bases de données sur le serveur Web voire par rapport à un profil individuel des utilisateurs. Malheureusement, ces différents types de d'objets dynamiques ne s'annoncent pas toujours comme tels vis-à-vis des caches. Les objets transportant des informations spécifiques au client sont facilement reconnaissables d'après les en-têtes HTTP (cookies). En revanche, les autres types d'objets dynamiques devraient idéalement être délivrés par les serveurs Web avec le champ 'Expires' correctement positionné (dans le cas d'un document qu'il ne faut pas mettre en cache, on peut le positionner sur l'instant de génération du document). Cependant, tous les serveurs Web ne délivrent pas ces en-têtes, et les caches sont obligés de recourir à des heuristiques plus ou moins efficaces pour identifier les objets dynamiques.

Les objets statiques. Les objets statiques trouvés sur le Web sont de types très variés. Il peut s'agir aussi bien de fichier texte, de fichiers binaires, d'images mais aussi de fichiers contenant de l'audio ou/et notamment de la vidéo, ... Ces fichiers seront amenés intégralement localement chez le client Web et pourront commencer à être visualisés ou joués soit en cours de chargement soit à la fin du chargement. Le serveur de cache peut stocker n'importe quel type de ces fichiers.

Systèmes existants.

Produit	CERN proxy server v3.0	Netscape Proxy Server v1.12	Harvest Caching server v1.4
Rafraîchissement	Vérifie toujours	Vérifie toujours	TTL
Protocoles cachés	HTTP, Gopher, FTP, Wais	HTTP, Gopher, FTP, WAIS	HTTP, Gopher, FTP
HTTP	HEAD, GET, POST : supportés, GET : caché	GET : caché, POST supporté	Tout sauf critères particuliers
HTTPS, SSL	non supportés	supportés	non supportés
Installation	Complexe	Simple	Complexe
Administration	Édition d'un fichier de configuration	pages HTML + scripts CGI	programme CGI
Sécurité	Accès restreint par adresse Internet	Accès restreint par adresse Internet	Accès restreint par adresse Internet
Communication avec clients Web	port TCP par défaut	port TCP par défaut	port TCP par défaut
Hiérarchie	oui	oui	oui
Architecture	UFS à partir des URL	structure de répertoire à trois niveaux basés sur MD5	cache en mémoire virtuelle
Équilibrage de charge sur plusieurs disques	Non	Non	circulaire
Documents sécurisés	non cachés	non cachés	non cachés
Actualiser une page	toujours récupérée à partir du serveur origine	GET If-modified-since	toujours récupérée à partir du serveur origine
CGI-scripts	résultats non cachés	résultats non cachés	configurable
Java applets	cachées	cachées	cachées

Exemples de performances. Le Wisconsin Proxy Benchmark mesure : la latence, le taux de succès par fichier et le taux de succès en octets.

Les tests de 1998 sur des requêtes HTTP (uniquement 1.0) sont effectués pour : CERN, Apache et Squid sur une grappe de 40 stations de travail Sun Sparc 20 avec chacune 2 CPU 66 Mhz, 64 Mo de mémoire, 2 disques de 1 Go interconnectées par des liens 100-BaseT Ethernet. Le taux de succès maximum observé a été de 25%. La latence moyenne varie en fonction du nombre de clients. Même en négligeant le délai du réseau vers le serveur Web, si le nombre de clients devient important, le temps d'un succès de cache devient plus long que le temps d'un défaut de cache. La latence varie entre 2 et 12s. Les différences d'implantations des différents

serveurs de cache influent sur la charge du CPU et des disques ainsi que sur le taux de succès, la latence client et les erreurs de connexions clientes. Le principal goulot d'étranglement est le disque. Néanmoins, l'expérience montre que le fait de rajouter un disque n'augmente pas significativement les performances de Squid.

Le Cache Parallèle Optimisé

Le CPO est un serveur de cache en grappe. La grappe est composée de seize nœuds ayant chacun un disque. Le but du CPO est de fournir une grande puissance de cache destinée aux utilisateurs d'un fournisseur d'accès Internet. La puissance de cache est obtenue grâce à une mémoire vive extensible en prenant en compte la mémoire de chacun des nœuds de la grappe et un espace de stockage lui aussi extensible en considérant l'ensemble des disques de la grappe comme un seul grand espace de stockage.

Le CPO est un système composé d'une grappe à base de composants standards tournant un système d'exploitation standard et d'un logiciel de serveur de cache.

Serveur de cache du CPO. Le logiciel de serveur de cache du CPO est constitué des modules suivants :

- module de distribution des requêtes,
- module de cache,
- module de gestion de cohérence intercache,
- module de gestion du stockage distribué.

Le module de distribution des requêtes dirige la requête vers un des nœuds de la grappe de manière à équilibrer globalement la charge entre les différents nœuds du CPO. Le module de cache prend alors en charge la requête du client Web et détermine si le CPO possède l'objet ou non. Si le CPO ne possède pas l'objet alors il adresse une requête au serveur origine (ou à un autre serveur de cache). Le CPO récupère alors l'objet et le transmet au module de gestion de la cohérence intercache. Ce module permet de gérer l'appartenance des objets aux différents nœuds de la grappe et leur mise à jour. Enfin le module de stockage distribué stocke l'objet. Si le CPO possède l'objet alors le module de gestion de cohérence intercache demande au module de stockage l'objet et le transmet au module de cache. Le module de cache transmet enfin un réponse au client Web.

Le protocole de communication entre le client Web, le serveur Web et le serveur de cache est HTTP. Malheureusement, à sa conception le protocole HTTP n'avait pas intégré le mécanisme de cache (voir le chapitre plus loin). C'est seulement à partir de la version 1.1 qu'il existe des mécanismes spécifiques liés à la gestion des caches encore peu utilisés. C'est pourquoi les interactions entre client Web et CPO se borneront aux possibilités actuelles des navigateurs : possibilité d'émettre des requêtes et de recevoir des réponses et possibilité de préciser que l'on veut la dernière version de l'objet.

Le CPO est un serveur de cache transparent pour le serveur Web. Son contenu est donc aléatoire, dépendant uniquement des objets téléchargés par les clients Web. Il devra ainsi prendre en charge la gestion de son contenu et la gestion de la cohérence des objets qu'il possède avec ceux des serveurs origines.

Nous reprenons par la suite en détail chacun des différents modules pour définir leurs fonctionnalités.

Description de la grappe de PC. L'architecture en grappe du CPO garantit son extensibilité. Elle permet de délivrer un grand nombre de requêtes en simultané, la montée en charge est possible par un simple ajout de modules matériels, sans nécessité de duplication de contenu et sans remettre en cause l'investissement initial.

La grappe de PC se base au niveau matériel et logiciel sur des composants standards donc fiables. Elle est composée de 16 processeurs Pentium II à 450 MHz répartis dans 8 serveurs bi-processeurs, mémoire totale de 4 Go, chaque processeur disposant de 256 Mo, d'un réseau hautes performances interne du système sera basé sur la technologie Myrinet.

Spécifications du client

Le module de distribution des requêtes est chargé de distribuer les requêtes sur les différents nœuds du serveur de cache en grappe. Pour des raisons de performance du serveur de cache, il est important que le nombre de connexions pour chaque requête client ne soit pas augmenté. D'autre part, il est préférable qu'un tel mécanisme de distribution de requêtes assure une distribution équilibrée de la charge du serveur sur l'ensemble des machines.

Il est préférable pour un serveur de cache en grappe de n'avoir qu'un seul nom de serveur public pour l'ensemble de la grappe afin que quelles que soient les modifications de configuration qui puissent intervenir à l'intérieur de la grappe, elles n'affectent pas les applications des clients. Si un seul nom est public pour l'ensemble du serveur en grappe, alors toutes les requêtes des machines clientes connectées à ce serveur de cache sont envoyées grâce à ce nom à la grappe et un mécanisme de distribution des requêtes assure que chaque requête est traitée par une machine à l'intérieur de la grappe.

Le protocole HTTP est un protocole niveau application construit pour le Web. Il est basé sur une architecture client/serveur. Pour chaque requête HTTP, une connexion TCP/IP est établie entre un client et un serveur. Le client envoie une requête au serveur qui répond en renvoyant l'information demandée. La connexion TCP/IP est alors fermée. Pour établir une connexion TCP, le navigateur demande au DNS (Domain Name Service) de faire correspondre le nom du serveur auquel on veut se connecter avec une adresse IP. L'adresse IP et le numéro de port (par défaut 80) forment l'adresse niveau transport du processus serveur HTTP.

Basé sur ce protocole, il y a plusieurs catégories de mécanismes de distribution de requêtes possibles (voir la partie A.3.2) :

1. l'approche code HTTP de redirection : un serveur occupé répond avec un code HTTP de redirection et fournit une nouvelle adresse de serveur à laquelle le client pourra soumettre à nouveau sa requête. Ceci est défini dans le protocole HTTP. Cette approche est transparente à l'utilisateur mais augmente le nombre de connexions nécessaires à la requête d'un client. Elle augmente donc le temps de réponse et le trafic sur le réseau.
2. l'approche côté client : nécessite que chaque client ait une certaine connaissance de la grappe pour que les requêtes soient envoyées aux différentes adresses du serveur de manière équilibrée. Par exemple, lorsqu'un navigateur Netscape est utilisé pour accéder

à la page d'accueil de Netscape, le navigateur tire un nombre N aléatoirement compris entre 1 et 32 et se connecte au serveur de l'adresse wwwN.netscape.com. Cette solution est pour nous difficilement réalisable.

Une autre approche côté client consiste à lancer une appliquette côté client. Cette appliquette collecte des informations notamment sur la charge des différents nœuds de la grappe et transmet la requête du client vers le nœud approprié suivant cette information. La tolérance aux pannes et l'extensibilité sont implantées côté client. Cette approche n'est bien sûr pas transparente pour le client et accroît le trafic sur le réseau.

3. l'approche DNS côté serveur : quand le serveur DNS reçoit une demande de translation d'adresse, il sélectionne l'adresse IP d'un des serveurs de manière circulaire. L'inconvénient de cette approche est que le résultat de la translation nom vers adresse peut être caché dans des DNS intermédiaires, de futurs clients auront donc toujours accès à l'adresse du même nœud sans bénéficier de la distribution circulaire. De plus, en cas de panne de ce nœud, des clients vont continuer à essayer d'accéder au nœud serveur en panne en utilisant l'adresse cachée.
4. l'approche une unique image IP côté serveur : pour le client, le serveur en grappe n'a qu'une seule adresse IP. L'approche routeur TCP [16] rend publique l'adresse d'un routeur côté serveur. Chaque requête client est alors envoyée au routeur qui redirige la requête vers le serveur approprié en fonction de caractéristiques de charge. Cette redistribution est effectuée en changeant l'adresse IP destination de chaque paquet IP entrant avec celle du nœud serveur sélectionné. De plus, le nœud serveur sélectionné doit mettre l'adresse du routeur à la place de sa propre adresse comme adresse source dans les paquets de réponse. L'avantage de cette approche est qu'elle n'augmente pas le nombre de connexions TCP et qu'elle est totalement transparente au client. Cependant, elle a pour inconvénient de modifier le code du noyau de chaque nœud serveur dans la grappe pour implanter le mécanisme de changement d'adresse au niveau TCP/IP.

   L'approche translation d'adresse réseau est une autre approche où le serveur n'a qu'une seule adresse IP vu du client. Dans cette approche, tous les changements d'adresse sont effectués par le routeur côté serveur. Comparée à l'approche routeur TCP, cette approche a pour avantage d'être transparente pour les nœuds du serveur [9].

Spécifications du module de cache

Le module de cache gère les fonctionnalités de cache à proprement parler. Il doit comprendre les différents modules suivants :

- un proxy Web : il reçoit des requêtes en provenance des clients Web et émet des requêtes vers les serveurs Web ;
- un filtre qui décide pour chaque message HTTP s'il concerne ou non le cache. Par exemple, c'est lui qui décidera de ne pas mettre en cache les objets générés dynamiquement ;
- un module de remplacement qui a pour charge de déterminer quel(s) objet(s) il convient d'expulser du cache quand il n'y a pas suffisamment d'espace libre pour stocker un nouvel objet. Cette fonctionnalité peut être réalisée par un grand nombre d'algorithmes

parmi lesquels nous retiendrons FIFO (l'objet le plus ancien dans le cache est expulsé), LRU (l'objet qui n'a pas été accédé depuis le plus longtemps est expulsé), et SIZE (le plus gros objet en cache est expulsé) ;
- un module de maintien de cohérence qui est chargé de faire en sorte que les objets délivrés aux clients soient à jour par rapport à leur version originale sur le serveur Web. Les algorithmes de cohérence forte sont généralement trop coûteux à mettre en oeuvre pour un système à grande échelle comme le Web. Les algorithmes les plus fréquemment utilisés sont les TTL et Alex. TTL affecte à chaque objet une durée de vie fixée à l'avance, et l'expulse à expiration de la durée de vie. Alex est une variante de TTL, qui affecte des durées de vie différentes suivant les objets : la durée de vie est proportionnelle à l'âge de l'objet. Ainsi, un objet modifié une heure auparavant héritera d'une durée de vie brève, tandis qu'un objet qui n'a pas été modifié depuis un mois héritera d'une durée de vie plus importante ;
- un module de coopération qui permet à un cache de tirer parti de la présence d'autres caches proches. Module de gestion de cohérence intercache Le module de gestion de cohérence intercache est chargé de garantir la cohérence des données stockées sur les différents nœuds du serveur de cache. Pour cela, il faut définir précisément quelle est la nature des données stockées par les différents nœuds ;
- un module de gestion du stockage distribué (NFS, BPFS) assurant le stockage sur les disques des différents nœuds des objets en cache.

Visualisation des performances

Pour pouvoir optimiser un cache Web, un préalable indispensable est de pouvoir comparer plusieurs configurations différentes. Les serveurs de cache ont trois types d'effets que l'on peut vouloir évaluer : ils améliorent les temps d'accès, ils diminuent le trafic réseau mais ils peuvent introduire des incohérences.

L'héritage des caches mémoire et des caches disque a fait que la première métrique sur laquelle les concepteurs et les utilisateurs de cache se sont penchés est le taux de succès. Comme son nom l'indique le taux de succès évalue la proportion de requêtes qui ont pu être résolues avec succès localement par le cache au lieu de contacter le serveur. Cette métrique a l'avantage d'être très simple à mesurer. Cependant, tous les succès n'ont pas la valeur, un succès sur un gros objet devrait compter davantage qu'un succès sur un petit objet et tous les échecs n'ont pas le même coût : un échec sur un objet facile à obtenir de la source est moins dommageable pour la performance du système qu'un échec sur un objet plus long à obtenir.

La caractérisation des temps d'accès intéresse en premier lieu l'utilisateur. C'est en espérant améliorer ses temps d'accès que l'utilisateur décide de configurer son navigateur pour passer par un serveur de cache. On peut distinguer au moins deux temps d'accès différents :
- la latence : c'est la durée entre le début de la requête et la réception du premier octet de la réponse ;
- le temps de retrait : c'est la durée entre le début de la requête et la réception du dernier octet de la réponse. Il est relativement facile de mesurer la latence ou le temps de retrait d'un document. En revanche, il est assez compliqué de comparer des jeux de

requêtes entiers entre eux.

La mesure du trafic réseau intéresse en premier lieu l'administrateur réseau. Un opérateur s'intéressera d'une part au trafic qu'il doit acheminer sur son infrastructure de réseau puis au trafic qui sort sur des liaisons à grand coût telles des connexions internationales.

Enfin, l'inconvénient de l'utilisation d'un serveur de cache est l'incohérence qu'il introduit. La métrique la plus simple pour évaluer l'incohérence est le taux d'objets périmés délivrés par le cache. En revanche, elle est difficile à mettre en oeuvre : il faut pouvoir en effet connaître la version à jour pour pouvoir décider si la version délivrée est périmée ou non.

A.3.2 Différentes techniques d'équilibrage de charge sur une grappe de PC

Dans cette partie, nous étudions spécifiquement les différentes techniques existantes d'équilibrage de charge sur une grappe de PC. L'équilibrage de charge consiste à distribuer des requêtes HTTP de manière équilibrée sur les différents nœuds d'une grappe de PC. Nous distinguons deux types de mécanismes : dans le premier cas, le client émettant les requêtes choisit le nœud serveur le moins chargé, dans le deuxième cas, le serveur lui-même équilibre les requêtes.

Approche côté client

C'est l'application côté client qui effectue des requêtes de manière équilibrée aux différents serveurs. Deux projets implantant cette solution existent :

- Netscape : Netscape Navigator choisit de manière aléatoire parmi les adresses `www1.netscape.com` à `www32.netscape.com` une adresse pour se connecter à la page d'accueil de Netscape. C'est une solution embarquée qui n'est pas récupérable.
- SmartClient (Berkeley) [74] : pour chaque service, une appliquette Java tourne dans le navigateur côté client qui se connectera au serveur le moins chargé. Ceci fonctionne notamment pour Netscape Navigator et Internet Explorer. Ce projet implante les services Telnet et FTP. HTTP est en projet.

Approche côté serveur

Une adresse IP unique est visible de l'extérieur pour une grappe de PC. Il existe plusieurs mécanismes permettant d'équilibrer les requêtes : un mécanisme basé sur le DNS, un mécanisme directement basé sur le protocole HTTP, un mécanisme de niveau application et enfin un mécanisme de niveau réseau.

DNS tournant. Quand un serveur DNS reçoit une requête, l'adresse IP de l'un des serveurs est fournie de manière circulaire. Ce principe est décrit dans [15].

L'avantage de cette solution est la transparence à l'utilisateur. Son inconvénient est que l'équilibrage de charge aléatoire pour des requêtes éventuellement très différentes.

Les différents projets implantant cette solution sont :

- NCSA project [37]
- DEC [47].

Redirection HTTP. Le serveur occupé répond un code HTTP-redirection et fournit une nouvelle adresse de serveur auquel le client pourra à nouveau soumettre sa requête.

L'avantage de cette solution est la transparence à l'utilisateur. Ses inconvénients sont nombreux :

- latence côté client doublée pour les petits messages,
- un unique point de panne,
- le serveur faisant la redirection peut être surchargé,
- seulement disponible pour HTTP,
- augmentation du trafic réseau.

Un projet implante cette solution : le projet SWEB [10].

Niveau application. C'est une application côté serveur qui se charge de rediriger les requêtes HTTP vers un serveur HTTP appartenant à la grappe. L'avantage de cette solution est la transparence à l'utilisateur. Les inconvénients sont qu'il faut deux connexions TCP pour chaque requête, qu'il y a un point unique de panne et le serveur redirigeant les requêtes peut être surchargé. Un projet implante cette solution : le projet Reverse proxy (Apache team) [25]. Il s'agit d'une modification du serveur HTTP Apache (version 1.3b6) pour qu'il retransmette les requêtes vers un autre serveur.

Niveau IP. La redirection des requêtes se fait au niveau IP côté serveur.

- NAT (Network Address Translator) : le routeur change l'adresse IP destination des paquets entrants avec l'adresse d'un nœud du serveur. Le nœud sélectionné envoie les paquets de réponse au routeur. Le routeur change l'adresse IP source des paquets de réponse avec sa propre adresse et envoie les données réponse au client. Ce système est décrit dans [36]. Les avantages de cette solution sont la transparence à l'utilisateur, la transparence côté serveur et enfin la possibilité pour n'importe quel système d'exploitation supportant TCP/IP de tourner côté serveur. Les inconvénients sont : la surcharge éventuelle du routeur et un unique point de panne. Les projets implantant cette solution sont : Magic Router (Berkeley) [9] et CISCO Local Director [16].
- NAT modifié : le routeur change l'adresse IP destination des paquets entrants avec l'adresse d'un nœud Le nœud sélectionné met l'adresse du routeur à la place de sa propre adresse comme adresse source des paquets IP réponse. Le nœud sélectionné envoie directement les réponses au client sans passer par le routeur. Ceci nécessite un changement d'adresse au niveau de la couche TCP/IP fait par le nœud sélectionné ce qui implique une modification du noyau du nœud. Les projets utilisant cette solution sont :
  - IBM TCP Router [32]
  - Approche hybride DNS tournant plus NAT modifié par IBM sur SP-2 [57] (tolérance aux pannes assez poussée).
- IP Tunneling : le routeur encapsule le datagramme IP et envoie la datagramme IP encapsulé au nœud sélectionné. Le nœud sélectionné décapsule le datagramme reçu et renvoie directement la réponse au client. Ce système est décrit dans [50, 61]. Les avantages de cette solution sont la transparence à l'utilisateur et la charge faible sur le routeur. L'inconvénient est que les nœuds serveurs doivent pouvoir supporter le

protocole IP Tunneling. Un projet implante cette solution : le projet Linux Virtual Server Project. Ce projet fournit deux implantations différentes : NAT et IP Tunneling. Des algorithmes d'équilibrage de charge sont implantés : circulaire et en fonction du nombre de connexions actives sur les serveurs. Il y a aussi un mécanisme de tolérance aux pannes.

A.3.3 Mesures de performances de serveurs et caches Web

Les mesures correctes de performances d'un serveur et d'un cache Web et les identifications de goulot d'étranglement dans ces systèmes sont souvent impossibles dans un environnement d'exécution réel à cause des nombreux facteurs qui sont susceptibles d'affecter leur comportement. De nombreux outils de mesures de performances ont alors fait leur apparition.

Il y a deux stratégies de mesures de performances principales. La première consiste à simuler les conditions d'un environnement d'exécution réel aussi proche de la réalité possible et de mesurer alors les performances. La deuxième stratégie consiste à effectuer les mesures de performance dans des conditions particulières qui ne cherchent pas à ressembler à la réalité. Cette deuxième stratégie a pour avantage de pas avoir à évaluer les caractéristiques d'un environnement d'exécution réel méconnu et permet d'identifier les goulots d'étranglement.

La plupart des outils de mesure de performance auront donc des fonctionnalités de simulation de conditions extérieures. On distingue néanmoins les outils de mesures de performances pour les serveurs Web et les outils de mesures pour caches Web. En effet, pour tester un serveur Web, l'outil de mesure de performances simulera la plupart du temps un client Web. En revanche, pour tester un cache Web, l'outil de mesure de performances fournira en général non seulement un simulateur de client Web mais aussi un simulateur de serveur Web. La plupart des outils existants mesurent donc les performances soit d'un serveur soit d'un cache Web mais rarement les deux d'autant que les critères d'évaluation peuvent être assez différents. Dans le cas d'un cache Web, on s'intéressera particulièrement au gain en trafic réseau apporté par la présence du cache. Pour un serveur Web, en revanche, il sera, par exemple, important de connaître le temps de retrait d'une page à contenu dynamique, en général non cachable par les caches Web. Il existe néanmoins des critères de performances communs aux deux systèmes. Un critère commun aux caches comme aux serveurs Web est, par exemple, le temps de retrait d'un document.

Si les outils de mesures de performances sont nombreux, ils sont bien souvent peu documentés et attachés à un outil. Un serveur ou un cache Web est, en effet, souvent fourni avec un outil de mesures de performances par son fabricant. Les métriques utilisées sont généralement assez floues et sont donc difficilement comparables entre elles.

Nous distinguerons donc par la suite les critères d'évaluation et les outils existants pour les serveurs Web de ceux pour les caches Web. Il s'agit donc dans un premier temps de bien définir les critères à évaluer et enfin les conditions dans lesquelles ceux-ci seront évalués.

Critères d'évaluation.

Il est possible de chercher à évaluer toutes les fonctionnalités d'un serveur et d'un cache Web. Le détail de ces fonctionnalités peut déjà constituer pour ces systèmes un premier critère que l'on pourra ramener par exemple pour un produit commercial au prix. En effet,

un serveur Web est par définition constitué de quelques fonctionnalités simples telles que la réception de connexion HTTP, l'analyse d'une requête HTTP, l'établissement d'une réponse HTTP et l'envoi de cette réponse. Dans la pratique, les serveurs Web supportent en général beaucoup plus de fonctionnalités. Pour évaluer un serveur, on peut donc être amené à se poser des questions telles que : Ce serveur Web supporte-t-il d'autres protocoles que le protocole HTTP ? Quels types de contenus dynamiques est-il capable de générer ? Supporte-t-il les connexions persistantes ? De la même manière, un cache Web peut supporter d'autres protocoles que le protocole HTTP, cacher du contenu dynamique, supporter des connexions persistantes et plus techniquement, il peut être capable ou non d'interpréter les nouveaux champs du protocole HTTP 1.1 tels que les dates d'expiration ou les en-têtes de contrôle de cache.

Si l'on considère les principales fonctionnalités d'un serveur et d'un cache Web, il est possible de distinguer les deux critères principaux suivants :

- le temps de retrait d'un document doit être faible à la fois dans le cas du serveur Web que dans le cas du cache Web ;
- le serveur Web et le cache Web doivent avoir la possibilité de fournir beaucoup de données en peu de temps que le volume de données soit lié au nombre de documents fournis mais aussi à la taille des documents fournis ; en plus, pour un cache Web,
- il faut pouvoir mesurer le gain en efficacité lié à la présence du cache en mesurant par exemple le taux de succès.

De plus, comme la nature du trafic Web est très fluctuante avec des pics d'affluence dans la journée ou liés à certains événements, l'adaptativité d'un serveur et d'un cache Web à plusieurs conditions de charge telles que la charge de la machine, la congestion du réseau, le type de requêtes entrantes et le nombre de connexions simultanées est un critère de qualité important.

Dans la suite de cette partie, nous allons détailler ces différents critères de mesures pour les serveurs Web dans un premier temps puis pour les caches Web.

Critères d'évaluations d'un serveur Web. Du point de vue d'un utilisateur, les principaux critères de performance pour un serveur Web sont le temps d'accès à un document et la disponibilité du serveur. Pour un administrateur, le critère principal sera le débit du serveur Web afin de trouver un dimensionnement du serveur approprié au trafic qu'il génère.

- Temps d'accès ou temps de réponse :
  Ce critère est très important pour un utilisateur. Il joue aussi un rôle déterminant dans la popularité d'un serveur Web. En effet, quel que soit l'intérêt du contenu du serveur Web, si le temps de réponse de ce serveur est très long, il viendra à bout de la patience de l'utilisateur qui ira alors chercher son information ailleurs.
  Le temps d'accès peut être caractérisé par deux métriques :
  - la latence qui est la durée entre le début de la requête et la réception du premier octet de la réponse ; cette latence mesurera donc le temps d'établissement de la connexion, l'analyse de la requête, la recherche de l'objet et l'établissement de la réponse ;
  - le temps de retrait qui est la durée entre le début de la requête et la réception du dernier octet de la réponse.

  Dans la plupart des outils de mesures de performances, le critère mesuré est le temps de

retrait. Pourtant, la latence permet d'identifier un goulot d'étranglement potentiel plus facilement que le temps de retrait. Elle mesure essentiellement la capacité du serveur Web à gérer les requêtes HTTP en fonction du trafic. En général, plus le nombre de requêtes est important, plus la latence est élevée. Le temps de retrait est une mesure plus globale prenant notamment en compte la latence et le débit.
Il y a de plus plusieurs manières de mesurer un temps de réponse. Le temps de réponse moyen est souvent biaisé par des valeurs très fortes et des valeurs très faibles. De plus, le temps de réponse varie en fonction de la taille du serveur Web, du nombre de requêtes en cours, ..

- Disponibilité : ce critère est aussi très important pour un utilisateur. En effet, un des succès du Web est de fournir l'information souhaitée tout de suite (temps de réponse) mais aussi tout le temps. Si le serveur fournissant les billets de train est indisponible une fois sur deux, ce service ne pourra pas être rendu correctement. La principale source d'indisponibilité d'un serveur Web est la difficulté de surmonter la montée en charge. Pour un grand nombre de requêtes HTTP, la latence deviendra telle que le serveur Web sera considéré hors service ou pire le serveur tombera réellement en panne. Ainsi l'adaptativité du serveur Web à un grand nombre de requêtes est un critère de performances très important.
- Débit : ce critère est très important dans le dimensionnement d'un serveur Web. Le débit mesure le nombre d'octets par seconde que le serveur Web est capable de fournir en sortie. Ainsi, en considérant une taille de document à délivrer moyenne, on pourra en déduire le nombre de requêtes par seconde que le serveur Web est capable de traiter.

Critères d'évaluations d'un cache Web. Du point de vue d'un utilisateur, les principaux critères de performance d'un cache Web sont, comme pour le serveur Web, le temps d'accès et la disponibilité. En revanche, pour mesurer le gain réel lié à l'existence d'un cache Web, mesurer le gain en temps d'accès d'une solution avec cache Web par rapport à une solution sans cache Web est une meilleure métrique. Un critère supplémentaire important du point de vue de l'utilisateur mais difficilement quantifiable par celui-ci est l'incohérence introduite dans les documents rapatriés par l'existence du cache. Un utilisateur peut en effet rapatrier une page Web qui n'est pas la dernière version de la page Web souhaitée. Pour un administrateur, le critère de mesures de performances important est le gain en trafic réseau.

- Taux de succès : ce critère est très souvent celui mesuré pour vérifier l'efficacité d'un cache Web. Il évalue la proportion de requêtes qui ont pu être résolues avec succès par le cache localement au lieu de contacter le serveur. Pourtant, un succès ne signifie pas nécessairement un gain en temps d'accès important ou un gain en trafic réseau important. Il est préférable d'avoir un succès pour un gros document difficilement accessible que pour un petit document facilement accessible. Ainsi si le taux de succès concerne majoritairement des petits documents facilement accessibles alors que les échecs ne concernent que des gros documents difficilement accessibles, les performances globales du cache ne seront guère améliorées par un taux de succès élevé. Ce critère est facile à mesurer et est par conséquent mesuré par la plupart des outils de mesures de performances ;
- Gain en temps d'accès : le gain en temps d'accès consiste à comparer le temps d'accès

à un document à un instant donné par l'intermédiaire d'un cache Web et le temps d'accès à ce document à ce même instant donné sans dispositif de cache.
Si le temps d'accès est assez facile à mesurer, le gain en temps d'accès est difficilement mesurable. Il faut pouvoir connaître le temps d'accès à cet objet sans utiliser de cache Web dans ces mêmes conditions. Ces conditions malheureusement sont assez difficilement définissables : selon l'instant, l'objet peut être récupéré plus ou moins facilement sans système de cache : disponibilité du serveur, encombrement du réseau, ...

- Gain en trafic réseau : un administrateur système s'intéressera sans doute en premier lieu au trafic sortant de son organisation sur des réseaux publics. Un opérateur s'intéressera d'une part au trafic qu'il doit acheminer sur son infrastructure réseau puis au trafic qui sort sur des liaisons coûteuses telles que les liaisons internationales.
  Une bonne métrique pour mesurer l'impact d'un cache sur le réseau est le ratio du trafic entrant dans le cache par le trafic sortant. Cette métrique est appelée 'Byte-Hit rate' et est équivalente à un taux de succès pondéré par la taille d'objets. Le but est que le cache Web possède le maximum de données intéressant l'utilisateur en local donc qu'il y ait le moins possible de données entrant dans le cache et le plus possible de données sortant du cache.
- Incohérence introduite : le contenu d'un document est susceptible de varier de manière plus ou moins fréquente. Le plus simple et le plus efficace pour un cache est de considérer que tous les documents sont invariants. Ceci amènera un fort degré d'incohérence et ne satisfera pas les utilisateurs. Le plus juste est de vérifier à chaque requête sur un document contenu à l'intérieur du cache sa validité auprès du serveur Web propriétaire du document. Ceci risque de faire baisser notablement le gain en temps d'accès et en trafic réseau du cache. La plupart des caches Web implantent un compromis entre ces deux solutions. C'est pourquoi il peut être important de mesurer l'incohérence introduite par la présence du cache. La métrique la plus simple pour évaluer l'incohérence est le taux de documents périmés mesurés par le cache Web. Cette métrique est difficile à mesurer : il faut pouvoir connaître la version à jour pour pouvoir décider si la version délivrée est périmée ou non.

Outils

Comme nous l'avons vu, de nombreux outils existent afin de mesurer les performances d'un serveur Web ou d'un cache Web. Tous les outils ne sont pas basés sur le même principe et ne mesurent pas les mêmes grandeurs. Dans cette partie, nous nous intéresserons tout d'abord au principe général de fonctionnement d'un outil de mesures de performances d'un serveur Web puis au principe de fonctionnement d'un tel outil pour un cache Web. Nous détaillerons ensuite les quelques outils existants disponibles et leurs caractéristiques tout d'abord pour les serveurs Web puis pour les caches Web.

Pour mesurer les performances d'un serveur Web, il faut un ensemble de machines connectées au serveur Web sur lesquelles tournent un ou plusieurs processus clients chargés de simuler le comportement de véritables utilisateurs sur le Web. La principale difficulté d'un outil de mesures de performances réside dans la simulation de l'utilisateur réel. De plus, l'outil doit fournir non seulement les processus clients mais aussi un ensemble de mesures effectuées

pendant la simulation sur le comportement du serveur. Ces mesures seront plus ou moins significatives selon le processus client fourni. En effet si le but des mesures est de caractériser le comportement du serveur Web dans un environnement réel, il faudra que les processus clients implantent au mieux le comportement d'un utilisateur réel. Si, en revanche, le but est d'identifier un certain nombre de goulots d'étranglement potentiels dans l'implantation du serveur Web, alors les processus clients pourront être l'origine de situations exceptionnelles sur le serveur Web telles qu'une très grand nombre de requêtes, des requêtes concernant de très gros documents, ...

Pour mesurer les performances d'un cache Web, il faut non seulement fournir des processus clients simulant des utilisateurs réels mais aussi des processus serveurs simulant des serveurs Web réels. Ces processus serveurs simulent en général en même temps le trafic du réseau s'étendant du cache Web au serveur Web. De même que pour les outils de mesures sur les serveurs Web, cet outil devra en plus fournir des mesures qui seront plus ou moins pertinentes selon les processus clients fournis mais aussi selon les processus serveurs fournis.

Mesures de performances d'un serveur Web. La plupart des fournisseurs de serveurs Web et de caches Web possèdent leur propre outil de mesures de performances. Ces outils sont généralement difficilement récupérables ou peu documentés. Il est difficile d'interpréter les résultats pour trouver les goulots d'étranglement d'un serveur Web quelconque. Pour mesurer les performances d'un serveur Web, il existe quelques outils connus.

Les outils de mesures de performances les plus utilisés sont WebStone et SPECWeb. Les autres outils développés essentiellement par des universitaires cherchent particulièrement à améliorer les caractéristiques de simulation des clients Web.

- WebStone est le premier outil de mesures de performances de serveur Web. Il a été développé par Silicon Graphics (`http ://www.mindcraft.com/webstone/`). L'approche générale de WebStone est de créer un processus maître qui envoie et collecte des données provenant de plusieurs processus clients. Le maître crée plusieurs clients qui ouvrent de manière répétée une connexion avec le serveur et téléchargent un fichier. WebStone mesure principalement le débit en envoyant le plus de requêtes possibles en un temps court.
- SPECWeb (`http ://www.specbench.org/osg/web99/`) est aussi construit sur un modèle client serveur mais plutôt que de créer des clients qui génèrent de manière aléatoire le plus de requêtes possibles, SPECWeb peut échelonner la charge afin de tester le serveur en faisant varier la charge. SPECWeb cherche à montrer les résultats d'un accroissement de charge sur le serveur jusqu'à ce qu'il ne puisse plus la gérer du tout. SPECWeb peut contrôler la génération de charge beaucoup mieux que WebStone parce qu'il contrôle le comportement des clients et non pas seulement leur nombre. La métrique reportée par SPECWeb est le nombre de connexions que le serveur peut exécuter par seconde. Cette métrique ne rend pas compte de la capacité du serveur Web d'envoyer des octets sur le réseau. Cet outil est payant, de 200 à 800$.
- DBench (`http ://207.86.247.147/html/dbench.html`) simule des utilisateurs à partir de fichiers d'enregistrement réels. Le D est pour dynamique dans le but de souligner la différence principale entre DBench et les autres outils de mesures de performances : la possibilité de mesurer les performances de serveurs fournissant un contenu généré

dynamiquement. Il peut simuler différents types de trafic des requêtes GET conditionnelle au trafic CGI ainsi que des requêtes générant des erreurs. Cet outil de mesures de performances rapporte les mesures de débit et le nombre de connexions par seconde mais aussi la latence ressentie au niveau du client.

- WebBench (`http ://www.zdnet.com/zdbop/webbench/webbench.html`) a une architecture basée sur celle de WebStone. Il utilise des processus légers clients pour faire monter la charge du serveur jusqu'à une valeur seuil maximum. Il donne deux valeurs de mesures : le nombre de requêtes par seconde et le débit.
- httperf (`http ://www.hpl.hp.com/personal/David_Mosberger/httperf.html`) a été développé dans les laboratoires de Hewlett Packard. C'est plutôt un outil d'intégration d'outils d'évaluation de performances. Il peut générer et soutenir une charge sur le serveur, supporte le protocole HTTP 1.1, peut être étendu à d'autres générateurs de charge et intégrer des mesures de performance.
- Apache Jmeter (`http ://java.apache.org/jmeter/`) est un outil de mesures de performances fourni par Apache et tout à fait disponible. Il sert à tester le serveur à la fois avec un contenu qu'avec un contenu dynamique. Il peut aussi être utilisé pour générer une charge sur le serveur et simuler une charge sur le réseau. Il possède une interface graphique en Java. Son principal inconvénient est d'être très peu documenté donc les résultats récupérés sont difficilement interprétables.
- Apache Benchmark est aussi un outil de mesures de performances fourni par Apache lors de son installation. La commande est ab. Il permet de demander un document sur un serveur Web une ou plusieurs fois et donne un certain nombre de statistiques telles que le nombre d'octets transférés, le temps écoulé, le débit de transfert et le nombre de requêtes par seconde obtenu. Son utilisation est très simple et permet de se faire une idée rapide sur les performances générales d'un serveur.
- WAGON (`http ://www-sop.inria.fr/mistral/personnel/Zhen.Liu/wagon.html`) est un outil fabriqué par l'INRIA et n'est pas encore disponible. Il possède a priori un générateur de trafic Web, un analyseur de requêtes et un outil de mesures de performances accessibles au travers d'une interface graphique évoluée.

Mesures de performances d'un cache Web. Pour mesurer les performances d'un cache Web, des sessions de tests effectués par un organisme ne fabriquant pas de caches Web se sont mises en place (`http ://bakeoff.ircache.net/`). Ces sessions se basent sur l'outil Polygraph. La première session (premier 'bake-off') a eu lieu le 15 mars 1999 et a permis de tester les caches de IBM, InfoLibria, Network Appliance, Novell, NLANR/Squid et University of Wisconsin/Peregrine.

- Polygraph (`http ://polygraph.ircache.net/`) est capable de générer des charges variées sur le cache Web, de simuler un environnement réel ou de tester un composant particulier du cache. Il est composé d'un programme client générant un flux de requêtes HTTP avec des propriétés particulières et d'un programme serveur générant un flux de réponses HTTP. Il fournit beaucoup de statistiques telles que le taux de réponse, le temps de réponse , le taux de succès et le nombre d'erreurs de transaction.
- Wisconsin Proxy benchmark (`http ://www.cs.wisc.edu/ cao/wpb1.0.html`) est un outil créé à l'université du Wisconsin. Il fournit un processus maître coordonnant les

actions des processus clients, des processus clients et des processus serveurs. Il donne principalement la latence vue par le client moyenne et le taux de succès.

Comparaison des outils en fonctions des différents critères.

	WebStone	SPECWeb	Dbench	Polygraph	Wisconsin Proxy Benchmark
simulateur serveur	//	//	//	OK	OK
simulateur client	ne gère que le nombre de clients	peut moduler la charge de chaque client	à partir de fichiers d'enregistrements réels	OK	OK
débit serveur	OK		OK		
connexions/s	OK	OK	OK	OK	
temps de réponse				OK	
débit client	OK				
latence client	OK		OK	OK	OK
taux de succès				OK	OK

Projet CHARM

Dans le cadre du projet CHARM, nous évaluerons de manière séparée les deux systèmes MHD et CPO.

Le MHD se comporte du point de vue de l'utilisateur comme un serveur Web classique en ce qui concerne les requêtes HTTP. Les performances du MHD seront donc mesurées grâce aux outils de mesures de performances existants.

Dans le cadre du projet CHARM, nous chercherons à évaluer : le gain en confort de l'utilisateur et les performances intrinsèques du serveur Web intégré dans le MHD.

Le gain en confort de l'utilisateur est surtout influencé par la pertinence du contenu du miroir. En effet, il est possible de supposer que le retrait en local d'un document sera dans tous les cas plus confortable que le retrait de ce même document sur un serveur distant. Ceci suppose bien sûr que le MHD supporte de manière correcte la montée en charge d'un point de vue nombre d'utilisateurs et nombre de requêtes.

C'est pourquoi pour mesurer les performances intrinsèques MHD, nous nous focaliserons particulièrement sur les performances de celui-ci dans des cas de surcharges en utilisateurs et en nombre de requêtes. Le MHD devra de plus assurer que dans des conditions moyennes d'utilisation, les mesures classiques telles que temps de retrait et débit seront bonnes.

Pour effectuer l'ensemble de ces mesures, la simulation au plus proche d'un utilisateur réel n'est pas nécessaire. L'utilisation d'un outil simple comme Apache Benchmark tournant sur plusieurs clients permet de mesurer un débit et un temps de retrait des divers cas d'utilisation.

De plus, une panne du MHD ne doit pas empêcher l'utilisateur d'accéder aux pages qu'il souhaite de manière transparente quitte à devoir accéder au serveur distant propriétaire de la page Web.

Du point de vue du fournisseur d'accès, il est possible de mesurer le gain en bande passante sur le réseau. Ce gain est néanmoins lui aussi essentiellement influencé par la pertinence du contenu du MHD.

Les mesures de performances du CPO seront effectuées grâce à l'outil Polygraph. Les sources de cet outil sont disponibles sur le Web. Cet outil paraît être le plus complet et

le plus utilisé. De plus, l'utilisation de cet outil permettra des comparaisons aisées avec les autres caches testés lors du premier 'bake-off'.

Dans le cadre du projet CHARM, nous chercherons ici à évaluer : le gain en confort du point de vue de l'utilisateur, les gains qu'amènent intrinsèquement les optimisations du CPO par rapport à un cache Web classique, et enfin le gain du point de vue du fournisseur d'accès Internet.

Le gain en confort de l'utilisateur consiste essentiellement à mesurer la disponibilité du service Internet, l'amélioration des temps de réponse et enfin la mesure de l'incohérence introduite dans les documents par le cache. En effet, une panne du cache Web ne doit en aucun cas interdire à un utilisateur l'accès au Web. L'amélioration des temps de réponse moyens pourra se mesurer au travers d'un enregistrement de jeux de tests enregistrés lors de sessions utilisateurs de même que la mesure de l'incohérence introduite dans les documents par le cache.

Le CPO est un cache parallèle optimisé. Les principales optimisations de ce cache sont :

- une grande capacité de stockage impliquant un taux de succès élevé,
- un grand nombre de serveurs impliquant un nombre de requêtes en cours de traitement élevé,
- un système de stockage distribué efficace impliquant un débit élevé,
- des communications à l'intérieur du serveur optimisée impliquant un nombre de requêtes par seconde important.

L'ensemble de ces mesures sont fournies par l'outil Polygraph.

En plus des mesures de performances proposées par Polygraph, la mesure du ratio du trafic entrant dans le cache par le trafic sortant est intéressante dans le cadre du projet CHARM. Elle permettra de donner une idée du gain apporté par le cache au niveau du fournisseur d'accès Internet.

A.4 Conclusion

Par l'intermédiaire de ces trois projets, j'ai acquis une connaissance diversifiée dans le domaine des grappes de PC. J'ai, tout d'abord, grâce au projet Medistar, installé un système permettant de développer des applications parallèles sur une grappe de PC.

Le projet PARSMED-3D, très directement lié avec mes travaux sur l'application Shear-Warp, m'a permis d'expérimenter sur le terrain divers algorithmes de reconstruction en trois dimensions d'images médicales et de démontrer la faisabilité d'une solution à base de grappe de PC dans le but d'obtenir des images en temps réel.

Enfin, le projet CHARM m'a permis d'étudier, dans le cadre des technologies liées à Internet, les possibilités fournies par des architectures à base de grappe de PC, qu'il s'agisse de systèmes faiblement couplés comprenant un seul point d'entrée redirigeant les requêtes ou de systèmes fortement couplés comme l'architecture du cache web optimisé.

Bibliographie

[1] T. S. Abdelrahman. « Latency hiding on COMA multiprocessors ». *Journal of Supercomputing* (1996). `http://www.eecg.toronto.edu/~tsa/jsuper96.ps`.

[2] T. S. Abdelrahman et G. Liu. « Overlap of Computation and Communication on Shared-Memory Networks-of-Workstations ». Dans *Parallel and Distributed Computing Pratices* (1999), volume 2, pp. 145–153.

[3] V. S. Adve. « Flowchart and Pipelines for Sweep3D ». `http://www.cs.rice.edu/~adve/sweep3D/flowchart.html`, 1998.

[4] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. J. Teller et M. K. Vernon. « POEMS : End-to-End Performance Design of Large Parallel Adaptive Computational Systems ». Dans *Proceedings of the First International Workshop on Software and Performance* (1998). `ftp://pcl.cs.ucla.edu/pub/papers/tse.poems.ps`.

[5] V. S. Adve et R. Sakellariou. « Application Representations for Multi-Paradigm Performance Modeling of Large-Scale Parallel Scientific Codes ». *International Journal of High-Performance and Scientific Applications* **14**, numéro 4 (2000). `http://www-sal.cs.uiuc.edu/~vadve/Papers/ijhsa.nasaspissue.ps`.

[6] A. Alexandrov, M. Ionescu, K. Schauser et C. Scheiman. « LogGP : Incorporating Long Messages into the LogP Model — One Step Closer Towards a Realistic Model for Parallel Computation ». Dans *7th Annual Symposium on Parallel Algorithms and Architecture* (Juillet 1995). `ftp://ftp.cs.ucsb.edu/pub/papers/schauser/95-spaa.ps`.

[7] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent et X. S. Li. « Analysis, Tuning and Comparison of Two General Sparse Solvers for Distributed Memory Computers ». Rapport technique, Lawrence Berkeley National Laboratory LBNL-45992, 2000. `ftp://ftp.enseeiht.fr/pub/numerique/AMESTOY/N7_RT_99_2.ps.Z`.

[8] M. B. Amin, A. Grama et V. Singh. « Fast Volume Rendering Using an Efficient Parallel Formulation of the Shear-Warp Algorithm ». Dans *Proceedings 1995 Parallel Rendering Symposium* (1995). `ftp://ftp.cs.umn.edu/dept/users/kumar/parallel_rendering.ps`.

[9] E. Anderson, D. Patterson et E. Brewer. « The Magicrouter, an Application of Fast Packet Interposing ». Document électronique, 1996. `http://www.cs.berkeley.edu/~eanders/projects/magicrouter/`.

[10] D. Andresen, T. Yang, V. Holmedahl et O. Ibarra. « SWEB : Towards a Scalable WWW Server on MultiComputers ». Dans *Proceedings of the 10th International Parallel Pro-*

cessing Symposium (IPPS'96) , Hawaii (1996). http://www.cs.ucsb.edu/~dandrese/papers/index.html.

[11] O. Aumage, G. Mercier et R. Namyst. « MPICH/Madeleine : a True Multi-Protocol MPI for High Performance Networks ». Rapport technique, INRIA, 2000. http://www.ens-lyon.fr/~bouge/Biblio/Aumage/AumMerNam01IPDPS2001.ps.gz.

[12] S. B. Baden et S. J. Fink. « Communication overlap in multi-tier parallel algorithms ». Dans *Proceedings of SuperComputing'98* (1998). ftp://ftp.cs.ucsd.edu/pub/scg/papers/1998/k2_sc98.ps.gz.

[13] R. Bagrodia, E. Deelman, S. Docy et T. Phan. « Performance Prediction of Large Parallel Applications using Parallel Simulations ». Dans *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Mai 1999). ftp://pcl.cs.ucla.edu/pub/papers/ppopp99.ps.

[14] T. Brandes et F. Desprez. « Implementing Pipelined Computation and Communication in an HPF Compiler ». Dans *Europar'96* (1996). http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/PAPERS/LOCCS/loccs_hpf.p%s.gz.

[15] T. Brisco. « DNS Support for Load Balancing ». RFC 1794, 1995. ftp://ftp.math.utah.edu/pub/rfc/rfc1794.txt.

[16] « Cisco Systems Scaling the Internet Web Servers ». Document électronique, 1997. http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/index.shtml.

[17] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian et T. von Eicken. « LogP : Towards a Realistic Model of Parallel Computation ». Dans *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Mai 1993). http://www.scs.carleton.ca/~bsp/literatur/C/C_logP.html.

[18] R. F. V. der Wijngaart, S. R. Sarukkai et P. Mehra. « The Effect of Interrupts on Software Pipeline Execution on Message-Passing Architectures ». Dans *International Conference on Supercomputing'96* (1996), pp. 189–196. http://www.nas.nasa.gov/~wijngaar/papers/sci.ps.

[19] F. Desprez. « A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures ». Dans *Conference on Programming Environments for Massively Parallel Distributed Systems* (1994). http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/PAPERS/LOCCS/TI.ps.gz.

[20] F. Desprez, S. Domas et B. Tourancheau. « Optimization of an LU Factorization Routine Using Communication/Computation Overlap ». Rapport technique, INRIA, 1997. ftp://ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR3094.ps.gz.

[21] F. Desprez et M. Garbey. « Numerical Simulation of a Combustion Problem on a Paragon Machine ». Dans *Proceedings of Parallel Computing* (1995), volume 21, pp. 495–508. http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/ParaMAp/PAPERS/PaWN1_com%bust.ps.gz.

[22] F. Desprez, P. Ramet et J. Roman. « Optimal Grain Size Computation for Pipelined Algorithms ». Dans *Europar'96* (1996). http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/PAPERS/LOCCS/opium.ps.gz%.

[23] F. Desprez et B. Tourancheau. « LOCCS : Low Overhead Communication and Computation Subroutines ». Rapport technique, LIP - ENS Lyon, 1992. `http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/PAPERS/LOCCS/loccs.ps.gz%`.

[24] H. C. Edwards. « MMPI : Asynchronous Message Management for the Message-Passing Interface ». Rapport technique TICAM Report 96-44, The University of Texas at Austin, 1996. `ftp://www.ticam.utexas.edu/pub/carter/mmpi.ps.Z`.

[25] R. S. Engelschall. « Load Balancing Your Web Site, Practical Approaches for Distributing HTTP Traffic ». Document électronique, 1998. `http://www.webtechniques.com/archives/1998/05/engelschall/`.

[26] P. Geoffray, L. Prylli et B. Tourancheau. « BIP-SMP : High Performance Message Passing over a Cluster of Commodity SMPs ». Dans *International Conference on Supercomputing'99* (1999). `http://lhpca.univ-lyon1.fr/~pgeoffra/Publi/Supercomputing99.ps.gz`.

[27] W. Gropp. « Tutorial on MPI : The Message-Passing Interface ». Document électronique. `http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/src3/jacobi/C/main.ht%ml`.

[28] S. Hiranandani, K. Kennedy et C.-W. Tseng. « Evaluation of Compiler Optimizations for Fortran D on MIMD ». Dans *Proceedings of the 1992 ACM International Conference on Supercomputing* (1992). `ftp://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR91196.ps.gz`.

[29] A. Hoisie, O. Lubeck et H. Wasserman. « Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters ». Dans *Frontiers of Massively Parallel Computing* (1999). `http://www.c3.lanl.gov/cic19/teams/par_arch/Web_papers/Wavefront/Wavefr%ont.html`.

[30] A. Hoisie, O. Lubeck et H. Wasserman. « Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications ». *The International Journal of High Performance Computing Applications* (2000). `http://www.c3.lanl.gov/cic19/teams/par_arch/pubs/IJHPC.pdf`.

[31] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini et H. Alme. « A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs ». Dans *Proceedings of International Conference of Parallel Processing 2000* (Août 2000). `http://www.c3.lanl.gov/cic19/teams/par_arch/pubs/icpp.pdf`.

[32] G. D. Hunta, G. S. Goldszmidta, R. P. Kinga, et R. Mukherjeeb. « IBM Network Dispatcher : a connection router for scalable Internet services ». Dans *Proceedings of the 7th International World Wide Web Conference* (1998). `http://decweb.ethz.ch/WWW7/1899/com1899.htm`.

[33] K. Ishizaki, H. Komatsu et T. Nakatani. « A Loop Transformation Algorithm for Communication Overlapping ». *International Journal of Parallel Programming* **28**, numéro 2 (2000).

[34] D. Jiang et J. P. Singh. « Improving Parallel Shear-Warp Rendering on Shared Address Space Multiprocessors ». Rapport technique, Princeton University, 1997.

[35] G. Johnson et J. Genetti. « Medical Diagnosis Using the Cray T3D ». Dans *1995 Spring Proceedings (Cray User Group)* (1995), pp. 70–77.

[36] C. C. K. Egevang et N. P. Francis. « The IP Network Address Translator (NAT) ». RFC 1631, 1994. `ftp://ftp.math.utah.edu/pub/rfc/rfc1631.txt`.

[37] E. D. Katz, M. Butler, et R. McGrath. « A Scalable HTTP Server : The NCSA Prototype ». Dans *Proceedings of the First International World-Wide Web Conference* (1994). `http://www.cern.ch/PapersWWW94/ekatz.ps`.

[38] A. E. Kaufman. « Volume Visualization ». *ACM Computing Surveys* **28**, numéro 1 (Mars 1996), 165–167.

[39] K. R. Koch, R. S. Baker et R. E. Alcouffe. « Solution of the First-Order Form of Threedimensional Discrete Ordinates Equations on a Massively Parallel Machine ». *Transactions of the American Nuclear Society* **65**, numéro 198 (1992).

[40] A. Krishnamurthy, K. E. Schauser, C. J. Scheiman, R. Y. Wang, D. E. Culler et K. Yelick. « Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines ». Dans *in Proceedings of Architectural Support for Programming Languages and Operating Systems* (1996). `http://http.cs.berkeley.edu/~arvindk/papers/asplos96.ps`.

[41] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of The Viewing Transformation*. PhD thesis, Stanford University, 1995. `http://www-graphics.stanford.edu/~lacroute/`.

[42] P. Lacroute. « Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization ». Dans *Proceedings of the 1995 Parallel Rendering Symposium* (1995). `http://www-graphics.stanford.edu/~lacroute/`.

[43] J. J. Li. *Algorithmes Parallèles pour la Synthèse d'Image sur Machine à Mémoire Distribuée*. Thèse de doctorat, Ecole normale supérieure de Lyon, 1992.

[44] K.-L. Ma et T. W. Crockett. « A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data ». Dans *Proceedings of the 1997 Parallel Rendering Symposium* (Octobre 1997), ACM SIGGRAPH, pp. 95–104. `ftp://ftp.icase.edu/pub/techreports/97/97-37.pdf`.

[45] K. McManus, S. Johnson et M. Cross. « Communication Latency Hiding in a Parallel Conjugate Gradient Method ». Dans *Proceedings of the 11th International Conference on Domain Decomposition Methods* (1998). `http://www.gre.ac.uk/~k.mcmanus/doc/dd11-98.ps`.

[46] S. Miguet et Y. Robert. « Elastic Load Balancing for Image Processing Algorithms ». Dans *Parallel Computation* (1991), H. Zima, éditeur, First International ACPC Conference.

[47] J. C. Mogul. « Network Behavior of a Busy Web Server and its Clients ». Rapport technique, 1995. `http://www.research.digital.com/wrl/publications/abstracts/95.5.html`.

[48] D. of Energy. « ASCI : Accelerated Strategic Computing Initiative ». Document électronique. `http://www.llnl.gov/asci`.

[49] D. J. Palermo. *Compiler Techniques for Optimizing Communication and Data Distribution for Distributed-Memory Multicomputers*. PhD thesis, University of Illinois at UrbanaChampaign, 1996. `http://www.ece.nwu.edu/cpdc/Paradigm/phd96.p.ps.Z`.

[50] C. Perkins. « IP Encapsulation within IP ». RFC 2003, 1996. `ftp://ftp.math.utah.edu/pub/rfc/rfc2003.txt`.

[51] M. Prieto, I. M. Llorente et F. Tirado. « Data Locality Exploitation in the Decomposition of Regular Domain Problems ». *IEEE Transactions on Parallel and Distributed Systems* **11**, numéro 11 (2000).

[52] L. Prylli. *BIP Messages User Manual for BIP 0.94*, Juin 1998. `http://www-bip.univ-lyon1.fr/bip.html`.

[53] L. Prylli. *Réseaux et systèmes de communication - Etude de logiciels de base*. Thèse de doctorat, École Normale Supérieure de Lyon, 1998. `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD1998/PhD1998-02.ps.Z`.

[54] M. J. Quinn et P. J. Hatcher. « On the Utility of Communication-Computation Overlap in Data-Parallel Programs ». *Journal of Parallel and Distributed Computing* (1996). `ftp://ftp.cs.unh.edu/pub/faculty/hatcher/jpdc96.ps.gz`.

[55] P. Ramet. *Optimisation de la Communication et de la Distribution des Données pour des Solveurs Parallèles Directs en Algèbre Linéaire Dense et Creuse*. Thèse de doctorat, LaBRI, Université Bordeaux I, Talence, France, Janvier 2000. `http://dept-info.labri.u-bordeaux.fr/~ramet/publications/these_ramet.ps%.gz`.

[56] D. Sarrut. « Imagerie Médicale et Segmentation du Cerveau ». Rapport technique, Laboratoire ERIC, Université Lyon 2, 1997. `http://eric.univ-lyon2.fr/francais/RR.html`.

[57] A. Scalable Highly Available Web Server. « Daniel M. Dias and William Kish and Rajat Mukherjee and Renu Tewari ». Dans *Proceedings of the Montreal INET'96 conference* (1996). `http://www.cs.utexas.edu/users/tewari/papers/compcon96.ps`.

[58] C. J. Scheiman et K. E. Schauser. « Evaluating Benefits of Communication Coprocessors ». *Journal of Parallel and Distributing Computing* **57** (1999), 236–256.

[59] R. Shahidi, R. Tombropoulos et R. P. Grzeszczuk. « Clinical Applications of Three Dimensional Rendering of Medical Data Sets ». Dans *Proceedings of the IEEE* (Mars 1998), volume 86.

[60] B. S. Siegell et P. Steenkiste. « Controlling application grain size on a network of workstations ». Dans *Supercomputing'95* (1995). `http://www.supercomp.org/sc95/proceedings/591_BSIE/SC95.PS`.

[61] W. Simpson. « IP in IP tunneling ». RFC 1853, 1995. `ftp://ftp.math.utah.edu/pub/rfc/rfc1853.txt`.

[62] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker et J. Dongarra. *MPI : the complete reference*. MIT Press, Cambridge, MA, USA, 1996. `http://www.netlib.org/utk/papers/mpi-book/mpi-book.ps`.

[63] A. Sohn et R. Biswas. « Communication Studies of DMP and SMP Machines ». Rapport technique NAS-97-005, NASA, 1997. `http://www-sci.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-97-005/NAS-%97-005.ps`.

[64] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H.Yamana, S. Sakai et Y. Yamaguchi. « Identifying the Capability of Overlapping Computation with Communication ». Dans *Proceedings of the PACT'96* (1996). http://www.cis.njit.edu/~sohn/papers/pact96.ps.gz.

[65] A. K. Somani et A. M. Sansano. « Achieving Robustness and Minimizing Overhead in Parallel Algorithms Through Overlapped Communication/Computation ». *The Journal of Supercomputing* **16** (2000), 27–52. http://www.icase.edu/Dienst/Repository/2.0/Body/ncstrl.icase/TR-97-8.

[66] V. Strumpen. « Communication Latency Hiding – Model and Implementation in High-Latency Computer Networks ». Rapport technique, Eidgennossische Technische Hochschule Zurich, 1994. ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/2xx/216.ps.gz.

[67] D. Sundaram-Stukel et M. K. Vernon. « Predictive Analysis of a Wavefront Application Using LogGP ». Dans *7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming* (Mai 1999). http://www.cs.wisc.edu/~vernon/papers/poems.99ppopp.ps.

[68] Y. Tanaka, M. Matsuda, K. Kubota et M. Sato. « Performance Improvement by Overlapping Computation and Communication on SMP Clusters ». Dans *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications* (1998), volume 1, pp. 275–282. http://pdplab.trc.rwcp.or.jp/pdperf/papers/pdpta98.ps.gz.

[69] O. Tatebe, Y. Kodama, S. Sekiguchi et Y. Yamaguchi. « Highly Efficient Implementation of MPI Point-To-Point Communication Using Remote Memory Operations ». Dans *Proceedings of 12th ACM International Conference on Supercomputing* (1998). http://www.etl.go.jp/~tatebe/research/paper/ICS98-A4.ps.gz.

[70] P. Teller et R. Oliver. « Specification of Sweep3D Memory Study ». http://pcl.cs.ucla.edu/projects/poems/memory.ps, 1998.

[71] M. K. Vernon. « LogP Modeling of Sweep3D Communication ». http://pcl.cs.ucla.edu/projects/poems/logP.ps, 1998.

[72] L. A. Westover. « SPLATTING : A Parallel, Feed-Forward Volume Rendering Algorithm ». Rapport technique TR91-029, Department of Computer Science, University of North Carolina - Chapel Hill, Juillet 1991. ftp://ftp.cs.unc.edu/pub/publications/techreports/91-029.ps.tar.Z.

[73] J. B. White et S. W. Bova. « Where's the Overlap? An Analysis of Popular MPI Implementations ». Rapport technique, ERDC MSRC, 1999. http://www.wes.hpc.mil/pet/tech_reports/reports/pdf/tr_9909.pdf.

[74] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, et D. Culler. « Using Smart Clients to Build Scalable Services ». Dans *Proceedings of the 1997 USENIX Conference* (1997). http://now.cs.berkeley.edu/SmartClients/usenix97.ps.

[75] J. Zory. *Contributions à l'optimisation de programmes scientifiques.* Thèse de doctorat, École des Mines de Paris, 1999. http://www.cri.ensmp.fr/doc/A-310.ps.gz.

[76] J. Zory et F. Desprez. « Performance des macros-pipelines dans les programmes data-paralleles ». Dans *Renpar'97* (1997). `http://www.cri.ensmp.fr/~zory/Files/dea_zory.ps.gz`.

Recouvrement des communications : du matériel au logiciel

Résumé

Le but de cette thèse est l'étude des recouvrements des communications sur des machines parallèles à mémoire distribuée. Les recouvrements des communications sont des techniques très utilisées dans le but d'accélérer les exécutions de programmes parallèles. Notre première contribution est l'étude détaillée d'une implantation efficace de la bibliothèque de communication MPI sur une grappe de PC interconnectée par un réseau rapide. Cette étude montre tout d'abord que le matériel permet de recouvrir le transfert des données sur le réseau. Dans un deuxième temps, nous montrons pourquoi le logiciel de communication inter-processus, MPI, empêche le recouvrement des communications par le calcul. La sémantique des communications et l'implication du processeur de calcul dans la communication en sont les principales causes.

Notre deuxième contribution est la validation des équations de gain d'un pipeline modélisées par Zory, par l'intermédiaire de l'application Sweep3D. Cette application nous a permis de montrer que sur des données de grande taille, l'utilisation de pipelines logiciels sur P processeurs, peut apporter un gain en temps d'exécution s'approchant d'un facteur P.

Notre troisième contribution est l'évaluation a priori de la meilleure configuration de l'espace des processeurs, c'est-à-dire mono, bi ou tridimensionnel. En établissant les équations des temps d'exécution d'une application par vague pour ces trois types de distribution en fonction des paramètres de la machine cible et de l'application, nous pouvons déterminer la distribution qui donnera le plus petit temps d'exécution.

Mots clés : parallélisme, MPI, grappe de PC, recouvrements des communications, pipelines logiciels.

Communication overlap : from hardware to software

Abstract

In this thesis we study communication overlap on distributed memory parallel machines. Communication overlap is very often used to reduce execution times of parallel programs.

We first study a fast implementation of the communication library MPI on a pile of PCs interconnected with a fast network. To begin with, we show that using the available hardware overlapping data transfer over the network is possible. Secondly, we explain why, despite of the overlap provided by the hardware and basic communication software the available MPI interface does not provide any overlap.

In a second time we validate equations modelising software pipelining gain given by Zory, through the Sweep3D application. Using this application we showed that using software pipelining on P processors on large data sets can reduce the execution time nearly P times.

The last part of this thesis tackles the problem of determining before execution the best processor configuration (mono, bi or tri-dimensional space) for wavefront applications. We modelise the problem with solvable time equations of wavefront for each processor configuration.

Keywords : MPI, pile of PCs, communication overlap, software pipelining.